Searching and sorting with Java

Peter Sestoft, Department of Mathematics and Physics
Royal Veterinary and Agricultural University, Denmark

1995
English version 1.04, 2003-05-21

Searching is the process of looking for a particular element in a collection of data. For instance, one may look for a particular name in the phone directory to find the associated phone number.

Sorting is the process of arranging data according to some ordering. For instance, the entries of the phone directory are organized in increasing alphabetic order of the names (but not the phone numbers).

In these notes we present several ways to do searching and sorting. Such 'ways' or general prescriptions for how to proceed are called *algorithms*. We use the programming language Java to present them, but the algorithms themselves are independent of any language. To illustrate this way to look at programming, we show how one may study the correctness and efficiency of algorithms in a general way, regardless of what programming language, what computer, etc. one uses.

Contents

1 Searching

This section presents two ways to search a list of items.

1.1 Linear search

When searching an *unordered* collection of data for a particular element, one must go through it from one end to the other, looking at a single piece of data at a time. This is called *linear search*. For instance, assume I want to use the phone directory 'backwards', to locate the name associated with phone number 31 19 76 01. Then I must read the phone directory from one end, until I locate the number or until there are no more entries.

How much time does it take to perform a linear search? When measuring the amount of work required for searching or sorting, it is customary to count the number of *comparisons* between data items. Assume there are 1,000,000 entries in the phone directory. In the best case, the number I look for is the first one in the directory; in this case only one comparison is required. If I look for a number chosen at random from those in the directory, I will have to go through half the directory to find the number. Thus on the average, 500,000 comparisons are required. In the worst case, the number I look for is the very last one in the directory, or it is not in the directory at all; in these cases 1,000,000 comparisons are required.

More generally: Linear search in an unordered list collection with n elements requires $\frac{n}{2}$ comparisons in the *average case*, and n comparisons in the *worst case*.

1.2 Binary search

When searching an *ordered* collection of data, one can proceed in a considerably more efficient manner.

Example: Let us search for 62 in the list 11, 11, 28, 35, 45, 50, 62, 78, 79, 117, 117, 251, in which the elements occur in increasing order.

We start by inspecting an element in the middle of the list, such as 50. Since it is smaller than 62, and the list is ordered, all elements in the first half of the list must be smaller than 62 also. Henceforth we need to look only at the second half of the list: 62, 78, 79, 117, 117, 251.

Now we repeat the same procedure, inspecting an element in the middle of this remaining part of the list, such as 79. This is greater than 62, so now we need look only at the first half of this part of the list: 62, 78.

Again we inspect an element in the 'middle' of the remaining list, such as 62. But that is the element we were looking for, so the search succeeded.

This procedure is called *binary search*, because the list is bisected in every step of the algorithm: the number of remaining elements is halved for every comparison made.

Approximately the same procedure is used when we, as humans, look up somebody in the phone directory. First we decide which volume to look in. Secondly, we open that volume on some page, and decide whether the wanted name must appear before, on, or after that page. If it appears before or after, then we go to a page somewhat before (or after) the current page, and repeat this manoeuvre. If the name must appear on the current page, then we first decide which column the name must be in, and then we use binary search on that column. (In the case of the phone directory we also exploit our knowledge of the structure of the alphabet and the likely distribution of names. If we look for Mr. Zerksis, then we immediately take out the last volume and look towards the end of that).

How much time does it take to perform a binary search? Assume that I look for a name in an ordered list with 1,000,000 elements. To begin with, I look at the middle element. Either this is the one I was looking for, or else I now know what half of the list I must continue looking in; in either case there are at most 500,000 elements left. Say that I must continue with the first half. Then I look at the middle element of the first half. Again I either find the name immediately, or decide what half (of the first half) to continue with; in either case there are at most 250,000 elements left. This way I continue until the name has been found (or I conclude that it is not in the list).

In each step of my search — following each comparison — the number of remaining elements is halved: 1,000,000, 500,000, 250,000, 125,000, 62,500, 31,250, 15,625, 7,812, 3,906, 1,953, 976, 488, 244, 122, 61, 30, 15, 7, 3, 1, and we see that in just 20 steps we are down to 1 remaining name. Either this is just the one I was looking for, or else that name is not on the list.

In the worst case (and on average), binary search in a list of 1,000,000 elements requires 20 comparisons; that is, 25,000 times less than linear search! Note that this works only if the list is *sorted*. That in itself should justify looking at sorting later in these notes.

The base 2 logarithm $\log_2(n)$ says how many times one may halve a number n and still get a result which is greater than or equal to 1. The base 2 logarithm can be computed as follows:

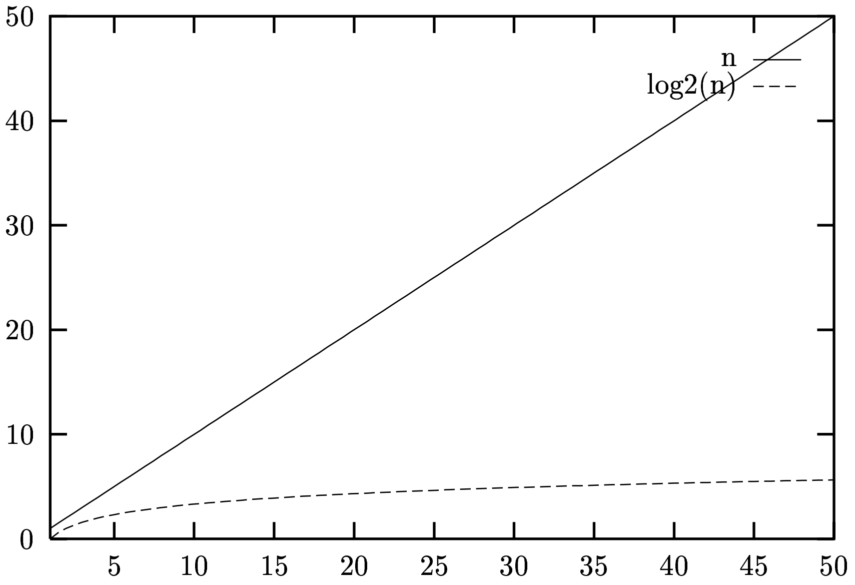
$$\log_2(n) = \frac{\log(n)}{\log(2)}$$

Since $\log_2(1,000,000) = 19.9$ we see that one needs to use approximately 20 comparisons to do binary search in a sorted list with 1,000,000 elements

More generally: Binary search in a sorted list with n elements requires on the average approximately $\log_2(n)$ comparisons, and the worst case is no worse than the average case.

1.3 Comparison of linear and binary search

The time required for linear search among n elements is proportional to n, whereas the time required for binary search is proportional to $\log(n)$. Since $\log(n)$ grows very slowly, binary search is *much* faster than linear search when n is large, as shown by the graph:



The graph shows the theoretical time consumption. The upper curve is for linear search; the lower one for binary search.

1.4 Common Java code for the search algorithms

Assume we have an array *arr* with n integers in the cells $0 \ldots n-1$. Below we present two Java methods to search such an array. The effect of the search is to modify the variables `i` and `found` (which have class scope):

```
static int i;
static boolean found;
```

If the search was successful (the looked-for number was found), then `found` is set to `true` and `i` is set to the index of the array cell (between 0 and $n-1$) which holds that number. If the search was unsuccessful, then `found` is set to `false`, and the value of `i` is immaterial.

1.5 Programming linear search

The linear search for x in $arr[0..(n-1)]$, where $n \geq 0$, can be done by this Java method:

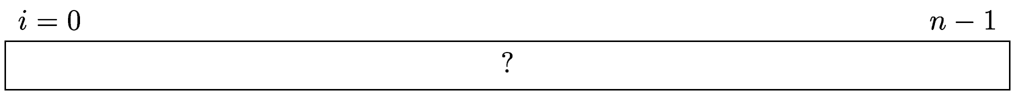
```
static void linsearch(int x, int[] arr, int n)
{
  i = 0; found = false;                /* pp1 */
  while (!found && i < n)
    {                                  /* pp2 */
      if (arr[i] != x) i++;
      else             found = true;   /* pp3 */
    }                                  /* pp4 */
}
```

When the `while` loop has terminated, it holds at program point pp4 that:

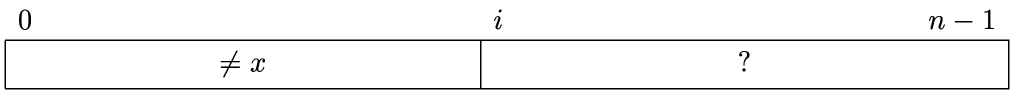
- if `found = true`, then $0 \leq i \leq n-1$ and $arr[i] = x$;
- if `found = false`, then x is not in $arr[0..(n-1)]$.

The loop can be understood by considering the state of the program's variables at the program points pp1, pp2, and pp3. We will use so-called *general snapshots* to describe the state of the program's variables.

To begin with, at pp1, we have $i = 0$ and we do not know anything about the array's elements:

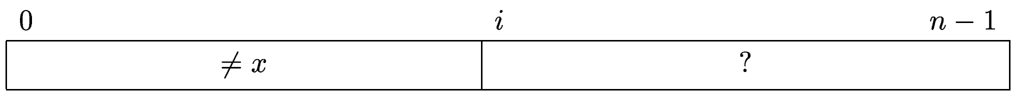


At pp2 it holds after i iterations of the loop that all elements of $arr[0..(i-1)]$ are different from x, and that we still need to look at $arr[i..(n-1)]$:

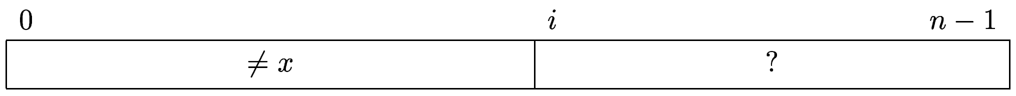


Namely, this is true from the beginning (after zero iterations) because $i = 0$. The sub-array $arr[0..(i-1)]$ is empty; it has no elements, and therefore no elements that equal x.

To show that the snapshot at pp2 continues to hold, we assume that it holds after i iterations, and show that it holds also after $i+1$ iterations. Thus assume that we are at pp2 and look at the array element at index i:



(1) If $arr[i] = x$, then we are done, and `found` is set to `true`, so the loop terminates: the search was successful.
(2) Otherwise it holds that $arr[i] \neq x$, and thus $arr[0..i] \neq x$. After we have incremented i by 1, it holds again at pp3 that all elements in $arr[0..(i-1)]$ are different from x:



If the loop continues, then the above snapshot holds again at pp2. Such a snapshot is called a *loop invariant*: it holds invariantly at entry to and exit from the body of the loop.

The `while` loop continues as long as the loop condition `(!found && i < n)` is true. When the loop terminates, the condition must have become false. Thus at least one of the terms `!found` and `i < n` must be false. If `found` is true, then it is clearly because $arr[i] = x$. If `found` is false, then `!found` is true, and therefore `i < n` must be false, so $i \geq n$. Together with the loop invariant, which says that x is not in $arr[0..(i-1)]$, we have that x is not in $arr[0..(n-1)]$.

