EFOP-3.4.3-16-2016-00014

READER

Selected Chapters from Algorithms

author: András Erik Csallner

Department of Applied Informatics

University of Szeged

reviewer: Attila Tóth

This teaching material has been made at the University of Szeged, and supported by the European Union. Project identity number: EFOP-3.4.3-16-2016-00014

University of Szeged
Address: 13 Dugonics tér, 6720 Szeged, Hungary
www.u-szeged.hu
www.szechenyi2020.hu

The aim of this reader and learning outcomes

The present reader can be used at higher level studies in algorithms at college and university courses. Although the topics include some higher mathematics and computer science, no more prior knowledge for understanding is necessary than a secondary school can provide. The aim is to teach the students those basic notions and methods that are used in informatics to develop and analyze algorithms and data structures.

This reader can also be considered as a uniform learning guide to basic algorithms studies. If used this way it aims at given **learning outcomes** which we list below.

Knowledge	Skills	Attitude	Autonomy/ responsibility
The student knows the elements and properties of algorithms, is aware of different algorithm description methods and special algorithmic approaches.	They are capable to develop simple algorithms of different approaches and recognizes a method not to be an algorithm.	They are curious to find the right approach to solve simple problems algorithmically, and is open to experiment with unusual solutions.	They apply algorithm description methods with responsibility. They autonomously convert iterative and recursive algorithms between each other.
The student is aware of the basic algorithmic complexity notation.	They can analyze both iterative and recursive algorithms and express the magnitudes of their complexity.	They strive to find the most efficient way of solving problems algorithmically or to find the best data structure for a given problem.	They independently state upper bounds on the running time of different types of simple algorithms.

The student knows the basic linear and nonlinear data structures, their operators, and the time complexity of the operators.	They are able to design an array or a linked list together with their operations using an algorithm description method.	They are open to use different linear data structure approaches to find the best solution with the fastest operators.	They independently implement stacks, queues, and other dynamic data structures using different basic linear data types.
The student knows some essential sorting algorithms, like the insertion, merge, heap and quick sorting algorithms, and is aware of the selection problem and its algorithmic solution in linear time.	They are capable of adopting the appropriate sorting algorithm for a given problem considering its properties like stability or being on-line or in-place.	They strive to apply the most appropriate sorting algorithm for a given application and is committed to take all necessary circumstances into account.	They can choose the right sorting method for a given problem on hir own. They check the necessity of using the five-step algorithm for the selection problem independently.
The student knows the methodology and elements of the dynamic programming approach	They are able to design a dynamic programming algorithm for a simple related problem.	They are committed to always use dynamic programming where a recursion would be inefficient.	They can decide whether a dynamic approach is necessary for a recursively defined problem.
The student is aware of higher amortized analysis methods,	They can analyze an algorithm or operators on a given data	They are curious about using the best method for an amortized	They can monitor the process of an iterative method on their own, and

like the aggregate analysis, the accounting and potential methods.	structure using any of the known amortized analysis method.	analysis to get the sharpest bound on the time complexity function.	find out if amortized analysis is necessary. They autonomously choose the best way to analyze the algorithm.
The student knows how and when greedy algorithms work, and is aware of the solution of the activity-selection problem and knows the algorithm to create Huffman codes.	They are capable to construct an algorithm for a corresponding problem and determine whether it delivers an optimal solution for it.	They strive to use the greedy approach where possible to get a fast and straightforward solution to an optimization problem.	They responsibly apply greedy algorithms after independently setting out the conditions to be met to guarantee an optimal solution for the given problem.
The student knows how to represent graphs on a computer, and knows basic graph walk algorithms including the solution of the shortest paths problem.	They are able to implement the graph representations and Dijkstra's shortest paths algorithm.	They strive to find the optimal way how to represent a graph for a particular problem and is curious of adopting graph walk methods to different applications.	They can check independently whether a graph model is appropriate for a simple problem.
The student is aware of basic problems of	They are able to give pseudocodes to particular	They are committed to use simple computer	They autonomously apply basic

computational geometry and their solution, including the problem of finding intersecting line segments and creating the convex hull of a set of points.	problems from computational geometry using cross products and logical formulae.	geometry algorithms instead of algebraic solutions where possible. They strive to solve geometry problems in an optimal way.	computer geometric solutions to given simple problems like finding the convex hull of polygons instead of finding the convex hull of points.

András Erik Csallner

Selected Chapters from Algorithms

Reader

Department of Applied Informatics
University of Szeged

Manuscript, Szeged, 2020

Contents

About Algorithms

In everyday life we could simply call a sequence of actions an algorithm, however, since we intend to talk about algorithms all through this course, we ought to define this notion more carefully and rigorously. A finite sequence of steps (commands, instructions) for solving a certain sort of problems is called an ***algorithm*** if it can provide the solution to the problem after executing a finite number of steps. Let us examine the properties and elements of algorithms closer.

The notion of ***finiteness*** occurs twice in the definition above. Not only does the number of steps have to be finite but each step also has to be finished in finite time (e.g., a step like "list all natural numbers" is prohibited). Another point is ***definiteness***. Each step has to be defined rigorously, therefore the natural languages that are often ambiguous are not suitable for an algorithm's description. We will use other methods for this purpose, like flow diagrams or pseudocodes (see later in this chapter). Moreover, an algorithm has to be ***executable***. This means that every step of the algorithm must be executable (e.g., a division by zero is not a legal step).

Each algorithm has an input and an output. These are special sets of data in a special format. The ***input*** is a set of data that has to be given prior to beginning the execution of the algorithm but can be empty in some cases (e.g., the algorithm "open the entrance door" has no further input, the algorithm itself determines the task). The ***output*** is a set of data, as well, however, it is never an empty set. The output depends on the algorithm and the particular input. Both the input and the output can be of any kind of data: numbers, texts, sounds, images, etc.

Structured programming

When ***designing*** an algorithm we usually follow the ***top-down*** strategy. This method breaks up the problem to be solved by the algorithm under design into subproblems, the subproblems into further smaller parts, iterating this procedure until the resulting building blocks can be solved directly. The basic method of the top-down program design was worked out by E.W. DIJKSTRA in the 1960s (1), and says that every algorithm can be broken up into steps coming from the following three basic classes of structural elements:

- ***Sequence*** is a series of actions to be executed one after another.

- ***Selection*** is a kind of decision where only one of a given set of actions can be executed, and the algorithm determines the one.
- ***Iteration*** (also called ***repetition***) means a frame that regulates the repeat of an action depending on a condition. The action is called the body of the iteration loop.

This means that all low or high level algorithms can be formulated as a series of structural elements consisting only of the three types above. Hence, for any algorithm description method it is enough to be able to interpret the three types above.

Algorithm description methods

There are various description methods. They can be categorized according on the age when they were born and the purpose they were invented for. From our present point of view the two most important and most widely used are flow diagrams and pseudocodes. There are several methods which support structured algorithm design better and also ones which reflect the object-oriented approach and programming techniques. But for describing structured algorithmic thoughts, methods, procedures or whatever we call them, the two mentioned above are the most convenient to use and most understandable of all.

We will demonstrate the two kinds of algorithm description methods on a simple problem, the problem of finding the least number in a given set of numbers. Thus, the input is a set of numbers, usually given as a sequence, and the output the least among them. The algorithm works as follows. We consider the first number in the sequence as the actual least one, and then check the remaining numbers. If a number less than the actual minimum is found, we replace our choice with the newly found. After all numbers have been checked, the actual minimum is the minimum of all numbers at the same time.

Flow diagrams

A ***flow diagram*** consists of plane figures and arrows connecting them in a directed way. There is more than one standard, yet we have a look at only one of them here.

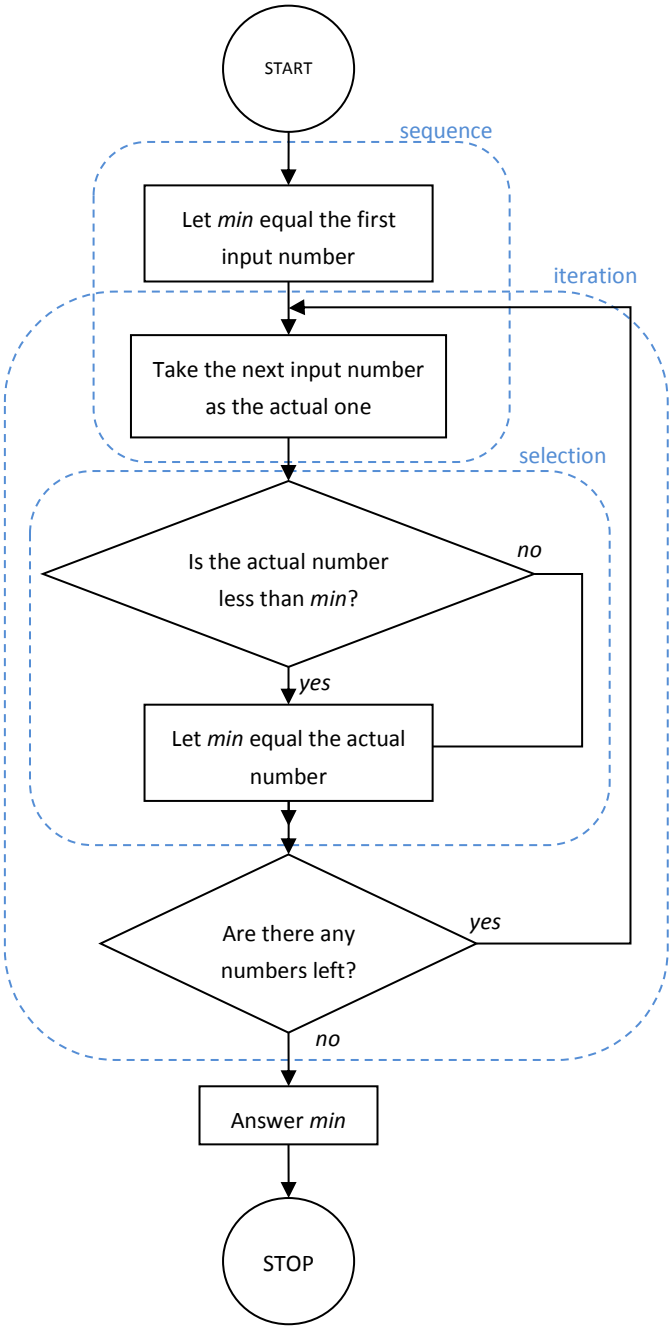


Figure 1. The flow diagram of the minimum finding problem.

Circle: A circle denotes a starting point or an endpoint containing one of the words START or STOP. There can be more than one endpoint but only one starting point.

Rectangle: A rectangle always contains an action (command).

Diamond: A diamond (rhombus) formulates a simple decision, it contains a yes/no question, and has two outgoing arrows denoted with a *yes* and a *no*, respectively.

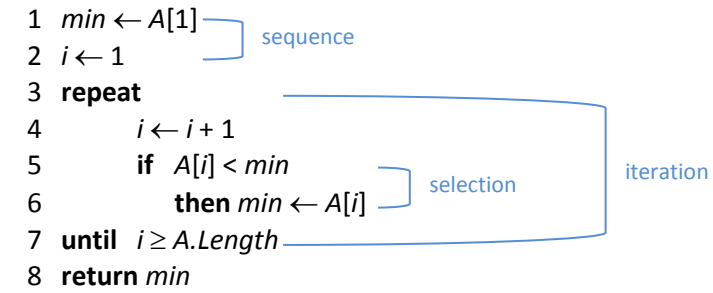
The flow diagram of the problem of finding the least element can be seen in *Figure 1*. Examples of the three basic structural elements can be found in the light dotted frames (the frames are not part of the flow diagram).

Pseudocode

The pseudocode is the closest description method to any general structured programming language such as Algol, Pascal or C, and the structured part of Java and C#. Here we assume that the reader is familiar with the basic structure of at least one of these. The exact notation of the pseudocode dialect we are going to use is the following.

- The assignment instruction is denoted by an arrow (←).
- The looping constructs (**while-do**, **for-do**, and **repeat-until**) and the conditional constructs (**if-then** and **if-then-else**) are similar to those in Pascal.
- Blocks of instructions are denoted by indentation of the pseudocode.
- Objects are handled as references (hence, a simple assignment between objects does not duplicate the original one, only makes a further reference to the same object). Fields of an object are separated by a dot from the object identifier (*object.field*). If an object reference is empty, it is referred to as a NIL value. Arrays are treated as objects but the indices are denoted using brackets, as usual.
- Parameters are passed to a procedure by value by default. For objects this means the value of the pointer, i.e., the reference.

The following pseudocode formulates the algorithm of finding the least number of a set of numbers given in an array *A*. Although the problem can be solved in several ways, the one below follows the structure of the flow diagram version given in *Figure 1*.



Exercises

1 Give a simple real-life example to the design of an algorithm using the top-down strategy. Draw a flow diagram for the algorithm.

2 Draw a flow diagram for the following task: Take a given n number of books from the shelf and put them into a bin, supposing you can hold at most h ($\leq n$) books at a time in your hands.

3 Write a pseudocode for the task described in the previous exercise.

4 Write a computer program simulating the task of exercise 1.

Type algorithms

Algorithms can be classified by more than one attribute, one of these is to consider the number and structure of their input and output. From this point of view four different types of algorithms can be distinguished (2).

1 Algorithms assigning a single value to a sequence of data.
2 Algorithms assigning a sequence to another sequence.
3 Algorithms assigning a sequence to more than one sequence.
4 Algorithms assigning more than one sequence to a sequence.

Some examples for the particular type algorithms are given in the following enumeration.

- Algorithms assigning a single value to a sequence are among others
  - *sequence calculations* (e.g. summation, product of a series, linking elements together, etc.),
  - *decision* (e.g. checking whether a sequence contains any element with a given property),

  - *selection* (e.g. determining the first element in a sequence with a given property provided we know that there exists at least one),
  - *search* (e.g. finding a given element),
  - *counting* (e.g. counting the elements having a given property),
  - *minimum or maximum search* (e.g. finding the least or the largest element).
- Algorithms assigning a sequence to another sequence:
  - *selection* (e.g. collect the elements with a given property of a sequence),
  - *copying* (e.g. copy the elements of a sequence to create a second sequence),
  - *sorting* (e.g. arrange elements into an increasing order).
- Algorithms assigning a sequence to more than one sequence:
  - *union* (e.g. linking two sequences together excluding duplicates – set union),
  - *intersection* (e.g. producing the set intersection of the elements of two sequences),
  - *difference* (e.g. producing the set difference of two sequences),
  - *uniting sorted sequences* (merging / combing two ordered sequences).
- Algorithms assigning more than one sequence to a sequence:
  - *filtering* (e.g. filtering out elements of a sequence having given properties).

Exercises

5 Write a pseudocode that calculates the product of a series given as an array parameter.

6 Write a pseudocode that determines the index of the second least element in an unsorted array consisting of pair-wise different values. (The second least element is that which is greater than exactly one other element in the array.)

7 Write a pseudocode that finds a number that is smaller than at least one of the elements preceding it in an input array.

8 Write a pseudocode that counts how many odd numbers there are in a given array.

9 Write pseudocodes for the set operations union, intersection and difference. The sets are stored in arrays.

10 Write a pseudocode that combs two sorted arrays in a third array.

Special algorithms

As we have seen, algorithms consist of sequences of algorithmic steps (instructions) where some series of steps can be repeatedly executed (iteration). Because of this latter feature these algorithms are jointly called ***iterative*** algorithms. An iterative algorithm usually has an initialization part consisting of steps (initialize variables, open files, etc.) to be executed prior to the iteration part itself, subsequently the loop construct is executed. However, there are algorithms slightly differing from this pattern, although they consist of simple steps in the end.

Recurrences

We call an algorithm ***recursive*** if it refers to itself. A recurrence can be ***direct*** or ***indirect***. It is direct if it contains a reference to itself, and it is indirect if two methods are mutually calling each other.

A recursive algorithm always consists of two parts, the base case and the recursive case. The ***base criterion*** decides which of them has to be executed next. Roughly speaking, if the problem is small enough, it is solved by the base case directly. If it is too big for doing so, it is broken up into smaller subproblems that have the same structure as the original, and the algorithm is recalled for these parts. The process obviously ends when all arising subproblems “melt away”.

A typical example of a recursive solution is the problem known as the ***Towers of Hanoi***, which is a mathematical puzzle. It consists of three rods, and a number of disks of different sizes which can slide onto any rod. The puzzle starts with the disks in a neat stack in ascending order of size on one rod, the smallest at the top, thus making a conical shape. The objective of the puzzle is to move the entire stack to another rod, obeying the following rules:

1 Only one disk may be moved at a time.
2 Each move consists of taking the upper disk from one of the rods and sliding it onto another rod, on top of the other disks that may already be on that rod.
3 No disk may be placed on top of a smaller disk.

The recursive solution of moving *n* disks from the first rod to the second using the third one can be defined in three steps as demonstrated in *Figure 2*.

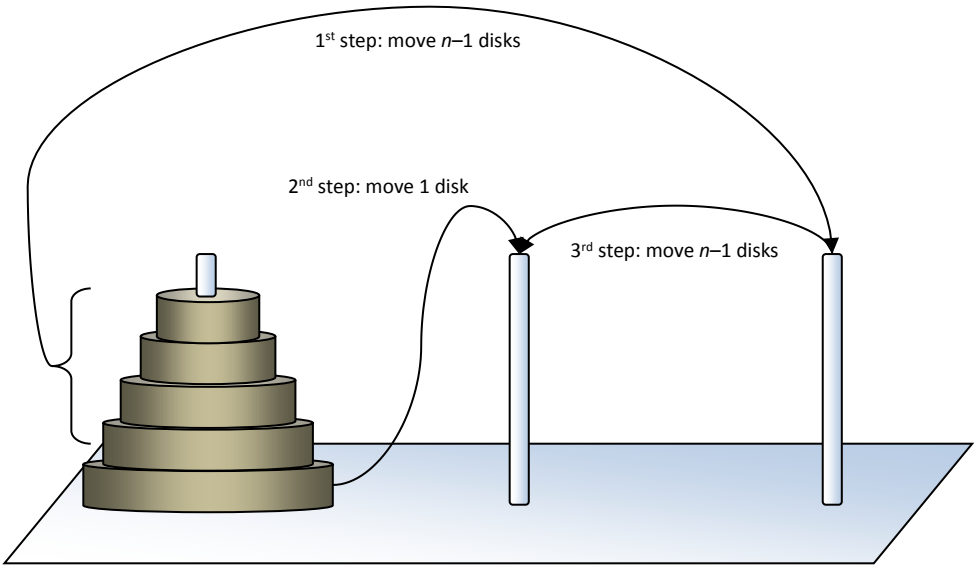


Figure 2. Three steps of recursive solution

The pseudocode of the procedure providing every single disk move is given below. In order to move *n* disks from the first to the second rod using the third one, the method has to be called with the parameters **TowersOfHanoi**(*n*,1,2,3).

```
TowersOfHanoi(n,FirstRod,SecondRod,ThirdRod)
1  if  n > 0
2      then  TowersOfHanoi(n – 1,FirstRod,ThirdRod,SecondRod)
3            write "Move a disk from " FirstRod " to " SecondRod
4            TowersOfHanoi(n – 1, ThirdRod,SecondRod,FirstRod)
```

The base criterion is the condition formulated in line 1. The base case is $n = 0$ when no action is necessary. If $n > 0$ then all disks are moved using recurrence in lines 2 and 4.

Backtracking algorithms

Backtracking algorithms are systematic trials. They are used when during the solution of a problem a selection has to be passed without the information which is needed for the algorithm to come to a decision. In this case we choose one of the possible branches and if our choice turns out to be false, we return (backtrack) to the selection and choose the next possible branch.

A typical example of backtracking algorithms is the solution to the ***Eight Queens Puzzle*** where eight chess queens have to be placed on a chessboard in such a way that no two queens attack each other.

To solve the problem we place queens on the chessboard column by column selecting a row for each new queen where it can be captured by none of the ones already placed. If no such row can be found for a new queen, the algorithm steps a column back and continues with trying to replace the previously placed queen to a new safe row in its column (backtracking). The pseudocode of this procedure drawing all possible solutions is given below.

EightQueens

```
 1  column ← 1
 2  RowInColumn[column] ← 0
 3  repeat
 4        repeat  RowInColumn[column] ← RowInColumn[column] + 1
 5        until    IsSafe(column,RowInColumn)
 6        if   RowInColumn[column] > 8
 7           then  column ← column – 1
 8           else  if  column < 8
 9                    then  column ← column + 1
10                          RowInColumn[column] ← 0
11                    else  draw chessboard using RowInColumn
12  until column = 0
```

Each element of the array *RowInColumn* stores the row number of the queen in the column given by its index, and equals 0 if there is none. The procedure **IsSafe**(*column*,*RowInColumn*) returns true if a queen placed to the coordinates (*column*,*RowInColumn*[*column*]) is not attacked by any of the queens in the previous columns or is outside the chessboard, i.e., *RowInColumn*[*column*] > 8 here. In the latter case a backtracking step is needed in line 7.

Exercises

11 Write a pseudocode solving the warder's problem: There is a prison having 100 cells all of which are locked up. The warder is bored, and makes up the following game. He opens all cells, then locks up every second, subsequently opens or locks up every third depending on its state, then every forth, etc. Which cells are open by the end of the warder's "game".

12 Write the pseudocode of **IsSafe**(*column*,*RowInColumn*) used in the pseudocode of **EightQueens**.

13 Let us suppose that there are 10 drivers at a car race. Write a pseudocode which lists all possibilities of the first three places. Modify your pseudocode so that it works for any n drivers and any first $k(\leq n)$ places. (We call this the ***k-permutation of n elements***.)

14 Let us suppose we are playing a lottery game: we have 90 balls in a pot numbered from 1 to 90, and we draw 5 of them. Write a pseudocode which lists all possible draws. Modify your pseudocode so that it works for any n balls and any $k(\leq n)$ drawn of them. (We call this the ***k-combination of n elements***.)

Analysis of algorithms

After an algorithm is constructed ready to solve a given problem, two main questions arise at once. The first is whether the method always gives a correct result to the problem, and the second is how it manages the resources of a computer while delivering this result. The answer to the first question is simple, it is either "yes" or "no", even if it is usually not easy to prove the validity of this answer. Furthermore, the second question involves two fields of investigations: that of the storage complexity and the time complexity of the algorithm.

Complexity of algorithms

Storage and time complexity are notions which express the amount of storage and time the algorithm consumes depending on the size of the input. In order to be able to interpret and compare functions describing this dependence, a piece of the used storage or time is considered as ***elementary*** if it does not depend on the size of the input. For instance, if we consider the problem of finding the minimum as seen previously, then if the cardinality of the set to be examined is denoted by n, the storage complexity turns out to be $n+1$, while the time complexity equals n. How do we come to these results?

The storage complexity of the algorithm is $n+1$ because we need n elementary storage places to store the elements to be examined, and we need one more to store the actual minimum. (For the sake of simplicity here we are disregarding the loop variable.) We do not use any unit of measurement since if an element consists of 4 bytes, then $(n+1)\cdot 4$ bytes are needed in all, but if a single element takes 256 bytes then it is $(n+1)\cdot 256$ bytes of storage that the algorithm occupies. Therefore the algorithm's tendency of storage occupation is $n+1$ independently of the characteristics of the input type.

Note that in reality the algorithm does not need to store all the elements at the same time. If we modified it getting the elements one by one during execution, the storage complexity would work out as 1.

The same methodology of analysis can be observed in calculating time complexity. In our example it is *n*. Where does this come from? For the analysis of the algorithm we will use the pseudocode of the method **Minimum** (page 5) (the *n*=*A.length* notation is used here). Lines 1 and 2 are executed together and only once so they can be considered as a single elementary algorithmic step due to our definition above, i.e. the execution time of the first two lines does not depend on the input's size. The same holds for the body of the iteration construct in lines 4 through 6, which is executed $n-1$ times (from $i=1$ to $i=n-1$ when entering the body). Hence the time complexity of the whole algorithm equals $1+(n-1)=n$.

The asymptotic notation

If, for example two algorithmic steps can be considered as one, or even a series of tens or hundreds of steps make up a single elementary step, the question arises whether there is any sense in distinguishing between *n* and $n+1$. Since $n=(n-1)+1$ while $n+1=(n-1)+2$, and we have just learned that it is all the same whether we execute one or two steps after having taken $n-1$; *n* and $n+1$ seem to denote the same complexity. This is exactly the reason for using asymptotic notation which eliminates these differences and introduces a uniform notation for the complexity functions.

