Software Design and Algorithms



Algorithms and data structures

EPAM Systems Inc.

Learn & Development

Software Design and Algorithms

[www.epam.com](http://www.epam.com/)

Table of Contents

[Algorithms and data structures 1](#_Toc70021535)

[1. algorithms 3](#_Toc70021536)

[1.1. algorithm characteristics 3](#_Toc70021537)

[1.2. Common algorithms 4](#_Toc70021538)

[1.2.1. Search algorithms 4](#_Toc70021539)

[1.2.2. Sorting algorithms 4](#_Toc70021540)

[1.2.3. Computational algorithms 4](#_Toc70021541)

[1.2.4. Collection algorithms 5](#_Toc70021542)

[1.3. Algorithm performance 5](#_Toc70021543)

[1.4. correctness of algorithm 7](#_Toc70021544)

[1.5. analyzing algorithm 8](#_Toc70021545)

[2. insertion sort 9](#_Toc70021546)

[2.1. asymptotic notation 14](#_Toc70021547)

[3. Divide and conquer 17](#_Toc70021548)

[3.1. merge sort 18](#_Toc70021549)

[3.2. Master method and theorem 20](#_Toc70021550)

[4. heap. heap sort. proirity queue 22](#_Toc70021551)

[4.1. heap 23](#_Toc70021552)

[4.2. Maintaining the heap property 24](#_Toc70021553)

[4.3. building a heap 26](#_Toc70021554)

[4.4. heap sort 27](#_Toc70021555)

[4.5. priority queue 28](#_Toc70021556)

[5. sorting in linear time 30](#_Toc70021557)

[5.1. counting sort 30](#_Toc70021558)

[5.2. bucket sort 32](#_Toc70021559)

# algorithms

So, what is algorithm? Non-programmers joke, that this is a word used by programmers when they do not want to explain what they did.

Informally, an **algorithm** is any well-defined computational procedure that takes some value, or set of values as input and produces some value, or set of values, as output. An algorithm is thus a sequence of computational steps that transform the input into the output.

We can also view an algorithm as a tool for solving a well-specified computational problem. The statement of the problem specifies in general terms the desired input/output relationship. The algorithm describes a specific computational procedure for achieving that input/output relationship.

You may ask a question why do we need all this? We are qualified enough specialists and have not encountered them. In most projects, if you just take it and do it, then you really don't need them, program may work without it. But if you approach writing code in a meaningful way, it is necessary.

In fact, any code you write is an evaluation-based algorithm. It's not just sorting and other things. You have input data and you give the result based on them. It is also important to understand how the language you are working with works. In JS example, it has a sort() function. If you look in the code, it uses the C++ quicksort function. And sometimes it makes sense not to use what has already been implemented, but implement our own sorting algorithm.

## algorithm characteristics

Algorithms usually have a specific set of input values ​​that it can work with to get a result. For example, sorting algorithms take collections of data values ​​and try to order them.

You can also talk about the classification of an algorithm using a variety of criteria. Some algorithms operate on their datasets sequentially, which means they are sequential in nature. Whereas a parallel algorithm can split a dataset into smaller pieces and then work with each one at the same time.

The algorithm can be exact, in which case it produces a known predictable value, or it can be approximate, in which case it tries to find an answer that may not be. For example, a face recognition algorithm may not give the same answer every time with the same face.

Algorithms can be deterministic, in which case they perform each step with an exact solution, and it can be non-deterministic if they try to find a solution using consecutive guesses that become more accurate over time.

## Common algorithms

### Search algorithms

This kind of algorithms find specific data in structure. (for example, a substring within a string). One of the most common types of algorithms you come across is search algorithms, which are used when you need to find a piece of data within a larger data structure. For example, searching for a substring within a larger string, or perhaps searching for a file in a set of subfolders in the file system.

### Sorting algorithms

Take a dataset and apply a sort to order it. Sorting algorithms are another very common type used when working with ordered datasets. And, you guessed it, they take a dataset and put them in a specific order.

### Computational algorithms

Given one set of data, calculate another basing on this set. Computational algorithms are used to get from one dataset to another. And a simple example would be calculating whether a given number is a prime number, or perhaps converting a temperature from one scale to another.

### Collection algorithms

Work with collections of data (count specific items, navigate among data elements, filter out unwanted data etc.). Finally, there are collection algorithms that involve manipulating or navigating between sets of data that are stored in a particular structure. It's easy to imagine examples here that count the number of specific items, filter out unwanted data, and so on.

## Algorithm performance

Since algorithms are designed to work with datasets and solve computational problems, it is important to understand how to talk about the performance of an algorithm. This is an important factor in how you choose a particular algorithm for solving a computational problem, as well as understanding how your program will behave in different circumstances.

So, we want to measure how the performance of an algorithm changes based on the size of the input dataset. You will often hear a term called Big-O notation that is used to describe the performance of an algorithm. This notation format is used to describe how a particular algorithm works as the input data set grows over time. And the reason the letter O is used is because the rate at which the complexity of an algorithm grows is also called order of operation. It usually describes a worst-case scenario of how long it will take to complete a given operation. And it's important to note that many different algorithms and data structures have more than one Big-O value. For example, data structures can usually perform several types of operations, such as inserting or searching for values, each with its own order of operations.

|  |  |  |
| --- | --- | --- |
| Notation | Description | Example |
| O(1) | Constant time | Looking for a single element in an array knowing index |
| O(log n) | Logarithmic | Finding an item in a sorted array with a binary search |
| O(n) | Linear time | Searching an unsorted array of for a specific value |
| O(n log n) | Log-linear | Complex sorting algorithms like heap sort and merge sort |
| O(n2) | Quadratic | Simple sorting algorithms, such as bubble sort, selection sort |

So let's take a look at some common Big-O notation terms to see what they mean in real-world scenarios. Each of these items are arranged in ascending order of complexity of time.

The simplest example is what's called constant time, and this corresponds to a Big-O equal to one. And in essence, this means that the operation in question does not depend on the number of elements in a given dataset. A good example of this is calculating whether a number is even or odd, or looking for a specific index of an element in an array.