1.6 Programming of binary search

Assume again that we want to search for x in the array $arr[0..(n-1)]$ with $n \geq 0$ elements, and that $arr[0..(n-1)]$ is ordered increasing. Then we can use binary search, implemented by this Java method:

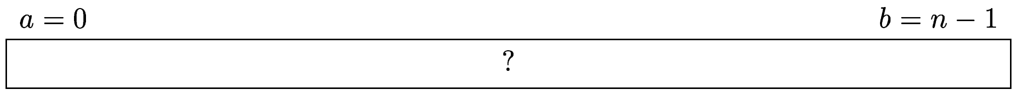
```
static void binsearch(int x, int[] arr, int n)
{
  int a = 0, b = n-1;
  found = false;                                    /* pp1 */
  while (!found && a <= b)
    {                                               /* pp2 */
      i = (a+b) / 2;
      if (x < arr[i]) b = i-1;
      else if (arr[i] < x) a = i+1;
      else found = true;                            /* pp3 */
    }                                               /* pp4 */
}
```

When the loop terminates, we have at pp4:

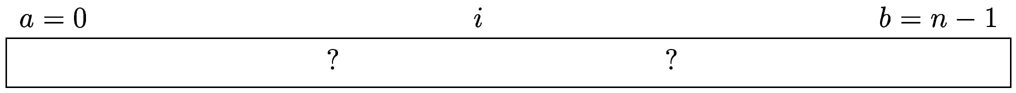
- if `found` is `true`, then $0 \leq i \leq n-1$ and $arr[i] = x$;
- if `found` is `false`, then x is not in $arr[0..(n-1)]$.

To fully understand the binary search method, we consider the program state at the program points `pp1`, `pp2`, and `pp3`.

To begin with, at `pp1`, we have $a = 0$ and $b = n - 1$, and we do not know anything about the elements of the array:

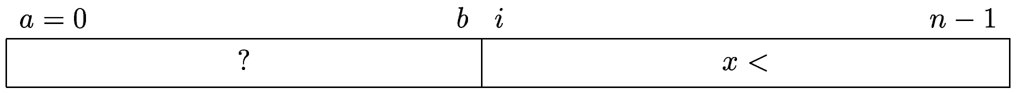


The first time we reach `pp2`, we consider the middle element (at index i):

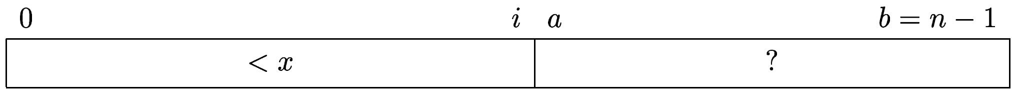


Now there are three cases:

Case (1) If $x < arr[i]$, then all elements to the right of i are greater than x, and we set b to $i - 1$:

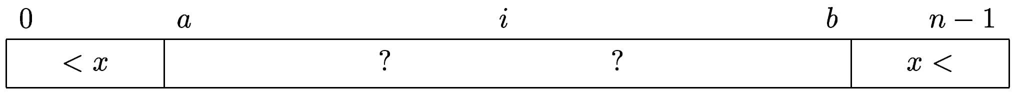


Case (2) If $arr[i] < x$, then all elements to the left of i are less than x, and we set a to $i + 1$:



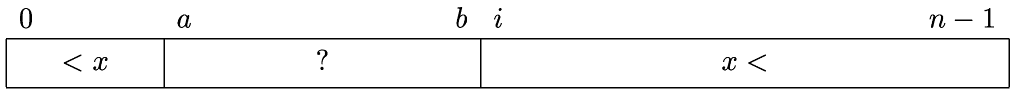
Case (3) If neither $x < arr[i]$ nor $arr[i] < x$, then $arr[i] = x$, and we set `found` to `true`, so that the loop terminates: the search was successful.

More generally, assume that we are at `pp2` after a number of iterations of the loop. Then the array has been divided into three sections: some elements less than x, some unknown elements in $arr[a..b]$, and some elements greater than x. The elements of $arr[a..b]$ are those we still need to search among:



Now either $a > b$, in which case the middle (unknown) section is empty; if so x is not in the array. Or else $a \leq b$ and the middle section is non-empty; if so we consider its middle element (at index i). Again there are three cases:

Case (1) If $x < arr[i]$, then all elements to the right of i are greater than x, and we set b to $i - 1$, so that we have at `pp3`:



Case (2) If $arr[i] < x$, then all elements to the left of i are less than x, and we set a to $i + 1$, so that we have at `pp3`:

0 i	a b	$n-1$
$< x$	?	$x <$

Case (3) If neither $x < arr[i]$ nor $arr[i] < x$, then $arr[i] = x$, so we set `found` to `true`, so that the loop terminates: the search was successful.

If the loop continues, then the above `pp3` will still hold when we reach `pp2` again. Thus the snapshot is a loop invariant.

At `pp4`, when the loop has terminated, the loop condition `(!found && a <= b)` must be false. Thus at least one of `!found` and `a <= b` must be false.

If `found` is `true`, then it is because $arr[i] = x$.

Conversely, if `found` is `false`, then `!found` is true, so `a <= b` must be false, and therefore $a > b$, so that $a - 1 \geq b$. We conclude that $arr[0..(a-1)]$ and $arr[(b+1)..(n-1)]$ cover all of $arr[0..(n-1)]$. Since the loop invariant says that all elements in $arr[0..(a-1)]$ are less than x, and that all elements in $arr[(b+1)..(n-1)]$ are greater than x, none of the two sections can contain x. Therefore $arr[0..(n-1)]$ does not contain x.

1.7 Exercises

1. Execute the linear search method (Section 1.5) manually with arr being the array below, and with $n = 19$:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
95	69	33	84	17	39	97	88	39	51	20	84	62	52	35	74	28	57	86

   Show the values of `i` and `found` at `pp3` in each iteration of the loop when searching for $x = 84$.

   Show the values of `i` and `found` at `pp3` in each iteration of the loop when searching for $x = 85$.

2. Execute the binary search program (Section 1.6) manually with arr being the sorted array below, and with $n = 19$:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
17	20	28	33	35	39	39	51	52	57	62	69	74	84	84	86	88	95	97

   Show the values of `a`, `b`, `i`, and `found` before the loop, at `pp3` in each iteration of the loop, and after the loop, when searching for $x = 84$.

   Show the values of `a`, `b`, `i`, and `found` before the loop, at `pp3` in each iteration of the loop, and after the loop, when searching for $x = 85$.

3. Do the programs for linear and binary search produce the right result when the array is empty ($n = 0$)?

4. Write a Java program to test the linear and binary search methods on the arrays shown above. The program may print 'Found at index i' if the element is in the array, and 'The element is not in the array' otherwise.

   The arrays may be declared and initialized as follows in Java:

```
int[] arr1 = {95, 69, 33, 84, 17, 39, 97, 88, 39, 51, 20, 84,
              62, 52, 35, 74, 28, 57, 86};

int[] arr2 = {17, 20, 28, 33, 35, 39, 39, 51, 52, 57, 62, 69,
              74, 84, 84, 86, 88, 95, 97};
```

2 Good programming style

Above we let the methods return their results via the variables `i` and `found` which are 'global': they have class scope. In a large program this anarchic programming style would soon lead to chaos. For instance, it it easy (and tempting) to use the variable `i` for some other purpose, or to forget to test the value of `found` before using the value of `i`.

It is safer to let the methods `linsearch` and `binsearch` return the result of the search. One possibility is to return the the index $i \in \{0, \ldots, n-1\}$ when the search is successful, and return -1 (that is, minus one) when the search fails: when the element we seek is not in the array. We can use the number -1 to represent failure because it cannot be a legal index into a Java array.

Thus we may write `binsearch` as follows:

```
static int binsearch(int x, int[] arr, int n)
{
  int a = 0, b = n-1;                              /* pp1 */
  while (a <= b)
    {                                              /* pp2 */
      int i = (a+b) / 2;
      if (x < arr[i]) b = i-1;
      else if (arr[i] < x) a = i+1;
      else return i;                               /* pp3 */
    }                                              /* pp4 */
  return -1;
}
```

Note that i is returned from inside the while loop directly when we find that the search was successful, thus terminating the loop. We do not need to set `found` to `true` to achieve this effect, so the variable `found` can be dispensed with. If the loop terminates for any other reason, it must be because the sought element x is not in $arr[0..(n-1)]$. In this case we return -1 to indicate that the search failed.

Now assume the following variable declaration:

```
  int[] myarray;
```

When using the method `binsearch` (or `linsearch`) one must check that the index returned is a legal non-negative array index, before one uses it:

```
int r = binsearch(key, myarray, myarray.length);
if (r >= 0)
  System.out.println("Found at index " + r + "\n");
else
  System.out.println("Not found\n");
```

This programming style is better. When there are fewer 'global' variables, the program becomes easier to modify and maintain. When one consistently checks whether the search is successful, the program program becomes more robust.

3 Sorting

In the following chapters we present three sorting algorithms: selection sort, Quicksort, and heap sort, and we study their properties.

3.1 Why sorting?

Sorting has a wide range of applications. Some examples:

- We have a list of items (e.g., books in a library), and want to locate a particular item (a book) in the list. If the list is sorted, then one can use binary search; otherwise, one must use linear search. As shown above, binary search is by far the fastest. For example, the phone directory is sorted by name, so that one can locate e.g. Mr. Smithson quickly, or decide that he is not in the phone directory at all. One the other hand, the phone directory is not sorted by phone number, so it is cumbersome (slow) to find out who has phone number 31 19 76 01, or to decide that nobody in the phone directory has that number.
- We have a list of items and want to check whether any item appears twice in the list. If the list is sorted, then all instances of the same item will be adjacent in the list. This may be used to check (in the Department of Motor Vehicles) that no license plate has been issued twice.
- We have two lists of items and want to check whether any item appears on both of them. If the lists are sorted (by the same ordering criterion), then it suffices to go through both lists sequentially, just once. This may be used to check whether anybody receives unemployment benefits and works for the government at the same time.
- We have a list of values, and want to find the ten least values in the list. This may be done by sorting the list in order of increasing values, and extract the ten first elements of the list.

Hence we must accept that sorting is useful. But why consider three different ways to sort; wouldn't it suffice to consider just one way? No: by considering several different ways, we see that different solutions to exactly the same problem may have very different properties, in terms of comprehensibility, speed of execution, and predictability of execution time for different inputs.

4 Selection sort

Selection sort is easy to understand and implement. However, it is extremely slow when the number of elements to sort is large (that is, larger than 20).

4.1 The basic idea in selection sort

In selection sort, one constructs the sorted list from one end, as follows:

Find the least element in the given list and remove it from the list. The sorted (partial) result now consists of just this element

Find the second least element (which must be the least of the remaining elements), remove it from the given list, and append it to the sorted (partial) result.

Find the third least element (which must be the least of the remaining elements), remove it from the given list, and append it to the sorted (partial) result.

One proceeds in this way until the given list is empty.

When sorting an array, one may construct the sorted (partial) result from left to right in the array, as shown in the example below:

0	1	2	3	4	5
35	62	28	50	11	45
11	62	28	50	35	45
11	28	62	50	35	45
11	28	35	62	50	45
11	28	35	45	62	50
11	28	35	45	50	62
11	28	35	45	50	62

The first line shows the unsorted array.