Let us define the following set of functions. We say that a function f has the ***asymptotic upper bound*** (or is asymptotically bounded from above by) g and we denote this as $f(x) = O(g(x))$ (correctly we should use the "set element of" notation: $f(x) \in O(g(x))$) if

$$(\exists C, x_0 > 0)\ (\forall x \geq x_0)\ 0 \leq f(x) \leq C \cdot g(x)$$

holds. The notation is called Landau (3) notation (or big O notation). The word asymptotic is used here because the bounding property is fulfilled only from a given threshold x_0 upwards, and only applying a multiplier *C* for the bounding function g. Using this definition all the functions $f_0(n) = n$, $f_1(n) = n + 1$, $f_2(n) = n + 2$, etc. are of $O(n)$ delivering the expected result, even a linear function such as $f(x) = 3x - 2$ is $O(n)$. However, it is easy to check that all linear functions are of $O(n^2)$ at the same time. This is no wonder because the *O*() notation formulates only an asymptotic upper bound on the function *f*, and *n* and

n^2 are both upper bounds of n, although the latter one is not a tight bound. To make this bound tight, we extend our definition for the same function to be also a lower bound. We say that the function *f asymptotically equals* function *g* and we denote this relation as $f(x) = \theta(g(x))$ if

$$(\exists c, C, x_0 > 0)\ (\forall x \geq x_0)\ 0 \leq c \cdot g(x) \leq f(x) \leq C \cdot g(x)$$

holds. Certainly, in this case $n \neq \theta(n^2)$ since no positive number c exists that satisfies the inequality $c \cdot n^2 \leq n$ for all natural numbers.

The notion of an asymptotic lower bound can be defined in a similar way, and the notion of the proper version of asymptotic lower and upper bounds also exists but we will use only the two defined above.

Formulating time complexity

When we use asymptotic notation, the algorithm for finding the minimum of n elements has a time complexity of $T(n) = O(n)$. A strange consequence of this notation is that due to the formal definition time complexity of finding the minimum of n numbers is the same as that of finding the minimum of $2n$ numbers, i.e. $T(2n) = O(n)$. Obviously, the execution of the algorithm would take twice as long in practice. Nevertheless, the asymptotic notation delivers only the tendency of the awaited time consumption, delivering this strange result. This means that if the time complexity is $O(n)$, i.e. linear in n, then having two, three or four times more data, the algorithm will approximately run two, three and four times longer, respectively. For instance, if an algorithm has $T(n) = \theta(n^2)$ time complexity, this means that increasing the size of its input to the double or triple, it will approximately run four and nine times longer, respectively.

If we merge an algorithm's groups of steps which are not repeated by a number of repetitions depending on the input's size, the time complexity can easily be formulated. However, if the algorithm under consideration is recursive, this problem can be far more difficult. Let us revisit the puzzle Towers of Hanoi described on page 8. In the pseudocode the solution consists of only three steps, however, the time complexity of the recursive calls in lines 2 and 4 is not explicitly known. Moreover, we make the observation that the formula for the time complexity of any recursive algorithm is itself recursive. In this case, if we take the time complexity of the pseudocode line by line, the time complexity of the method **TowersOfHanoi** is the following.

$$T(n) = 1 + T(n-1) + 1 + T(n-1) = 2T(n-1) + 2 = 2T(n-1) + 1$$

The last equation looks weird from a mathematical point of view, but since this is a time complexity function, any constant number of steps can be merged into one single step.

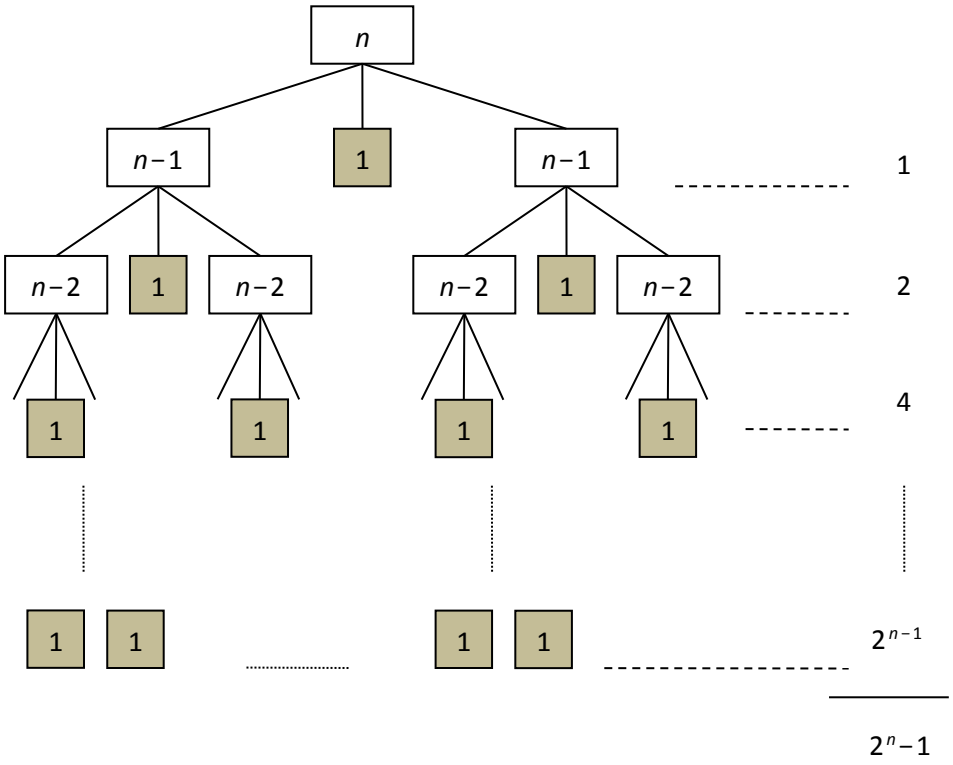


Figure 3. Recursion tree of the Towers of Hanoi puzzle

The difficulty of calculating this function is due to the recurrence in it. In this particular case it is not too hard to find the explicit formula $T(n) = 2^n - 1$ but in general it is a very difficult, sometimes insoluble problem. A simple way of determining the time complexity of a recursive algorithm, or at least giving an asymptotic upper bound of it is using a ***recursion tree***. The recursion tree of the algorithm for the problem Towers of Hanoi is shown in *Figure 3*.

In the recursion tree we draw the hierarchic system of the recurrences as a tree graph. Subsequently we calculate the effective number of algorithmic steps (in

this example the number of disks being moved is meant, denoted by shaded boxes in *Figure 3*) for each level, i.e. "row of the tree". At the end, these sums are summarized, delivering the time complexity of the algorithm.

In our examples above the algorithms have always been supposed to work in one way only. In other words, given *n* elements, the way of finding the minimum does not depend on where we find it among the elements; or given *n* disks on one of three rods, the way of moving the disks to another rod does not depend on anything except for the initial number of disks. Nevertheless, there exist algorithms where the procedure depends also on the quality of the input, not only on the quantity of it. If the problem is to find a certain element among others (e.g., a person in a given list of names), then the operation of the algorithm can depend on where the wanted person is situated in the list. Let us consider the easiest search algorithm, the so-called linear search, taking the elements of the list one by one until it either finds the wanted one or runs out of the list.

LinearSearch(*A,w*)
1 $i \leftarrow 0$
2 **repeat** $i \leftarrow i + 1$
3 **until** $A[i] = w$ **or** $i = A.Length$
4 **if** $A[i] = w$ **then return** *i*
5 **else return** NIL

The ***best case*** is obviously if we find the element at the first position. In this case $T(n) = O(1)$. Note that to achieve this result it is enough to assume to find *w* in one of the first positions (*i* is a constant). The ***worst case*** is if it is found at either the last place or if it isn't found at all. Thus, the number of iterations in lines 2 and 3 of the pseudocode equals the number *n* of elements in the list, hence $T(n) = \theta(n)$. In general, none of the two extreme cases occur very frequently but a kind of average case operation happens. We think that this is when the element to be found is somewhere in the middle of the list, resulting in a time complexity of $T(n) = (n + 1)/2 = \theta(n)$. Indeed, the ***average case*** time complexity is defined using the mathematical notion of mean value taking all possible inputs' time complexities into consideration. Since on a finite digital computer the set of all possible inputs is always a finite set, this way the mean value is unambiguously defined. In our example if we assume that all possible input situations including the absence of the searched element occur with the same probability this is the following.

$$T(n) = \frac{1 + 2 + 3 + \cdots + n + n}{n + 1} = \frac{n(n+1) + 2n}{2(n+1)} = \frac{n}{2} + \frac{n}{n+1} \leq \frac{n}{2} + 1 = O(n)$$

Exercises

15 Prove that the function $f(x) = x^3 + 2x^2 - x$ is $\theta(x^3)$.

16 Prove that the function $f(x) = 2x^2 - 3x + 2$ is $O(x^3)$ but is not $\theta(x^3)$.

17 Prove that the time complexity of the Towers of Hanoi is $\theta(2^n)$.

18 Prove that in the asymptotic notation $\theta(\log n)$ the magnitude of the function is independent from the base of the logarithm.

19 Determine an upper bound on the time complexity of the algorithm called the Sieve of Eratosthenes for finding all primes not greater than a given number *n*.

20 Determine the time complexity of the recursive algorithm for calculating the n^{th} Fibonacci number. Compare your result with the time complexity of the iterative method.

Data Structures

Most algorithms need to store data, and except for some very simple examples, sometimes we need to store a huge amount of data in structures which serve our purposes best. Depending on the problem our algorithm solves, we might need different operations on our data. And because there is no perfect data structure letting all kinds of operation work fast, the appropriate data structure has to be chosen for each particular problem.

Linear data structures

Arrays vs. linked lists

Arrays and linked lists are both used to store a set of data of the same type in a linear ordination. The difference between them is that the elements of an ***array*** follow each other in the memory or on a disk of the computer directly, while in a ***linked list*** every data element (***key***) is completed with a link that points at the next element of the list. (Sometimes a second link is added to each element pointing back to the previous list element forming a ***doubly linked list***.) Both arrays and linked lists can manage all of the most important data operations such as searching an element, inserting an element, deleting an element, finding the minimum, finding the maximum, finding the successor and finding the predecessor (the latter two operations do not mean finding the neighbors in the linear data structure, but they concern the order of the base set where the elements come from). Time complexity of the different operations (in the worst case if different cases are possible) on the two different data structures is shown in *Table 1*.

Arrays are easier to handle because the elements have a so-called ***direct-access arrangement***, which means they can be directly accessed knowing their indices in constant time, whereas an element of a linked list can only be accessed indirectly through its neighbor in the list finally resulting in a linear time complexity of access in the worst case. On the other hand, an array is inappropriate if it has to be modified often because the insertion and the deletion both have a time complexity of $O(n)$ even in the average case.

	Search	Insert	Delete	Minimum	Maximum	Successor	Predecessor
Array	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Linked list	$O(n)$	$O(1)$	$O(1)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$

Table 1. Time complexity of different operations on arrays and linked lists (worst case).

Representation of linked lists

Linked lists can be implemented using record types (struct or class) with pointers as links if available in the given programming language but simple pairs of arrays will do, as well (see later in this subsection). A linked list is a series of elements each consisting of at least a key and a link. A linked list always has a head pointing at the first element of the list (upper part of *Figure 4*). Sometimes dummy head lists are used with single linked lists where the dummy head points at the leading element of the list containing no key information but the link to the first proper element of the list (lower part of *Figure 4*).

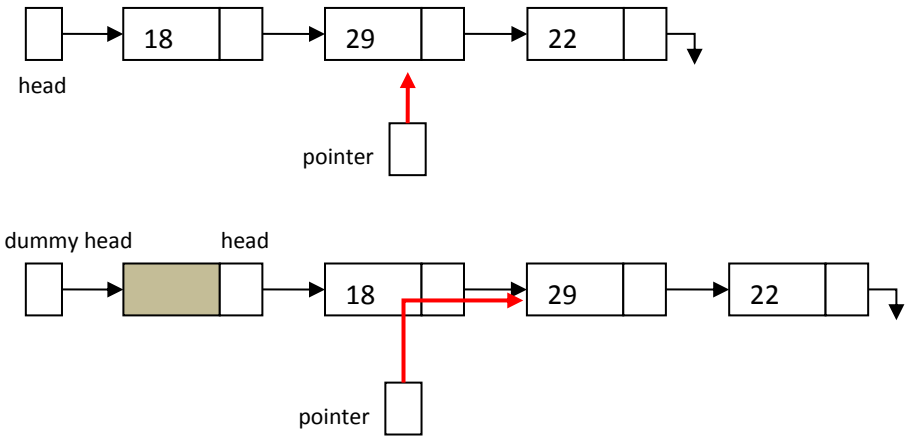


Figure 4. The same keys stored in a simple linked list and a dummy head linked list.

All kinds of linked lists can be implemented as arrays. In *Figure 5* the dummy head linked list of *Figure 4* is stored as a pair of arrays. The dummy head contains the index 3 in this example.

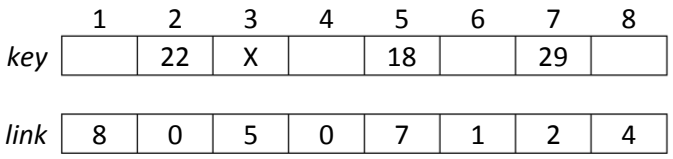


Figure 5. A dummy head linked list stored in a pair of arrays. The dummy head is pointing at 3. The head of the garbage collector list is pointing at 6.

The advantage of dummy head linked lists is that they promote the usage of double indirection (red arrow in the lower part of *Figure 4*) by making it possible to access the first proper element in a similar way like any other element. On the other hand, double indirection maintains searching a position for an insertion or a deletion. This way after finding the right position, the address of the preceding neighbor is kept stored in the search pointer. For instance, if the element containing the key 29 in *Figure 4* is intended to be deleted, then the link stored with the element containing 18 has to be redirected to point at the element containing 22. However, using a simple linked list (upper part of *Figure 4*) the address of 18 is already lost by the time 29 is found for deletion.

The following two pseudocodes describe the algorithm of finding and deleting an element from a list on a simple linked list and a dummy head linked list, respectively, both stored in a pair of arrays named *key* and *link*.

```
FindAndDelete(toFind,key,link)
 1  if  key[link.head] = toFind
 2      then  toDelete ← link.head
 3            link.head ← link[link.head]
 4            Free(toDelete,link)
 5      else  toDelete ← link[link.head]
 6            pointer ← link.head
 7            while toDelete ≠ 0  and  key[toDelete] ≠ toFind
 8                  do pointer ← toDelete
 9                     toDelete ← link[toDelete]
10            if  toDelete ≠ 0
11               then  link[pointer] ← link[toDelete]
12                     Free(toDelete,link)
```

Procedure **Free**(*index,link*) frees the space occupied by the element stored at *index*. In lines 1-4 given in the pseudocode above the case is treated when the key to be deleted is stored in the first element of the list. In the **else** clause beginning in line 5 two pointers (indices) are managed, *pointer* and *toDelete*. Pointer *toDelete* steps forward looking for the element to be deleted, while *pointer* is always one step behind to enable it to link the element out of the list if once found.

The following realization using a dummy head list is about half as long as the usual one. It does not need an extra test on the first element and does not need two pointers for the search either.

```
FindAndDeleteDummy(toFind,key,link)
 1  pointer ← link.dummyhead
 2  while link[pointer] ≠ 0  and  key[link[pointer]] ≠ toFind
 3        do   pointer ← link[pointer]
 4  if   link[pointer] ≠ 0
 5     then   toDelete ← link[pointer]
 6            link[pointer] ← link[toDelete]
 7            Free(toDelete,link)
```

Dummy head linked lists are hence more convenient to use for storing lists, regardless of whether they are implemented with memory addresses or indices of arrays.

In *Figure 5* the elements seem to occupy the space randomly. Situations like this often occur if several insertions and deletions are done continually on a dynamic linked list stored as arrays. The problem arises where a new element should be stored in the arrays. If we stored a new element at position 8, positions 1, 4 and 6 would stay unused, which is a waste of storage. Nevertheless, if after this a new element came again, we would find no more free storage (position 9 does not exist here). To avoid situations like these, a kind of ***garbage collector*** can be used. This means that the unused array elements are threaded to a linked list, simply using the index array *link* (in *Figure 5* the head of this list is pointing at 6). Thus, if a new element is inserted into the list, its position will be the first element of the garbage list; practically the garbage list's first element is linked out of the garbage list and into the proper list (**Allocate**(*link*)) getting the new data as its key. On the other hand, if an element is deleted, its position is threaded to the garbage list's beginning (**Free**(*index,link*)). Initially, the list is empty and all elements are in the garbage list.

The following two pseudocodes define the algorithms for **Allocate**(*link*) and **Free**(*index,link*).

```
Allocate(link)
 1  if  link.garbage = 0
 2      then  return  0
 3      else  new ← link.garbage
 4            link.garbage ← link[link.garbage]
 5            return  new
```

If the garbage collector is empty (*garbage* = 0), there is no more free storage place in the array. This storage overflow error is indicated by a 0 return value.

The method **Free** simply links in the element at the position indicated by *index* to the beginning of the garbage collector.

```
Free(index,link)
 1  link[index] ← link.garbage
 2  link.garbage ← index
```

Stacks and queues

There are some data structures from which we require very simple operations. The simplest basic data structures are ***stack***s and ***queue***s. Both of them have only two operations: putting an element in and taking one out, but these operations have to work very quickly, i.e. they need to have constant time complexity. (These two operations have extra names for stacks; ***push*** and ***pop***, respectively.) The difference between stacks and queues is only that while a stack always returns the last element which was put into it (this principle is called ***LIFO*** = Last In First Out), a queue delivers the oldest one every time (***FIFO*** = First In First Out). An example for stacks is the garbage collector of the previous subsection where the garbage stack is implemented as a linked list. Queues are used in processor “pipelines”, for instance, where instructions are waiting to be executed, and the one that has been waiting most is picked up from the queue and executed next.

Both stacks and queues can be implemented using arrays or linked lists. If a stack is represented with an array, an extra index variable is necessary to store where the last element is. If a linked list is used, the top of the stack is at the beginning of the list (no dummy head is needed). For the implementation of queues as linked lists an extra pointer is defined to point at the last element of the list. Hence new

elements can be put at the end of the list and picked up at the beginning, both in constant time. If an array stores a queue, the array has to be handled cyclically (the last element is directly followed by the first). This way it can be avoided to come to the end of the array while having plenty of unused space at the beginning of it.

For both stacks and queues two erroneous operations have to be handled: underflow and overflow. ***Underflow*** happens if an element is intended to be extracted from an empty data structure. ***Overflow*** occurs if a new element is attempted to be placed into the data structure while there is no more free space available.

The following pseudocodes implement the push and pop operations on a stack stored in a simple array named *Stack*. The top of the stack index is stored in the *Stack* object's field *top* (in reality it can be the 0^{th} element of the array, for instance).

Push(*key,Stack*)
```
1  if  Stack.top = Stack.Length
2      then  return  Overflow error
3      else  Stack.top ← Stack.top + 1
4            Stack[Stack.top] ← key
```

If *Stack.top* is the index of the last element in the array, then the stack is full and the overflow error is indicated by the returned value. The next pseudocode indicates the underflow error in the same way. Note that in this case the value indicating the error has to differ from all possible proper output values that can normally occur (i.e. values that are stored in the stack).

Pop(*Stack*)
```
1  if  Stack.top = 0
2      then  return  Underflow error
3      else  Stack.top ← Stack.top − 1
4            return  Stack[Stack.top + 1]
```

The next two pseudocodes define the two basic operations on a queue stored in a simple array. The beginning and the end of the queue in the array are represented by two indices; *Queue.end* is the index of the last element, and *Queue.beginning* is the index of the element preceding the queue's first element in a cyclic order (see *Figure 6*).

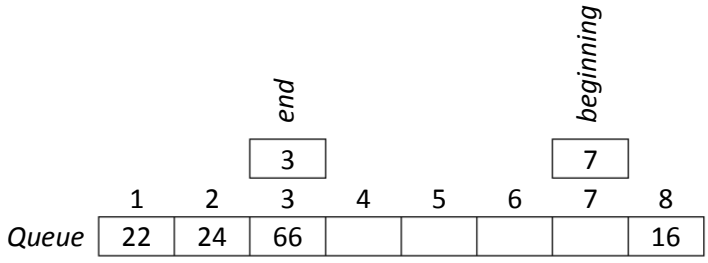


Figure 6. A possible storing of the sequence 16, 22, 24 and 66 in a queue coded in an array.

The coding of the empty queue has to be planned thoughtfully. A good solution can be to let the *beginning* be the index of the array's last element (*Length*) and *end* be 0. Hence a difference can be made between an empty and a full queue.

```
Enqueue(key,Queue)
 1  if  Queue.beginning = Queue.end
 2      then  return  Overflow error
 3      else  if  Queue.end = Queue.Length
 4                then  Queue.end ← 1
 5                else  Queue.end ← Queue.end + 1
 6            Queue[Queue.end] ← key
```

Stepping cyclically in the array is realized in lines 3-5 in both of the pseudocodes **Enqueue** and **Dequeue**.

```
Dequeue(Queue)
 1  if  Queue.end = 0
 2      then  return  Underflow error
 3      else  if  Queue.beginning = Queue.Length
 4                then  Queue.beginning ← 1
 5                else  Queue.beginning ← Queue.beginning + 1
 6            key ← Queue[Queue.beginning]
 7            if  Queue.beginning = Queue.end
 8                then  Queue.beginning ← Queue.Length
 9                      Queue.end ← 0
10            return  key
```

In lines 7-10 the coding of the empty queue is restored if it was the last element that has just been dequeued.

Exercises

21 Write a pseudocode that inserts a given key into a sorted linked list, keeping the order of the list. Write both versions using simple linked lists and dummy head lists stored in a pair of arrays.

22 Write a pseudocode for both the methods push and pop if the stack is stored as a linked list coded in a pair of arrays. Does it make sense to use garbage collection in this case? Does it make sense to use a dummy head linked list? Why?

Hash tables

Many applications require a dynamic set that supports only the dictionary operations insert, search, and delete. For example, a compiler that translates a programming language maintains a symbol table, in which the keys of elements are arbitrary character strings corresponding to identifiers in the language.

Direct-address tables

Direct addressing is a simple technique that works well when the universe U of keys is reasonably small. Suppose that an application needs a dynamic set in which each element has a key drawn from the universe U, where $|U|$ is not too large. We shall assume that no two elements have the same key.

To represent the dynamic set, we use an array, or ***direct-address table***, denoted by T of the same size as U, in which each position, or slot, corresponds to a key in the universe U. For example if $U = \{1,2, \dots ,10\}$, and from the universe U the keys $\{2,5,7,8\}$ are stored in the direct-address table, then each key is stored in the slot with the corresponding index, i.e. the key 2 is stored in $T[2]$, 5 in $T[5]$, etc. The remaining slots are empty (e.g. they store a NIL value). It is similar to a company's parking garage, where every employee has its own parking place (slot). Obviously, all three operations can be performed in $O(1)$, namely in constant time.

Exercises

23 Suppose that a dynamic set S is represented by a direct-address table T of length m. Describe a procedure that finds the maximum element of S. What is the worst-case performance of your procedure?

24 A ***bit vector*** is simply an array of bits (0s and 1s). A bit vector of length m takes much less space than an array of m numbers. Describe how to use a bit vector to represent a dynamic set of distinct elements. Dictionary operations should run in constant time.

Hash tables

The downside of direct addressing is obvious: if the universe U is large, storing a table T of size $|U|$ may be impractical, or even impossible, given the memory available on a typical computer. Furthermore, the set K of keys ***actually stored*** may be so small relative to U that most of the space allocated for T would be wasted. In our parking garage example, if the employees of the firm work in shifts, then there might be many employees in all, still only a part of them uses the parking garage concurrently.

With direct addressing, an element with key k is stored in slot k. With hashing, this element is stored in slot $h(k)$; that is, we use a so-called ***hash function*** h to compute the slot of the key k. Here, h maps the universe U of keys into the slots of a ***hash table*** T:

$$h: U \to \{1,2, \dots, |T|\},$$

where the size of the hash table is typically much less than $|U|$. We say that an element with key k ***hashes*** to slot $h(k)$; we also say that $h(k)$ is the ***hash value*** of key k.

There is one hitch: two keys may hash to the same slot. We call this situation a ***collision***. Fortunately, we have effective techniques for resolving the conflict created by collisions.