Next comes the order of log n, which is called logarithmic time. And a typical example of this kind of operations is searching for a specific value in a sorted array using binary search. Thus, as the number of elements in the sorted array grows, it only takes a logarithmic ratio of time to find any given element.

The next is linear time, which corresponds to Big-O of n, and this level of time complexity corresponds to a typical example of finding an element in an unsorted array.

We also have an order of n times log(n) or the so-called logarithmic time complexity. Examples would be some sorting algorithms such as heap sort and merge sort.

And finally, the order of n squared, which is called the quadratic time complexity, and as you probably guessed, this is not a very good level of performance, because it means that as the number of elements in the dataset increases, the time it takes to process they increase in the square of this number. An example would be several simpler sorting algorithms such as bubble sort and selective sort.



Figure 1

Believe it or not, there is actually even worse than quadratic dependence. But this is a good list of the levels of difficulty that you are likely to encounter in your work. As you can see, there is a graph (Figure 1) representatively showing how much the number of operations will change depending on the number of input elements.

## correctness of algorithm

An algorithm is considered correct if, at any admissible (for a given problem) input, it finishes its work and produces a result that meets the requirements of the problem. In this case, the algorithm is said to solve the given computational problem. An incorrect algorithm (for some input) may not stop at all or give an incorrect result, but this does not mean that such algorithms are completely useless. If errors are rare enough or it is possible to control the frequency of errors, we may admit the use of incorrect algorithms. It may be that initially we have a specific task with one data set and we have compiled an algorithm. Then some new data begins to arrive, there are not many of them, but with them the algorithm slows down significantly. But since there is little such data so far, the algorithm is quite working.

## analyzing algorithm

***Analyzing*** an algorithm has come to mean predicting the resources that the algorithm requires. Occasionally, resources such as memory, communication bandwidth, or computer hardware are of primary concern, but most often it is computational time that we want to measure. Generally, by analyzing several candidate algorithms for a problem, we can identify a most efficient one. Such analysis may indicate more than one viable candidate, but we can often discard several inferior algorithms in the process.

Before we can analyze an algorithm, we must have a model of the implementation technology that we will use, including a model for the resources of that technology and their costs. For this lecture, we will assume a generic one processor, ***random-access machine (RAM)*** model of computation as our implementation technology and understand that our algorithms will be implemented as computer programs. In the RAM model, instructions are executed one after another, with no concurrent operations.

# insertion sort

Insertion sort is a simple sorting algorithm that works similar to the way you sort playing cards in your hands. The array is virtually split into a sorted and an unsorted part. Values from the unsorted part are picked and placed at the correct position in the sorted part.

To sort an array of size n in ascending order (Figure 2):

* Iterate from arr[1] to arr[n] over the array.
* Compare the current element (key) to its predecessor.
* If the key element is smaller than its predecessor, compare it to the elements before. Move the greater elements one position up to make space for the swapped element.



Figure 2

The running time of the *insertionSort* procedure depends on the set of input values: it takes longer to sort thousands of numbers than to sort three numbers. In the general case, the running time of the algorithm increases with an increase in the amount of input data, so it is common practice to represent the running time of a program as a function depending on the number of input elements. For this, the concepts of “algorithm time” and “size of input data” need to be determined more precisely.

The most adequate concept of input data size depends on the problem in question. For each task considered below, the way of measuring the size of the input data will be indicated. In the case of insertionSort, the number of input elements is considered as the size of the input data.

The running time of an algorithm on a particular input is the number of primitive operations or “steps” executed. It is convenient to define the notion of step so that it is as machine-independent as possible. For the moment, let us adopt the following view. A constant amount of time is required to execute each line of our code.

In the following discussion, our expression for the running time of *insertionSort* will evolve from a messy formula that uses all the statement costs to a much simpler notation that is more concise and more easily manipulated. This simpler notation will also make it easy to determine whether one algorithm is more efficient than another.

To begin with, we introduce for the insertionSort procedure a time “cost” of each instruction and the number of their repetitions. Let be the number of condition checks in the while loop. If the for and while loops are completed normally (that is, when the condition specified in the loop header ceases to be met), the condition is checked for one more time than the loop body is executed.



This is the insertionSort implementation, but we have placed the C with the index and the number of repetitions how many times we pay this price.

The running time of the algorithm is the sum of running times for each statement executed; a statement that takes steps to execute and executes n times will contribute n to the total running time. To compute T(n) the running time of *insertionSort* on an input of n values, we sum the products of the *cost* and *times* columns, obtaining:

Even for inputs of a given size, an algorithm’s running time may depend on *which* input of that size is given. For example, in *insertionSort*, the best case occurs if the array is already sorted. For each we then find that in line 4 when has it’s initial value of . Thus for , and the best-case running time is:

We can express this running time as for *constants a* and *b* that depend on the statement costs . It is thus a ***linear function*** of n.

If the array is in reverse sorted order—that is, in decreasing order—the worst case results. We must compare each element with each element in the entire sorted subarray , and so for . The worst case, the running time of *insertionSort* is:

We can express this worst-case running time as for constants *a*, *b*, and *c* that again depend on the statement costs . It is thus a ***quadratic function*** of n.

In our analysis of insertion sort, we looked at both the best case, in which the input array was already sorted, and the worst case, in which the input array was reverse sorted. For the remainder of this book, though, we shall usually concentrate on finding only the worst-case running time, that is, the longest running time for any input of size n. We give three reasons for this orientation.

The worst-case running time of an algorithm gives us an upper bound on the running time for any input. Knowing it provides a guarantee that the algorithm will never take any longer. We need not make some educated guess about the running time and hope that it never gets much worse.

For some algorithms, the worst case occurs fairly often. For example, in searching a database for a particular piece of information, the searching algorithm’s worst case will often occur when the information is not present in the database. In some applications, searches for absent information may be frequent.