In the second line, we have found the least element (11) and have moved it to cell 0 by swapping it with 35. The sorted part of the array now is $arr[0..0]$.

In the third line, we have found the second least element (28) and moved it to cell 1 by swapping it with 62. The sorted part of the array now is $arr[0..1]$.

We proceed this way until the sorted part comprises the entire array $arr[0..(n-1)]$.

4.2 The efficiency of selection sort

Selection sort spends most of the time looking for the least element among those not yet sorted.

First it must find the least among n elements, then the least among $n-1$ elements, then the least among $n-2$ elements, and so on, until only one element remains.

To find the least element among m elements requires $m-1$ comparisons. Namely, if $m=1$, then there is only one element, which is obviously the least one; it requires 0 comparisons to establish that. If we can find the least among k elements by using $k-1$ comparisons, then we can find the least among $k+1$ elements by using k comparisons: $k-1$ comparisons to find the least among the first k elements, and 1 comparison to decide whether the last element is less than the least one of the others. In all, $(k-1)+1=k$ comparisons. By induction we have shown that the least among m elements can be found by using $m-1$ comparisons.

The total number of comparisons required for sorting n elements thus is

$$T(n) = \sum_{m=1}^{n} (m-1) = (n-1) + (n-2) + \cdots + 1 + 0 = \frac{n(n-1)}{2} = \frac{1}{2}n^2 - \frac{1}{2}n$$

The term $\frac{1}{2}n^2$ dominates for large n, so the number of comparisons is approximately proportional to n^2 for large n. We say that the time consumption for sorting n elements using selection sort is *asymptotically proportional* to n^2.

Together with results from later sections, this shows that selection sort is slow compared to other sorting algorithms. The basic reason for this is the inefficient way in which we find the least element among the remaining ones. If we improve just that part of the algorithm, we obtain another sorting algorithm which is surprisingly good; see Section 6 on heap sort below.

4.3 Auxiliary declarations for sorting algorithms

Assume that we have an array $arr[0..(n-1)]$ with n elements that we want to sort. For simplicity we shall sort arrays of integers (the Java basic type `int`). Clearly one may sort more interesting types of data; we shall see how later.

Very often we need to swap two elements of arr at given indexes s and t. Therefore it is useful to declare a method `swap`, so that the call `swap(`arr`, `s`, `t`)` will swap $arr[s]$ and $arr[t]$.

```
static void swap(int[] arr, int s, int t)
{
  int tmp = arr[s];  arr[s] = arr[t];  arr[t] = tmp;
}
```

Note that the method uses a local auxiliary variable `tmp`.

4.4 Programming selection sort

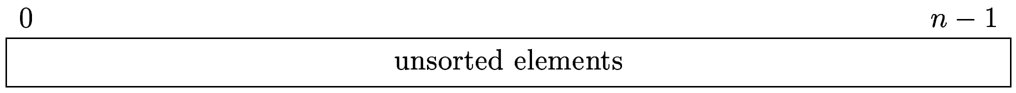
The goal is to sort the values in the sub-array $arr[0..(n-1)]$, where $0 \leq n$. When $n = 0$, the sub-array $arr[0..(n-1)]$ is empty.

Selection sort can be programmed using two nested `for` loops. Every iteration of the outer loop adds one more element to the sorted part of the array. The inner loop finds the least element among those not yet in the sorted part of the array.

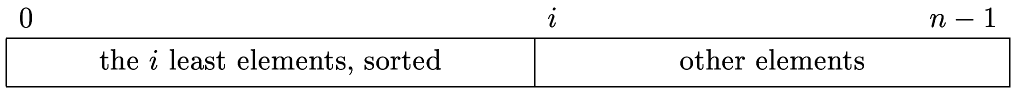
```
public static void selsort(int[] arr, int n)
{                                                  /* pp1 */
  for (int i = 0; i < n; i++)
    {                                              /* pp2 */
      int least = i;
      for (int j = i+1; j < n; j++)
        {
          if (arr[j] < arr[least])
            least = j;
        }
      swap(arr, i, least);                         /* pp3 */
    }                                              /* pp4 */
}
```

4.5 Snapshots for the outer loop

At `pp1` it holds that



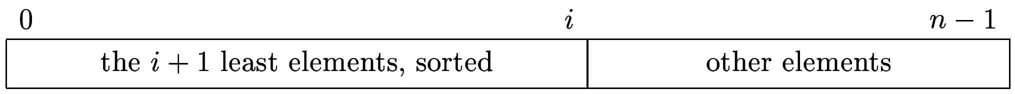
We shall see that at `pp2` it holds after i iterations



Namely, this holds to begin with (after zero iterations), because $i = 0$, and it is true that the empty sub-array $arr[0..(i-1)]$ contains $i = 0$ sorted elements.

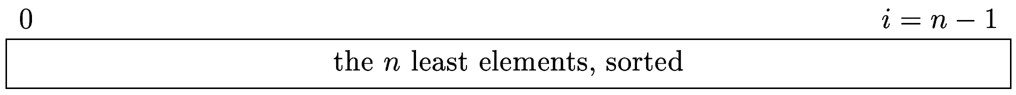
As we shall see in the next subsection, the inner loop finds an index $i \leq \mathit{least} < n$, so that $arr[\mathit{least}]$ is least among $arr[i..(n-1)]$. We will write this as $arr[\mathit{least}] \leq arr[i..(n-1)]$. Because $arr[0..(i-1)]$ already contains the i least elements, the swapping `swap(arr, i, least)` of $arr[i]$ and $arr[\mathit{least}]$ will ensure that $arr[0..i]$ contains the $i+1$ least elements. Furthermore, $arr[0..(i-1)] \leq arr[\mathit{least}]$ holds before the swapping because of `pp2`, and therefore $arr[0..(i-1)] \leq arr[i]$ after the swapping, so that $arr[0..i]$ remains sorted. Finally, the swapping makes sure, in case that $\mathit{least} > i$, that the original element $arr[i]$ is not just thrown away (overwritten), but gets moved to 'other elements'.

All in all it holds at `pp3`:



Now, if the outer loop continues, then i is increased by 1 before we reach `pp2` again, and the array is unmodified, so the snapshot at `pp2` holds again. We see that the snapshot at `pp2` holds in the first iteration, and that if it holds in the i'th iteration, then it will hold also in the $(i+1)$'th iteration. Therefore it holds in every iteration. It follows that `pp3` holds in every iteration too. Thus the snapshot at `pp3` is a loop invariant.

In the last iteration of the outer loop we have $i = n-1$, and the loop invariant at `pp3` continues to hold, so at `pp4` after the loop it holds that:



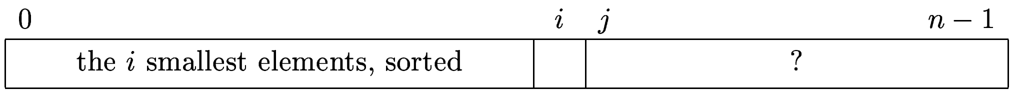
That is, when the outer loop terminates, the array contains precisely the sorted result.

4.6 Snapshots for the inner loop

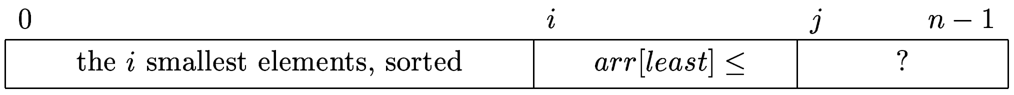
Now let us focus on the inner loop and the associated program points. We want to show that it finds the index *least* of the least element in $arr[i..(n-1)]$:

```
int least = i;                              /* pp21 */
for (int j = i+1; j < n; j++)
  {                                         /* pp22 */
    if (arr[j] < arr[least])                /* pp23 */
      least = j;
  }                                         /* pp24 */
```

At pp21 it holds — because of the assignment statement — that $least = i$ and hence $arr[least] = arr[i]$:



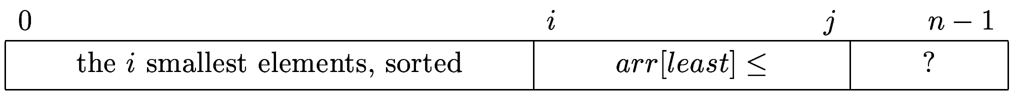
At pp22 it holds that $arr[least]$ is less than or equal to all elements in $arr[i..(j-1)]$, which we will write as $arr[least] \le arr[i..(j-1)]$:



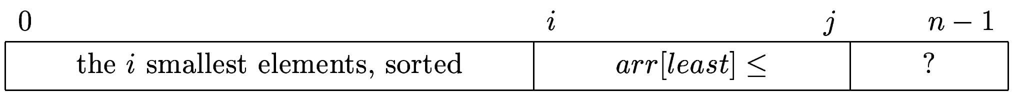
In particular, this holds in the first iteration, because $j = i+1$ and hence $arr[i..(j-1)] = arr[i]$.

In general, there are two possibilities:

(1) $arr[j] < arr[least]$. Together with $arr[least] \le arr[i..(j-1)]$ from pp22 this shows that $arr[j] < arr[i..(j-1)]$, and since $arr[j] \le arr[j]$ by definition, we have that $arr[j] \le arr[i..j]$. Putting $least$ equal to j, it holds at pp23 that $arr[least] \le arr[i..j]$:

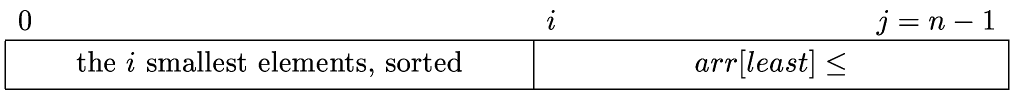


(2) Otherwise, $arr[least] \le arr[j]$. Together with $arr[least] \le arr[i..(j-1)]$ from pp22 this shows that $arr[least] \le arr[i..j]$, so at pp23 we have:



In both cases the snapshot holds at pp23. If the loop continues, then j is incremented by 1 before we reach pp22 again, so if pp23 holds in this iteration, then pp22 will hold in the next iteration, and since pp22 implies pp23, then pp23 holds in the next iteration too. Now pp22 holds in the first iteration, so pp22 and pp23 will hold in all iterations: they are loop invariants.

In the last iteration of the inner loop, $j = n-1$ and together with the loop invariant pp23 this shows that $arr[least] \le arr[i..(n-1)]$ at pp24 as required:



These considerations show that the inner loop works as expected: it finds an index $i \le least \le n-1$ so that $arr[least] \le arr[i..(n-1)]$. That is, it finds the least of the yet unsorted elements.

4.7 Example 1: Sorting lines of text

Assume we have a file containing a number of lines of text, for instance a list of addresses:

```
Karl-Erik                Lillegade 15
Børge                    Lillegade 9
Peter                    Ørnebakken 40
Anna                     Lillegade 8
Dines                    Fredsvej 11
Neil                     Bukkeballevej 88
Jørgen                   Skjoldagervej 20
Christian                Bondehavevej 135
Peter                    Begoniavej 20
```

We want a program to read in this list from a file, sort it alphabetically, and print out the sorted list. There are three distinct tasks: reading, sorting, and writing.