Of course, the ideal solution would be to avoid collisions altogether. We might try to achieve this goal by choosing a suitable hash function h. Because $|U| > |T|$, however, there must be at least two keys that have the same hash value; avoiding collisions altogether is therefore impossible.

Collision resolution by chaining

In ***chaining***, we place all the elements that hash to the same slot into the same linked list, as *Figure 7* shows. Slot *j* contains a pointer to the head of the list of all stored elements that hash to j; if there are no such elements, slot j contains NIL.

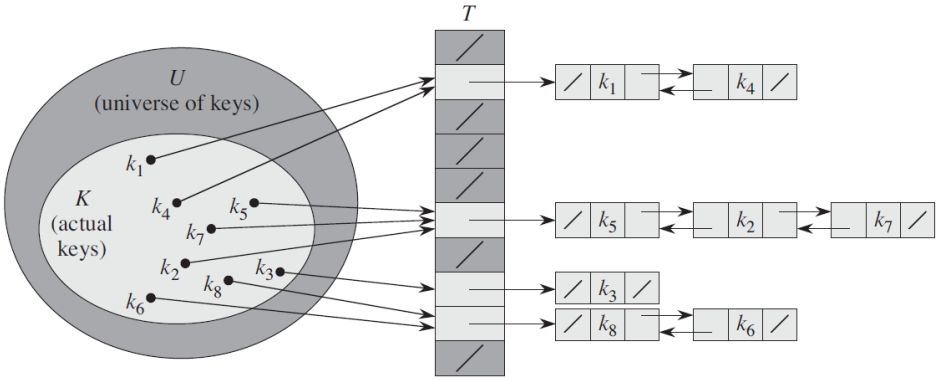


Figure 7. Collision resolution by chaining. Each hash-table slot $T[j]$ contains a linked list of all the keys whose hash value is j. For example, $h(k_1) = h(k_4)$ and $h(k_5) = h(k_2) = h(k_7)$. The linked list can be either singly or doubly linked; we show it as doubly linked because deletion is faster that way.

How fast are the operations if chaining is used? Insertion takes obviously constant time by inserting the key k as the new leading element of the linked list of slot $h(k)$. Note, that this is only possible if it is sure that k is not already present in the hash table. Otherwise, we have to search for k first. The same is true for deletion. If we know the position (i.e. the address) of key k to be deleted, then it simply has to be linked out of its list. Otherwise, we have to find it first. The question remains how long it takes to search for an element in a hash-table. Assuming that calculating the hash function h takes constant time the time complexity of finding an element in an unsorted list depends mainly on the length of the list.

The worst-case behavior of hashing with chaining is terrible: all $|U|$ keys hash to the same slot. The worst-case time for searching is thus not better than if we used one linked list for all the elements.

The average-case performance of hashing depends on how well the hash function h distributes the set of keys to be stored among the slots, on the average. Therefor we shall assume that any given element is equally likely to hash into any of the slots, independently of where any other element has hashed to. We call this the assumption of ***simple uniform hashing***. Let us define the ***load factor*** α for T as $|U|/|T|$. Due to the simple uniform hashing assumption the expected value of a single chain's length in our hash table equals α. If we add the constant time of calculating the hash function h, we have the average case of operations of a hash table as $O(1 + \alpha)$. Thus, if we assume that $|U| = O(|T|)$, i.e. U is only linearly

bigger than T, then the operations in a hash table can be reduced to an average time complexity of $O(1)$.

But how does a good hash function look like? The simplest way to create a hash function fulfilling the simple uniform hashing is using the so-called ***division method***. We assume that the keys are coded with the natural numbers $\mathbb{N} = \{0,1,2,\dots,|U|-1\}$, and define the hash function as follows: for any key $k \in U$ let $h(k) = k \bmod |T|$. Another solution is if the keys k are random real numbers independently and uniformly distributed in the range $0 \leq k < 1$, then the hash function can be defined as $h(k) = \lfloor k \cdot |T| \rfloor$. This satisfies the condition of simple uniform hashing, as well.

Exercises

25 Demonstrate what happens when we insert the keys 5, 28, 19, 15, 20, 33, 12, 17, 10 into a hash table with collisions resolved by chaining. Let the table have 9 slots, and let the hash function be $h(k) = k \bmod 9$.

26 Professor Marley hypothesizes that he can obtain substantial performance gains by modifying the chaining scheme to keep each list in sorted order. How does the professor's modification affect the running time for successful searches, unsuccessful searches, insertions, and deletions?

27 Suppose that we are storing a set of n keys into a hash table of size m. Show that if the keys are drawn from a universe U with $|U| > nm$, then U has a subset of size n consisting of keys that all hash to the same slot, so that the worst-case searching time for hashing with chaining is $\theta(n)$.

Binary search trees

All structures above in this section are linear, preventing some basic operations from providing better time complexity results in the worst case than *n* (the number of stored elements). Binary search trees have another structure.

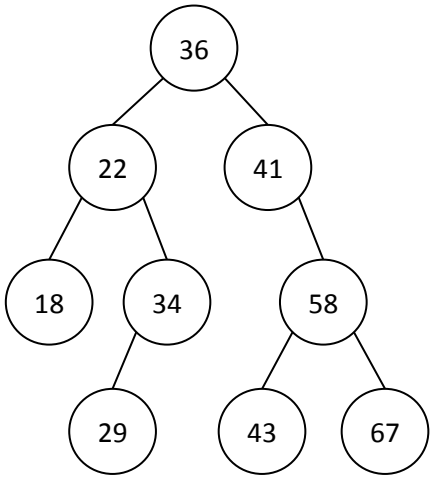


Figure 8. A binary search tree.

Binary search trees are rooted trees (***tree***s are cycleless connected graphs), i.e. one of the vertices is named as the root of the tree. A rooted tree is called binary if one of the following holds. It is either empty or consists of three disjoint sets of vertices: a root and the left and right subtrees, respectively, which are themselves binary trees. A binary tree is called a search tree if a key is stored in each of its vertices, and the ***binary search tree property*** holds, i.e. for every vertex all keys in its left subtree are less and all in its right subtree are greater than the key stored in it. Equality is allowed if equal keys can occur.

The vertices of binary trees can be classified into ***level***s depending on their distance from the root (distance is the number of edges on the path) hence the root alone constitutes the 0^{th} level. For instance in *Figure 8* the vertex containing the number 29 is at level 3. The ***depth*** of a binary tree (sometimes also called ***height***) is the number of the deepest level. Furthermore a vertex directly preceding another on the path starting at the root is called its ***parent***, and the vertex following its parent directly is called ***child***. The two children of the same vertex are called ***twins*** or ***siblings***. A vertex without children is a ***leaf***.

Binary search trees can be represented with dynamic data structures similar to doubly linked lists but in this case every tree element (vertex) is accompanied by three links, two for the left and the right child, respectively, and one for the parent.

Binary search tree operations

For a binary search tree, the following operations are defined: walk (i.e. listing), search, minimum (and maximum), successor (and predecessor), insert, delete.

In a linear data structure it is no question which the natural order of walking the elements is. However, a binary tree has no such obvious order. A kind of tree walk can be defined considering the binary tree as a triple consisting of the root and its two subtrees. The ***inorder tree walk*** of a binary tree is defined recursively as follows. First we walk the vertices of the left subtree using an inorder tree walk, then visit the root, and finally walk the right subtree using an inorder walk again. There are so-called preorder and postorder tree walks, which differ from the inorder only in the order of how these three, well-separable parts are executed. In the preorder case the root is visited first before the two subtrees are walked recursively, and in the postorder algorithm the root is visited last. It is easy to check that in a binary search tree the inorder tree walk visits the vertices in an increasing order regarding the keys. This comes from the simple observation that all keys visited prior to the root of any subtree are less than the key of that root, whilst any key visited subsequently is greater.

The pseudocode of the recursive method for the inorder walk of a binary tree is the following. It is assumed that the binary tree is stored in a dynamically allocated structure of objects where *Tree* is a pointer to the root element of the tree.

```
InorderWalk(Tree)
1  if  Tree ≠ NIL
2     then  InorderWalk(Tree.Left)
3           visit Tree, e.g. check it or list it
4           InorderWalk(Tree.Right)
```

The ***tree search*** highly exploits the special order of keys in binary search trees. First it just checks the root of the tree for equality with the searched key. If they are not equal, the search is continued at the root of either the left or the right subtree, depending on whether the searched key is less than or greater than the key being checked. The algorithm stops if either it steps to an empty subtree or the searched key is found. The number of steps to be made in the tree and hence the time complexity of the algorithm equals the depth of the tree in worst case. The following pseudocode searches key *toFind* in the binary search tree rooted at *Tree*.

```
TreeSearch(toFind,Tree)
 1 while Tree ≠ NIL and Tree.key ≠ toFind
 2       do if toFind < Tree.key
 3             then Tree ← Tree.Left
 4             else Tree ← Tree.Right
 5 return Tree
```

Note that the pseudocode above returns with NIL if the search was unsuccessful.

The vertex containing the minimal key of a tree is the leftmost leaf of it (to check this simply let us try to find it using the tree search algorithm described above). The algorithm for finding this vertex simply keeps on stepping left starting at the root until it arrives at an absent left child. The last visited vertex contains the ***minimum*** in the tree. The maximum is symmetrically on the other side of the binary search tree, it is the rightmost leaf of it. Both algorithms walk down in the tree starting at the root so their time complexity is not more than the depth of the tree.

```
TreeMinimum(Tree)
 1 while Tree.Left ≠ NIL
 2       do Tree ← Tree.Left
 3 return Tree
```

To find the ***successor*** of a key stored in a binary search tree is a bit harder problem. While, for example, the successor 41 of key 36 in *Figure 8* is its right child, the successor 29 of 22 is only in its right subtree but not a child of it, and the successor 36 of 34 is not even in one of its subtrees. To find the right answer we have to distinguish between two basic cases: when the investigated vertex has a nonempty right subtree, and when it has none. To define the problem correctly: we are searching the minimal key among those greater than the investigated one; this is the successor. If a vertex has a nonempty right subtree, then the minimal among the greater keys is in its right subtree, furthermore it is the minimum of that subtree (line 2 in the pseudocode below). However, if it has none, then the searched vertex is the first one on the path leading upwards in the tree starting at the investigated vertex which is greater than the investigated one. At the same time this is the first vertex we arrive at through a left child on this path. In lines 4-6 of the following pseudocode the algorithm keeps stepping upwards until it either finds a parent-left child relation, or runs out of the tree at the top (this could only happen if we tried to find the successor of the greatest key in the tree, which

obviously does not exist; in this case a NIL value is returned). Parameter *Element* in the following pseudocode contains the address of the key's vertex whose successor is to be found.

```
TreeSuccessor(Element)
 1 if  Element.Right ≠ NIL
 2     then  return  TreeMinimum(Element.Right)
 3     else  Above ← Element.Parent
 4           while Above ≠ NIL  and  Element = Above.Right
 5                 do Element ← Above
 6                    Above ← Above.Parent
 7 return  Above
```

Since in both cases we walk in just one direction in the tree (downwards or upwards), the time complexity equals the depth of the tree in worst case. Finding the ***predecessor*** of a key results simply in the mirror images of the search paths described above. Hence changing the words "minimum" and "right" to "maximum" and "left" in the pseudocode results in the algorithm finding the predecessor.

The principle of ***inserting*** a new vertex into a binary search tree is the same as when we try to search for the new key. As soon as we find the place where the new key should be, it is linked into the tree as a new leaf. For this reason we say that a binary search tree is always grown at its leaf level. Since the time complexity of the tree search equals the depth of the tree in worst case, so does the insertion of a new element.

The procedure of ***deleting*** an element from a known position in a binary search tree depends on the number of its children. If it has no children at all, i.e. it is a leaf, the vertex is simply deleted from the data structure (lines 1-8 of the pseudocode **TreeDelete**). If it has only one child (as e.g. key 41 in *Figure 8*), the tree's structure resembles a linked list locally: the vertex to be deleted has just one vertex preceding it (its parent) and another following it (the only child of it). Thus we can link it out of the data structure (lines 9-18 if the left child is missing and lines 19-28 if the right child is missing in the pseudocode **TreeDelete**). The most sophisticated method is needed in the case when the vertex to be deleted has two children. The problem is solved using a subtle idea: instead of rearranging the tree's structure near the place where the vertex has been deleted, the structure is preserved by substituting the deleted vertex by another one from the

tree, which can easily be linked out from its previous position and contains an appropriate key for its new position. A good decision is the successor or the predecessor of the key to be deleted; on one hand it can easily be linked out because it has one child at most, on the other hand it is the nearest key to the deleted one in the considered order of keys. (You can check each step in the pseudocode below. The tree successor is taken as a substitute in line 30. In lines 31-35 the substitute is linked out from its present position, and in lines 36-45 it is linked into its new position, i.e. where the deleted element has been till now.)

The pseudocode given below is redundant, its verbosity serves better understanding.

```
TreeDelete(Element,Tree)
 1  if  Element.Left = NIL  and  Element.Right = NIL
 2     then  if  Element.Parent = NIL
 3              then  Tree ← NIL
 4              else  if  Element = (Element.Parent).Left
 5                       then  (Element.Parent).Left ← NIL
 6                       else  (Element.Parent).Right ← NIL
 7           Free(Element)
 8           return  Tree
 9  if  Element.Left = NIL  and  Element.Right ≠ NIL
10     then  if  Element.Parent = NIL
11              then  Tree ← Element.Right
12                    (Element.Right).Parent ← NIL
13              else  (Element.Right).Parent ← Element.Parent
14                    if  Element = (Element.Parent).Left
15                       then  (Element.Parent).Left ← Element.Right
16                       else  (Element.Parent).Right ← Element.Right
17           Free(Element)
18           return  Tree
19  if  Element.Left ≠ NIL  and  Element.Right = NIL
20     then  if  Element.Parent = NIL
21              then  Tree ← Element.Left
22                    (Element.Left).Parent ← NIL
23              else  (Element.Left).Parent ← Element.Parent
24                    if  Element = (Element.Parent).Left
25                       then  (Element.Parent).Left ← Element.Left
26                       else  (Element.Parent).Right ← Element.Left
```

```
27            Free(Element)
28            return Tree
29  if  Element.Left ≠ NIL  and  Element.Right ≠ NIL
30    then  Substitute ← TreeSuccessor(Element)
31            if  Substitute.Right ≠ NIL
32              then  (Substitute.Right).Parent ← Substitute.Parent
33            if  Substitute = (Substitute.Parent).Left
34              then  (Substitute.Parent).Left ← Substitute.Right
35              else  (Substitute.Parent).Right ← Substitute.Right
36            Substitute.Parent ← Element.Parent
37            if  Element.Parent = NIL
38              then  Tree ← Substitute
39              else  if  Element = (Element.Parent).Left
40                      then  (Element.Parent).Left ← Substitute
41                      else  (Element.Parent).Right ← Substitute
42            Substitute.Left ← Element.Left
43            (Substitute.Left).Parent ← Substitute
44            Substitute.Right ← Element.Right
45            (Substitute. Right).Parent ← Substitute
46            Free(Element)
47            return Tree
```

The time complexity issues of the deletion are the following. If a leaf is deleted, it can be done in constant time. A vertex with only one child can be linked out in constant time, too. If the element has two children, the successor or the predecessor of it has to be found and linked into its place. Finding the successor or predecessor does not cost any more steps than the depth of the tree as seen earlier, hence after completing some pointer assignment instructions in constant time we have a time complexity proportional to the depth of the tree at most.

Summarizing the results for the time complexity of the operations that are usually executed on binary search trees, we find that all have the time complexity $T(n) = O(d)$ where *d* denotes the depth of the tree. But how does *d* depend on *n*? It can be proven that the depth of any randomly built binary search tree on *n* distinct keys is $d = O(\log n)$ (4). (Note that the base of the logarithm in the formula is inessential since changing the base is equivalent to a multiplication by a constant that does not influence the asymptotic magnitude, see *Exercise* 18 on page 15). This means that all the basic operations on a binary search tree run in $T(n) = O(\log n)$ time.

Binary search

If the data structure is not intended to be extended or to be deleted from frequently, then the keys can be stored simply in an ordered array. Such operations as minimum, maximum, successor and predecessor are obvious on ordered arrays, moreover, they run in constant time. Search can be made in a similar way as in binary search trees, obtaining the same $T(n) = O(\log n)$ time in the worst case. Let us imagine our array is a coding of a binary tree where the root's key is stored in the central element of the array, and the left and right subtrees' keys in the first and second half of it, respectively, in a similar way. The so-called binary search can be implemented using the following pseudocode. It returns the index of the searched key in the array, and zero if it was not found.

```
BinarySearch(A,key)
 1  first ← 1
 2  last ← A.Length
 3  while first ≤ last
 4        do  central ← (first + last) / 2
 5            if  key = A[central]
 6                then  return central
 7                else  if  key < A[central]
 8                          then  last ← central – 1
 9                          else  first ← central + 1
10  return  0
```

The binary search algorithm can also be implemented easily with a recursive code. In practice, however, if the same problem can be solved recursively and in a straightforward way with similar difficulty at the same time, then it is always decided to use the straightforward one contrary to using recurrence because of the time consuming administrative steps arising when running recursive codes.

Exercises

28 Write the pseudocode of **TreeInsert**(*Element,Tree*) that inserts *Element* into the binary search tree rooted at *Tree*.

Sorting

The problem of ***sorting*** is the following. A set of input data has to be sorted using an order defined on the base set of the input. A simple example is to arrange a list of names in an alphabetical order. In this example the order of strings (texts) is the so-called ***lexicographical order***. This means that if a sequence consists of symbols having a predefined order themselves (the letters of the alphabet here certainly have), then an order can be defined on such sequences in the following way. We first compare the first symbols (letters); if they are equal, then the second ones, etc. Hence the first difference determines the relation of the two sequences considered.

The problem of sorting is easy to understand but the solution is not obvious. Furthermore, plenty of algorithms exist to solve it, hence it is an appropriate subject for investigating algorithms and algorithmic properties. In the following we are going to study some of them.

Insertion sort

One of the simplest sorting algorithms is insertion sort. Its principle can be explained through the following example. Let us imagine we are carrying a stack of paper in which the sheets have a fixed order. Suddenly we drop the stack and we have to pick the sheets up to reconstruct the original order. The method that most people use for this is insertion sort. We do not have any task with the first sheet, we simply pick it up. When we have at least one sheet in our hands, the algorithm turns the sheets in our hands one by one starting from the end or from the beginning searching for the correct place of the new sheet. If the position is found, the new sheet is inserted there.

The algorithm is very flexible; it can be implemented by using both dynamic storage (linked lists) without direct access possibility, and arrays. It can be used on-line (an algorithm is called ***on-line*** if it delivers the solution to subproblems arising at every stage of execution underway, while it is ***off-line*** if it needs the whole input data set prior to execution), since after each step the keys that are already inserted form an ordered sequence. A possible implementation of the insertion sort on arrays is given in the following pseudocode. Array *A* is divided into the sorted front part and the unsorted rear part by variable *i*; *i* stores the index of the unsorted part's first key. Variable *ins* stores the next key to be moved from the unsorted part to the sorted part. Variable *j* steps backwards in the sorted

part until either the first element is passed (*ins* is the least key found until now) or the place of insertion is found earlier. In the last line *ins* is inserted.

InsertionSort(*A*)

```
1  for  i ← 2 to A.Length
2       do  ins ← A[i]
3           j ← i – 1
4           while  j > 0  and  ins < A[j]
5                  do  A[j + 1] ← A[j]
6                      j ← j – 1
7           A[j + 1] ← ins
```

The time complexity of the algorithm depends on the initial order of keys in the input. If after each iteration of the **while** loop *ins* can be inserted at the end of the sorted part, i.e. the **while** loop's body is not executed at all, the time complexity of the **while** loop is constant. Thus, the whole time complexity equals $T(n) = 1 + 1 + \cdots + 1 = n - 1 = \theta(n)$. This is the best case, and it occurs if the keys are already ordered in the input. If, on the other hand, the **while** loop has to search through the whole sorted part in each iteration of the **for** loop yielding a time complexity *i* for the i^{th} iteration, the time complexity of the whole algorithm will be $T(n) = 2 + 3 + \cdots + n = n(n+1)/2 - 1 = \theta(n^2)$. The difference between this worst and the best case is significant. In practice the most important result is the average case time complexity telling us what can be expected in performance in most of the cases.

For the insertion sort the best case occurs if only one of the sorted part's keys has to be examined, while the worst case means searching through the whole sorted part for the insertion point. On average we can insert *ins* somewhere in the middle of the sorted part consisting of *i* elements in the i^{th} iteration resulting in $T(n) = 2/2 + 3/2 + \cdots + n/2 = (n(n+1)/2 - 1)/2 = \theta(n^2)$ time complexity, which is not better asymptotically than the worst case.

Exercises

29 Demonstrate how insertion sort works on the following input: (5, 1, 7, 3, 2, 4, 6).

30 What kind of input yields a best case and which a worst case behavior for the insertion sort? Give examples.

31 A sorting algorithm is called ***stable*** if equal keys keep their order. Is insertion sort stable?

Merge sort

Merge sort (also known as comb sort) is a typical example for the so-called ***divide-and-conquer*** strategy. The idea is to halve the set of elements first and let the arising two halves be sorted recursively, subsequently the two presorted parts are merged. Obviously the algorithm is recursive and so is the function of its time complexity. However, it is worth investigating. The recursion tree of the algorithm looks as follows.

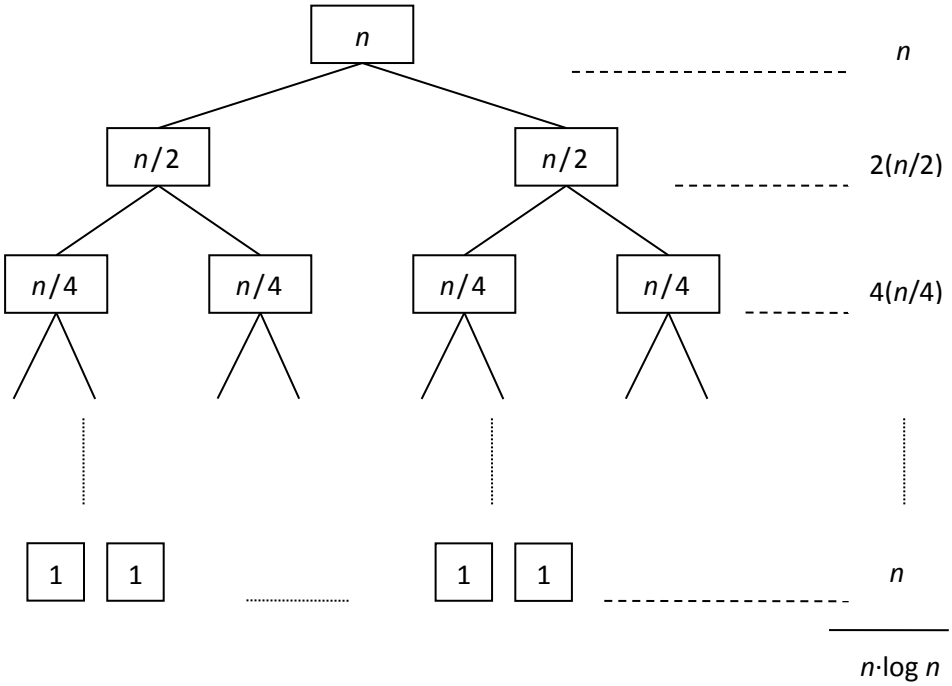


Figure 9. Recursion tree of the merge sort

Merging *n* elements costs *n* steps, and this is all what has to be done at that level. Because every level of the recursion tree costs *n* altogether, the question remains: how many levels does the tree have? The answer to this question is: as many as many times *n* has to be halved to decrease to 1. Thus, solving $n/2^{\ell} = 1$ for ℓ we have $T(n) = \theta(n \cdot \log n)$. It can be proven that for comparison sorts (i.e. sorting algorithms working using pair comparisons only) this time complexity is a lower bound, in other words, merge sort's worst case speed is optimal.

Unfortunately, merge sort is not in-place (a sorting algorithm is called ***in-place sorting*** if it does not need any auxiliary storage of a size comparable to the size of the input; e.g. if another array with the same number of elements as the input is needed, the algorithm is not in-place) but it has more than one implementation to keep its storage handling simple and relatively efficient.

Exercises

32 Demonstrate how merge sort works on the following input: (4, 2, 8, 1, 6, 7, 3, 5).

33 Write a pseudocode implementing merge sort.

Heapsort

Although heapsort is a well-known and theoretically very efficient sorting algorithm indeed, this is not the only reason why it is worth studying. It also uses a special data structure called heap, which is appropriate for several other problems in algorithm theory.