The “average case” is often roughly as bad as the worst case. Suppose that we randomly choose n numbers and apply insertion sort. How long does it take to determine where in subarray to insert element ? On average, half the elements in are less than , and half the elements are greater. On average, therefore, we check half of the , and so is about . The resulting average-case running time turns out to be a quadratic function of the input size, just like the worst-case running time.

In some particular cases, we shall be interested in the average-case running time of an algorithm; you could look for the technique of probabilistic analysis applied to various algorithms. The scope of average-case analysis is limited, because it may not be apparent what constitutes an “average” input for a particular problem. Often, we shall assume that all inputs of a given size are equally likely. In practice, this assumption may be violated, but we can sometimes use a randomized algorithm, which makes random choices, to allow a probabilistic analysis and yield an expected running time.

We used some simplifying abstractions to ease our analysis of the insertionSort procedure. First, we ignored the actual cost of each statement, using the constants to represent these costs. Then, we observed that even these constants give us more detail than we really need: we expressed the worst-case running time as for some constants a, b, and c that depend on the statement costs . We thus ignored not only the actual statement costs, but also the abstract costs .

We shall now make one more simplifying abstraction: it is the rate of growth, or order of growth, of the running time that really interests us. We therefore consider only the leading term of a formula (e.g., ), since the lower-order terms are relatively insignificant for large values of n. We also ignore the leading term’s constant coefficient, since constant factors are less significant than the rate of growth in determining computational efficiency for large inputs. For insertion sort, when we ignore the lower-order terms and the leading term’s constant coefficient, we are left with the factor of from the leading term. We write that insertion sort has a worst-case running time of

We usually consider one algorithm to be more efficient than another if its worst case running time has a lower order of growth. Due to constant factors and lower- order terms, an algorithm whose running time has a higher order of growth might take less time for small inputs than an algorithm whose running time has a lower order of growth. But for large enough inputs, a algorithm, for example, will run more quickly in the worst case than a algorithm.

We found that the worst-case running time of insertion sort is . Let us formalize what this notation means.

## asymptotic notation

Consider what theta is. We found our worst case 𝜃 (n2). For a given function g (n) - that is, for our algorithm. We denote 𝜃 (g (n)) as the set of the following functions when there is a set of positive constants C and n0 such that the inequality holds for every n greater than zero. Theta is an accurate estimate (it must be bounded at both the top and bottom). For example, an algorithm requiring Ω (n logn) requires at least n log(n) time, but the upper bound is not known. An algorithm requiring Θ (n logn) is preferable because it requires at least n log(n) (Ω (n log(n))) and at most n log(n) (O (n log(n))).

**For a given function we denote by the *set of functions :***

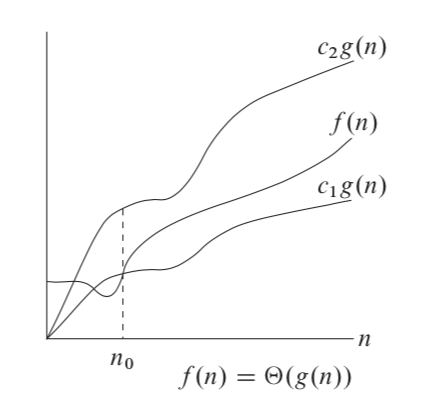


Figure 3

Theta notation is asymptotically tied to top and bottom functions, that is, it keeps it in a narrower framework. If we only have an upper bound, we use big O notation. For the functions g (n), a positive constant must be satisfied and the inequality holds for each n greater than 0. O large is most useful because it represents the worst case.

**For a given function we denote by O the *set of functions :***

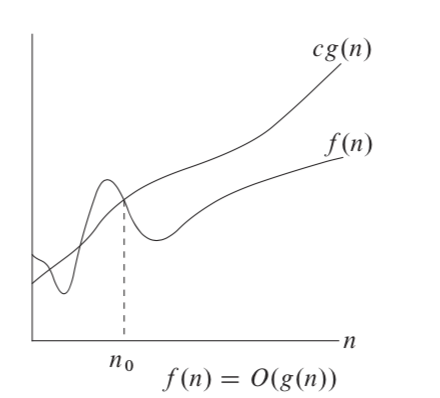


Figure 4

Omega notation. If big O is notation for the upper bound, then omega is for the lower one. It is defined in a similar way to big O, but we restrict the function from below. That is, we show the best implementation, for example, an already sorted collection in the input.

**For a given function we denote by the *set of functions :***

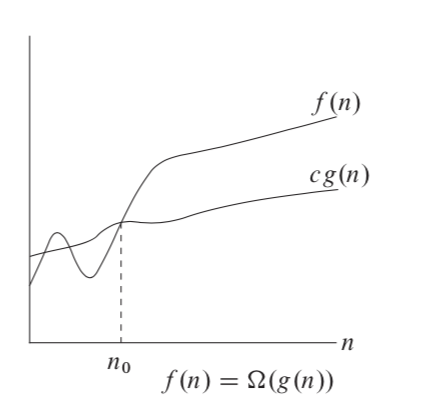


Figure 5

After that, you can already analyze your code, put down the execution time, etc. We've covered *insertionSort* using an incremental approach, i.e. each time we took the next element and considered for each sorted array. We remember that the quadratic option is not the best and let's see what else we can use.

# Divide and conquer

We can choose from a wide range of algorithm design techniques. For insertion sort, we used an incremental approach: having sorted the subarray A[1 .. j-1], we inserted the single element A[j] into its proper place, yielding the sorted subarray A[1 .. j].

Here we will examine an alternative design approach, known as “divide-and-conquer”. We’ll use divide-and-conquer to design a sorting algorithm whose worst-case running time is much less than that of insertion sort. One advantage of divide-and-conquer algorithms is that their running times are often easily determined.

Many useful algorithms are recursive in structure: to solve a given problem, they call themselves recursively one or more times to deal with closely related sub-problems. These algorithms typically follow a divide-and-conquer approach: they break the problem into several subproblems that are similar to the original problem but smaller in size, solve the subproblems recursively, and then combine these solutions to create a solution to the original problem.