To begin with, we must modify the method `swap` so that it can swap strings (of type `String`) instead of integers (of type `int`):

```
private static void swap(String[] arr, int s, int t)
{
  String tmp = arr[s];  arr[s] = arr[t];  arr[t] = tmp;
}
```

Reading the address list from a file is the first subproblem, which can be solved by this method `readfile`:

```
public static int readfile(String[] arr, String filename)
     throws FileNotFoundException, IOException
{
  int n = 0;
  Reader inp = new FileReader(filename);
  StreamTokenizer tstream = new StreamTokenizer(inp);
  tstream.wordChars(' ', ' ');
  tstream.parseNumbers();
  tstream.nextToken();
  while (n < arr.length && tstream.ttype != StreamTokenizer.TT_EOF)
    {
      arr[n] = tstream.sval; tstream.nextToken();
      n++;
    }
  return n;
}
```

Calling the method with `readfile(arr, "addrlist.txt")` will open the file `addrlist.txt` and read lines from the file, storing them in `arr[0]`, `arr[1]`, and so on. When there are no more lines in the file, or no more room in the array, it will return the number of lines actually read from the file.

Sorting is done by the method `selsort`, which must be modified also. The ordering '<' cannot be used on strings in Java. Instead we use the string method `compareTo`, which compares two strings and returns -1, 0, or 1, according as the first string is lexicographically smaller than, equal to, or larger than the second string:

```
public static void selsort(String[] arr, int n)
{                                             /* pp1 */
  for (int i = 0; i < n; i++)
    {                                         /* pp2 */
      int least = i;
      for (int j = i+1; j < n; j++)
        {
          if (arr[j].compareTo(arr[least]) < 0)
            least = j;
        }
      swap(arr, i, least);                    /* pp3 */
    }                                         /* pp4 */
}
```

Printing can be done by yet another method `printout`, which takes as arguments the array and the number of elements to print:

```
public static void printout(String[] arr, int n)
{
  for (int i=0; i < n; i++)
    System.out.println(arr[i]);
}
```

The main method simply invokes the above-mentioned methods:

```
public static void main(String[] args)
     throws FileNotFoundException, IOException
{
  String[] lines = new String[100];
  int n = readfile(lines, "addrlist.txt");
  selsort(lines, n);
  printout(lines, n);
}
```

Before the call to `readfile`, *lines* is an array with room for 100 strings. After the call, some strings have been read into the first n elements, that is, in $lines[0..(n-1)]$. The call to `selsort` sorts these elements, and the call to `printout` prints them on the display.

4.8 Example 2: Sorting records

Assume we have a list of people, giving for each person the date of birth and the name. The date of birth is given in the eight-digit format yyyymmdd, where yyyy represents the year, mm represents the month (01-12), and dd the date (01-31). The list is given unsorted on a text file, e.g.:

```
19640627 Carsten
19470206 Niels
19031001 Edith
19660106 Carsten
19070206 Ingeborg
19360114 Kirsten
19360630 Henrik
19551202 Harald
19340930 Jørgen
19641209 Hanne
```

We must read, sort, and print out this list, and as before we have three distinct subproblems.

But first we must design the data structures. In a Java program we can represent each person as an object of a simple class with two fields, `date` and `name`:

```
class Person {
  int date;
  String name;
}
```

A variable of type `int` can hold a nine-digit number, and therefore suffices for storing the date.

As usual, we need a version of the `swap` method, to swap two elements of a Person array:

```
private static void swap(Person[] arr, int s, int t)
{
  Person tmp = arr[s];  arr[s] = arr[t];  arr[t] = tmp;
}
```

Reading the file. A method in the style of `readfile` from above can be used here as well, but now we need to read a date as well as a name from the same line of the text file. It can be done like this:

```
public static int readfile(Person[] arr, String filename)
     throws FileNotFoundException, IOException
{
  int n = 0;
  Reader inp = new FileReader(filename);
  StreamTokenizer tstream = new StreamTokenizer(inp);

  tstream.parseNumbers();
  tstream.nextToken();
  while (n < arr.length && tstream.ttype != StreamTokenizer.TT_EOF)
    {
      arr[n].date = (int)tstream.nval; tstream.nextToken();
      arr[n].name = tstream.sval;      tstream.nextToken();
      n++;
    }
  return n;
}
```

For each line read from the file, the date and name are stored in the fields *date* and *name* of the `Person` object *arr*[*n*].

Sorting is done by method `selsort`, but we must adapt it for the new `Person`, and we must decide what ordering to use: should we sort by date or by name?

If we want to sort by date, then the condition in `selsort` must be:

```
if (arr[j].date < arr[least].date)
```

Printing should display *arr*[*i*].*date* as well as *arr*[*i*].*name* for every record, and follows the outline given by `printout` above.

The main method just invokes the three methods corresponding to the subproblems:

```
public static void main(String[] args)
     throws FileNotFoundException, IOException
{
  Person[] people = new Person[100];
  for (int i=0; i<people.length; i++)
    people[i] = new Person();
  int n = readfile(people, "birthday.txt");
  selsort(people, n);
  printout(people, n);
}
```

4.9 Parametrizable sorting routines

It is cumbersome to have many different versions of the sorting methods, each corresponding to one particular type of elements to be sorted, and one particular ordering relation. An object-oriented solution to this problem is to define a class `Ordered` of ordered values, which must have a method `less` for comparing two values. Then one may define general methods, able to sort arrays of elements of type `Ordered`. It looks a bit complicated, so we postpone this till Section 8.

4.10 Exercises

1. Manually execute selection sort (Section 4.4) on this array:

0	1	2	3	4	5
35	62	28	50	11	45

   Show the values of i and $least$, and the contents of the array, in each iteration of the outer loop.
2. Selection sort would work correctly also if the outer loop had read

```
for (int i=0; i < n-1; i++) ...
```

   Why? Give an informal explanation. Then give an explanation using the snapshot at `pp3`.
3. Selection sort would work correctly if the conditional in the inner loop had been

```
if (arr[j] <= arr[least]) least = j;
```

   Why? Give an informal explanation. Then give an explanation in terms of the snapshots at `pp22` and `pp23`. Is there a reason for preferring one form of the conditional over the other in `selsort`?
4. In Java one may fill an array $arr[0..(n-1)]$ with pseudo-random integers between 0 and 29999 by using the library method `Math.random()` as shown below.

```
public static int[] fillarray(int n)
{
  int [] arr = new int[n];
  for (int i = 0; i < n; i++)
    arr[i] = (int) (Math.random() * 30000);
  return arr;
}
```

   Use this method to fill an array with 0, 50, and 100 random numbers. Sort them using selection sort, then print them, to see whether selection sort works.
5. Remove the printout from the program constructed above. Run it, using a wristwatch or similar to measure the time to sort n =1000, 2000, 3000, 4000, 5000, 6000, ... random numbers by selection sort. Tabulate the execution time as a function of n.
6. Complete the program from Section 4.7, and run it on the input file shown there.
7. Complete the program from Section 4.8, and run it on the file shown there.
8. Modify the program from Section 4.8 so that it sorts by age (earliest date of birth first).
9. Modify the program from Section 4.8 so that it sorts by birthday (day of the year) instead. That is, people born in January should precede those born in February, and so on. (Hint: `m % 10000` gives the last four digits of the integer `m`).

5 Quicksort

Quicksort usually is a very fast sorting algorithm, much faster than selection sort, except for arrays with few (< 20) elements. However, for certain data sets, Quicksort may be just as slow as selection sort — but such data sets are rare. Quicksort was invented by C.A.R. Hoare in 1962.

5.1 The basic idea in Quicksort

Quicksort works in two steps. First it divides the data into two parts: the ‘small’ elements and the ‘large’ elements. Second, it sorts each of the two parts separately, and combine them to a sorted whole.

More precisely: To sort the array $arr[0..(n-1)]$, choose an element x (usually from the middle of the array). This element is called the *pivot*. Now do the *partitioning*: Move the elements of the array so that all elements less than or equal to x move to the left, and all elements greater than or equal to x move to the right:

elements $\leq x$	x	elements $\geq x$

Now sort the two parts of the array separately. When the two parts have been sorted, then the entire array has been sorted also.

Quicksort solves the sorting problem by ‘divide and conquer’: a problem is solved by splitting it into several smaller subproblems that may be solved separately, and whose sub-solutions may be combined to a solution to the original problem. Very small problems have trivial solutions: an array with 0 or 1 element is sorted already and requires no processing at all.

The partitioning is most easily done by traversing the array from the left and from the right, until one finds a left-hand element $\geq x$ and a right-hand element $\leq x$. Then these elements are swapped so that they end up in the proper place relative to the pivot x. This procedure is repeated until all elements have been compared with x.

Here is an example, with the pivot underlined at every step:

0	1	2	3	4	5
35	62	$\underline{28}$	50	11	45
11	62	$\underline{28}$	50	35	45
11	$\underline{28}$	62	50	35	45
11	28	62	$\underline{50}$	35	45
11	28	45	$\underline{50}$	35	62
11	28	45	35	$\underline{50}$	62
11	28	$\underline{45}$	35	50	62
11	28	35	45	50	62
11	28	35	45	50	62

Note that the pivot may be moved around during the partitioning.

5.2 The efficiency of Quicksort

How many comparisons are needed to sort n elements using Quicksort? First, note that the partitioning phase always requires n comparisons, since all elements must be compared with the pivot x.

The total time consumption crucially depends on the partitioning, which in turn depends on the choice of pivot.

In the worst case the pivot x is always chosen so that x ends up at the very left (resp. the very right) end of the array after the partitioning. This happens if x is the least (resp. greatest) element of the array:

x	elements $\geq x$

In this case the remainder of the array contains $n-1$ elements, which must be sorted in turn. If we again choose the pivot as the least (resp. greatest) of the remaining elements, then the third round must sort $n-2$ elements, and so on. The total number of comparisons required would be

$$T_{worst}(n) = \sum_{m=1}^{n} m = n + (n-1) + \cdots + 2 + 1 = \frac{n(n+1)}{2} = \frac{n^2}{2} + \frac{n}{2}$$

in the worst case. The worst-case running time $T_{worst}(n)$ is asymptotically proportional to n^2.

In the best case the pivot x is always chosen so that x ends up in the middle of the array after the partitioning:

elements $\leq x$	x	elements $\geq x$

In this case the second round requires two sub-sorts, each sorting approximately $\frac{n}{2}$ elements. If the pivot is chosen well in both of these sub-sorts, then those give rise to four sub-sub-sorts, each sorting approximately $\frac{n}{4}$ elements, and so on.

That is, the number of elements to be sorted by a sub-sort is halved for each level of recursion. This can happen at most $\log_2(n)$ times before we reach a sub-array of length 0 or 1, which is sorted by definition.

Thus the best running time $T_{best}(n)$ for sorting of n elements is

$$\begin{aligned} T_{best}(1) &= 0 \\ T_{best}(n) &= n + 2 \cdot T_{best}(\tfrac{n}{2}) \quad \text{if } n > 1 \end{aligned}$$

and therefore

$$\begin{aligned} T_{best}(n) &= n + 2 \cdot T_{best}(\tfrac{n}{2}) \\ &= n + 2 \cdot (\tfrac{n}{2} + 2 \cdot T_{best}(\tfrac{n}{4})) \\ &= n + n + 4 \cdot T_{best}(\tfrac{n}{4}) \\ &= n + n + 4 \cdot (\tfrac{n}{4} + 2 \cdot T_{best}(\tfrac{n}{8})) \\ &= n + n + n + 8 \cdot T_{best}(\tfrac{n}{8}) \\ &= \underbrace{n + n + \cdots + n}_{\log_2(n)\text{ terms}} \\ &= n \log_2(n) \end{aligned}$$

In the best case, the running time is asymptotically proportional with $n\log_2(n)$. More importantly, this is also the average case, assuming that all permutations of the data set are equally likely.