Heaps

An array *A* can be considered as a ***heap*** if for all of its elements inequalities $A[i] \geq A[2i]$ and $A[i] \geq A[2i+1]$ hold (this pair of inequalities constitutes the so-called ***heap property***). To be able to imagine what this means, we will fill in the elements of the array into a binary tree structure row by row; the first element will be the root, the following two elements one after the other will form the first level, the next four elements the second level, etc. (see an example in *Figure 10*).

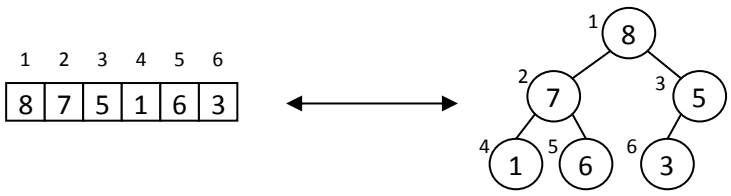


Figure 10. Equivalent coding of a heap in an array and in a binary tree structure.

The heap property in the binary tree means that every element's key is not less than any of its children's.

A heap is a special data structure supporting the implementation of ***priority queue***s, which support only the operations insert, maximum (or minimum) and extract maximum (or extract minimum). (The variants in parentheses presume a so-called minimum heap, which is defined with a similar heap property with reverse inequalities.)

Before an array can be used as a heap, the order of its elements might be modified to fulfill the heap property, i.e. we have to ***build a heap***. First let us consider the obvious case when only the root's key may infringe the heap property. In this case we ***sink*** it to a position where it fits in the following way (the pseudocode below sinks the key at index *k* in the array to a proper position). First we compare it with its children (lines 1-5 in the pseudocode). If it is the greatest, we are done. Otherwise we exchange it for the greatest of them and carry on this procedure (lines 7-8). At the end the key that was originally situated in the root arrives at a place where it fulfills the heap property.

Sink(*k,A*)
```
1  if  2*k ≤ A.HeapSize  and  A[2*k] > A[k]
2      then  greatest ← 2*k
3      else  greatest ← k
4  if  2*k + 1 ≤ A.HeapSize  and  A[2*k + 1] > A[greatest]
5      then  greatest ← 2*k + 1
6  if  greatest ≠ k
7      then  Exchange(A[greatest],A[k])
8            Sink(greatest,A)
```

To mend all elements of the array this way, we begin from the last array element having any children in the binary tree representation (its index is obviously $n/2$ where *n* denotes the number of keys) and do the sinking procedure for all elements backwards from it in the array (direction root). Because we move upwards in the tree, by the time a given element is visited all the elements in its subtrees have already been mended. Hence the sinking procedure can be applied to mending all elements using this order.

BuildHeap(*A*)
```
1  A.HeapSize ← A.Length
2  for  k ← A.Length / 2  downto  1
3       do  Sink(k,A)
```

What is the time complexity of building a heap of an unordered array consisting of *n* elements? An upper bound comes right from the observation that sinking any element of the tree cannot cost more than the whole tree's depth, which is $O(\log n)$ for heaps. Since $n/2$ elements have to be sunk, an upper bound for the time complexity of building a heap is $T(n) = n/2 \cdot O(\log n) = O(n \log n)$. However, it can be proven that although this bound is correct, the tight bound equals $T(n) = \theta(n)$, thus a heap can be built in linear time.

To find the ***maximum in a heap*** we need only constant time, i.e. $T(n) = O(1)$, because due to the heap property the greatest key is always stored in the root element.

To ***extract the maximum from a heap*** we first read out the key of the root and then delete the root element by replacing it with the last element of the array. To mend the heap we call the sinking procedure for the root. The time complexity of extracting the maximum is hence $T(n) = O(\log n)$, the depth of the tree.

Exercises

34 Write the pseudocode **ExtractMaximum**(*A*) if *A* is assumed to be an array coded heap.

35 Demonstrate how a heap is built from the following input: (3, 1, 7, 5, 9, 6, 4, 8, 2).

Sorting in a heap

Heaps also provide a convenient tool for sorting elements. The idea is very simple. After building a heap from the input array the first and the last elements are exchanged. Since the leading element of an array coding a heap is the maximum, the greatest key is hence put into its final position and this last element of the array is excluded from the heap. The remaining part is mended by sinking the new root because this is the only element infringing the heap property. After mending the heap the first element is exchanged for the last one in the new, shorter heap effecting the second greatest element of the original array to come to its final position too. If we iterate this procedure till the heap's size decreases to 1, the array becomes sorted.

```
HeapSort(A)
 1  BuildHeap(A)
 2  for  k ← A.Length  downto  2
 3       do  Exchange(A[1],A[A.HeapSize])
 4           A.HeapSize ← A.HeapSize – 1
 5           Sink(1,A)
```

The time consumption of the heapsort aggregates from the time complexities of building a heap and iteratively exchanging and sinking elements resulting in $T(n) = O(n) + (n-1) \cdot O(\log n) = O(n) + O(n \log n) = O(n \log n)$. This means that the heapsort algorithm is optimal. Moreover it is an in-place sorting.

Exercises

36 Demonstrate how a heapsort works on the following input: (3, 1, 7, 5, 9, 6, 4, 8, 2).

37 Is heapsort a stable sorting algorithm (see Exercise 31 on page 35)?

Quicksort

Although heapsort is in-place and has an optimal asymptotic time complexity among sorting algorithms, with small size inputs it is not very efficient because of the relatively expensive heap building at the beginning of it. Quicksort is much worse in the worst case, still it performs better on real-life size problems in the average case.

Partition algorithm

Quicksort (5) uses a partition algorithm that, roughly speaking, puts the small keys in an array at the beginning and the large keys at the end of the array in-place in linear time. Because this algorithm is used by other algorithms, too, it is treated separately here.

First, let us choose any element of the array, which will be the so-called ***pivot key***. Keys not greater than the pivot key will be considered as small and those not smaller will be the large keys. To arrange the elements following the rule above, loops are started from both the beginning and from the end of the array. Every time a large key is found in the first part of the array and a small key in the second part they are exchanged. The procedure ends if the indices of the two loops meet. The meeting point's index indicating the border between the small and the large elements is then returned by the method.

```
Partition(A,first,last)
 1  left ← first – 1
 2  right ← last + 1
 3  pivotKey ← A[RandomInteger(first,last – 1)]
 4  repeat
 5        repeat  left ← left + 1
 6        until   A[left] ≥ pivotKey
 7        repeat  right ← right – 1
 8        until   A[right] ≤ pivotKey
 9        if  left < right
10           then  Exchange(A[left],A[right])
11           else  return  right
12  until  false
```

The time complexity of this algorithm is obviously $T(n) = \theta(n)$ since every *n* element of the array is visited by exactly one of the two loops.

Note that there is no guarantee for a good balance between the size of the two parts. It can occur that the array is halved but either of the parts can even consist of only one element. How the algorithm proceeds depends on the choice of the pivot key; if it is too small, the second part will be very long, and if it is two large, then the first part gets long. Unfortunately, to find a good pivot element is not easy. To avoid consistent unbalanced output on certain series of inputs, the pivot element is chosen randomly in general.

Exercises

38 Demonstrate how the partition algorithm works on the following input: (4, 1, 9, 3, 7, 8, 2, 5, 6). Try to use different elements as the pivot key.

39 What does the partition algorithm do if all keys in the input array are the same?

Sorting with quicksort

Quicksort is the most widely used sorting algorithm. It executes the divide-and-conquer principle in a very simple but efficient way. It calls the partition algorithm for the input array and then calls itself to the two parts provided by the partition. The size of the occurring subarrays constitutes the base criterion; if an array consists of only one single element, it is already sorted so no further sorting of it is needed.

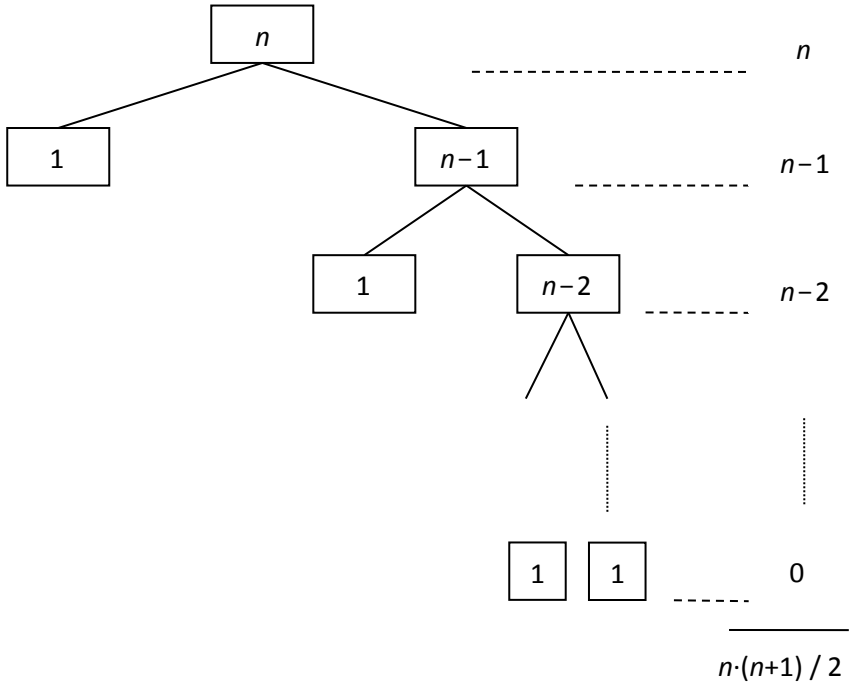


Figure 11. Recursion tree of the quicksort in worst case.

The time complexity of quicksort highly depends on the balance of the partitions. If the partition algorithm can halve the arrays occurring during the execution of the sorting algorithm, the recursion tree will look like that of the merge sort seen in *Figure 9*, and since the partition algorithm's time complexity is linear like that of merging in merge sort, the time complexity of the sorting itself will be $T(n) = \theta(n \log n)$, as well. This is the best case.

However, if the partitions are unbalanced, e.g. at each step one of the parts contains only one element, the recursion tree will look like *Figure 11* resulting in a quadratic time complexity $T(n) = n \cdot (n+1)/2 = \theta(n^2)$. This is the worst case.

It is not easy to prove that quicksort is as good in the average case as in the best case. However, to have an idea about how quicksort works in the special case when the balance of the occurring parts of partition is not worse than a given fixed ratio, we investigate this case in details. Therefore, let us assume that in none of the executions of the partition algorithm during quicksorting will be the ratio of

the size of the occurring two parts worse than $(1-\lambda):\lambda$ for a given $\lambda \in]0{,}1[$ (we shall refer to this later as the $\boldsymbol{\lambda}$ **assumption**). This means that if, e.g., $\lambda = 0.25$ then none of the partitions will contain less than a quarter of the partitioned subarray of elements, or equivalently none of the partitions will be longer than three quarters of the subarray. Since λ can be arbitrarily close to either 0 or 1, let us say 0.99999, this assumption does not seem to be unfeasible. If we set $\lambda = 1 - 1/N$ where N is an upper bound on the number of elements we can store in an array at a time on our computer, then even the case of separating only one element from the others during the **Partition** function will fulfill the λ assumption.

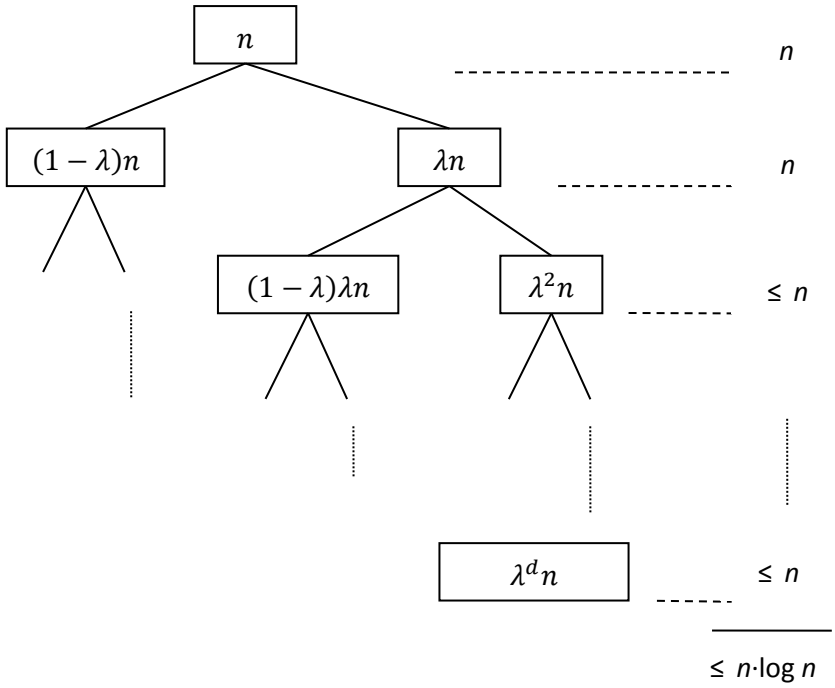


Figure 12. Recursion tree of the quicksort in a special case.

For the sake of simplicity in the following let us assume without the loss of generality that $\lambda \geq 0.5$, otherwise $(1-\lambda)$ and λ may exchange places. As a result the bigger partition (the λ proportion part) will always be the upper part and hence the right subtrees of the recursion tree will be deeper than the left subtrees for all vertices (see *Figure 12*). Thus, the depth *d* of the recursion tree depends on for which d exponent $\lambda^d n = 1$ is obtained. The answer is $d = \log_{1/\lambda} n$ yielding $T(n) \leq n \cdot \log_{1/\lambda} n = O(n \log n)$ time complexity for that special case.

The pseudocode of quicksort is nevertheless very simple.

```
QuickSort(A,first,last)
  1  if  first < last
  2      then  border ← Partition(A,first,last)
  3            QuickSort(A,first,border)
  4            QuickSort(A,border + 1,last)
```

It can be proven that a good pivot key for the partition algorithm can always be found in linear time which assures that no partition will be longer than a fixed constant ratio. Hence quicksort can be completed so that it will work in $O(n \log n)$ time even in worst case.

Exercises

40 Demonstrate how quicksort works on the following input: (4, 1, 9, 3, 7, 8, 2, 5, 6). Use the subarrays' leading elements as pivot keys.

41 What does quicksort do if the input is already sorted? Is it a best case or a worst case?

42 Is quicksort a stable sorting algorithm (see Exercise 31 on page 35)?

Sorting in Linear Time

We have now introduced several algorithms that can sort n numbers in $O(n \log n)$ time. Merge sort and heapsort achieve this upper bound in the worst case; quicksort achieves it on average. Moreover, for each of these algorithms, we can produce a sequence of n input numbers that causes the algorithm to run in $n \log n$ time. These algorithms share an interesting property: the sorted order they determine is based only on comparisons between the input elements. We called such sorting algorithms comparison sorts. All the sorting algorithms introduced thus far are comparison sorts.

Lower bounds for sorting

In a comparison sort, we use only comparisons between elements to gain order information about an input sequence. Hence, to answer the question how many algorithmic steps there are necessary to sort the sequence, we simply have to find out how many comparisons we need to analyze the input. This analysis can be demonstrated using the so-called decision trees.

A decision tree is a binary tree that represents the comparisons between elements that are performed by a particular algorithm to determine the order of the input elements. *Figure 13* shows a possible decision tree of an input sequence of three elements *a*, *b*, and *c*.

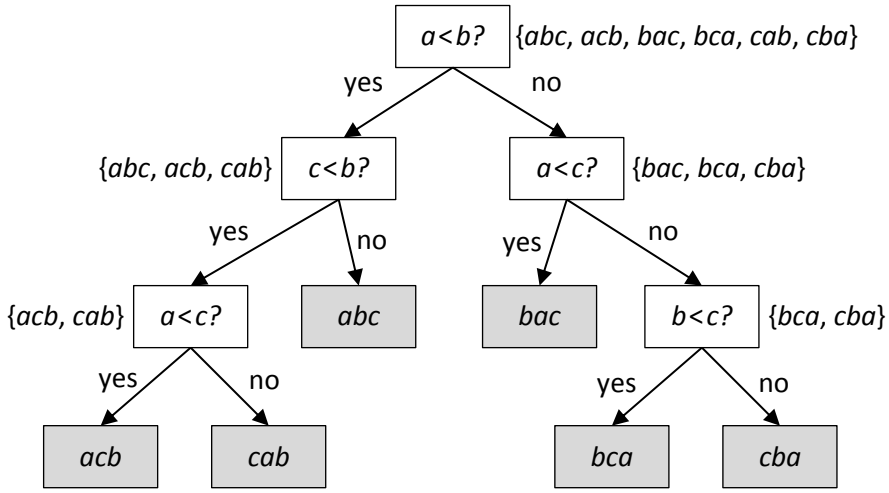


Figure 13. Decision tree of analyzing the order of three elements: a, b, and c. The white rectangles represent the questions of decisions, the lists in curly brackets the possible orders remaining, and the shaded rectangles the last, only possible order on that route.

The execution of a sorting algorithm corresponds to tracing a simple path from the root of the decision tree down to a leaf. Each internal node indicates a comparison. When we come to a leaf, the sorting algorithm has established the ordering. Because any correct sorting algorithm must be able to produce each permutation of its input, each of the $n!$ permutations on n elements must appear as one of the leaves of the decision tree for a comparison sort to be correct. The length of the longest simple path from the root of a decision tree to any of its leaves represents the worst-case number of comparisons that the corresponding sorting algorithm performs. Consequently, the worst-case number of comparisons for a given comparison sort algorithm equals the depth of its decision tree. A lower bound on the depths of all decision trees in which each permutation appears as a leaf is therefore a lower bound on the running time of any comparison sort algorithm.

It is obvious that a binary tree of depth d can have at most 2^d leaves. Because we now have exactly $n!$ leaves, $2^d \geq n!$ follows, and thus

$$d \geq \log_2(n!) =$$

$$= \sum_{k=1}^{n} \log_2 k \geq \int_1^n \log_2 x \, dx =$$

$$= (\ln 2)^{-1}[x \ln x - x]_1^n =$$

$$= \Theta(n \log n)$$

That means that any comparison sort algorithm requires at least $n \log n$ comparisons in the worst case. As a consequence, heapsort and merge sort are asymptotically optimal comparison sorts.

Exercises

43 Draw the decision tree of a sorting algorithm sorting four elements: *a*, *b*, *c*, and *d*.

44 What is the smallest possible depth of a leaf in a decision tree for a comparison sort?

Counting sort

However, if we have and exploit more information on the input than just their pairwise relations, sorting algorithms with $\theta(n)$, i.e. linear time complexity can be constructed. One of them is the counting sort, which assumes that each of the n input elements is an integer in the range 1 to k, for some integer k. When $k = O(n)$, the sort runs in $\theta(n)$ time.

Counting sort determines, for each input element x, the number of elements less than x. It uses this information to place element x directly into its position in the output array. For example, if 17 elements are less than x, then x belongs in output position 18. We must modify this scheme slightly to handle the situation in which several elements have the same value, since we do not want to put them all in the same position.

In the code for counting sort, we assume that the input is an array *A*. We require two other arrays: an array *B* as a temporary working storage, and the array *C* for the sorted output. Note, that *B* has only k elements. The following pseudocode implements the counting sort algorithm.

```
CountingSort(A,C,k)
 1  for i ← 1 to k
 2      do B[i] ← 0
 3  for i ← 1 to A.Length
 4      do B[A[i]] ← B[A[i]] + 1
 5  for i ← 2 to k
 6      do B[i] ← B[i] + B[i – 1]
 7  for i ← A.Length downto 1
 8      do C[B[A[i]]] ← i
 9         B[A[i]] ← B[A[i]] – 1
```

After the for loop of lines 1–2 initializes the array *B* to all zeros, the for loop of lines 3–4 inspects each input element. If the value of an input element is *i*, we increment *B*[*i*]. Thus, after line 4, *B*[*i*] holds the number of input elements equal to *i* for each integer *i* = 1,…,*k*. Lines 6–7 determine for each *i* = 1,…,*k* how many input elements are less than or equal to *i* by keeping a running sum of the array *B*. Finally, the for loop of lines 7–9 places each element into its correct sorted position in the output array *C*. If all input elements are distinct, then when we first enter line 7, for each *A*[*i*], the value *B*[*A*[*i*]] is the correct final position of *A*[*i*] in the output array, since there are *B*[*A*[*i*]] elements less than or equal to *A*[*i*]. Because the elements might not be distinct, we decrement *B*[*A*[*i*]] each time we place a value *A*[*i*] into the *C* array. Decrementing *B*[*A*[*i*]] causes the next input element with a value equal to *A*[*i*], if one exists, to go to the position immediately before *A*[*i*] in the output array. Since the for loop of lines 7–9 runs through the input elements in a reverse order (beginning with the last, ending with the first), this way the original order of equal elements is preserved, and so the resulting algorithm is stable.

Note, that in line 8 instead of the value of *A*[*i*] only the index *i* is stored in the output array *C*. This makes it possible to follow up on the order of equal elements in the final sorted order. *Figure 14* illustrates counting sort. Array *C* is called a ***permutation vector*** of the input vector *A*, and it is very useful if a whole database is sorted by a single field's values. When using permutation vectors, you don't have to change the order of whole records in a database physically, you just store another order of them. This way also different orders can be stored at the same time without modifying the input database.

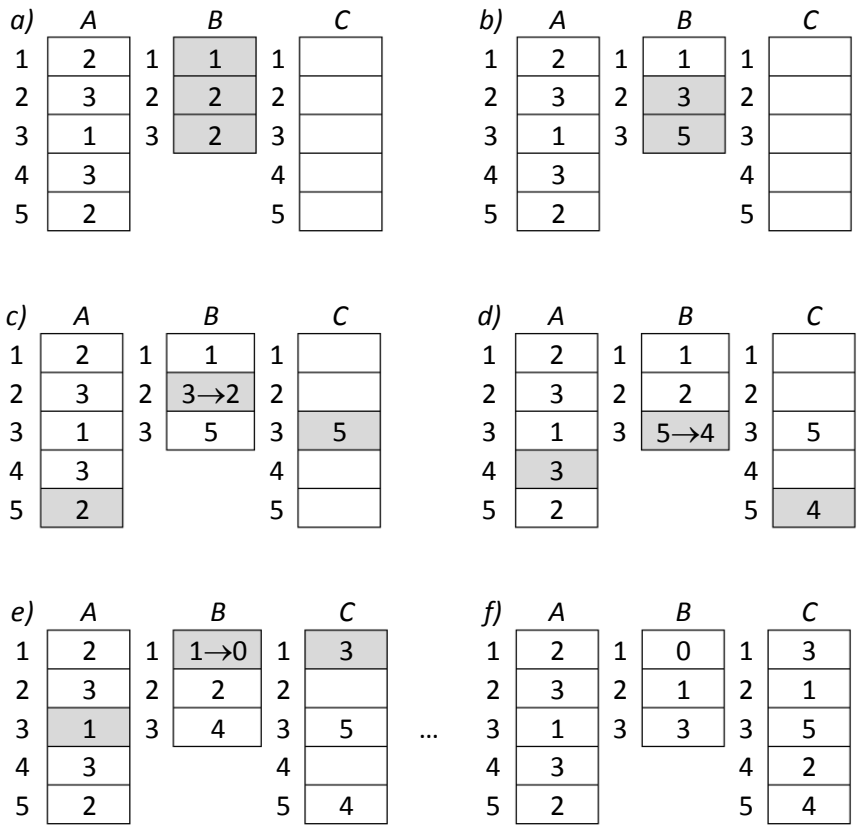


Figure 14. Counting sort on an example. Figure a) shows array B with the counted values (shaded) made by lines 3–4 of the pseudocode, Figure b) the same with cumulative values after lines 5–6. Figures c)–e) demonstrate some repetitions of lines 8–9 of the pseudocode, and f) the result. C contains the permutation vector of the sorted order of A.

How much time does counting sort require? The for loop of lines 1–2 takes time $\theta(k)$, the for loop of lines 3–4 takes time $\theta(n)$, the for loop of lines 5–6 takes time $\theta(k)$, and the for loop of lines 7–9 takes time $\theta(n)$. Thus, the overall time is $\theta(k + n)$. In practice, we usually use counting sort when we have $k = O(n)$, in which case the total running time is $T(n) = \theta(O(n) + n) = \theta(n)$.