Figure 6

The divide-and-conquer paradigm involves three steps at each level of the recursion:

* **Divide** the problem into a number of subproblems that are smaller instances of the same problem.
* **Conquer** the subproblems by solving them recursively. If the subproblem sizes are small enough, however, just solve the subproblems in a straightforward manner.
* **Combine** the solutions to the subproblems into the solution for the original problem.

The ***merge sort*** algorithm closely follows the divide-and-conquer paradigm. Intuitively, it operates as follows.

* **Divide:** Divide the n-element sequence to be sorted into two subsequences of n/2 elements each.
* **Conquer:** Sort the two subsequences recursively using merge sort.
* **Combine:** Merge the two sorted subsequences to produce the sorted answer



Figure 7

## merge sort



This is how mergeSort will look like. We take half the length of the array and divide it into two parts. And we call the merge function to the same sorting from the left and right sides. The merge function itself compares and merges our sequences.

Although the code for merge sort works correctly when the number of elements is not even, our recurrence-based analysis is simplified if we assume that the original problem size is a power of 2. Each divide step then yields two subsequences of size exactly n/2. This assumption does not affect the order of growth of the solution to the recurrence.

We reason as follows to set up the recurrence for T(n), the worst-case running time of merge sort on n numbers. Merge sort on just one element takes constant time. When we have n > 1 elements, we break down the running time as follows.

Divide: The divide step just computes the middle of the subarray, which takes constant time. Thus, D(n) = (1)

Conquer: We recursively solve two subproblems, each of size n/2, which contributes 2T(n/2) to the running time.

Combine: We have already noted that the MERGE procedure on an n-element subarray takes time (n), and so C(n)= (n).

When we add the functions D(n) and C(n) for the merge sort analysis, we are adding a function that is (n)and a function that is (1). This sum is a linear function of n, that is (n). Adding it to the 2T(n/2) term from the “conquer” step gives the recurrence for the worst-case running time T(n) of merge sort:

## Master method and theorem

The master method provides a “cookbook” method for solving recurrences of the form: , whereand are constants and is an asymptotically positive function.

According to the master theorem:

* If for some constant , then
* If then
* If for some constant , and if for some constant and all sufficiently large n,
* then

Let’s use the master method to solve the “divide and conquer” recurrence: . Here, we have a=2, b=2, \, and thus we have that: . Case 2 applies, since , and so we have the solution:

# heap. heap sort. proirity queue

The ***(binary) heap*** data structure is an array object that we can view as a nearly complete binary tree. Each node of the tree corresponds to an element of the array. The tree is completely filled on all levels except possibly the lowest, which is filled from the left up to a point. An array A that represents a heap is an object with two attributes: A.*length*, which (as usual) gives the number of elements in the array, and A.*heap*-*size*, which represents how many elements in the heap are stored within array A. That is, although A[1 .. A.*length]* may contain numbers, only the elements in A[1 .. A.*heap*-*size]*, where0 ≤ A.*heap*-*size ≤* A.l*ength*, are valid elements of the heap. The root of the tree is A[1], and given the index i of a node, we can easily compute the indices of its parent, left child, and right child:

* Parent: i/2
* Left: 2i
* Right: 2i+1

There are two kinds of binary heaps: max-heaps and min-heaps. In both kinds, the values in the nodes satisfy a ***heap property***, the specifics of which depend on the kind of heap. In a ***max-heap***, the ***max-heap property*** is that for every node i other than the root: A[Parent(i)] ≥ A[i]. Thus, the largest element in a max-heap is stored at the root, and the subtree rooted at a node contains values no larger than that contained at the node itself. A ***min-heap*** is organized in the opposite way; the ***min-heap property*** is that for every node i other than the root: A[Parent(i)] ≤ A[i]. The smallest element in a min-heap is at the root. For the heapsort algorithm, we use max-heaps. Min-heaps commonly implement priority queues.

## heap

What does this heap look like? We have an array that we can represent as a tree. The numbers above the element in the array and the heap are the same, you can see how they are substituted into the leaves.