In practice, Quicksort is a very fast sorting algorithm, but it may be slow in extreme cases.

5.3 Programming Quicksort

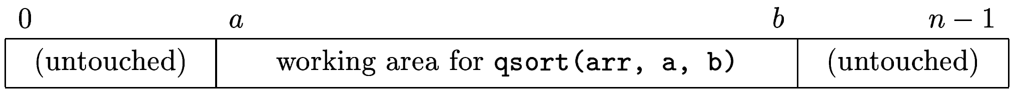
The basic idea in Quicksort is (1) to partition into smaller sub-arrays, and (2) to sort these separately. Therefore it is convenient to program it as a *recursive* method: after the partitioning, the method simply calls itself to sort the sub-arrays.

A call `qsort(arr, a, b)` to method `qsort` will sort the sub-array $arr[a..b]$:

```
private static void qsort(int[] arr, int a, int b)
{
  if (a < b)
    {
      int i = a, j = b;
      int x = arr[(i+j) / 2];                /* pp1 */
      do {                                   /* pp2 */
       while (arr[i] < x) i++;               /* pp3 */
       while (arr[j] > x) j--;               /* pp4 */
       if (i <= j)
         {
           swap(arr, i, j);
           i++; j--;
         }                                   /* pp5 */
      } while (i <= j);                      /* pp6 */
      qsort(arr, a, j);                      /* pp7 */
      qsort(arr, i, b);                      /* pp8 */
    }                                        /* pp9 */
}
```

5.4 Snapshots for method `qsort`

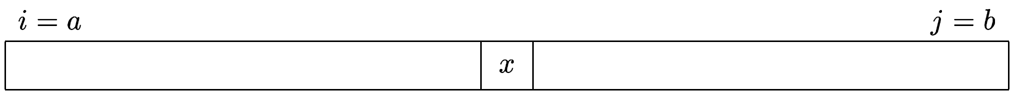
In general, a call to method `qsort` will work on a sub-array $arr[a..b]$ of $arr[0..(n-1)]$, so that the other sub-arrays $arr[0..(a-1)]$ and $arr[(b+1)..(n-1)]$ are left untouched:



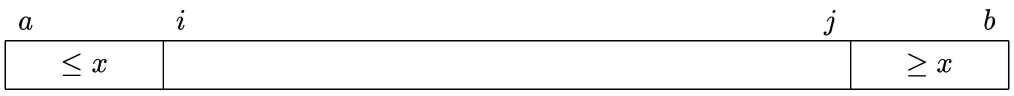
In the snapshots below, we shall therefore focus on the sub-array $arr[a..b]$ and ignore the rest.

First we note that if $a \geq b$, then $arr[a..b]$ has zero or one element, so $arr[a..b]$ is sorted already. In this case, the program execution skips all statements before `pp9`.

Otherwise, we have $a < b$, so that $arr[a..b]$ has at least two elements, and $a \leq (a+b)/2 \leq b$. At `pp1` it holds, when $x = arr[(a+b)/2]$:

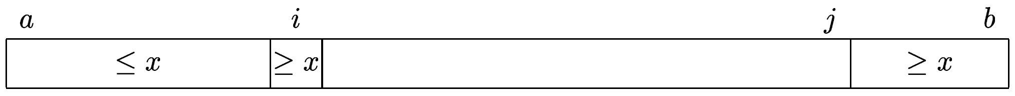


We shall see that it holds at `pp2`:



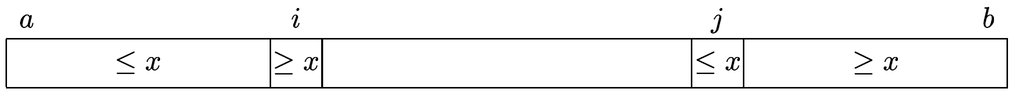
In particular this holds in the first iteration of the do-while loop, with $i = a$ and $j = b$, for in this case the sub-arrays at the left and right end are empty.

If that snapshot holds at pp2, then we have at pp3, because i will be moved to the right so long as $arr[i] < x$:



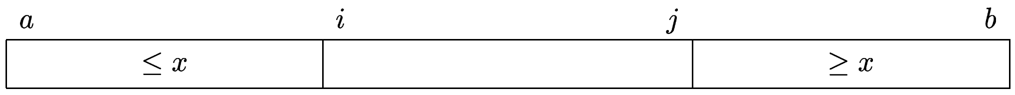
The element at index i must be $\geq x$; otherwise the first while loop would have continued.

If that snapshot holds at pp3, then the following will hold at pp4:



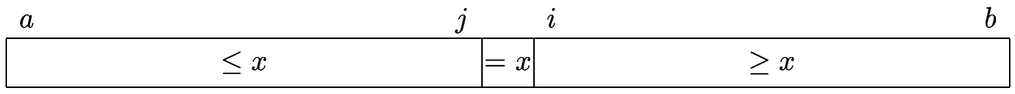
The element at index j must be $\leq x$; otherwise the second while loop would have continued.

If that snapshot holds at pp4, and $i \leq j$, then the elements $arr[i]$ and $arr[j]$ will be swapped, and i and j will be moved one step to the right (resp. left), so we have at pp5:



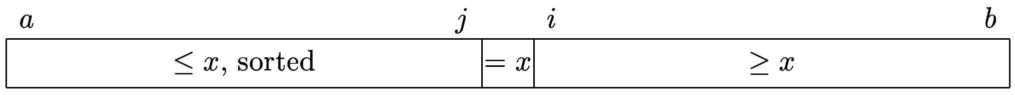
If the do-while loop continues, then the same snapshot will hold at pp2 again in the next iteration. We now see that pp2 holds from the beginning; that pp2 implies pp3, pp4, and pp5; and that pp5 implies that pp2 holds in the next iteration. Therefore pp2, pp3, pp4, and pp5 hold in every iteration of the do-while loop: they are loop invariants.

If the do-while loop terminates, then it is because $i > j$, and therefore $i - 1 \geq j$. Since pp5 holds in every iteration, also the last one, we must have at pp6:

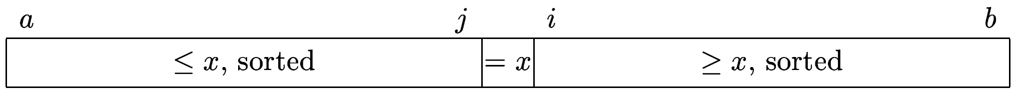


Since $arr[a..i] \leq x$ and $arr[j..b] \geq x$, the common elements, in $arr[j..i]$, must equal x.

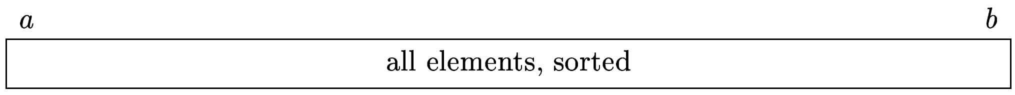
Inductively we can assume that `qsort(arr, a, j)` does in fact sort $arr[a..j]$, so that at pp7 it holds:



Similarly, after the call `qsort(b, i)` it holds at pp8 and therefore pp9 that:



But that is the same as saying:



We have shown that method qsort does in fact sort the elements in $arr[a..b]$, regardless whether $a \geq b$ or $a < b$.

5.5 Why do the loops terminate?

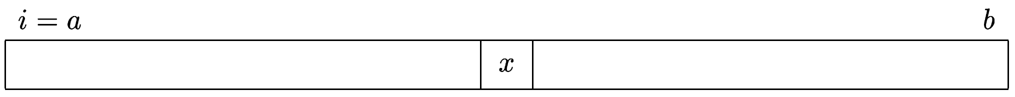
However, we have not yet shown that method `sort` ever terminates.

We can see that the `do-while` loop terminates, by observing that i is increased and j is decreased by at least 1 in every iteration, so the difference $j - i$ decreases by at least 2 in every iteration. The loop terminates when $i > j$, that is, when $0 > j - i$. One can subtract 2 from a number only finitely many times before it gets negative. Consequently there can be at most finitely many iterations of the `do-while` loop before it terminates.

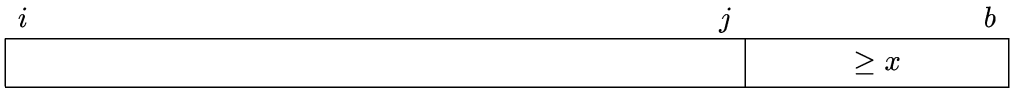
What about the two `while` loops? Apparently nothing prevents the loop

```
while (arr[i] < x) i++;
```

from looping forever. But in fact it holds just before the loop, at `pp2`, that there is a $y \in arr[i..b]$ for which $y \geq x$, so the loop condition will become false at the latest when $i = b$, and the loop will stop. Namely, in the first iteration of the outer `do-while` loop we have $i = a$, and x is chosen so that $x \in arr[a..b]$:



In all later iterations it follows from `pp5` and $j < b$ that $arr[b] \geq x$:



Similar arguments show that the second `while` loop terminates.

Note that it would be rather difficult to understand this argument without the snapshots presented above.

Avoiding the separate bounds tests $i < b$ and $j > a$ in the inner `while` loops contribute much to the practical efficiency of Quicksort. However, if we could not convince ourselves that the loops terminate anyway, it would be irresponsible to leave out the tests. In other words, loop invariants can also contribute to the creation of highly efficient programs!

We still need to explain why we do not get an infinite chain of recursive calls to `qsort`. But that is easy; `qsort` calls itself only if the sub-array $arr[a..b]$ has at least two elements, and calls itself only on sub-arrays $arr[a..j]$ and $arr[i..b]$ that are strictly smaller than the original array $arr[a..b]$. This can happen only finitely many times before the sub-arrays have less than two elements. Consequently there are only finitely many recursive calls.

Quicksort is correct: it terminates, and it terminates with the right answer: a sorted array.

5.6 Quicksort

The overall Quicksort algorithm simply calls on method `qsort` to sort $arr[0..(n-1)]$:

```
public static void quicksort(int[] arr, int n)
{
  qsort(arr, 0, n-1);
}
```

5.7 Improving Quicksort

Above we remarked that Quicksort is very slow if the pivot x is chosen inappropriately: as the least or the greatest element in the sub-array. The risk that this happens can be reduced by choosing the pivot x in `qsort` as the median of the three values $arr[a]$, $arr[(a+b)/2]$, and $arr[b]$. That is, x is chosen as one of these three values such that one of them is less than or equal to x, and the other is greater than or equal to x. For instance, 35 is the median of 35, 28, and 45, because $28 \leq 35$ and $45 \geq 35$.