Certainly, counting sort cannot only be used if the input consists of integers of a given range. Any finite base set's elements can be coded as integers. If, e.g., the base set consists of the letters A, B, and C, then we can code them as A $\leftrightarrow$ 1,

B ↔ 2, and C ↔ 3. Thus, we can sort the numbers instead of the letters using counting sort.

Exercises

45 Using *Figure 14* as a model, illustrate the operation of CountingSort on the array $A = (6, 1, 3, 1, 2, 4, 5, 6, 2, 4, 3)$.

46 Suppose that we were to rewrite the for loop header in line 7 of the CountingSort as

```
7  for i ← 1 to A.Length
```

Show that the algorithm still works properly. Is the modified algorithm stable?

Radix sort

If sequences of symbols are to be sorted in a lexicographical order, it is convenient to use radix sort. Radix sort can only be used if the sequences are of equal length. It sorts the sequences by their last symbols first, then by their last but one symbols, etc., using any sorting algorithm on the symbols. If at any position equal symbols occur, radix sort will only work correctly if the applied sorting algorithm is stable.

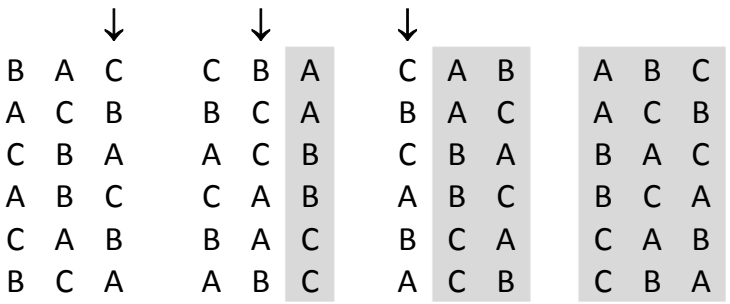


Figure 15. Radix sort. The arrows show which column will be used next to sort the sequences, and the shaded parts are those subsequences that are already sorted.

Figure 15 demonstrates how radix sort works. After the first round the sequences are sorted by their last symbols. After the second round they are sorted by the subsequences of their last two symbols, etc. It is obvious that for sorting the particular symbols a stable algorithm is needed, otherwise in the last round, e.g., ABC and ACB could change their order ending in an incorrect result.

If the length of the sequences is denoted by d (the number of digits in a sequence), and the time complexity of the sorting algorithm used by $T(n)$, then the time complexity of radix sort is $d \cdot T(n)$, where n stands for the number of sequences. If we assume that d can be considered as a constant for a given class of problems, and we use a stable, linear time sorting algorithm (e.g. counting sort), then the time complexity of the radix sort becomes $d \cdot T(n) = d \cdot \theta(n) = \theta(n)$, i.e., linear.

Exercises

47 Using *Figure 15* as a model, illustrate the operation of the radix sort on the following list of English words: COW, DOG, SEA, RUG, ROW, MOB, BOX, TAB, BAR, EAR, TAR, DIG, BIG, TEA, NOW, FOX.

48 Which of the following sorting algorithms are stable: insertion sort, merge sort, heapsort, and quicksort?

49 Show how to sort n integers in the range 0 to $n^3 - 1$ in $O(n)$ time.

Medians and Order Statistics

The i^{th} ***order statistic*** of a set of n elements is the i^{th} smallest element. For example, the ***minimum*** of a set of elements is the first order statistic ($i = 1$), and the ***maximum*** is the *n*th order statistic ($i = n$). A ***median***, informally, is the "halfway point" of the set. When n is odd, the median is unique, occurring at $i = (n + 1)/2$. When n is even, there are two medians, occurring at $i = n/2$ and $i = n/2 + 1$. Thus, regardless of the parity of n, medians occur at $i = \lfloor (n + 1)/2 \rfloor$ (the ***lower median***) and $i = \lceil (n + 1)/2 \rceil$ (the ***upper median***). For simplicity in this text, however, we consistently use the phrase "the median" to refer to the lower median.

Now let us consider the problem of selecting the i^{th} order statistic from a set of n distinct numbers. We assume for convenience that the set contains distinct numbers, although virtually everything that we do extends to the situation in which a set contains repeated values. We formally specify the selection problem as follows:

Input: A set *A* of n (distinct) numbers and an integer i, with $1 \leq i \leq n$.

Output: The element $x \in A$ that is larger than exactly $i - 1$ other elements of *A*.

We can solve the selection problem in $O(n \log n)$ time, since we can sort the numbers using heapsort or merge sort and then simply index the i^{th} element in the output array. Here we present faster algorithms.

Minimum and maximum

How many comparisons are necessary to determine the minimum of a set of n elements? We can easily obtain an upper bound of $n - 1$ comparisons as seen on page 11. We can, of course, find the maximum with $n - 1$ comparisons, as well. This is the best we can do, these upper bounds are tight.

In some applications, however, we must find both the minimum and the maximum of a set of n elements. For example, a graphics program may need to scale a set of (x, y) data to fit onto a rectangular display screen or other graphical output device. To do so, the program must first determine the minimum and maximum value of each coordinate.

At this point, it should be obvious how to determine both the minimum and the maximum of n elements using $\theta(n)$ comparisons, which is asymptotically optimal: simply find the minimum and maximum independently, using $n - 1$ comparisons for each, for a total of $2n - 2$ comparisons.

In fact, we can find both the minimum and the maximum using at most $3\lfloor n/2 \rfloor$ comparisons. We do so by maintaining both the minimum and maximum elements seen thus far. Rather than processing each element of the input by comparing it against the current minimum and maximum at a cost of 2 comparisons per element, we process elements in pairs. We compare pairs of elements from the input first with each other, and then we compare the smaller with the current minimum and the larger to the current maximum, at a cost of 3 comparisons for every 2 elements.

How we set up initial values for the current minimum and maximum depends on whether n is odd or even. If n is odd, we set both the minimum and maximum to the value of the first element, and then we process the rest of the elements in pairs. If n is even, we perform 1 comparison on the first 2 elements to determine the initial values of the minimum and maximum, and then process the rest of the elements in pairs as in the case for odd n.

Let us analyze the total number of comparisons. If n is odd, then we perform $\frac{3(n-1)}{2}$ comparisons. If n is even, we perform 1 initial comparison followed by $\frac{3(n-2)}{2}$ comparisons, for a total of $\frac{3(n-2)}{2} + 1 = \frac{3\left(n-\frac{4}{3}\right)}{2}$. Thus, in either case, the total number of comparisons is at most $3\lfloor n/2 \rfloor$.

Selection in expected linear time

The general selection problem appears more difficult than the simple problem of finding a minimum. Yet, surprisingly, the asymptotic running time for both problems is the same: $\theta(n)$. In this section, we present a divide-and-conquer algorithm for the selection problem. The select algorithm is modelled after the quicksort algorithm. As in quicksort, we partition the input array recursively. But unlike quicksort, which recursively processes both sides of the partition, the select algorithm works on only one side of the partition. This difference shows up in the analysis: whereas quicksort has an expected running time of $\theta(n \log n)$, the expected running time of the select algorithm is $\theta(n)$, assuming that the elements are distinct.

The following pseudocode of the select algorithm uses the procedure **Partition** introduced on page 41, and returns the *i*th smallest element of the array *A*.

```
Select(A,first,last,i)
 1  if  first = last
 2      then  return A[first]
 3  border ← Partition(A,first,last)
 4  k ← border – first + 1
 5  if  i ≤ k
 6      then  Select(A,first,border,i)
 7      else  Select (A,border + 1,last,i – k)
```

If there are more than one element in the remaining subarray (otherwise the i^{th} element has been found), **Select** calls the **Partition** procedure which arrange smaller elements in the first, larger elements in the second part of its input array, and returns with the index of the border element between the two parts. The size of the smaller elements' part is stored in *k*, and if $i \leq k$, i.e. the i^{th} element is in the first part, then the recursive call goes to the first part. Otherwise we carry on with the second part, where this time we are looking for the $(i - k)^{th}$ element since the first *k* elements have been left in the first part.

The worst-case running time for **Select** is $\theta(n^2)$, even to find the minimum, because we could be extremely unlucky and always partition around the largest remaining element, and partitioning of the subarrays shrinking step by step takes $n + (n-1) + \cdots + 1 = \frac{n(n+1)}{2} = \theta(n^2)$ time.

However, if we follow the idea of the λ assumption that none of the partition ratios will be worse during execution than a given $(1-\lambda):\lambda$ for some fixed $\lambda \in]0{,}1[$ (see on page 43), it turns out that the expected time complexity is linear. If $\lambda \geq 0.5$, then a worst behavior in this case results in a series of partitions of subarrays of the following sizes: $n, \lambda n, \lambda^2 n, \ldots, \lambda^d n$, where *d* stands for the depth of the recursion tree of the algorithm, and $\lambda^d n = 1$ (c.f. *Figure 12* on page 43). Hence the time consumption of the consecutive partitions is

$$n + \lambda n + \lambda^2 n + \cdots + \lambda^d n = \left(1 + \lambda + \lambda^2 + \cdots + \lambda^d\right) n = \frac{\lambda^{d+1} - 1}{\lambda - 1} n.$$

But from $\lambda^d n = 1$ it follows that $d = \log_{1/\lambda} n$, and so

$$\frac{\lambda^{d+1} - 1}{\lambda - 1} = \frac{\lambda^{\log_{1/\lambda} n} \cdot \lambda - 1}{\lambda - 1} = \frac{\frac{\lambda}{n} - 1}{\lambda - 1},$$

where the latter equality follows from the identity $a^{\log_{1/a} b} = \frac{1}{b}$. Multiplying this with n we get

$$\frac{\frac{\lambda}{n} - 1}{\lambda - 1} \cdot n = \frac{n - \lambda}{1 - \lambda} = O(n),$$

i.e., linear time complexity.

Selection in worst-case linear time

As we have seen, the select algorithm's worst case occurs if at every partition the part in which the selection follows is very large in proportion to the other. This balance depends on the pivot element of the partition algorithm. If a pivot element not too small, not too large could be found quickly, then the λ assumption could be fulfilled and thus the linear time complexity gained. In the following we show a modified version of the select algorithm where the pivot element is chosen in a tricky way.

Five-step algorithm:

1. If there is only one element in the input, then return it as the result. Otherwise divide the n elements of the input array into $\lfloor n/5 \rfloor$ groups of 5 elements each and at most one group made up of the remaining $n \bmod 5$ elements.

2. Find the median of each of the $\lceil n/5 \rceil$ groups by first insertion-sorting the elements of each group (of which there are at most 5) and then picking the median from the sorted list of group elements.
3. Use the **Five-step algorithm** recursively to find the median x of the $\lceil n/5 \rceil$ medians found in step 2.
4. Partition the input array around the median-of-medians x using the **Partition** algorithm. Let k be the number of elements on the low side of the partition.
5. Use the **Five-step algorithm** recursively to find the i^{th} smallest element on the low side if $i \leq k$, or the $(i-k)^{\text{th}}$ smallest element on the high side if $i > k$.

Now we show that the λ assumption holds for the algorithm above.

At least half of the medians found in step 2 are greater than or equal to the median-of-medians x. Thus, at least half of the $\lceil n/5 \rceil$ groups contribute at least 3 elements that are greater than x, except for the one group that has fewer than 5 elements if 5 does not divide n exactly, and the one group containing x itself. Discounting these two groups, it follows that the number of elements greater than x is at least

$$3\left(\left\lceil \frac{1}{2}\left\lceil \frac{n}{5} \right\rceil \right\rceil - 2\right) \geq \frac{3n}{10} - 6.$$

Because at least $\frac{3n}{10} - 6$ elements are greater than x, at most $n - (\frac{3n}{10} - 6) = \frac{7n}{10} + 6$ elements, i.e., the remaining elements are less than x.

Similarly, at least $\frac{3n}{10} - 6$ elements are less than x at the same time, and hence at most $\frac{7n}{10} + 6$ elements are greater than x. Note, that if $n \geq 60$ then $\frac{7n}{10} + 6 \leq \frac{8n}{10}$ holds which means that the λ assumption is fulfilled for the Five-step algorithm with the value $\lambda = 0.8$, and thus, the time complexity in all cases is $O(n)$, linear.

Exercises

50 Show how quicksort can be made to run in $O(n \log n)$ time in the worst case, assuming that all elements are distinct.

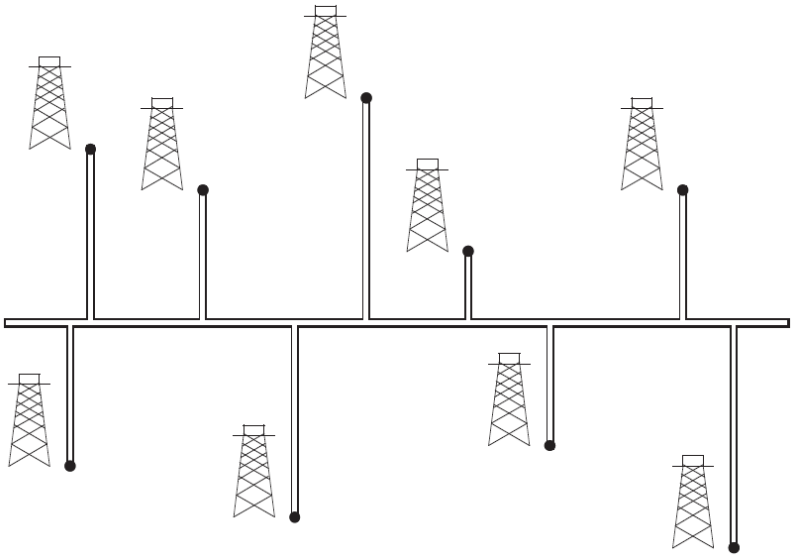


Figure 16. Professor Olay needs to determine the position of the east-west oil pipeline that minimizes the total length of the north-south spurs.

51 Professor Olay is consulting for an oil company, which is planning a large pipeline running east to west through an oil field of n wells. The company wants to connect a spur pipeline from each well directly to the main pipeline along a shortest route (either north or south), as shown in *Figure 16*. Given the x- and y-coordinates of the wells, how should the professor pick the optimal location of the main pipeline, which would be the one that minimizes the total length of the spurs? Show how to determine the optimal location in linear time.

52 For n distinct elements $x_1, x_2, \ldots, x_n$ with positive weights $w_1, w_2, \ldots, w_n$ such that $\sum_{i=1}^{n} w_i = 1$, the ***weighted (lower) median*** is the element x_k satisfying

$$\sum_{x_i<x_k} w_i < \frac{1}{2}$$

and

$$\sum_{x_i>x_k} w_i \leq \frac{1}{2}.$$

For example, if the elements are $0.1, 0.35, 0.05, 0.1, 0.15, 0.05, 0.2$ and each element equals its weight (that is, $w_i = x_i$ for $i = 1,2, \dots,7$), then the median is 0.1, but the weighted median is 0.2.

a. Argue that the median of $x_1, x_2, \dots, x_n$ is the weighted median of the x_i with weights $w_i = 1/n$ for $i = 1,2, \dots, n$.
b. Show how to compute the weighted median of n elements in $O(n \log n)$ worst-case time using sorting.
c. Show how to compute the weighted median in $\theta(n)$ worst-case time using a linear-time median algorithm such as the Five-step algorithm.

The ***post-office location problem*** is defined as follows. We are given n points $p_1, p_2, \dots, p_n$ with associated weights $w_1, w_2, \dots, w_n$. We wish to find a point p (not necessarily one of the input points) that minimizes the sum $\sum_{i=1}^{n} w_i d(p, p_i)$ where $d(a, b)$ is the distance between points a and b.

d. Argue that the weighted median is a best solution for the 1-dimensional post-office location problem, in which points are simply real numbers and the distance between points a and b is $d(a, b) = |a - b|$.
e. Find the best solution for the 2-dimensional post-office location problem, in which the points are (x, y) coordinate pairs and the distance between points $a = (x_1, y_1)$ and $b = (x_2, y_2)$ is the ***Manhattan distance*** given by $d(a, b) = |x_1 - x_2| + |y_1 - y_2|$.

Dynamic Programming

Dynamic programming, like the divide-and-conquer method, solves problems by combining the solutions to subproblems. (“Programming” in this context refers to a tabular method, not to writing computer code.) Divide-and-conquer algorithms partition the problem into disjoint subproblems, solve the subproblems recursively, and then combine their solutions to solve the original problem. In contrast, dynamic programming applies when the subproblems overlap — that is, when subproblems share subsubproblems. In this context, a divide-and-conquer algorithm does more work than necessary, repeatedly solving the common subsubproblems. A dynamic-programming algorithm solves each subsubproblem just once and then saves its answer in a table, thereby avoiding the work of recomputing the answer every time it solves each subsubproblem.

We typically apply dynamic programming to ***optimization problem***s. Such problems can have many possible solutions. Each solution has a value, and we wish to find a solution with the optimal (minimum or maximum) value. We call such a solution an optimal solution to the problem, as opposed to the optimal solution, since there may be several solutions that achieve the optimal value.

When developing a dynamic-programming algorithm, we follow a sequence of four steps:

1. Characterize the structure of an optimal solution.
2. Recursively define the value of an optimal solution.
3. Compute the value of an optimal solution, typically in a bottom-up fashion.
4. Construct an optimal solution from computed information.

Steps 1–3 form the basis of a dynamic-programming solution to a problem. If we need only the value of an optimal solution, and not the solution itself, then we can omit step 4. When we do perform step 4, we sometimes maintain additional information during step 3 so that we can easily construct an optimal solution, or even all of them.

Rod cutting

Our example uses dynamic programming to solve a simple problem in deciding where to cut steel rods. Sterling Enterprises buys long steel rods and cuts them

into shorter rods, which it then sells. Each cut is free. The management of Sterling Enterprises wants to know the best way to cut up the rods.

We assume that we know, for $i = 1,2, \ldots$, the price p_i in dollars that Sterling Enterprises charges for a rod of length i inches. Rod lengths are always an integral number of inches. For instance, they have the following price table:

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30

Figure 17. A sample price table for rods. Each rod of length i inches earns the company p_i dollars of revenue.

The rod-cutting problem is the following. Given a rod of length n inches and a table of prices p_i for $i = 1,2, \ldots$, determine the maximum revenue r_n obtainable by cutting up the rod and selling the pieces. Note that if the price p_n for a rod of length n is large enough, an optimal solution may require no cutting at all.

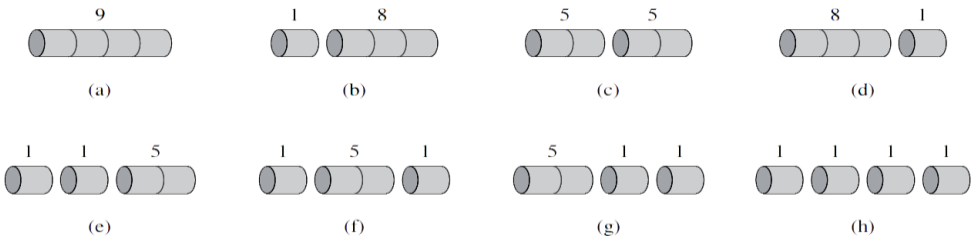


Figure 18. The 8 possible ways of cutting up a rod of length 4. Above each piece is the value of that piece, according to the sample price chart of Figure 17. The optimal strategy is part (c)—cutting the rod into two pieces of length 2—which has total value 10.

Consider the case when $n = 4$. *Figure 18* shows all the ways to cut up a rod of 4 inches in length, including the way with no cuts at all. We see that cutting a 4-inch rod into two 2-inch pieces produces revenue $p_2 + p_2 = 5 + 5 = 10$, which is optimal.

We can cut up a rod of length n in 2^{n-1} different ways, since we have an independent option of cutting, or not cutting, at distance i inches from the left end, for $i = 1,2, \ldots, n - 1$. We denote a decomposition into pieces using ordinary additive notation, so that $7 = 2 + 2 + 3$ indicates that a rod of length 7 is cut into three pieces — two of length 2 and one of length 3. If an optimal solution cuts the

rod into k pieces, for some $1 \leq k \leq n$, then an optimal decomposition $n = i_1 + i_2 + \cdots + i_k$ of the rod into pieces of lengths $i_1, i_2, \ldots, i_k$ provides maximum corresponding revenue $r_n = p_{i_1} + p_{i_2} + \cdots + p_{i_k}$.

For our sample problem, we can determine the optimal revenue figures r_i, for $i = 1,2, \ldots ,10$, by inspection, with the corresponding optimal decompositions

$r_1 = 1$ from solution $1 = 1$ (no cuts),
$r_2 = 5$ from solution $2 = 2$ (no cuts),
$r_3 = 8$ from solution $3 = 3$ (no cuts),
$r_4 = 10$ from solution $4 = 2 + 2$,
$r_5 = 13$ from solution $5 = 2 + 3$,
$r_6 = 17$ from solution $6 = 6$ (no cuts),
$r_7 = 18$ from solution $7 = 1 + 6$ or $7 = 2 + 2 + 3$,
$r_8 = 22$ from solution $8 = 2 + 6$,
$r_9 = 25$ from solution $9 = 3 + 6$,
$r_{10} = 30$ from solution $10 = 10$ (no cuts).

More generally, we can frame the values r_n for $n \geq 1$ in terms of optimal revenues from shorter rods:

$$r_n = \max(p_n, (r_1 + r_{n-1}), (r_2 + r_{n-2}), \ldots, (r_{n-1} + r_1)).$$

The first argument, p_n, corresponds to making no cuts at all and selling the rod of length n as is. The other $n - 1$ arguments to max correspond to the maximum revenue obtained by making an initial cut of the rod into two pieces of size i and $n - i$, for each $i = 1,2, \ldots, n - 1$, and then optimally cutting up those pieces further, obtaining revenues r_i and r_{n-i} from those two pieces. Since we don't know ahead of time which value of i optimizes revenue, we have to consider all possible values for i and pick the one that maximizes revenue. We also have the option of picking no i at all if we can obtain more revenue by selling the rod uncut.

Note that to solve the original problem of size n, we solve smaller problems of the same type, but of smaller sizes. Once we make the first cut, we may consider the two pieces as independent instances of the rod-cutting problem. The overall optimal solution incorporates optimal solutions to the two related subproblems, maximizing revenue from each of those two pieces. We say that the rod-cutting

problem exhibits ***optimal substructure***: optimal solutions to a problem incorporate optimal solutions to related subproblems, which we may solve independently.

In a related, but slightly simpler, way to arrange a recursive structure for the rod-cutting problem, we view a decomposition as consisting of a first piece of length i cut off the left-hand end, and then a right-hand remainder of length $n - i$. Only the remainder, and not the first piece, may be further divided. We may view every decomposition of a length-n rod in this way: as a first piece followed by some decomposition of the remainder. When doing so, we can couch the solution with no cuts at all as saying that the first piece has size $i = n$ and revenue p_n and that the remainder has size 0 with corresponding revenue $r_0 = 0$. We thus obtain the following simpler formula for r_n:

$$r_n = \max_{1 \le i \le n} (p_i + r_{n-i}).$$

In this formulation, an optimal solution embodies the solution to only one related subproblem — the remainder — rather than two.

Recursive top-down implementation

The following procedure implements the latter formula for r_n in a straightforward, top-down, recursive manner.

```
CutRod(p,n)
 1  if  n = 0
 2      then  return 0
 3  q ← – ∞
 4  for  i ← 1  to  n
 5       do  q ← max(q, p[i] + CutRod(p, n – i))
 6  return q
```

Procedure **CutRod** takes as input an array $p[1..n]$ of prices and an integer n, and it returns the maximum revenue possible for a rod of length n. If $n = 0$, no revenue is possible, and so **CutRod** returns 0 in line 2. Line 3 initializes the maximum revenue q to $-\infty$, so that the for loop in lines 4–5 correctly computes $q = \max_{1 \le i \le n}(p_i + \text{CutRod}(p, n - i))$; line 6 then returns this value. A simple induction on n proves that this answer is equal to the desired answer r_n.

If you were to code up **CutRod** in your favorite programming language and run it on your computer, you would find that once the input size becomes moderately large, your program would take a long time to run. For $n = 40$, you would find that your program takes at least several minutes, and most likely more than an hour. In fact, you would find that each time you increase n by 1, your program's running time would approximately double.

Why is **CutRod** so inefficient? The problem is that **CutRod** calls itself recursively over and over again with the same parameter values; it solves the same subproblems repeatedly. *Figure 19* illustrates what happens for $n = 4$: **CutRod**(*p*,*n*) calls **CutRod**(*p*,*n-i*) for $i = 1,2, \ldots, n$. Equivalently, **CutRod**(*p*,*n*) calls **CutRod**(*p*,*j*) for each $j = 0,1, \ldots, n-1$. When this process unfolds recursively, the amount of work done, as a function of n, grows explosively.