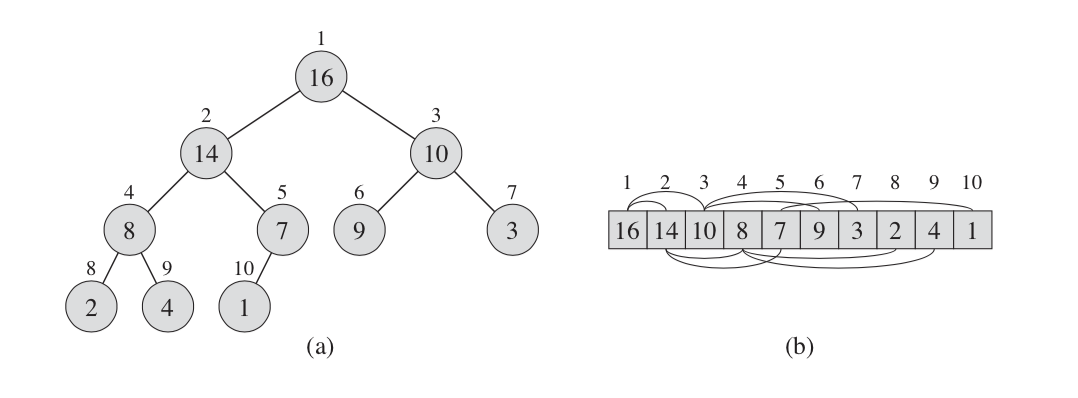


Figure 8

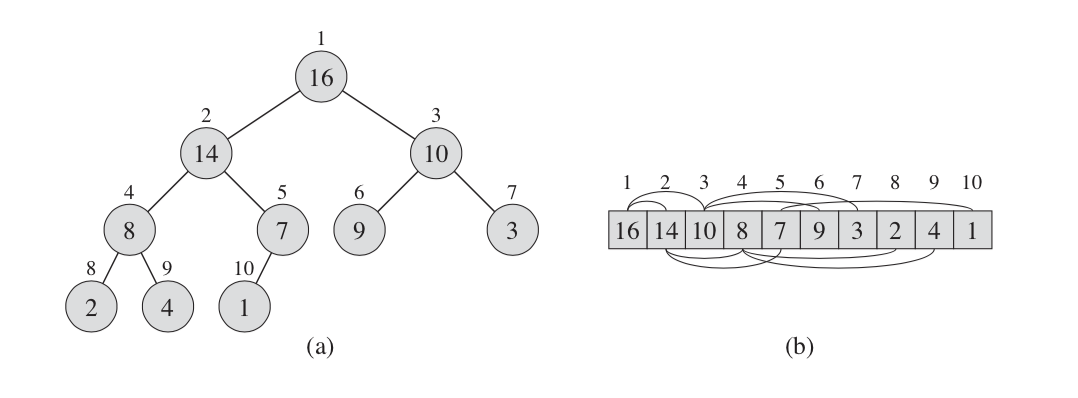


Figure 9

It doesn't have to be an array, different programming languages ​​have different data structures. For following example it is more convenient to consider it as an array.



Array A, which will represent our heap, also has attributes such as length - the number of elements in the array. And heap-size, the number of elements in the heap. The root of the tree is the first element in the array. We can easily calculate all the indexes of the nodes. The parent node or node is i / 2. The left child is 2i and the right is 2i + 1.

There are two types of our heaps. If max hip has the largest root node and the heap goes down, min has a minimum root node and goes up.

For heap sorting, max heap is usually used, and min heap is used for building a priority queue.

## Maintaining the heap property

In order to maintain the max-heap property, we call the procedure *maxHeapify*. When it is called, *maxHeapify* assumes that the binary trees rooted at LEFT(i) and RIGHT(i) are max-heaps, but that A[i] might be smaller than its children, thus violating the max-heap property. Procedure *maxHeapify* lets the value at A[i] “float down” in the max-heap so that the subtree rooted at index “I” obeys the max-heap property.

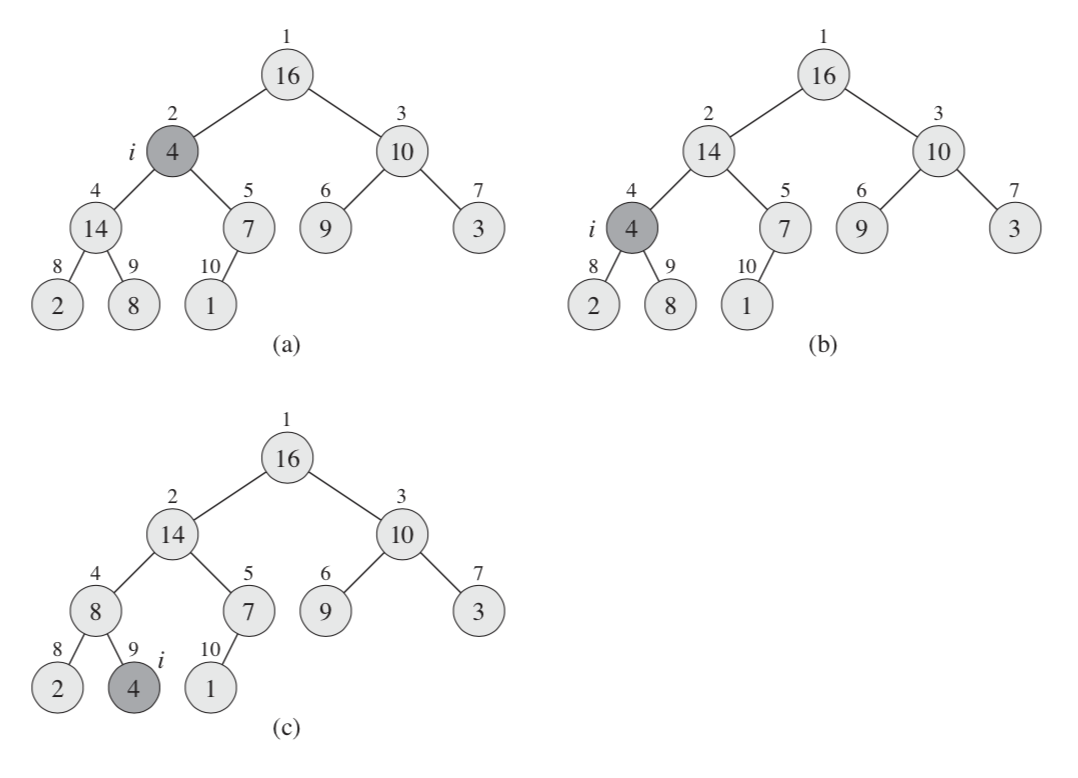


Figure 10

This heap must somehow support itself, for example, when inserting or when deleting elements a heap rebuild should take place. We have a procedure, let's call it maxHeapify. When it is called, we assume that the left and right nodes are also maxHeap. But these nodes can be smaller than its children and we have to let it go down below. As we can see, the second element has ceased to meet the requirement and we check the left heap until we reach the sheet.

At each step, the largest of the elements A[i], A[LEFT(i)], and A[RIGHT(i)] is determined, and its index is stored in *largest*. If A[i] is largest, then the subtree rooted at node “I” is already a max-heap and the procedure terminates. Otherwise, one of the two children has the largest element, and A[i] is swapped with A[*largest]*, which causes node i and its children to satisfy the max-heap property. The node indexed by *largest*, however, now has the original value A[i], and thus the subtree rooted at *largest* might violate the max-heap property. Consequently, we call *maxHeapify* recursively on that subtree.

The running time of *maxHeapify* on a subtree of size n rooted at a given node i is the time to fix up the relationships among the elements A[i], A[LEFT(i)], and A[RIGHT(i)], plus the time to run *maxHeapify* on a subtree rooted at one of the children of node i (assuming that the recursive call occurs). The children’s subtrees each have size at most 2n/3—the worst case occurs when the bottom level of the tree is exactly half full—and therefore we can describe the running time of *maxHeapify* by the recurrence:

The solution to this recurrence, by case 2 of the master theorem is .