If the pivot x were chosen as the median of the *entire* data set, the partitioning would split the data set in two equally large parts, and Quicksort would be guaranteed to be fast. Indeed it is possible to choose the pivot this way, getting a better worst-case execution time, but it complicates the algorithm and in practice it becomes slower. Usually we are satisfied with just reducing the *risk* that Quicksort will be slow, by picking the pivot as the median of a few elements, possibly chosen at random.

5.8 Exercises

1. Execute `quicksort` manually on the array:

0	1	2	3	4	5
35	62	28	50	11	45

   Show the values of a, b, i, j, and x and the array's contents at appropriate steps of the execution.

2. Execute `quicksort` manually on the array:

0	1	2	3	4	5
35	35	35	35	35	35

3. Would method `qsort` work also if the conditional in the inner `if` statement were

```
if (i < j) ...
```

4. Use method `fillarray` from Exercise 4 in Section 4.10 to fill an array with 0, 50, and 100 random numbers. Sort them using Quicksort and print them to check that it works.
5. Remove the print statements from the above program, and use a watch to time the execution of n =1.000, 2.000, 3.000, 4.000, 5.000, 10.000, 20.000, 30.000 random integers. Tabulate the execution time as a function of n.
6. Use Quicksort instead of selection sort in the program from Section 4.7.
7. Use Quicksort instead of selection sort in the program from Section 4.8.
8. Write conditional statements that choose the pivot x as the median of $arr[a]$, $arr[(a+b)/2]$ and $arr[b]$, cf. Section 5.7.
9. Execute method `quicksort` manually on this array:

0	1	2	3	4	5
15	13	11	12	14	16

6 Heap sort

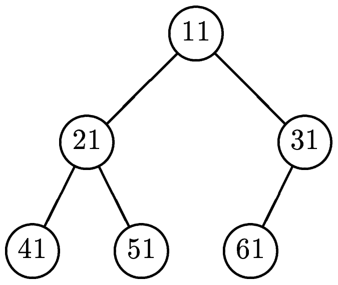
Heap sort is related to selection sort: the resulting sorted array is built gradually, by repeatedly selecting an element from the as yet unsorted part of the array.

However, heap sort is much faster, just because the selection is performed in a more intelligent way than in selection sort. Another difference is that heap sort builds the sorted array backwards, continually selecting the *largest* remaining element rather than the smallest one. One could define heap sort to select the smallest element instead, but that version is rather cumbersome.

Heap sort is somewhat slower than Quicksort in practice, but has the advantage that its execution time and memory consumption are completely predictable. Heap sort was invented by J. Williams in 1962 and improved by R.W. Floyd in 1964.

6.1 Trees and heaps

A *binary tree* consists of a *root* which has zero, one, or two *branches*, which are themselves trees. The roots of (sub-)trees are called *nodes*, and contain values, such as numbers. A node without branches is called a *leaf*. The tree below has the value 11 in the root (the top-most node), the values 21 and 31 the nodes of its branches, and 41, 51 and 61 in the leaves:



Computer science trees usually have their root at the top by convention.

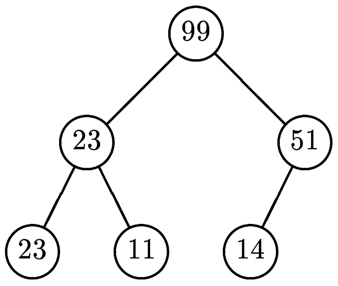
An array $arr[0..(n-1)]$ can be thought of as a binary tree, whose root is in $arr[0]$ and whose branches are in $arr[1]$ and $arr[2]$. More generally, if a sub-tree has its root in $arr[i]$, then its left branch (if any) is in $arr[2i+1]$ and its right branch (if any) is in $arr[2i+2]$.

The tree above corresponds to this array:

0	1	2	3	4	5
11	21	31	41	51	61

A node in a tree satisfies the *heap condition* if the value in the node is greater than or equal to the values in its branches (if any). A leaf trivially satisfies the heap condition because it has no branches. In the tree above only the leaves satisfy the heap condition.

A *heap* is a binary tree in which all nodes satisfy the heap condition. For instance, this tree is a heap, since $99 \geq 23$, $99 \geq 51$, $23 \geq 23$, $23 \geq 11$ and $51 \geq 14$:



In a heap, the value in the root is greater than or equal to the values in the top branches, which in turn are greater than or equal to the values in their own branches, and so on. Consequently the value in the root is the greatest value in the heap. It is just this property of heaps that is used in heap sort.

In an array $arr[0..(n-1)]$ the index i satisfies the heap condition if $arr[i] \geq arr[2i+1]$ and $arr[i] \geq arr[2i+2]$ when $2i+1 < n$ and $2i+2 < n$.

Correspondingly, an array $arr[0..(n-1)]$ is a *heap*, if for all $i = 0 \ldots (\frac{n}{2} - 1)$ it holds that $arr[i] \geq arr[2i+1]$ and $arr[i] \geq arr[2i+2]$. In an array which is a heap, it holds that $arr[0] \geq arr[0..(n-1)]$.

The array corresponding to the above heap is:

0	1	2	3	4	5
99	23	51	23	11	14

6.2 Operations on trees and heaps

6.2.1 Heapifying a node

Assume that we have a binary tree in which all nodes, except the root node, satisfy the heap condition. Then the value $arr[i]$ in the root must be smaller than one or both of the values in the branch nodes $arr[2i+1]$ and $arr[2i+2]$. In this case we can swap $arr[i]$ with the largest of the values $arr[2i+1]$ and $arr[2i+2]$ from the branch nodes, say, $arr[2i+1]$. This makes the root $arr[i]$ satisfy the heap condition, but now it is possible that $arr[2i+1]$ no longer satisfies the heap condition. Then one must heapify node $arr[2i+1]$; clearly this is a recursive process.

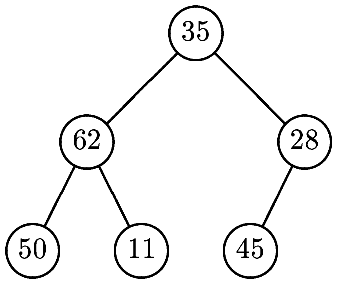
Since the index i doubles every time one goes down a branch, one can perform at most $\log_2(n)$ operations before reaching a leaf (which has index $> \frac{n}{2}$), when the heap has n elements. That is, it requires at most $\log_2(n)$ operations to heapify the root node, if that is the only node not satisfying the heap condition

6.2.2 Heapifying an entire tree

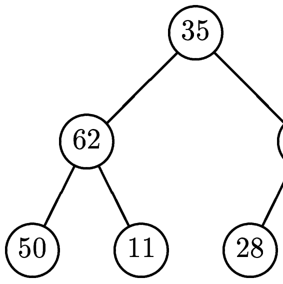
An entire tree may be heapified (turned into a heap) by heapifying its nodes from below. The leaves require no attention; they trivially satisfy the heap condition. One starts by heapifying those nodes whose branches are leaves, then those nodes whose branches have already been heapified, and so on, until one reaches the root.

A tree with n nodes has $\frac{n}{2}$ non-leaf nodes, and heapifying a single nodes requires at most $\log_2(n)$ operations, so the entire tree can be heapified with at most $\frac{n}{2}\log_2(n)$ operations.

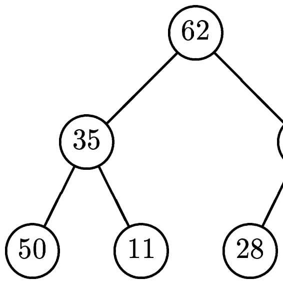
Example: Let us heapify this tree, which corresponds to the array [35, 62, 28, 50, 11, 45]:



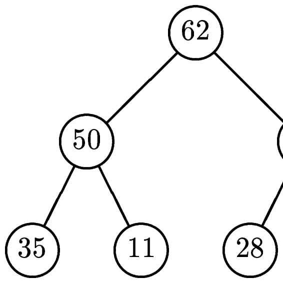
The leaves already satisfy the heap condition, so the heapification starts with the node layer next to the bottom. The sub-tree with node 62 satisfies the heap condition already. The sub-tree with node 28 must be heapified because $28 < 45$. This is done by swapping 28 and 45:



We proceed up the tree, and find that the root, node 35, does not satisfy the heap condition. The greatest branch node value is 62, so we swap 35 and 62:



Now the root (62) satisfies the heap condition, but node 35 does not. The greatest branch node value is 50, so we swap 35 and 50, and now all nodes satisfy the heap condition. The tree is a heap:



Could the heapification of node 35 possibly cause the root (62) to fail the heap condition, by replacing node 35 by some node value greater than 62? No. The sub-tree rooted at 35 originally had root value 62, and that sub-tree did satisfy the heap condition, so all node values in that sub-tree must be less than or equal to 62.

Let us see how the manipulations of the tree would be reflected in the corresponding array $arr[0..5]$. The node being heapified is underlined in every line:

0	1	2	3	4	5
35	62	$\underline{28}$	50	11	45
$\underline{35}$	62	45	50	11	28
62	$\underline{35}$	45	50	11	28
62	50	45	35	11	28

6.2.3 Extracting the greatest element from a heap

In heap sort we repeatedly need to extract the greatest element from a heap. We know where to find it: in the root of the tree (that is, element $arr[0]$ in the array). We cannot simply remove the root, then the tree breaks in two. However, we can replace the root by the heap's bottom right-most leaf (that is, element $arr[n-1]$ of the array). This new root node may fail the heap condition, but this may be fixed just by heapifying the new root node, as explained in Section 6.2.1.

That is, extracting the greatest element from a heap, and reestablishing the heap property, may be done in at most $\log_2(n)$ operations.

6.3 Heap sort = heapification phase + extraction phase

In heap sort one thinks of the array $arr[0..(n-1)]$ as a tree, and proceeds in two phases:

(1) Heapify the entire tree (array), as shown in Section 6.2.2.
(2) Extract the greatest element and re-heapify the remaining elements so that they satisfy the heap condition, as described in Section 6.2.3; extract the second-greatest element and re-heapify; extract the ... and so on, until the tree is empty.

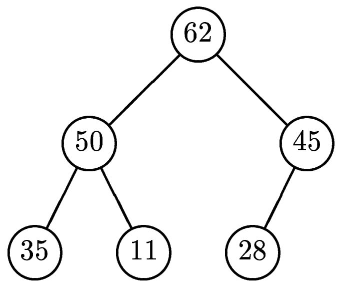
The sorted array is built from right to left, as the elements are extracted in decreasing order. This means that the current heap can be kept at the left-hand section of the array, and the sorted elements can be kept at the right-hand section of the array:

0	m $\qquad\qquad\qquad$ $n-1$
other elements, as a heap	the $n-1-m$ greatest elements, sorted

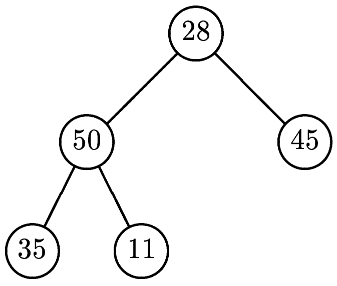
Extraction from the heap can be done simply by swapping $arr[0]$ and $arr[m]$. That is, we swap the top-most (greatest) node with the bottom right-most one. Re-heapification of the tree is done by working on the array section $arr[0..(m-1)]$ which represents the tree after removing the bottom right-most node. When the tree (left-hand array section) is empty, then the array $arr[0..(n-1)]$ contains the entire sorted result.