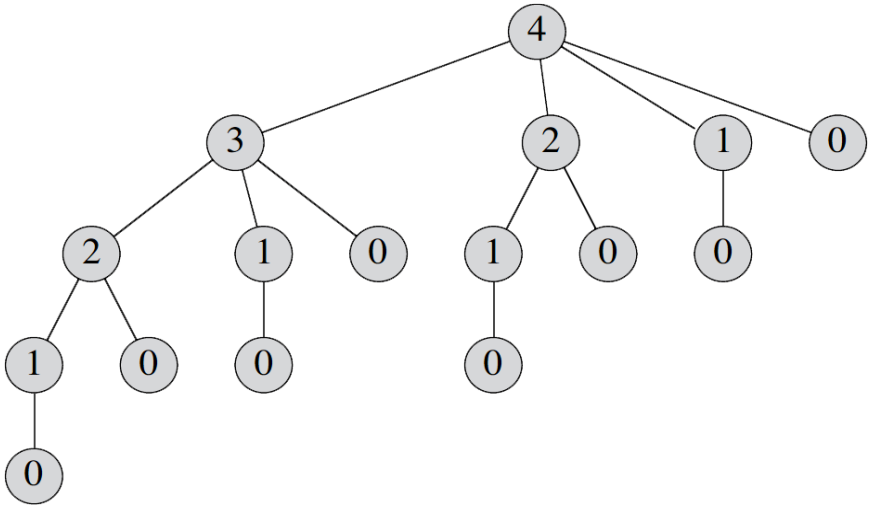


Figure 19. The recursion tree showing recursive calls resulting from a call CutRod(p,n) for $n = 4$. Each node label gives the size n of the corresponding subproblem, so that an edge from a parent with label s to a child with label t corresponds to cutting off an initial piece of size $s - t$ and leaving a remaining subproblem of size t . A path from the root to a leaf corresponds to one of the 2^{n-1} ways of cutting up a rod of length n. In general, this recursion tree has 2^n nodes and 2^{n-1} leaves.

To analyze the running time of **CutRod**, let $T(n)$ denote the total number of calls made to **CutRod** when called with its second parameter equal to n. This expression equals the number of nodes in a subtree whose root is labeled n in the recursion tree. The count includes the initial call at its root. Thus, $T(0) = 1$ and $T(n) = 1 + \sum_{j=0}^{n-1} T(j)$.

The initial 1 is for the call at the root, and the term $T(j)$ counts the number of calls (including recursive calls) due to the call **CutRod**(*p*,*n-i*), where $j = n - i$.

It is easy to show (see Exercise 53 on page 66) that $T(n+1)/T(n) = 2$ always holds. From this and from the initial condition $T(0) = 1$ it follows that $T(n) = 2^n$, and so the running time of **CutRod** is exponential in n.

In retrospect, this exponential running time is not so surprising. **CutRod** explicitly considers all the 2^{n-1} possible ways of cutting up a rod of length n. The tree of recursive calls has 2^{n-1} leaves, one for each possible way of cutting up the rod. The labels on the simple path from the root to a leaf give the sizes of each remaining right-hand piece before making each cut. That is, the labels give the corresponding cut points, measured from the right-hand end of the rod.

Using dynamic programming for optimal rod cutting

We now show how to convert **CutRod** into an efficient algorithm, using dynamic programming.

The dynamic-programming method works as follows. Having observed that a naive recursive solution is inefficient because it solves the same subproblems repeatedly, we arrange for each subproblem to be solved only once, saving its solution. If we need to refer to this subproblem's solution again later, we can just look it up, rather than recompute it. Dynamic programming thus uses additional memory to save computation time; it serves an example of a time-memory trade-off. The savings may be dramatic: an exponential-time solution may be transformed into a polynomial-time solution. A dynamic-programming approach runs in polynomial time when the number of distinct subproblems involved is polynomial in the input size and we can solve each such subproblem in polynomial time.

There are usually two equivalent ways to implement a dynamic-programming approach. We shall illustrate both of them with our rod-cutting example.

The first approach is ***top-down with memoization***. In this approach, we write the procedure recursively in a natural manner, but modified to save the result of each subproblem (usually in an array or hash table). The procedure now first checks to see whether it has previously solved this subproblem. If so, it returns the saved value, saving further computation at this level; if not, the procedure computes the

value in the usual manner. We say that the recursive procedure has been ***memoized***; it "remembers" what results it has computed previously.

The second approach is the ***bottom-up method***. This approach typically depends on some natural notion of the "size" of a subproblem, such that solving any particular subproblem depends only on solving "smaller" subproblems. We sort the subproblems by size and solve them in size order, smallest first. When solving a particular subproblem, we have already solved all of the smaller subproblems its solution depends upon, and we have saved their solutions. We solve each subproblem only once, and when we first see it, we have already solved all of its prerequisite subproblems.

These two approaches yield algorithms with the same asymptotic running time, except in unusual circumstances where the top-down approach does not actually recurse to examine all possible subproblems. The bottom-up approach often has much better constant factors, since it has less overhead for procedure calls.

Here is the pseudocode for the top-down CutRod procedure, with memorization added:

```
MemCutRod(p,n)
 1  for   i ← 0  to  n
 2        do   r[i] ← – ∞
 3  return MemCutRodAux(p,n,r)


MemCutRodAux(p,n,r)
 1  if   r[n] ≥ 0
 2     then   return  r[n]
 3  if   n = 0
 4     then   q ← 0
 5     else   q ← – ∞
 6              for   i ← 1  to  n
 7                   do   q ← max(q, p[i] + MemCutRodAux(p, n – i,r))
 8  r[n] ← q
 9  return q
```

Here, the main procedure **MemCutRod** initializes a new auxiliary array $r[0..n]$ with the value $-\infty$, a convenient choice with which to denote "unknown." (Known

revenue values are always nonnegative.) It then calls its helper routine, **MemCutRodAux**.

The procedure **MemCutRodAux** is just the memoized version of our previous procedure, **CutRod**. It first checks in line 1 to see whether the desired value is already known and, if it is, then line 2 returns it. Otherwise, lines 3–7 compute the desired value *q* in the usual manner, line 8 saves it in *r*[*n*], and line 9 returns it.

The bottom-up version is even simpler:

```
BottUpCutRod(p,n)
 1  r[0] ← 0
 2  forj ← 1  to  n
 3      do   q ← – ∞
 4           for   i ← 1  to  j
 5                do   q ← max(q, p[i] + r[j – i])
 6           r[j] ← q
 7  return r[n]
```

For the bottom-up dynamic-programming approach, **BottUpCutRod** uses the natural ordering of the subproblems: a problem of size i is "smaller" than a subproblem of size j if $i < j$. Thus, the procedure solves subproblems of sizes $j = 0,1, \ldots n$, in that order.

Line 1 of procedure **BottUpCutRod** initializes $r[0]$ to 0, since a rod of length 0 earns no revenue. Lines 2–5 solve each subproblem of size j, for $j = 1,2, \ldots, n$, in order of increasing size. The approach used to solve a problem of a particular size j is the same as that used by **CutRod**, except that line 5 now directly references array entry $r[j - i]$ instead of making a recursive call to solve the subproblem of size $j - i$. Line 6 saves in $r[j]$ the solution to the subproblem of size j. Finally, line 7 returns $r[n]$, which equals the optimal value r_n.

The bottom-up and top-down versions have the same asymptotic running time. The running time of procedure **BottUpCutRod** is $\theta(n^2)$, due to its doubly-nested loop structure. The number of iterations of its inner for loop, in lines 4–5, forms an arithmetic series. The running time of its top-down counterpart, **MemCutRod**, is also $\theta(n^2)$, although this running time may be a little harder to see. Because a recursive call to solve a previously solved subproblem returns immediately, **MemCutRod** solves each subproblem just once. It solves subproblems for sizes $0,1, \ldots, n$. To solve a subproblem of size n, the for loop of lines 6–7 of

MemCutRodAux iterates n times. Thus, the total number of iterations of this for loop, over all recursive calls of **MemCutRodAux**, forms an arithmetic series, giving a total of $\theta(n^2)$ iterations, just like the inner for loop of **BottUpCutRod**.

Reconstructing a solution

Our dynamic-programming solutions to the rod-cutting problem return the value of an optimal solution, but they do not return an actual solution: a list of piece sizes. We can extend the dynamic-programming approach to record not only the optimal value computed for each subproblem, but also a choice that led to the optimal value. With this information, we can readily print an optimal solution.

Here is an extended version of **BottUpCutRod** that computes, for each rod size j, not only the maximum revenue r_j, but also s_j, the optimal size of the first piece to cut off:

```
ExtBottUpCutRod(p,n)
 1  r[0] ← 0
 2  for  j ← 1  to  n
 3       do  q ← – ∞
 4           for  i ← 1  to  j
 5                do  if  q < p[i] + r[j – i]
 6                        then  q ← p[i] + r[j – i]
 7                              s[j] ← i
 8           r[j] ← q
 9  return r  and  s
```

This procedure is similar to **BottUpCutRod**, except that it uses a local array s, what it regularly updates in line 7 to hold the optimal size i of the first piece to cut off when solving a subproblem of size j.

The following procedure takes a price table p and a rod size n, and it calls **ExtBottUpCutRod** to compute the array $s[1..n]$ of optimal first-piece sizes and then prints out the complete list of piece sizes in an optimal decomposition of a rod of length n:

PrintCutRodSol(*p*,*n*)
```
1  (r,s) ← ExtBottUpCutRod(p,n)
2  while n > 0
3        do print s[n]
4           n ← n – s[n]
```

In our rod-cutting example, the call **ExtBottUpCutRod**(*p*,10) would return the following arrays:

i	0	1	2	3	4	5	6	7	8	9	10
$r[i]$	0	1	5	8	10	13	17	18	22	25	30
$s[i]$	0	1	2	3	2	2	6	1	2	3	10

A call to **PrintCutRodSol**(*p*,10) would print just 10, but a call with $n = 7$ would print the cuts 1 and 6, corresponding to the first optimal decomposition for r_7 given earlier.

Exercises

53 Show that $T(n) = 2^n$ follows from $T(n) = 1 + \sum_{j=0}^{n-1} T(j)$ and the initial condition $T(0) = 1$.

54 Modify **MemCutRod** to return not only the value but the actual solution, too.

55 Give an $O(n)$-time dynamic-programming algorithm to compute the *n*th Fibonacci number.

Amortized Analysis

In an amortized analysis, we average the time required to perform a sequence of data-structure operations over all the operations performed. With amortized analysis, we can show that the average cost of an operation is small, if we average over a sequence of operations, even though a single operation within the sequence might be expensive. Amortized analysis differs from average-case analysis in that probability is not involved; an amortized analysis guarantees the average performance of each operation in the worst case.

Two examples

In the following we introduce two problems over which the various methods of amortized analysis will be shown later. The traditional worst-case analysis fails to provide tight bounds on the time complexity of operations, while amortized analysis delivers much better results.

Augmented stack operations

In our first example of amortized analysis, we analyze stacks that have been augmented with a new operation. Traditional stacks have two fundamental operations, each of which takes $O(1)$ time:

Push(*key,Stack*) pushes object *key* onto stack *Stack*.

Pop(*Stack*) pops the top of stack *Stack* and returns the popped object. Calling Pop on an empty stack generates an error.

Since each of these operations runs in $O(1)$ time, let us consider the cost of each to be 1. The total cost of a sequence of n Push and Pop operations is therefore n, and the actual running time for n operations is therefore $\theta(n)$.

Now we add the stack operation **MultiPop**(*Stack*,*k*), which removes the k top objects of stack *Stack*, popping the entire stack if the stack contains fewer than k objects. **MultiPop** simply calls **Pop** k times. What is the running time of **MultiPop**(*Stack*,*k*) on a stack of s objects? The actual running time is linear in the number of **Pop** operations actually executed, and thus we can analyze **MultiPop** in terms of the abstract costs of 1 each for Push and Pop. The number of calls of

Pop by **MultiPop** equals $\min(s, k)$. Thus, the total cost of **MultiPop** is $\min(s, k)$, and the actual running time is a linear function of this cost.

Let us analyze a sequence of n **Push**, **Pop**, and **MultiPop** operations on an initially empty stack. The worst-case cost of a **MultiPop** operation in the sequence is $O(n)$, since the stack size is at most n. The worst-case time of any stack operation is therefore $O(n)$, and hence a sequence of n operations costs $O(n^2)$, since we may have $O(n)$ **MultiPop** operations costing $O(n)$ each. Although this analysis is correct, the $O(n^2)$ result, which we obtained by considering the worst-case cost of each operation individually, is not tight.

Incrementing a binary counter

As another example, consider the problem of implementing a k-bit binary counter that counts upward from 0. We use an array $A[0..k-1]$ of bits, where $A.Length = k$, as the counter. A binary number x that is stored in the counter has its lowest-order bit in $A[0]$ and its highest-order bit in $A[k-1]$, so that $x = \sum_{i=0}^{k-1} A[i] \cdot 2^i$. Initially, $x = 0$, and thus A contains all zeros. If we increment the counter anytime, all the one bits in the array A are flipped to zeros from left to right (in writing we use the opposite direction since the lower figures are on the right then, while in our array they are on the left) until we find a zero bit, which is then flipped to one, as in the following example:

That's how we add in writing:

1	1	0	0	0	1	1	1
+							1
1	1	0	0	1	0	0	0

The same in the array A:

i	0	1	2	3	4	5	6	7
$A[i]$	1	1	1	0	0	0	1	1
	↓	↓	↓	↓				
	0	0	0	1	0	0	1	1

We assume that we implement this algorithm with a function called **Increment**. As with the stack example, a cursory analysis of **Increment** yields a bound that is correct but not tight. A single execution of **Increment** takes time $\theta(k)$ in the worst case, in which array A contains all 1s. Thus, a sequence of n **Increment** operations on an initially zero counter takes time $O(nk)$ in the worst case.

Aggregate analysis

In ***aggregate analysis***, we show that for all n, a sequence of n operations takes worst-case time $T(n)$ in total. In the worst case, the average cost, or amortized cost, per operation is therefore $T(n)/n$. Note that this amortized cost applies to

each operation, even when there are several types of operations in the sequence. The other two methods we shall study in this chapter, the accounting method and the potential method, may assign different amortized costs to different types of operations.

Aggregate analysis of the augmented stack operations

Using aggregate analysis, we can obtain a better upper bound than $O(n^2)$ that considers the entire sequence of n operations. In fact, although a single **MultiPop** operation can be expensive, any sequence of n **Push**, **Pop**, and **MultiPop** operations on an initially empty stack can cost at most $O(n)$. Why? We can pop each object from the stack at most once for each time we have pushed it onto the stack. Therefore, the number of times that **Pop** can be called on a nonempty stack, including calls within **MultiPop**, is at most the number of **Push** operations, which is at most n. For any value of n, any sequence of n **Push**, **Pop**, and **MultiPop** operations takes a total of $O(n)$ time. The average cost of an operation is $O(n)/n = O(1)$. In aggregate analysis, we assign the amortized cost of each operation to be the average cost. In this example, therefore, all three stack operations have an amortized cost of $O(1)$.

We emphasize again that although we have just shown that the average cost, and hence the running time, of a stack operation is $O(1)$, we did not use probabilistic reasoning. We actually showed a worst-case bound of $O(n)$ on a sequence of n operations. Dividing this total cost by n yielded the average cost per operation, or the amortized cost.

Aggregate analysis of incrementing a binary counter

We can tighten our analysis to yield a worst-case cost of $O(nk)$ for a sequence of n **Increment** operations instead of by observing that not all bits flip each time **Increment** is called. As *Figure 20* shows, $A[0]$ does flip each time **Increment** is called. The next bit up, $A[1]$, flips only every other time: a sequence of n **Increment** operations on an initially zero counter causes $A[1]$ to flip $\lfloor n/2 \rfloor$ times. Similarly, bit $A[2]$ flips only every fourth time, or $\lfloor n/4 \rfloor$ times in a sequence of n **Increment** operations. In general, for $i = 0,1, \ldots, k-1$, bit $A[i]$ flips $\lfloor n/2^i \rfloor$ times in a sequence of n **Increment** operations on an initially zero counter. For $i \geq k$, bit $A[i]$ does not exist, and so it cannot flip. The total number of flips in the sequence is thus

$$\sum_{i=0}^{k-1} \left\lfloor \frac{n}{2^i} \right\rfloor < n \sum_{i=0}^{\infty} \frac{1}{2^i} = 2n.$$

The worst-case time for a sequence of n **Increment** operations on an initially zero counter is therefore $O(n)$. The average cost of each operation, and therefore the amortized cost per operation, is $O(n)/n = O(1)$.

Exercises

56 If the set of stack operations included a **MultiPush** operation, which pushes k items onto the stack, would the $O(1)$ bound on the amortized cost of stack operations continue to hold?

57 Show that if a **Decrement** operation were included in the k-bit counter example, n operations could cost as much as $O(nk)$ time.

58 Suppose we perform a sequence of n operations on a data structure in which the *i*th operation costs *i* if *i* is an exact power of 2, and 1 otherwise. Use aggregate analysis to determine the amortized cost per operation.

Counter value	A[7]	A[6]	A[5]	A[4]	A[3]	A[2]	A[1]	A[0]	Total cost
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1	1
2	0	0	0	0	0	0	1	0	3
3	0	0	0	0	0	0	1	1	4
4	0	0	0	0	0	1	0	0	7
5	0	0	0	0	0	1	0	1	8
6	0	0	0	0	0	1	1	0	10
7	0	0	0	0	0	1	1	1	11
8	0	0	0	0	1	0	0	0	15
9	0	0	0	0	1	0	0	1	16
10	0	0	0	0	1	0	1	0	18
11	0	0	0	0	1	0	1	1	19
12	0	0	0	0	1	1	0	0	22
13	0	0	0	0	1	1	0	1	23
14	0	0	0	0	1	1	1	0	25
15	0	0	0	0	1	1	1	1	26
16	0	0	0	1	0	0	0	0	31

Figure 20. An 8-bit binary counter as its value goes from 0 to 16 by a sequence of 16 Increment operations. Bits that flip to achieve the next value are shaded. The running cost for flipping bits is shown at the right. Notice that the total cost is always less than twice the total number of Increment operations.

The accounting method

In the ***accounting method*** of amortized analysis, we assign differing charges to different operations, with some operations charged more or less than they actually cost. We call the amount we charge an operation its ***amortized cost***. When an operation's amortized cost exceeds its actual cost, we assign the difference to specific objects in the data structure as ***credit***. Credit can help pay for later operations whose amortized cost is less than their actual cost. Thus, we can view the amortized cost of an operation as being split between its actual cost and credit that is either deposited or used up. Different operations may have different amortized costs. This method differs from aggregate analysis, in which all operations have the same amortized cost.

We must choose the amortized costs of operations carefully. If we want to show that in the worst case the average cost per operation is small by analyzing with amortized costs, we must ensure that the total amortized cost of a sequence of operations provides an upper bound on the total actual cost of the sequence. Moreover, as in aggregate analysis, this relationship must hold for all sequences of operations. If we denote the actual cost of the *i*th operation by c_i and the amortized cost of the *i*th operation by $\hat{c}_i$, we require

$$\sum_{i=1}^{n} \hat{c}_i \geq \sum_{i=1}^{n} c_i \quad \Leftrightarrow \quad \sum_{i=1}^{n} \hat{c}_i - \sum_{i=1}^{n} c_i \geq 0$$

for all sequences of n operations. The total credit stored in the data structure is the difference between the total amortized cost and the total actual cost, which must be nonnegative at all times. If we ever were to allow the total credit to become negative (the result of undercharging early operations with the promise of repaying the account later on), then the total amortized costs incurred at that time would be below the total actual costs incurred; for the sequence of operations up to that time, the total amortized cost would not be an upper bound on the total actual cost. Thus, we must take care that the total credit in the data structure never becomes negative.

Accounting method for the augmented stack operations

To illustrate the accounting method of amortized analysis, let us return to the stack example. In the table below we list the actual costs of the particular operations together with some suggested amortized costs:

Operation	Actual cost	Amortized cost
Push(*key*,*Stack*)	1	2
Pop(*Stack*)	1	0
MultiPop(*Stack*,*k*)	$\min(s, k)$	0

Note that the amortized cost of **MultiPop** is a constant (namely, zero), whereas the actual cost is variable. Here, all three amortized costs are constant. In general, the amortized costs of the operations under consideration may differ from each other, and they may even differ asymptotically.

We shall now show that we can pay for any sequence of stack operations by charging the amortized costs. Suppose we use a dollar bill to represent each unit of cost. We start with an empty stack. When we push a key on the stack, we use 1 dollar to pay the actual cost of the push and are left with a credit of 1 dollar (out of the 2 dollars charged), which we leave in the stack together with the key stored there. At any point in time, every key in the stack has a dollar of credit stored with it.

The dollar stored with the key serves as prepayment for the cost of popping it from the stack. When we execute a **Pop** operation, we charge the operation nothing and pay its actual cost using the credit stored in the stack. To pop a key, we take the dollar of credit off the stack and use it to pay the actual cost of the operation. Thus, by charging the **Push** operation a little bit more, we can charge the **Pop** operation nothing. Moreover, we can also charge **MultiPop** operations nothing because **MultiPop** does nothing more than calling the **Pop** operation several times; each of which costs nothing since it has already been prepaid for when pushing the element to be popped off the stack.

From this point on we can use a traditional worst-case analysis but this time with the amortized costs. The resulting bound will be an upper bound on the actual worst-case time complexity, too, because of the relation between the total amortized and actual costs. The worst-case time complexity of one single operation is 2 (the **Push** operation), and assuming this worst case occurs during all n operations it follows that $T(n) \leq 2n = O(n)$.

Accounting method for incrementing a binary counter

As we observed earlier, the running time of this operation is proportional to the number of bits flipped, which we shall use as our cost for this example. Let us once

again use a dollar bill to represent each unit of cost (the flipping of a bit in this example).

For the amortized analysis, let us charge an amortized cost of 2 dollars to set a bit to 1. When a bit is set, we use 1 dollar (out of the 2 dollars charged) to pay for the actual setting of the bit, and we place the other dollar on the bit as credit to be used later when we flip the bit back to 0. At any point in time, every 1 in the counter has a dollar of credit on it, and thus we can charge nothing to reset a bit to 0; we just pay for the reset with the dollar bill on the bit. The single steps' costs are listed in the table below:

Step	Actual cost	Amortized cost
Flip a bit from 0 to 1	1	2
Flip a bit from 1 to 0	1	0

Now we can determine the amortized cost of incrementing the counter. In one **Increment** operation some of the bits are flipped to 0, but only one bit is flipped to 1. That means that the amortized cost of one **Increment** operation always equals 2, having this as an upper bound for the actual costs. After n times incrementing the counter the time complexity turns out to be $T(n) \leq 2n = O(n)$.

Exercises

59 Redo Exercise 58 on page 70 using an accounting method of analysis.

60 Show how to implement a queue with two ordinary stacks so that the amortized cost of each **Enqueue** and each **Dequeue** operation is $O(1)$.

The potential method

The ***potential method*** is very similar to the accounting method only instead of representing prepaid work as credit stored with specific objects in the data structure, the potential method of amortized analysis represents the prepaid work as "potential energy," or just "potential," which can be released to pay for future operations. We associate the potential with the data structure as a whole rather than with specific objects within the data structure.

The potential method works as follows. We will perform n operations, starting with an initial data structure D_0. For each $i = 1,2, \dots, n$, we let c_i be the actual cost of the *i*th operation and D_i be the data structure that results after applying the *i*th operation to data structure D_{i-1}. A ***potential function*** Φ maps each data structure

D_i to a real number $\Phi(D_i)$, which is the potential associated with data structure D_i. The amortized cost $\hat{c}_i$ of the *i*th operation with respect to potential function Φ is defined by

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}).$$

The amortized cost of each operation is therefore its actual cost plus the change in potential due to the operation. Hence, the total amortized cost of the n operations is

$$\sum_{i=1}^{n} \hat{c}_i = \sum_{i=1}^{n} \left(c_i + \Phi(D_i) - \Phi(D_{i-1})\right) = \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0).$$

The latter equality comes from the sum of the potentials' differences being a telescope.

If we can define a potential function Φ so that $\Phi(D_n) \geq \Phi(D_0)$, then the total amortized cost $\sum_{i=1}^{n} \hat{c}_i$ gives an upper bound on the total actual cost $\sum_{i=1}^{n} c_i$.