And so, let's analyze the algorithm. We define the largest element of the three, then swap and so on, omit the element. For our first node, we have a Tetta from n. For a single node, then we split into under trees and go down under the trees. We have a maximum number of child sub-trees, in the worst case, 2n / 3. This is when we go from half to the very bottom.

Thus, the following time is obtained: . Turning to the master theorem, we get the result: Lg (6) = 2.6. This is a very good result, we need to spend less resources than our operations.

## building a heap

The heap should be built using method buildMaxHeap. To do this, you need to run the entire array through the maxHeapify function. As we see the complexity becomes n\*lg(n), we have a linear dependency - we apply a function for each element and call maxHeapify itself. We can use the procedure *maxHeapify* in a bottom-up manner to convert an array A[1 . . n], where n = A.*length*, into a max-heap. The elements in the subarray A[(n/2+1) .. n] are all leaves of the tree, and so each is a 1-element heap to begin with. The procedure buildMaxHeap goes through the remaining nodes of the tree and runs *maxHeapify* on each one.

We can compute a simple upper bound on the running time of buildMaxHeap:

* Each call to *maxHeapify* costs time.
* Procedure buildMaxHeap makes such calls.
* Thus, the running time is . This upper bound, though correct, is not asymptotically tight.

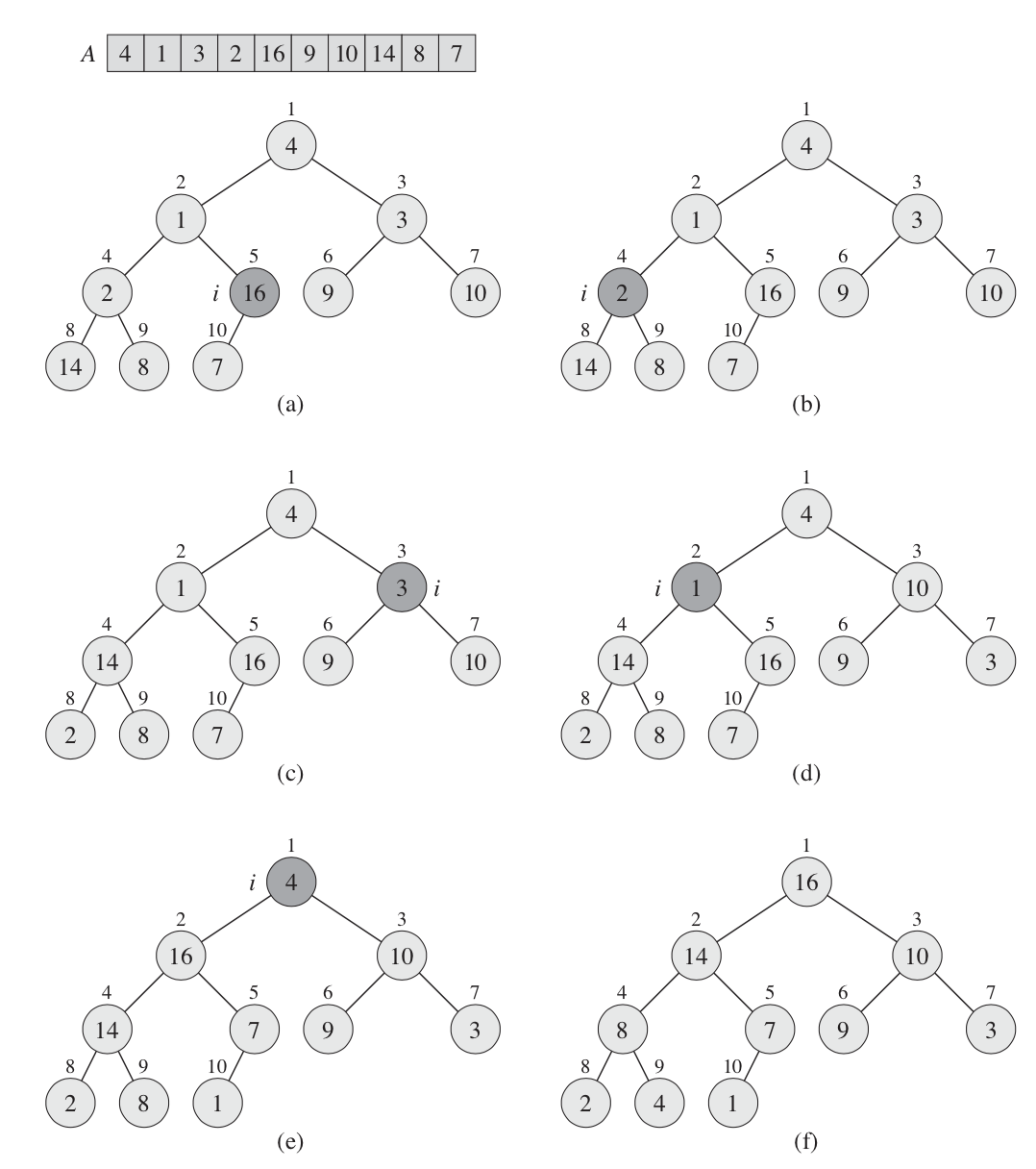


Figure 11

We have an unsorted array and we usually start in the middle. Each time the offset is made, the heap gets a complete look.

## heap sort

The heapsort algorithm starts by using *buildMaxHeap* to build a max-heap on the input array A[1..n], where n = A.*length*. Since the maximum element of the array is stored at the root A[1], we can put it into its correct final position by exchanging it with A[n]. If we now discard node n from the heap — and we can do so by simply decrementing A.*heap*-*size* — we observe that the children of the root remain max-heaps, but the new root element might violate the max-heap property. All we need to do is to restore the max-heap property, however, is call *maxHeapify*, which leaves a max-heap in A[1..n-1]. The heapsort algorithm then repeats this process for the max-heap of size n - 1 down to a heap of size 2.