Note that the invariant above is closely related to that of selection sort (pp3 in Section 4.4).

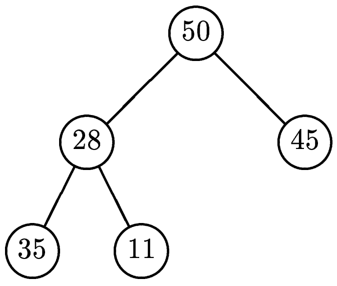
Example: Let us sort the array containing [35, 62, 28, 50, 11, 45]. In the above example we went through phase (1), by heapifying the corresponding tree. The result was:



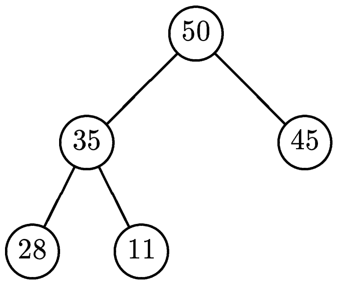
We continue with phase (2). Extract the root 62. Replace it with the bottom right-most element 28, whose node is removed from the tree:



Now we must re-heapify at the root, 28. This is done by swapping 28 and 50, which is the greatest of the branch nodes:



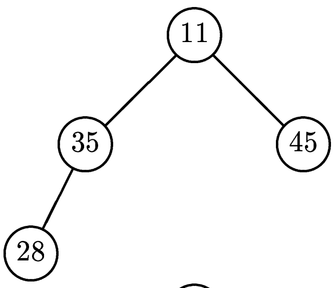
Now 28 and 35 must be swapped to heapify the left sub-tree:

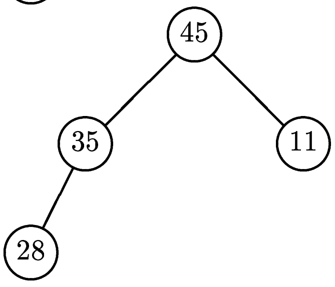


The corresponding array manipulations are shown below. Note that the sorted elements (so far just the element 62) are stored to the right in the array:

62	50	45	35	11	28
$\underline{28}$	50	45	35	11	62
50	$\underline{28}$	45	35	11	62
50	35	45	28	11	62

Extract the second-greatest element, 50, and make the sorted section [50, 62]. We replace 50 with the bottom left-most element (11) and re-heapify by swapping 11 and 45:

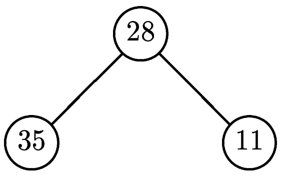


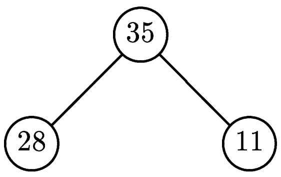


The corresponding array manipulations are:

<u>11</u>	35	45	28	50	62
45	35	11	28	50	62

Extract 45 and make the sorted section [45, 50, 62]. Swap 45 and 28, and re-heapify by swapping 28 and 35:

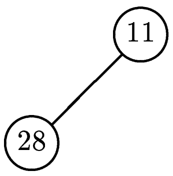


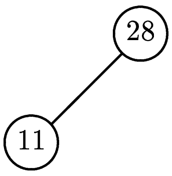


The corresponding array manipulations are:

<u>28</u>	35	11	45	50	62
35	28	11	45	50	62

Extract 35 and make the sorted section [35, 45, 50, 62]. Replace 35 by 11, and re-heapify by swapping 11 and 28:





The corresponding array manipulations are:

<u>11</u>	28	35	45	50	62
28	11	35	45	50	62

Extract 28 and make the sorted section [28, 35, 45, 50, 62]. The heap now contains a single element, 11. The corresponding array is:

11	18	35	45	50	62

When the $n - 1$ greatest elements are in the sorted section, then the remaining element must be the least one, so the sorting process is complete:

11	18	35	45	50	62

6.4 The efficiency of heap sort

You will be forgiven for thinking that heap sort is very cumbersome and therefore slow. Perhaps surprisingly, theoretical considerations show that heap sort is guaranteed to be fast in all cases (in contrast to Quicksort, which is fast only 'on the average', and may be slow in some cases).

Using heap sort on the array $arr[0..(n-1)]$ involves two phases. Phase (1) heapifies the array. Phase (2) repeatedly extracts the heap's greatest element and re-heapifies the remaining elements, until the heap is empty, for a total of n extractions.

Phase (1) takes time at most proportional to $n \log_2(n)$, as argued in Section 6.2.2.

Phase (2) involves n extractions from the heap, each followed by heapification of the root node $arr[0]$. As argued in Section 6.2.3 each re-heapification requires at most $\log_2(n)$ operations. In total the time consumption of this phase is at most proportional to $n \log_2(n)$.

Together, the two phases take time $2n \log_2(n)$, so the total time consumption is at most proportional to $n \log_2(n)$. This is a worst case bound, and the average case is similar (certainly it cannot be worse).

Theoretically speaking, heap sort is as good as Quicksort in the average case, and much better in the worst case. In practice, heap sort is on the average somewhat slower than Quicksort because of the many manipulations on the heap.

6.5 Programming heap sort

6.5.1 Heapifying a tree

The node *arr*$[i]$ in array *arr*$[0..k]$ can be heapified by a call to the method `heapify(arr,i,k)`, which works as described in Section 6.2.1:

```
private static void heapify(int[] arr, int i, int k)
{
  int j = 2 * i + 1;                          /* pp1 */
  if (j <= k)
    {
      if (j+1 <= k && arr[j] < arr[j+1])
        j++;                                  /* pp2 */
      if (arr[i] < arr[j])
        {
          swap(arr, i, j);                    /* pp3 */
          heapify(arr, j, k);                 /* pp4 */
        }
    }                                         /* pp5 */
}
```

At `pp1` variable j equals the index of the left branch of *arr*$[i]$, if any. If $j > k$, then *arr*$[i]$ has no branches; it is a leaf and therefore satisfies the heap condition, so we skip to `pp5`. Otherwise $j \leq k$. If *arr*$[i]$ has a right branch *arr*$[j+1]$, and if its node value is greater than that of the left branch *arr*$[j]$, then make j point to the right branch by incrementing j by 1.

At `pp2` we know that *arr*$[j]$ is the greatest branch node value.

If *arr*$[i] \geq$ *arr*$[j]$, then *arr*$[i]$ already satisfies the heap condition, and we skip to `pp5`. Otherwise *arr*$[i] <$ *arr*$[j]$, and we swap *arr*$[i]$ and *arr*$[j]$.

At `pp3` we know that *arr*$[i]$ satisfies the heap condition, but now the branch *arr*$[j]$ may no longer satisfy it, so we call `heapify(arr, j, k)` recursively if necessary to re-heapify *arr*$[j]$.

At `pp4` the heap condition for *arr*$[j]$ has been re-established. Because the original sub-tree *arr*$[j]$ satisfied the heap condition, and *arr*$[i]$ is the original value of *arr*$[j]$, we have *arr*$[j] \leq$ *arr*$[i]$.

In all three cases *arr*$[i]$ satisfies the heap condition at `pp5`, as desired.

6.5.2 Heap sort

The method `heapsort` follows the informal description of heap sort closely:

(1) Heapification of the array is done by heapifying all nodes from the bottom up, except for the leaves, as described in Section 6.2.2. The bottom nodes correspond to the highest indexes in *arr*. Thus the leaves are in *arr*$[\frac{n}{2}..(n-1)]$. To heapify the array we therefore execute `heapify(arr, m, n-1)` for $m = (\frac{n}{2} - 1) \ldots 0$.

(2) When $m + 1$ unsorted elements remain in the heap, then the greatest element in *arr*$[0]$ must be moved to index m; we do that by swapping *arr*$[0]$ and *arr*$[m]$. The heap now has m elements. Subsequently the heap condition must be re-established for *arr*$[0]$; we do that by executing `heapify(arr, 0, m-1)`. Both of these steps must be performed for $m = (n-1) \ldots 1$.

```
public static void heapsort(int[] arr, int n)
{
  for (int m=n/2; m >= 0; m--)
    heapify(arr, m, n-1);
                                                /* pp1 */
  for (int m=n-1; m >= 1; m--)
    {                                           /* pp2 */
      swap(arr, 0, m);                          /* pp3 */
      heapify(arr, 0, m-1);                     /* pp4 */
    }                                           /* pp5 */
}
```

After the second `for` loop it holds that $m = 0$, and the heap contains a single element, which must be the least one. Thus it is already in the right place: at the head of the sorted array.

6.6 Snapshots for heap sort

At `pp1` all nodes satisfy the heap condition, because the leaves do, and `heapify(arr, m, n-1)` has been called on all non-leaves, from the bottom up. Thus the entire array $arr[0..(n-1)]$ is a heap:

0 … $n-1$
n elements, as a heap

We shall see that it holds at `pp2`:

0	m … $n-1$
other elements, as a heap	the $n-1-m$ greatest elements, sorted

In particular, this holds initially with $m = n-1$ and the sorted section being empty.

Since $arr[0..m]$ is a heap, $arr[0]$ is greatest among $arr[0..m]$.

Since $arr[(m+1)..(n-1)]$ contains the $n-1-m$ greatest elements, it holds that $arr[0] \leq arr[(m+1)..(n-1)]$.

After swapping $arr[0]$ and $arr[m]$ it holds at `pp3`:

0	m … $n-1$
other elements	the $n-m$ greatest elements, sorted

The array section $arr[m..(n-1)]$ is sorted because $arr[m] \leq arr[(m+1)..(n-1)]$, and it contains the $n-m$ greatest elements because $arr[m] \geq arr[0..m-1]$.

After the swap $arr[0]$ may not satisfy the heap condition, but after the call `heapify(arr, 0, m-1)` we have at `pp4`:

0	m … $n-1$
other elements, as a heap	the $n-m$ greatest elements, sorted

If the loop continues, then m is decremented by 1 before we reach `pp2`, so that the snapshot at `pp2` holds again.

We see that pp2 holds initially, that pp2 implies pp3 and pp4, and that pp4 implies that pp2 holds again in the next iteration. Thus pp2, pp3, and pp4 hold in every iteration; they are loop invariants.

When the loop terminates we have $m = 0$, and it follows from pp4 that we have at pp5:

0	1 $\quad\quad\quad$ $n-1$
	the $n-1$ greatest elements, sorted

The remaining element $arr[0]$ must be the least one in the array, so it holds at pp5 that:

0 $\quad\quad\quad$ $n-1$
all elements, sorted

Heap sort works.

6.7 Exercises

1. Execute heap sort phase (1) manually on this array:

0	1	2	3	4	5
35	62	28	50	11	45

   Show the value of m in `heapsort` and show what calls to `heapify` are made. Show the values of j (in method `heapify`), and show the array's contents at each step of the execution.
2. Execute heap sort phase (2) manually on the array resulting from the above exercise:

0	1	2	3	4	5
62	50	45	35	11	28

3. Execute heap sort manually on this array:

0	1	2	3	4	5
35	35	35	35	35	35

4. Use method `fillarray` from the exercises in Section 4.10 to fill an array with 0, 50, and 100 random numbers. Sort them with heap sort and print them to check that it works.
5. Remove the print statements from the above program, and time the execution of heap sort for n =1.000, 2.000, 3.000, 4.000, 5.000, 10.000, 20.000, 30.000 random numbers. Tabulate the execution time as a function of n. Compare with your results for Quicksort.
6. Use heap sort instead of selection sort in the program from Section 4.8.