In practice, we do not always know how many operations might be performed. Therefore, if we require that $\Phi(D_i) \geq \Phi(D_0)$ for all i, then we guarantee, as in the accounting method, that we pay in advance. We usually just define $\Phi(D_0)$ tobe 0 and then show that $\Phi(D_i) \geq 0$ for all i.

Intuitively, if the potential difference $\Phi(D_i) - \Phi(D_{i-1})$ of the *i*th operation is positive, then the amortized cost $\hat{c}_i$ represents an overcharge to the *i*th operation, and the potential of the data structure increases. If the potential difference is negative, then the amortized cost represents an undercharge to the *i*th operation, and the decrease in the potential pays for the actual cost of the operation.

Potential method for the augmented stack operations

We define the potential function Φ on a stack to be the number of objects in the stack. For the empty stack D_0 with which we start, we have $\Phi(D_0) = 0$. Since the number of objects in the stack is never negative, the stack D_i that results after the *i*th operation has nonnegative potential, and thus $\Phi(D_i) \geq 0$.

The total amortized cost of n operations with respect to Φ therefore represents an upper bound on the actual cost.

Let us now compute the amortized costs of the various stack operations. If the *i*th operation on a stack containing s objects is a **Push** operation, then the potential difference is $\Phi(D_i) - \Phi(D_{i-1}) = (s+1) - s = 1$. The amortized cost of the **Push** operation is therefore $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 1 + 1 = 2$.

Suppose that the *i*th operation on the stack is **MultiPop**(*Stack*,*k*), which causes $k' = \min(s, k)$ objects to be popped off the stack. The actual cost of the operation is k', and the potential difference is $\Phi(D_i) - \Phi(D_{i-1}) = -k'$. Thus, the amortized cost of the **MultiPop** operation is $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = k' - k' = 0$.

Similarly, the amortized cost of an ordinary **Pop** operation is 0.

The amortized cost of an augmented stack operation in worst case equals 2 so assuming the worst case for all the n operations the total amortized cost is $2n$. Since we have already argued that $\Phi(D_i) \geq \Phi(D_0)$, the total amortized cost of n operations is an upper bound on the total actual cost. The worst-case cost of n operations is therefore $T(n) \leq 2n = O(n)$.

Potential method for incrementing a binary counter

This time, we define the potential of the counter after the *i*th **Increment** operation to be b_i, the number of 1s in the counter after the *i*th operation.

Let us compute the amortized cost of an **Increment** operation. Suppose that the *i*th **Increment** operation resets t_i bits. The actual cost of the operation is therefore $c_i = t_i + 1$, since in addition to resetting t_i bits, it sets at most one bit to 1. Thus, $b_i = b_{i-1} - t_i + 1$, and the potential difference is $\Phi(D_i) - \Phi(D_{i-1}) = b_i - b_{i-1} = (b_{i-1} - t_i + 1) - b_{i-1} = 1 - t_i$. The amortized cost is therefore $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = (t_i + 1) + (1 - t_i) = 2$.

If the counter starts at zero, then $\Phi(D_0) = b_0 = 0$. Since $\Phi(D_i) = b_i \geq 0$ for all i, the total amortized cost of a sequence of n **Increment** operations is an upper bound on the total actual cost, and so the worst-case cost of n **Increment** operations is $T(n) \leq 2n = O(n)$.

Exercises

61 Suppose we have a potential function Φ such that $\Phi(D_i) \geq \Phi(D_0)$ for all i, but $\Phi(D_0) \neq 0$. Show that there exists a potential function Φ' such that $\Phi'(D_0) = 0$, and the amortized costs using Φ' are the same as the amortized costs using Φ.

62 Redo Exercise 58 on page 70 using a potential method of analysis.

63 Redo Exercise 60 on page 73 using a potential method of analysis.

Greedy Algorithms

The field of optimization problems is a wide an important branch of mathematics and algorithm theory. Plenty of mathematical models for real-life problems from mechanical design to economic planning require optimization. In most of the cases it is very difficult and time consuming to solve these problems. However, there are optimization problems that can be solved in a very straightforward and simple way using so-called greedy algorithms. The question arising here first is how it can be decided from an optimization problem whether it can be solved in a greedy way or not.

Elements of the greedy approach

Greedy algorithms solve problems where usually a somehow optimal subset of the finite set of feasible decisions has to be selected. They set out from an empty subset and extend it element by element always taking the step that is locally optimal at that point. Note that this approach excludes revisiting subproblems once solved. This principle obviously cannot be applied for all optimization problems. If, e.g., our aim is to find the shortest road route from Los Angeles to Las Vegas, then it would be really stupid to use a greedy strategy and find the closest town to our starting point and drive there first, for it would lead us to Santa Monica, which is situated in the opposite direction as our destination.

An optimization problem has to comply with the following two properties to be soluble by a greedy algorithm:

1. ***Greedy choice property***: If a greedy choice is made first, it can always be completed to achieve an optimal solution to the problem.
2. ***Optimal substructure property***: Any substructure of an optimal solution provides an optimal solution to the adequate subproblem.

The example of finding the shortest road route from L.A. to Vegas has an optimal substructure property because considering any subroute of a shortest way provides a shortest way to the subproblem indicated by the subroute. For instance if a shortest way from L.A. to Vegas leads through Victorville and Barstow, then the road-section between Victorville and Barstow is obviously a shortest way from Victorville to Barstow, i.e. an optimal solution to the subproblem.

However, the shortest road route problem does not comply with the greedy choice property because if we first head towards Santa Monica as described above, we cannot correct our faults and the route can never be accomplished to an optimal one anymore.

A simple example for greedy algorithms is an ***activity-selection problem***. Let us discuss the problem through an example. Let us assume we have plenty of channels on our TV and we want to spend a day watching TV. First we select the programs we would like to watch; this will be the base set (feasible choices). If there are time overlaps between programs on different channels, we have to decide. Let us suppose we want to watch as many of the programs as we can (we maximize for the number of programs seen and not for the time spent watching TV). The correct greedy algorithm is the following. First we watch the program which ends at the earliest time. If we are finished, we choose the next program with the earliest finishing time from those that have not begun yet. We iterate this procedure till the end, i.e. until no more programs are available.

The algorithm above is greedy in the sense that we always choose the program that ends at the earliest time because the earlier we finish it the more time we have left for other programs. We will now verify that the solution delivered by this method is optimal. Our first decision of choosing the earliest ending TV program does not bungle the whole solution since no other first choice would give us more possibilities for the rest of the time. This means that the problem complies with the greedy choice property, and also that the first program in any optimal solution for the problem can always be replaced by the greedy choice. Because of this, on the other hand, if we exclude the first program together with those programs overlapping it from an optimal solution, the rest still delivers an optimal solution for this reduced base set (otherwise if we extended an existing better solution for the reduced base set with our original first choice, we would get a better solution to the original problem than the considered optimal one). This is exactly the optimal substructure property, which also means that after the first choice the rest of the TV programs can be processed in a similar way, leading to an optimal solution at the end.

Exercises

64 Demonstrate how the greedy algorithm reviewed above for solving the activity-selection problem works on the following base set: {(8,11), (8,12), (9,10), (10,13), (11,12), (12,13), (12,14), (14,16), (15,18), (19,20)}.

65 Verify that the greedy approach of choosing the available activity which starts first (instead the one that ends first) does not comply with the elements of the greedy approach. Give a counter example where this modified algorithm fails.

66 Do the same as in the previous exercise but for the greedy approach of choosing the shortest of the available activities.

Huffman coding

Many people have already heard about Huffman coding as a data compression method. The notion "data compression" is a bit confusing since it is not the data themselves that are compressed but rather another coding is applied that delivers a shorter code for a file than a previous coding. Hence data compression is always relative. However, we can say that Huffman codes are optimal among the codes delivered by prefix-free character coding methods. A file coding is called character coding if every file consists of characters coming from a fixed set (an ***alphabet***), and each character has its own code in the coding. The file thus consists of a concatenation of such codes. Moreover, a character coding is called prefix-free (or shortly ***prefix coding***) if none of the characters' codewords is the beginning of any other codeword. This latter notion is certainly significant only if the codewords are of different length. In this case the codes are called ***variable-length codes*** (cf. the good old ASCII coding which consists of ***fixed-length***, i.e. 7 bit codes for characters).

For better tractability we introduce the notion of coding trees. A binary tree is called a ***coding tree*** if its leaves represent the characters of a given alphabet, and the paths leading from the root to the leaves define the character codes in the following way. Each edge of the tree has a label. This label is 0 if it leads to a left child and 1 if a right child is reached through it. The code of any character is simply the sequence of zeros and ones on the path leading from the root to the leaf representing the character.

Note that a coding defined by a coding tree is always prefix-free. Moreover, a prefix coding never needs delimiters between the character codes because after beginning with reading a character code in a file using the coding's tree it definitely ends at a leaf of the tree, hence the next bit must belong to the file's next character.

To be able to formulate the optimality of a coding defined by a coding tree some notations need to be introduced. In the following let us fix a file to be coded

consisting of the characters of a given *C* alphabet (a set of characters). Then for any $c \in C$ character the number of its occurrences in the file (its frequency) is denoted by $f(c)$. If a T coding tree is used for the character codes then the length of the code of character *c* (which equals the depth of the leaf representing it in the tree) is denoted by $d_T(c)$. Hence, the (bit)length of the file using the coding defined by the coding tree *T* is $B(T) = \sum_{c \in C} f(c) d_T(c)$.

When trying to find optimal prefix-free character codings (T codings with minimal $B(T)$), the first observation is that no coding tree containing vertices with only one child can be optimal. To verify this, imagine there is a vertex having only one child. Then this vertex can be deleted from the tree resulting in decreasing the depth of all leaves which were in the subtree of the deleted vertex, thus shortening the codes of all characters represented by these leaves (see character *c* in *Figure 21* after deleting the shaded vertex).

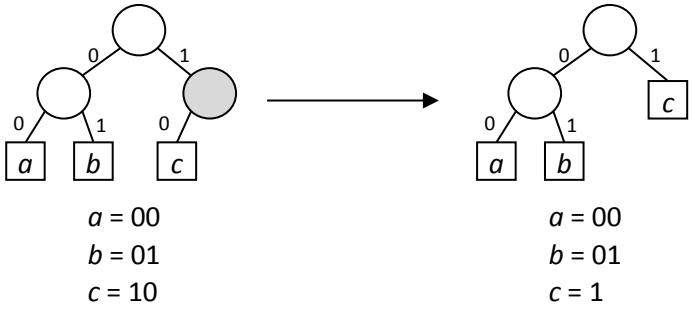


Figure 21. Deleting vertices having only one child from a coding tree shortens some of the codes.

However, the basic idea for Huffman coding is that we use different codes for different files: if a character occurs frequently in the given file, then we code it with a short codeword, whilst rare characters get long codewords. Following this principle the Huffman algorithm first sorts the elements of the alphabet according to their frequencies in an increasing order. Subsequently it joins the two leading elements of this list, and replaces the two elements with a single virtual one representing them with a frequency that is the sum of their frequencies preserving the order of the list. Then the first two elements of the new (one element shorter) list are joined the same way, and this process is iterated until the list consists of only one element representing all elements of the alphabet (see *Figure 22*). Hence the tree of the coding is built beginning at the leaves and the two rarest characters are represented by twins at maximal depth in the tree.

If we can prove that this problem complies with the two properties guaranteeing the optimality of the greedy approach, then the Huffman code must be optimal.

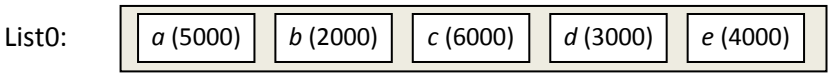
Our first assertion is that there exists an optimal solution where the two rarest characters are deepest twins in the tree of the coding, thus the greedy choice property is fulfilled by the Huffman coding. Indeed, taking any optimal coding tree and exchanging any two deepest twins for the two rarest characters' vertices, the total bitlength of the file code cannot decrease, because rarer characters get longer (or at least not shorter) codewords, and at the same time more frequent characters' codes become shorter (not longer).

The second assertion says that merging two (twin) characters leads to a problem similar to the original one, delivering the optimal substructure property for the greedy approach. The assertion itself is obvious. Thus, if an optimal coding tree is given, then joining any deepest twins the new tree provides an optimal solution to the reduced problem, where the two characters represented by these twins are replaced by a single common virtual character having the sum of the twins' frequencies.

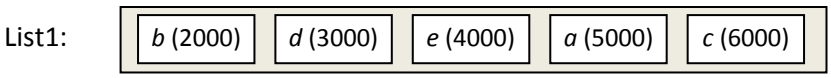
The two assertions above prove the optimality of the Huffman codes.

The following example demonstrates how Huffman's algorithm works. Let C be an alphabet consisting of five characters: $C = \{a, b, c, d, e\}$. Let us assume that their frequencies in the file to be coded are $f(a) = 5000$, $f(b) = 2000$, $f(c) = 6000$, $f(d) = 3000$ and $f(e) = 4000$. The list and the elements linked to the list elements are the following during the execution of the Huffman algorithm.

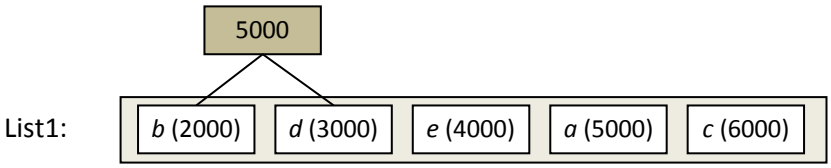
Note that any fixed-length character coding to code five characters would need at least three bit codewords, hence a 60.000 bit long coded file, while for the Huffman code above only *B*(*T*) = 45.000 bits are needed.



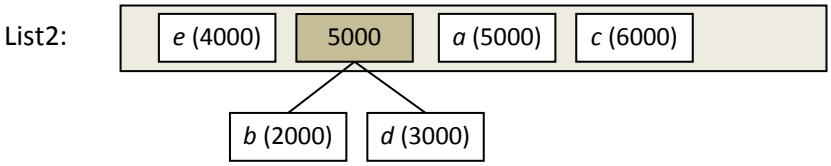
Sort the elements of the list.



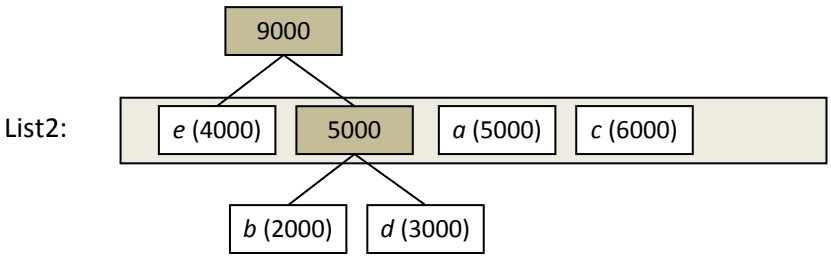
Join the two leading elements of the list.



Extract the two leading elements and insert the joint element.



Join the two leading elements of the list.



Etc.

Etc.

At the end, List5 consists of one single element, and the tree of the coding is finished.

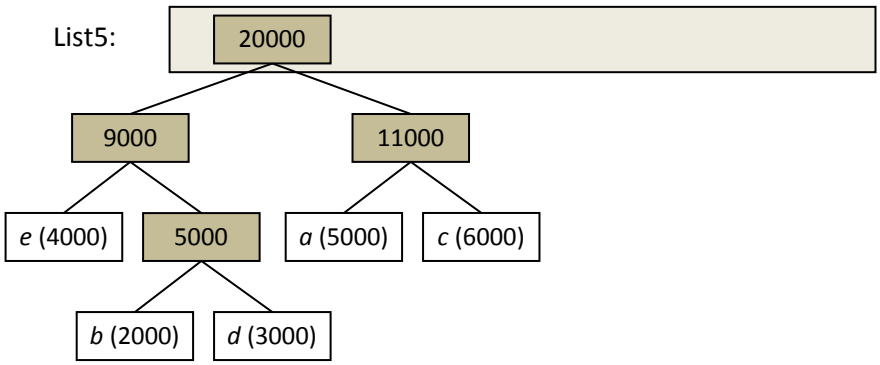


Figure 22. An example for Huffman codes, the resulting tree yields the codes: $a = 10, b = 010, c = 11, d = 011$ *and* $e = 00$.

Exercises

67 Demonstrate how the Huffman algorithm works on an alphabet whose elements have the following frequencies in the given file: 8000, 2000, 1000, 6000, 3000, 9000.

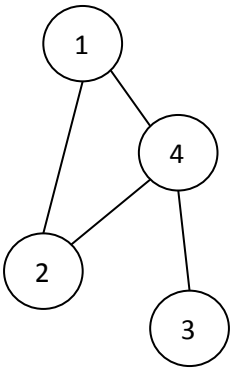
68 What kind of coding tree is built by the Huffman algorithm if the alphabet consists of *n* characters having the frequencies of the first *n* Fibonacci numbers?

Graphs

Graphs can represent different structures, connections and relations. The edges connecting the vertices can represent e.g. a road-network of a country or a flow structure of a chemical plant. In these cases the edges can have different numerical values assigned to them, representing the distances along road-sections and capacities or actual flow rates of pipelines, respectively. Such graphs are called ***weighted graph***s, where the values of the edges are the ***weight***s. Whatever we are modeling with graphs, we have to store them on a computer and be able to make calculations on them.

Graphs and their representation

The two most important graph representation types are the ***adjacency-matrix representation*** and the ***adjacency-list representation***. For both types the vertices are numbered and are referred to with their serial numbers, as you can see in the following example.



the graph

	1	2	3	4
1	0	1	0	1
2	1	0	0	1
3	0	0	0	1
4	1	1	1	0

the adjacency-matrix representation

1	2	4	
2	4	1	
3	4		
4	1	3	2

the adjacency-list representation

Figure 23. Different representations of a graph.

In the matrix representation if there is an edge pointing from vertex *i* to vertex *j* in the graph, it is represented by a 1 value at the i^{th} row's j^{th} position in the matrix. Otherwise there is a 0 at that position. One advantage of this representation is that connections can be checked or modified in constant time. Moreover, weighted graphs can easily be stored by simply replacing the 1 values in the matrix

with the weights of the edges. A serious drawback of adjacency-matrices is that if a graph has very few edges compared to its vertices, then a plenty of 0 values are unnecessarily stored in the matrix. Moreover, undirected graphs (where the edges have no direction) have symmetric adjacency matrices, causing further redundancy.

The list representation stores a list for each vertex *i*, consisting of the vertices adjacent with vertex *i*. This is the most storage saving method for storing graphs, however, some operations can last somewhat longer than on adjacency-matrices. To check whether there exists an edge pointing from vertex *i* to vertex *j*, the list of *i* has to be searched through for *j*. In worst case this list can contain nearly all vertices resulting in a time complexity linear in the number of vertices of the graph.

Single-source shortest path methods

Graphs have a plenty of applications, of which we will investigate only one here: the problem of finding the shortest path from a vertex to another (typically used in route planners, among others). Since there is no difference in worst case time complexity whether we look for the shortest path from a given vertex to a single other one or to all others (and there is no separate algorithm either), we investigate the problem of finding the shortest paths from a single vertex (also called the ***source***).

Breadth-first search

We have seen in subsection “Binary search trees” how binary trees can be walked. The same problem, i.e. walking the vertices arises on graphs in general. Two basic methods are the depth-first search and the breadth-first search. The ***depth-first search*** is a backtracking algorithm (see page 8). It starts from the source and goes along a path as far as it can without revisiting vertices. If it gets stuck, it tries the remaining edges running out from the actual vertex, and when there are no more, it steps back one vertex on its original path coming from the source and keeps on trying there.

The ***breadth-first search*** is not only simpler to implement than the depth-first search but it is also the basis for several important graph algorithms, therefore we are going to investigate this in details in the following. Breadth-first search can be imagined as an explosion in a mine where the edges of the graph represent the galleries of the mine which are assumed to have equal lengths. After the explosion

a shockwave starts from the source reaching the vertices adjacent to it first, then the vertices adjacent to these, etc. The breadth-first search algorithm usually cannot process all neighbors of a vertex at the same time on a computer, as in the mine example (except for the case of parallel computing), hence they are processed in a given order.

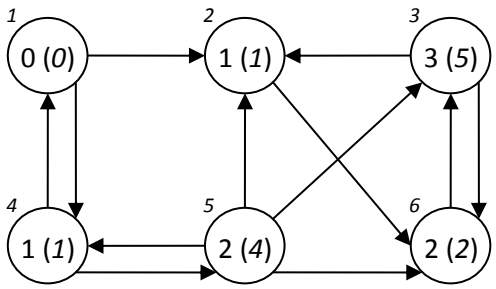


Figure 24. Breadth-first search from the source 1 in a graph. The labels written in the vertices denote the shortest distance from the source and the predecessor on a shortest path from the source in parentheses, respectively

The following pseudocode executes a breadth-first search from the source vertex *s* in a graph given with its adjacency matrix *A*.

```
BreadthFirstSearch(A,s,D,P)
 1  for  i ← 1 to A.CountRows
 2       do  P[i] ← 0
 3           D[i] ← ∞
 4  D[s] ← 0
 5  Q.Enqueue(s)
 6  repeat
 7       v ← Q.Dequeue
 8       for  j ← 1 to A.CountColumns
 9            do  if  A[v,j] > 0  and  D[j] = ∞
10                    then  D[j] ← D[v] + 1
11                          P[j] ← v
12                          Q.Enqueue(j)
13  until Q.IsEmpty
```

Parameters *D* and *P* are references to two one-dimensional arrays where the procedure stores the vertices' distances from the source during the search and their predecessors on the paths where they were reached. Initially it stores 0 in

all elements of *P* indicating that no predecessors have been assigned to the vertices yet, and ∞ in the elements of *D* (in reality any value greater than any distance that can be found during the search, e.g. the number *A.CountRows* of vertices of the graph) to indicate that the vertices have not been searched yet. The algorithm supplies a FIFO queue *Q*, where it puts the vertices that have already been reached but whose neighbors have not been processed yet. In the for-loop it checks all the neighbors *j* of vertex *v* (in line 9 it verifies whether vertex *j* is adjacent to *v* and if it has not been visited yet) and sets the values for the distance of the source and the predecessor on the path coming from the source.

Knowing the predecessors stored in array *P* of the vertices, the path leading to a vertex from the source can be restored recursively any time.

Dijkstra's algorithm

The breadth-first search immediately delivers shortest paths from a given source to any of the vertices, however, only if the length of all edges is considered as 1. In reality the edges of a network where shortest paths are to be determined have different weights. A very similar algorithm to the breadth-first search is Dijkstra's algorithm which can provide the shortest paths provided that every weight is positive which is mostly the case in real-life applications.

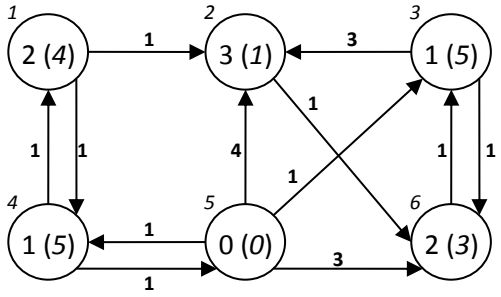


Figure 25. The result of Dijkstra's shortest path method started from source vertex 5. The bold values over the edges are the weights, the rest of the notation is similar to that of Figure 24.

The difference between the breadth-first search and Dijkstra's algorithm is that whilst the former never modifies a (shortest) distance value once given to a vertex, Dijkstra's algorithm visits all vertices adjacent to the vertex just being checked, independently from whether any of them have already been labeled or not.

However, to achieve an optimal solution it always chooses the one with the least distance value among the unprocessed vertices (those whose neighbors have not been checked yet). It realizes this idea using a minimum priority queue (that can be implemented using minimum heaps, see Page 37) denoted by *M* in the following pseudocode. The priority queue's order is defined by the values of array *D* of distances.

Dijkstra(*A,s,D,P*)

```
 1  for  i ← 1 to A.CountRows
 2       do  P[i] ← 0
 3           D[i] ← ∞
 4  D[s] ← 0
 5  for  i ← 1 to A.CountRows
 6       do  M.Enqueue(i)
 7  repeat
 8       v ← M.ExtractMinimum
 9       for  j ← 1 to A.CountColumns
10            do  if  A[v,j] > 0
11                    then  if  D[j] > D[v] + A[v,j]
12                              then  D[j] ← D[v] + A[v,j]
13                                    P[j] ← v
14  until M.IsEmpty
```

The idea mentioned above is simply an extension of the breadth-first search. We can even visualize this if the weights are natural numbers as in the example of *Figure 25*. In this case, if we insert virtual vertices in the graph the problem is reduced to a breadth-first search in an unweighted graph (see *Figure 26*, the virtual vertices are shaded). The choice formulated by taking the vertex with the minimal distance value in line 8 of the pseudocode translates in the converted problem to the order of vertices in the FIFO queue *Q* of the breadth-first search.