The *heapsort* procedure takes time since the call to *buildMaxHeap takes time*  and each of the n - 1 calls to *maxHeapify* takes time

Based on this data structure, we can build a sort algorithm called “heap sort”. We have to build our heap from some dataset, and then starting from the first elements we put it in the first place and reduce the length with our heap, it remains in the array, but it is not part of the heap. Thus, each time we rebuild our heap, we will sort our array and each time we sort the smaller array. heapsort is thus executed for n\*lg(n).

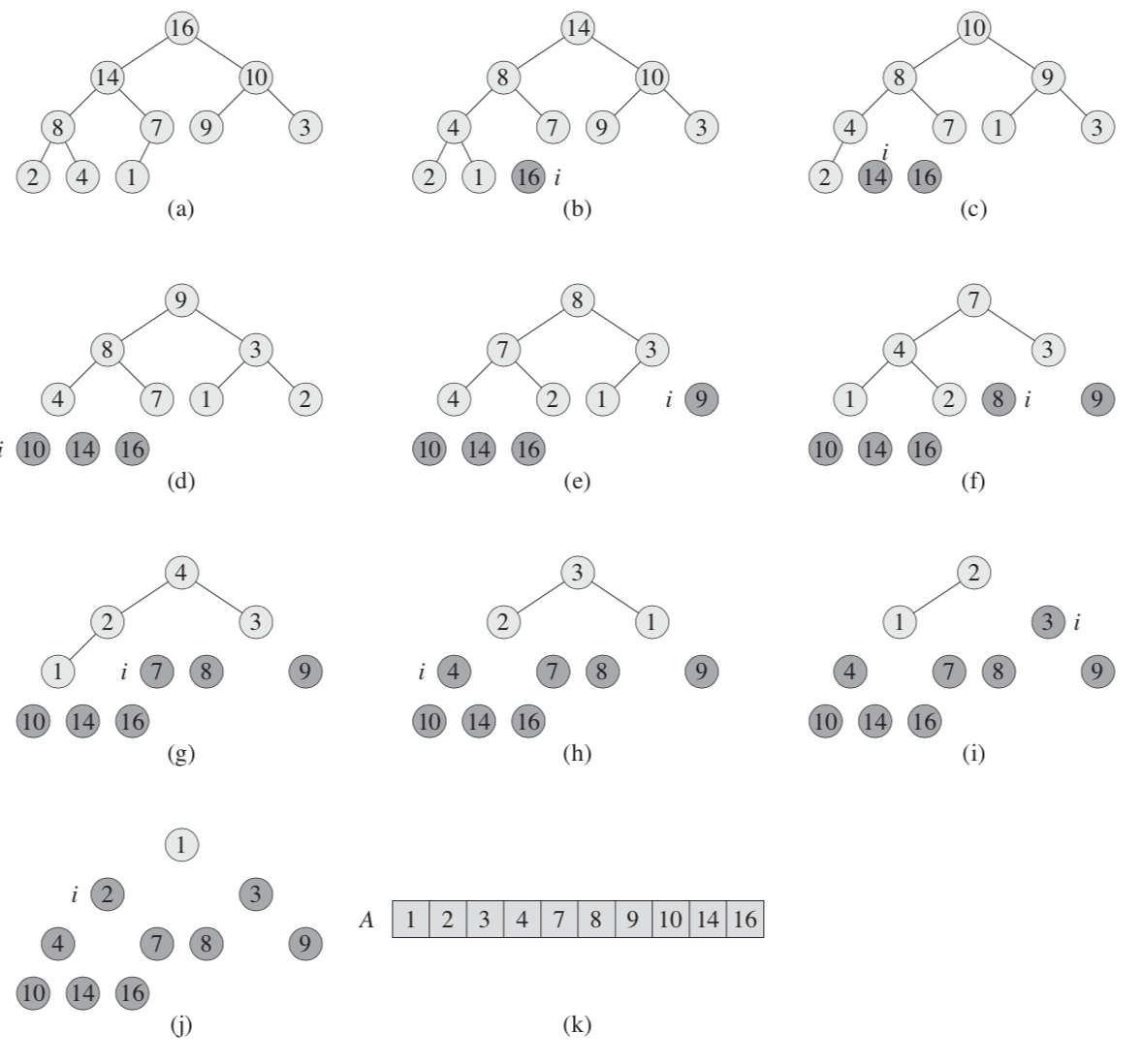


Figure 12

We take the first element and remove it from the heap when we put it first in the array. The tree is automatically rebuilt, now we take the first one again and put it in second place, and so on, until the heap ceases to exist and our array is completely sorted.

## priority queue

A ***priority queue*** is a data structure for maintaining a set S of elements, each with an associated value called a ***key***. A ***max-priority queue*** supports the following operations:

* INSERT(S, x) inserts the element x into the set S
* MAXIMUM(S) returns the element of S with the largest key.
* EXTRACT-MAX(S) removes and returns the element of S with the largest key.
* INCREASE-KEY(S,x,k) increases the value of element x’s key to the new value k, which is assumed to be at least as large as x’s current key value.

This is a very good sort, but in most cases it is not the most optimal sort. But for some reason we are considering it, and all because on the basis of the heap it is very good to build a queue with priority. This is how it is implemented in most programs. For example, where you can use this queue - schedule a job on the server or use an event-driven approach when modeling physical processes with great complexity. For example, an event occurs with a minimum priority, it is executed, the time is recalculated, inserted into the queue, and the event we need appears at the top. Also, there are queues with the highest priority or with the lowest. You can insert an element into it, you can take the maximum element, extract the maximum element, and you can change the priority of some element. For example, processor time, there is a queue of tasks that it executes in one clock cycle, it performs a task, the priority changes - we insert it into the queue and take it with the next priority.

# sorting in linear time

## counting sort

We do not compare the elements, we go through the entire array once and count the number of elements that is less than the given one. We are building another array where we will store temporary data. This array contains the indices that are contained in the original collection.

***Counting sort*** assumes that each of the n input elements is an integer in the range 0 to k, for some integer k. When k = O(n), the sort runs in 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[1 . . N], and thus A.*length =* n. We require two other arrays: the array B[1 . . n] holds the sorted output, and the array C[0 : : k] provides temporary working storage.