The time complexity of Dikstra's algorithm adds up from initializing the data structures (setting initial values in lines 1-4 and enqueuing all vertices into *M* in lines 5-6) and executing the search itself (the loop construct in lines 7-14). Initialization of arrays *D* and *P* takes $O(n)$ time (if denoting the number of vertices in the graph by *n*). Building a heap (using this optimal implementation of priority queue *M*) in lines 5-6 is linear, so this is $O(n)$ again (see page 39). The repeat loop is executed *n* times, for each iteration cycle consuming at most $O(\log n + n)$ time,

resulting in $O(n(\log n + n)) = O(n^2)$ in worst case. Since the initialization part does not worsen this, the final result is $T(n) = O(n^2)$.

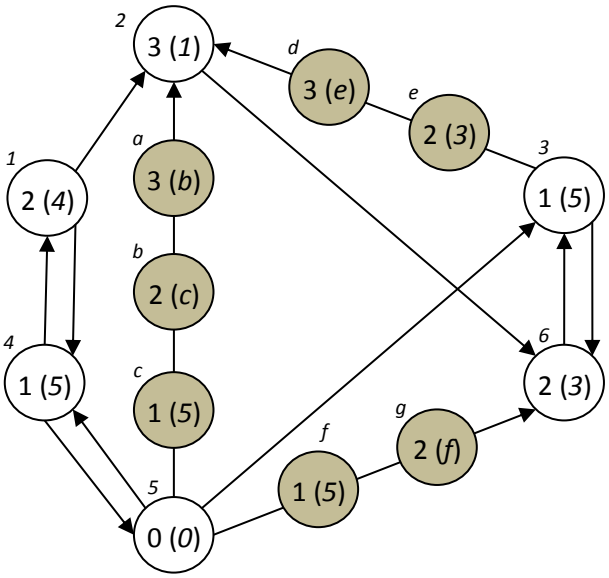


Figure 26. Visualizing the basic idea of Dijkstra's algorithm by converting the example of Figure 25 to a breadth-first search problem in an unweighted graph using virtual vertices.

Exercises

69 Demonstrate how the breadth-first search works on the example graph of *Figure 24*.

70 Demonstrate how Dijkstra's algorithm works on the example graph of *Figure 25*.

71 Write the pseudocode of a procedure that lists a shortest path from the source to a given vertex calling Dijkstra's algorithm.

Computational Geometry

Computational geometry is the branch of computer science that studies algorithms for solving geometric problems. In modern engineering and mathematics, computational geometry has applications in such diverse fields as computer graphics, robotics, VLSI design, computer-aided design, molecular modeling, metallurgy, manufacturing, textile layout, forestry, and statistics. The input to a computational-geometry problem is typically a description of a set of geometric objects, such as a set of points, a set of line segments, or the vertices of a polygon in counterclockwise order. The output is often a response to a query about the objects, such as whether any of the lines intersect, or perhaps a new geometric object, such as the convex hull (smallest enclosing convex polygon) of the set of points.

Cross products

Consider vectors p_1 and p_2, shown in *Figure 27*. We can interpret the ***cross product*** $p_1 \times p_2$ as the signed area of the parallelogram formed by the points $(0,0), p_1, p_2$, and $p_1 + p_2$. An equivalent, but more useful, definition gives the cross product as the determinant of a matrix:

$$p_1 \times p_2 = \det\begin{pmatrix} x_1 & x_2 \\ y_1 & y_2 \end{pmatrix} = x_1 y_2 - x_2 y_1.$$

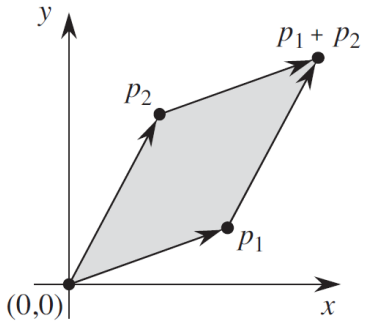


Figure 27. The cross product of vectors p_1 and p_2 is the signed area of the parallelogram.

If $p_1 \times p_2$ is negative, then p_2 is clockwise from p_1 with respect to the origin $(0,0)$; if this cross product is positive, then p_2 is counterclockwise from p_1 (positive and negative mathematical turning directions, resp.). A boundary condition arises if

the cross product is 0; in this case, the vectors are ***colinear***, pointing in either the same or opposite directions.

To determine whether a directed segment $\overrightarrow{p_0p_1}$ is closer to a directed segment $\overrightarrow{p_0p_2}$ in a clockwise direction or in a counterclockwise direction with respect to their common endpoint p_0, we simply translate to use p_0 as the origin. That is, we use the cross product to determine the turning direction of $p_1 - p_0$ to $p_2 - p_0$ with respect to $(0,0)$:

$$(p_1 - p_0) \times (p_2 - p_0) = (x_1 - x_0)(y_2 - y_0) - (x_2 - x_0)(y_1 - y_0).$$

If this cross product is negative, then $\overrightarrow{p_0p_2}$ is clockwise from $\overrightarrow{p_0p_1}$; if positive, it is counterclockwise.

Determining whether consecutive segments turn left or right

Our next question is whether two consecutive line segments $\overrightarrow{p_0p_1}$ and $\overrightarrow{p_1p_2}$ turn left or right at point p_1. Cross products allow us to answer this question without computing the angle. We simply translate both vectors to start from the origin $(0,0)$, resulting the vectors $p_1 - p_0$ and $p_2 - p_1$, respectively. The problem then reduces to the turning direction of a position vector to another one. But this, as we have already seen, can be stated by using the cross product:

$$(p_1 - p_0) \times (p_2 - p_1) = (x_1 - x_0)(y_2 - y_1) - (x_2 - x_1)(y_1 - y_0).$$

Determining whether two line segments intersect

To determine whether two line segments intersect, we check whether each segment ***straddles*** the line containing the other. A segment $\overline{p_1p_2}$ straddles a line if point p_1 lies on one side of the line and point p_2 lies on the other side. If a point lies directly on the line, it counts for both sides.

How can we detect whether, e.g., line segment $\overline{p_3p_4}$ straddles the line containing $\overline{p_1p_2}$. Let's imagine the line of $\overline{p_1p_2}$ is a river, and $\overline{p_1p_2}$ is our boat floating on it. The points p_3 and p_4 are trees on the banks of the river. How can we find out if they are on different sides of the river or not? Let us suppose we are sitting on the point p_1 looking in the direction of p_2. If we turn our eyes first towards p_3, and then again from p_2 to p_4, then obviously if the two turns had different directions (as in *Figure 28*), then the two trees lie on different banks of the river, what means, points p_3 and p_4 straddle the line segment $\overline{p_1p_2}$. The same construction works for

otherwise. Note, that if a point is directly on the line (a tree stands in the river), then it counts to both sides.

The Boolean expression we have to check is simply the following. If $(p_2 - p_1) \times (p_3 - p_1)$ and $(p_2 - p_1) \times (p_4 - p_1)$ have different signs (or any of them equals zero), then p_3 and p_4 straddle the line segment $\overline{p_1 p_2}$. Similarly if $(p_4 - p_3) \times (p_1 - p_3)$ and $(p_4 - p_3) \times (p_2 - p_3)$ have different signs (or any of them equals zero), then also p_1 and p_2 straddle the line segment $\overline{p_3 p_4}$. Summarized in one formula if

$$\big((p_2 - p_1) \times (p_3 - p_1)\big) \cdot \big((p_2 - p_1) \times (p_4 - p_1)\big) \leq 0$$
$$\text{and } \big((p_4 - p_3) \times (p_1 - p_3)\big) \cdot \big((p_4 - p_3) \times (p_2 - p_3)\big) \leq 0,$$

then the two line segments intersect.

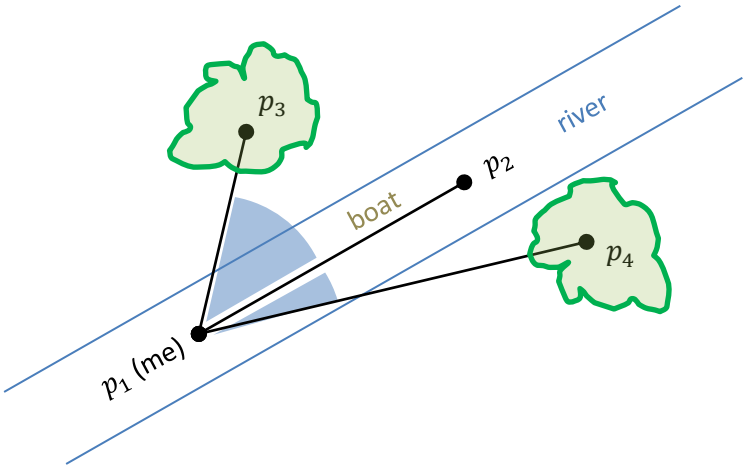


Figure 28. Representing the situation how we can find out whether two trees lie on different banks of the river we are boating on.

However, the boundary case when any of the "trees stand in the river" has to be examined more carefully. It is obvious, that in that case the corresponding cross product equals zero making the regarding inequality to be fulfilled, delivering the correct answer of existing intersection. But if all four points are colinear, i.e., all four cross products equal zero, this answer can be false, as shown in *Figure 29*.

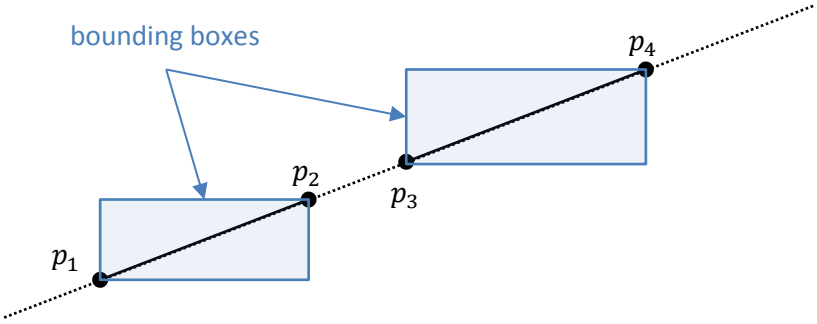


Figure 29. Both pairs of points straddle the lines of the others', but the two line segments still not intersect. This case can be excluded from further investigations by observing that their bounding boxes do not intersect.

To avoid this case, prior to checking whether the pairs of points mutually straddle each other's lines we test if the bounding boxes of the two line segments intersect (*Figure 29*). This can be done easily, since they only intersect if both of their coordinate intervals (their projections to both coordinate axes), intersect. In other words, these two Boolean expressions have to be true:

$[x_1, x_2] \cap [x_3, x_4] \neq \emptyset$ and $[y_1, y_2] \cap [y_3, y_4] \neq \emptyset$.

Exercises

72 Prove that if $p_1 \times p_2$ is negative, then p_2 is clockwise from p_1 with respect to the origin $(0,0)$; if this cross product is positive, then p_2 is counterclockwise from p_1.

73 Professor van Pelt proposes that only the expression $[x_1, x_2] \cap [x_3, x_4] \neq \emptyset$ needs to be tested in the boundary case. Show why the professor is wrong.

Determining whether any pair of segments intersect

This section presents an algorithm for determining whether any two line segments in a set of segments intersect. The algorithm uses a technique known as "sweeping," which is common to many computational-geometry algorithms. Moreover, this algorithm, or simple variations of it, can help solve other computational-geometry problems.

The algorithm determines only whether or not any intersection exists; it does not print all the intersections.

In ***sweeping***, an imaginary vertical ***sweep line*** passes through the given set of geometric objects, usually from left to right. We treat the spatial dimension that the sweep line moves across, in this case the x-dimension, as a dimension of time. Sweeping provides a method for ordering geometric objects, usually by placing them into a dynamic data structure, and for taking advantage of relationships among them. The line-segment-intersection algorithm in this section considers all the line-segment endpoints in left-to-right order and checks for an intersection each time it encounters an endpoint.

Ordering segments

We can order the segments that intersect a vertical sweep line according to the y-coordinates of the points of intersection. If a line segment is vertical then we treat the bottom endpoint of it as if it were a left endpoint and the top endpoint as if it were a right endpoint.

To be more precise, consider two segments s_1 and s_2. We say that these segments are ***comparable*** at x if the vertical sweep line with x-coordinate x intersects both of them. We say that s_1 is above s_2 at x, if s_1 and s_2 are comparable at x and the intersection of s_1 with the sweep line at x is higher than the intersection of s_2 with the same sweep line, or if s_1 and s_2 intersect at the sweep line.

Moving the sweep line

Sweeping algorithms typically manage two sets of data:

1. The ***sweep-line status*** gives the relationships among the objects that the sweep line intersects.

2. The ***event-point schedule*** is a sequence of points, called ***event points***, which we order from left to right according to their x-coordinates. As the sweep progresses from left to right, whenever the sweep line reaches the x-coordinate of an event point, the sweep halts, processes the event point, and then resumes. Changes to the sweep-line status occur only at event points.

For some algorithms, the event-point schedule develops dynamically as the algorithm progresses. The algorithm at hand, however, determines all the event points before the sweep, based solely on simple properties of the input data. In

particular, each segment endpoint is an event point. We sort the segment endpoints by increasing x-coordinate and proceed from left to right. (If two or more endpoints are ***covertical***, i.e., they have the same x-coordinate, we break the tie by putting all the covertical left endpoints before the covertical right endpoints. Within a set of covertical left endpoints, we put those with lower y-coordinates first, and we do the same within a set of covertical right endpoints.) When we encounter a segment's left endpoint, we insert the segment into the sweep-line status, and we delete the segment from the sweep-line status upon encountering its right endpoint. Whenever two segments first become consecutive in the total preorder, we check whether they intersect.

The sweeping line's algorithm manages an ordered data structure (e.g. a doubly linked list, see page 16) as the sweep-line status to store the segments actually intersecting the sweep-line. We use the following operations: **Insert** and **Delete** to insert a new line segment and delete one if the right endpoint has been reached, respectively. Furthermore, we use the operations **Above** and **Below** to determine which segment is above and which is below a given segment. These are elements directly preceding or supervening the actual segment in the ordered structure.

If the sweeping line comes to an event-point, it checks whether it is a left or a right endpoint. If it is a left endpoint, it inserts it to the sweep-line status, and examines if it intersects either the segment above or below it. If not, it proceeds with the next event point. If it is a right endpoint, it examines if the line segments above and below it intersect each other. If not, it proceeds with the next event point. If at any time this procedure finds an intersection, it stops and gives a positive answer. Otherwise it gives a negative answer.

Correctness and running time

If two line segments intersect, then one of the two line segments is inserted to the sweep-line status just before the other. When processing the left endpoint of the second line segment, the first line segment is already stored in the sweep-line status, and the second is inserted either directly above or below it. Hence, the test will find out that they intersect (part *a)* of *Figure 30*). The only exception is if a third line segment which is still stored in the sweep-line status swags between the two intersecting line segments. In this case when the sweeping line reaches the right endpoint of the sagging line segment, it will find the intersecting line segments above and below it in the sweep-line status (part *b)* of *Figure 30*).

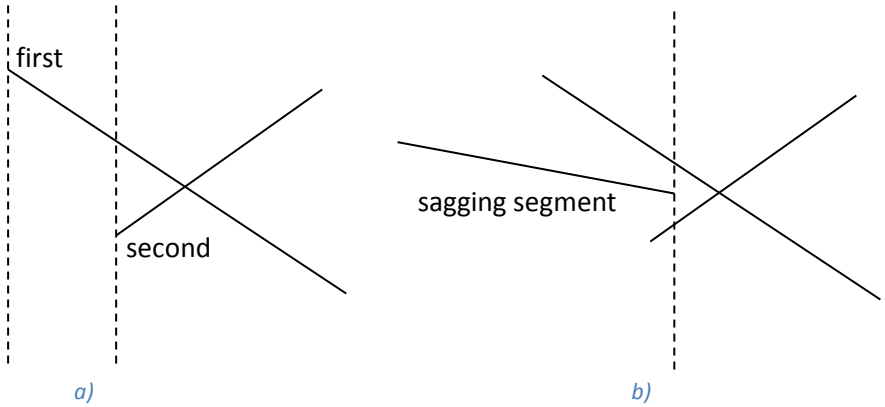


Figure 30. How the algorithm encounters intersection. a) After the second segment's left endpoint is reached, the first segment is directly above the second. b) When a sagging segment ends, the intersecting segments are one above the other in the sweep-line status.

Note, that in both cases the algorithm stops before passing by the first intersection point. Thus, the order of the segments stored in the sweep-line status never changes during the execution of the algorithm, thus, no resorting is necessary.

What is the time complexity of the sweeping line's algorithm? It first sorts the endpoints resulting in the event point schedule by their x-coordinates. This sorting takes $O(n \log n)$ time in worst case if using a sorting algorithm with an optimal time complexity.

At each event point it either inserts a new line segment into the sweep-line status or deletes one from it together with checking the intersection of one or two pairs of line segments. In rest of this section we assume that the sweep-line status is stored in a balanced binary search tree. A binary tree is called ***balanced*** if for any of its nodes the left and right subtree's depth do not differ more than by one. If a binary tree is balanced, then its depth depends logarithmically on the number of its vertices).

If a left endpoint comes next, we insert the line segment into the search tree in $O(\log n)$ time. Note, that although referring to a previous remark the order of the line segments does not change while the algorithm is running, during the insertion of a line segment into the sweep-line status the intersection of the sweep line with all of the involved nodes of the search tree have to be recalculated. This is necessary because the height (the y-coordinate) of the intersection of the line

segments with the sweep line changes continually (see *Figure 31*). This takes at most a constant time multiplied by $O(\log n)$. Then we have to find the line segments intersecting the sweep line directly under and above the new line segment (these are the predecessor and successor, respectively, in the search tree) in $O(\log n)$ time each, and at the end we check the intersection of the new one with these in constant time. In total, the whole number of steps for a single left endpoint does not exceed $O(\log n)$, hence for all n left endpoints the time complexity is $O(n \log n)$.

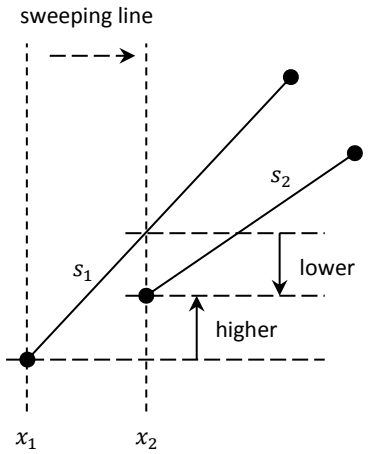


Figure 31. Although the intersection of line segment s_2 with the sweeping line at event point x_2 is higher than the intersection of s_1 in x_2, s_2 is in fact below s_1 in x_2.

In the case of a right endpoint the intersection of the line segments above and under the given line segment is checked prior to deleting it. In the binary search tree finding each of the line segments above and under the given line segment takes $O(\log n)$ time (finding the successor and predecessor, respectively), while we can check the intersection itself in constant time. For the at most n right endpoints to be checked it makes $O(n \log n)$ time.

Thus, the whole algorithm's time complexity equals $T(n) = 3 \cdot O(n \log n) = O(n \log n)$, assuming that our binary search tree stays balanced through the whole execution of the algorithm.

Exercises

74 Show that a set of n line segments may contain $n(n-1) = \theta(n^2)$ intersections.

75 Draw an example where the consecutive insertions of line segments into the sweep-line status results in an unbalanced binary search tree.

Finding the convex hull

The ***convex hull*** of a set Q of points, denoted by $\mathrm{CH}(Q)$, is the smallest convex polygon P for which each point in Q is either on the boundary of P or in its interior. We implicitly assume that all points in the set Q are unique and that Q contains at least three points which are not colinear. Intuitively, we can think of each point in Q as being a nail sticking out from a board. The convex hull is then the shape formed by a tight rubber band that surrounds all the nails. *Figure 32* shows a set of points and its convex hull.

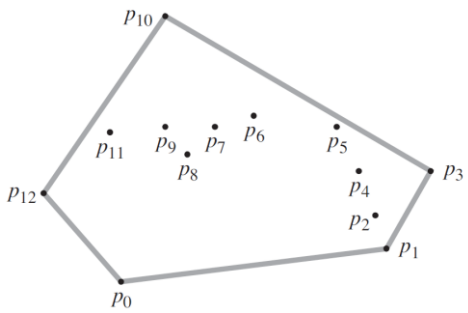


Figure 32. A set of points $Q = \{p_0, p_1, \ldots, p_{12}\}$ with its convex hull $CH(Q)$ in gray.

The convex hull problem is simply to pick up those points from the input set Q which are vertices of the convex hull.

Graham's scan

Graham's scan solves the convex-hull problem by maintaining a stack S of candidate points. It pushes each point of the input set Q onto the stack one time, and it eventually pops from the stack each point that is not a vertex of $\mathrm{CH}(Q)$. When the algorithm terminates, stack S contains exactly the vertices of $\mathrm{CH}(Q)$, in counterclockwise order of their appearance on the boundary.

Two additional stack operations are introduced. The first is **Top**(*S*), which just returns the object on the top of the stack S without extracting it from the stack. The second is **Top2**(*S*), doing the same to the object on the stack next to the object on the top (the second element on the stack from above). Initially stack S is empty.

The algorithm first renumbers the input points. The initial point, p_0 will be the point with the minimal y-coordinate value (if there are more than one, then the point with the minimal x-coordinate among them will be chosen). The remaining points are sorted by polar angle in counterclockwise order around p_0 (if more than one point has the same angle, remove all but the one that is farthest from p_0). This can be imagined as if a laser ray was shot out of p_0 to the right, in the direction of positive infinity parallel to the x-axis. Then the ray is turned counterclockwise slowly around p_0, and the order of hitting the particular points with the laser delivers the order of the numbering of them. Note, that this can be implemented with any comparison sorting algorithm, since we can compare two points p_i and p_j by determining the turning direction of $\overrightarrow{p_0p_j}$ from $\overrightarrow{p_0p_i}$. If this direction is left, then p_j is greater than p_i, otherwise it is less. The equal elements are filtered out at the end. After this, the first three points, p_0, p_1, and p_2 are pushed to the stack.

The algorithm then iteratively repeats the following steps for all remaining points p_i $(i = 3,4, \dots n-1,$ where $n = |Q|)$ of Q in their new order. While the consecutive vectors $\overrightarrow{\mathbf{Top2}(S)\,\mathbf{Top}(S)}$ and $\overrightarrow{\mathbf{Top}(S)\,p_i}$ form a right turn or are colinear, the top element of the stack is popped. That's because in this case the point **Top**(*S*) would be a dent on the hull, so it must be inside the convex hull, not being a vertex of it. (If the vectors are colinear, then **Top**(*S*) is an inner point of an edge of the convex hull's polygon, thus, also not a vertex of it.) After finding the first left turn, p_i is pushed to the stack, and the iteration takes the next point ($i \leftarrow i+1$).

The running time accumulates from the sorting of the points, which takes $O(n \log n)$, and the iterative part described above. Because each of the $\theta(n)$ points appear at most once on the stack, the time consumption of the iterations is $\theta(n)$ in all. Thus, the time complexity of Graham's scan is $O(n \log n)$.

Exercises

76 Prove that in the procedure of Graham's scan, points p_1 and p_{n-1} must be vertices of $\mathrm{CH}(Q)$.

References

1. **Dijkstra, E. W.** *Notes on Structured Programming.* Eindhoven : T.H. Report, 1969.

2. **Mágoriné Huhn, Ágnes.** *Algoritmusok és adatszerkezetek.* Szeged : JGYF Press, 2005. p. 167. ISBN: 9789639167360.

3. **Landau, Edmund.** *Handbuch der Lehre von der Verteilung der Primzahlen.* Leipzig : B. G. Teubner, 1909.

4. **Cormen, Thomas H., et al.** *Introduction to Algorithms, Third Edition.* US : The MIT Press, 2009. ISBN-10:0-262-03384-4, ISBN-13:978-0-262-03384-8.

5. *"Partition: Algorithm 63," "Quicksort: Algorithm 64," and "Find: Algorithm 65.".* **Hoare, C. A.R.** 1961, Comm. ACM 4(7), pp. 321-322.

Index