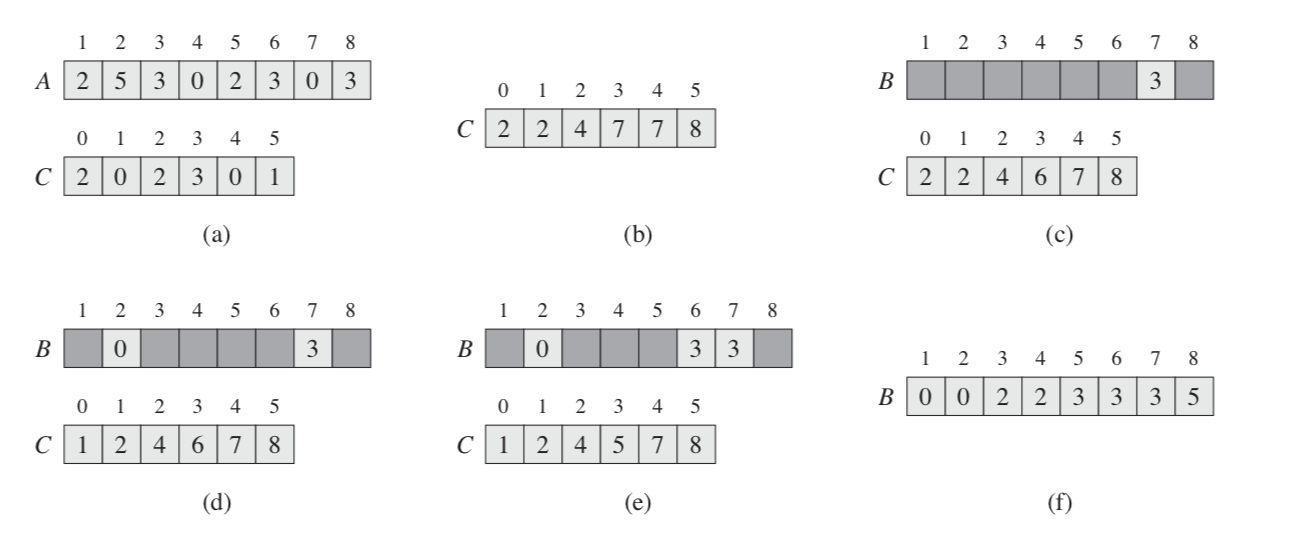


Figure 13

We take the maximum and minimum values ​​and create an array. Then we go through the original array, perceiving its elements as indices. And add from zero to the number of occurrences, then the number of elements is less than it.

 Thus, in the resulting array, we contain the indices and assign the value from the original array by indices. Thus, although not in one pass, but without nesting, we sorted it. We get a constant multiplied by n. And as it was said, the constant can be neglected at large values. A special case may be when we calculate not the number of elements that are greater than or equal, but the number of occurrences and then, by the number of occurrences, make up the resulting array. But there is one problem, our numbers are not always integers. It is very often necessary to sort the fractional ones. Very often you can write fractional numbers in some form from 0 to one.



Figure 14

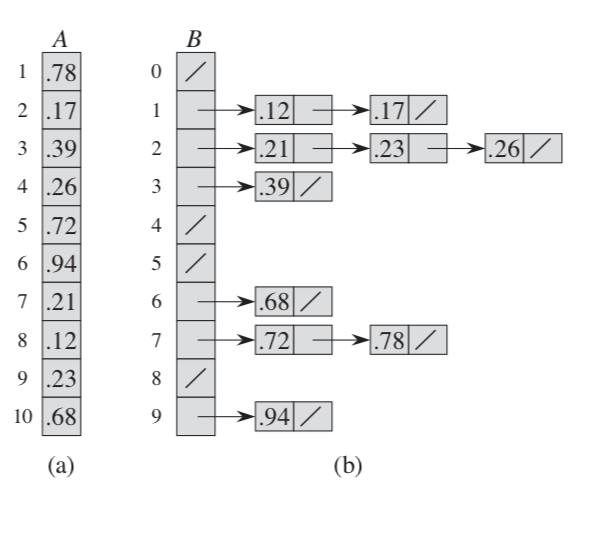
## bucket sort

We move on to the bucket sort. Here we do not add the number of elements, but we divide into some baskets, for example, from 0 to 0.1, from 0.1 to 0.2 to one. More often they hit a larger number, especially if there are patterns in the incoming data. They don't have to be uniform. We have an array of these buckets, each of which is also an array into which we will write our values.

***Bucket sort*** assumes that the input is drawn from a uniform distribution and has an average-case running time of O(n). Like counting sort, bucket sort is fast because it assumes something about the input. Whereas counting sort assumes that the input consists of integers in a small range, bucket sort assumes that the input is generated by a random process that distributes elements uniformly and independently over the interval [0; 1).

Bucket sort divides the interval [0; 1) into n equal-sized subintervals, or ***buckets***, and then distributes the n input numbers into the buckets. Since the inputs are uniformly and independently distributed over [0; 1), we do not expect many numbers to fall into each bucket. To produce the output, we simply sort the numbers in each bucket and then go through the buckets in order, listing the elements in each.

Our code for bucket sort assumes that the input is an n-element array A and that each element A[i] in the array satisfies 0 ≤ A[i] < 1. The code requires an auxiliary array B [0 . . n – 1] of linked lists (buckets) and assumes that there is a mechanism for maintaining such lists.



**Figure 15**

For example, we have values. We will add this to our buckets. Most often, we have information about the regularity of the input values and most often the sizes are selected so as to minimize the size of the bucket, contain 1-2 elements. Since there are few of them inside the bucket, we can ignore their sorting time. We simply merge the sorted buckets into an array.



The tricky part is that we have to sort the elements within the bucket. Even though we chose the size of the buckets very well, it so happens that there are a lot of input buckets. For example, we simulate a physical process, and we simulate several million molecules, since there are a lot of input data in buckets, there are also a large number of elements and, as you can see in the example, insertionSort is indicated in the sources, but it may not be able to cope with them. In those cases, you can use a data structure other than an array.



Figure 16