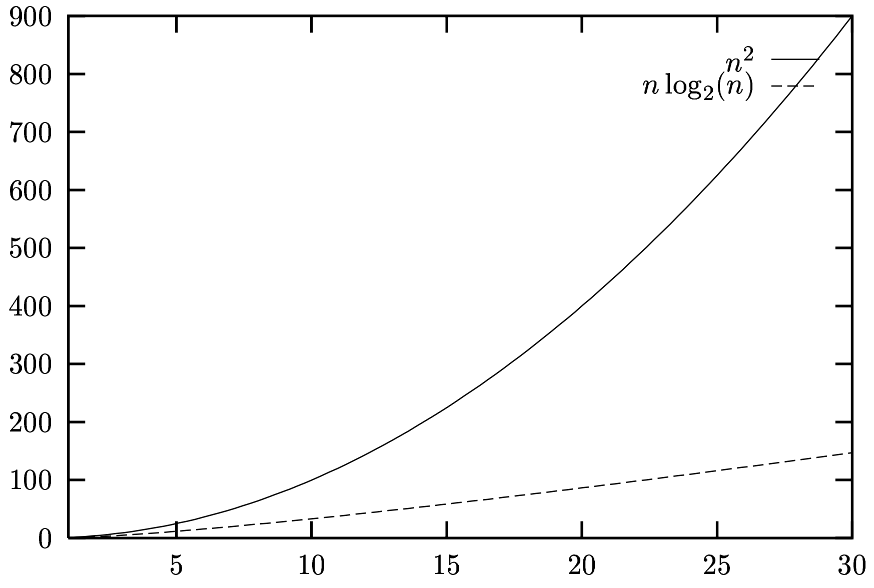
7 Comparing the three sorting algorithms

7.1 Theoretical execution times

Previously we found that the following theoretical execution times for the three sorting algorithms, as functions of the number n of elements to be sorted:

	Average	Worst
Selection sort	n^2	n^2
Quicksort	$n\log_2(n)$	n^2
Heap sort	$n\log_2(n)$	$n\log_2(n)$

Since n^2 grows much faster than $n\log_2(n)$ as a function of n, selection sort is *much* slower than Quicksort and heap sort, except for small data sets. This is clear from the function graphs:



The graph shows the theoretical average execution time for sorting n elements. The upper curve is for selection sort; the lower one is for Quicksort and heap sort.

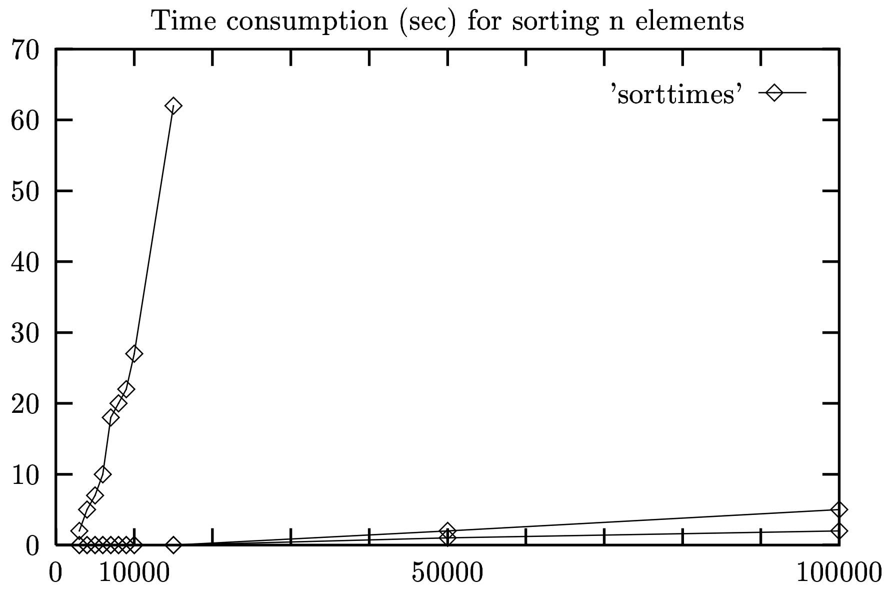
7.2 Determining (average) execution time experimentally

To see whether the theoretical results agree with practice, we have made some experiments with arrays of pseudo-random numbers. The results are shown below. They seem to agree with the theoretical estimates:

n	Selection sort	Quicksort	Heap Sort
3,000	2	0	0
4,000	5	0	0
5,000	7	0	0
6,000	10	0	0
7,000	18	0	0
8,000	20	0	0
9,000	22	0	0
10,000	27	0	0
15,000	62	0	0
50,000	766	1	2
100,000	3 995	2	5
500,000	(21 hours)	11	30
1,000,000	(111 hours)	27	66

The execution times are in seconds; figures within parentheses are estimates. We see that heap sort is 2 to 3 times slower than Quicksort on random data sets. Already at 100,000 elements, selection sort is approximately 2,000 times slower than Quicksort. At 1,000,000 elements it is nearly 15,000 times slower. That is the same as the difference between a two-week summer school and one minute.

The above data can be shown graphically:



The graph shows actual measured execution times for sorting n pseudo-random numbers. They can be expected to match the theoretical average cases.

The upper curve is for selection sort, the middle one is for heap sort, and the lower one is for Quicksort.

7.3 Perspective

We have seen two algorithms for searching an array: linear search and binary search. The former works for any array; the latter requires the array to be sorted, but is much faster.

We have seen three algorithms for sorting an array: selection sort, Quicksort, and heap sort. Selection sort is simple, but too slow if there are more than a few elements. Quicksort is the fastest one in practice, but may degenerate on certain data sets, in which case it uses much time and memory. Heap sort has a highly predictable time consumption, but is 2 to 3 times slower than Quicksort in practice.

It is remarkable that the three sorting algorithms solve exactly the same problem, but in entirely different ways, and with different execution times. From this we learn that even a precisely defined problem may have several, often widely different, solutions. In fact, there are even more sorting algorithms than those we looked at.

We have seen also that the speed of a program may be calculated theoretically, regardless of what computer will be used to run it. Moreover, the theoretical considerations are confirmed by experiments.

Finally, we have seen that snapshots and invariants can help understanding program loops, and help arguing that the loops work.

7.4 Exercises

1. Another (poor) sorting algorithm is *bubble sort*, which works as follows: The array is traversed from left to right, and neighbour elements that are out of order get swapped locally. This is repeated until all elements appear in increasing order. An example:

0	1	2	3	4	5
35	62	28	50	11	45
35	28	62	50	11	45
35	28	50	62	11	45
35	28	50	11	62	45
35	28	50	11	45	62

0	1	2	3	4	5
35	28	50	11	45	62
28	35	50	11	45	62
28	35	11	50	45	62
28	35	11	45	50	62
28	11	35	45	50	62

0	1	2	3	4	5
28	11	35	45	50	62
11	28	35	45	50	62

Estimate the best and worst execution time for bubble sort. (Hint: consider what happens if the array is ordered from the beginning; consider what happens if the array is inversely ordered from the beginning). Program bubble sort in Java and measure its execution time on pseudo-random data.

2. Yet another (poor) sorting algorithm is *insertion sort*, which works as follows: The left-hand section of the array is kept sorted; to begin with this section is empty. The left section is extended with one element at a time. Inserting the new element at its proper place in the sorted section may require one to move up some of the existing elements of the sorted section. This is repeated until all elements have been added to the sorted section. In the example below, the elements are inserted in the order 35, then 62, then 28, and so on. Note that to insert 28 at its proper place one has to move 35 and 62 to the right.

0	1	2	3	4	5
35	62	28	50	11	45
35	62	28	50	11	45
35	62	28	50	11	45
28	35	62	50	11	45
28	35	50	62	11	45
11	28	35	50	62	45
11	28	35	45	50	62

Estimate the best and worst execution time for insertion sort. (Hint: consider what happens if the array is ordered from the beginning; consider what happens is the array is inversely ordered from the beginning). Program insertion sort in Java and measure its execution time on pseudo-random data.

8 General sorting routines

As shown in Section 4.8 it is necessary to create a new version of the sorting methods for every new element type (such as `int`, `String`, `Person`), and for every new sorting criterion (such as 'sort by date of birth', 'sort by name'). In this section we describe general sorting methods, that can handle elements of all (object) types and all sorting criteria.

To read this section, the reader should be familiar with inheritance and interfaces.

8.1 Ordered data

One cannot sort data unless they have an ordering relation, such as '<'. Thus we can assume that the general sorting methods work on arrays of ordered elements. In Java we can express this assumption by requiring that the data are objects of a type `Ordered`, which has a method `less` to decide whether a given element is less than another element. We can declare the type `Ordered` as an interface with a method `less` for comparing two objects:

```
interface Ordered {
  public boolean less(Ordered x);
}
```

If `a` and `b` are objects of type `Ordered`, then `a.less(b)` will be `true` if `a` is less than `b` according to the sorting criterion, and `false` otherwise. The sorting methods themselves should use the predicate `less` instead of e.g. `a < b` or `a.compareTo(b) < 0`.

8.2 General selection sort

Having introduced the interface `Ordered`, we can declare a general method for selection sort as follows:

```
public static void selsort(Ordered[] arr, int n)
     // sort arr[0..n-1]
{                                           /* pp1 */
  for (int i = 0; i < n; i++)
    {                                       /* pp2 */
      int least = i;
      for (int j = i+1; j < n; j++)
        {
          if (arr[j].less(arr[least]))
            least = j;
        }
      swap(arr, i, least);                  /* pp3 */
    }                                       /* pp4 */
}
```

The only changes relative to Section 4.4 are in the method header (`Ordered[]` instead of `int[]`) and in the `if` statement (`less` instead of '<'). It is equally straightforward to define general versions of Quicksort and heap sort.

In any case one needs a general version of `swap` from Section 4.3, which can swap objects of type `Ordered` rather than `int`:

```
private static void swap(Ordered[] arr, int s, int t)
{
  Ordered tmp = arr[s];  arr[s] = arr[t];  arr[t] = tmp;
}
```

The idea now is to declare a class `Objsort` containing the above `selsort` and `swap` as class methods (`static` methods):

```
class Objsort {

  private static void swap(Ordered[] arr, int s, int t)
  { ... }

  public static void selsort(Ordered[] arr, int n)
  { ... }
}
```

The method `swap` is `private` because it will be used only inside the class, whereas `selsort` is `public` because it should be usable from the outside, called by `Objsort.selsort(arr, n)`.

8.3 Example: Sorting text strings

The general sorting method can be used to sort strings, for instance. To do this, define a class `OrderedString` which implements the interface `Ordered`, and which contains a string `s`. The sorting criterion `less` should just compare two strings lexicographically, using `compareTo`. The class has a constructor which stores its argument `s` in the class:

```
class OrderedString implements Ordered {
  String s;

  OrderedString(String s)
  { this.s = s; }

  public boolean less(Ordered t)
  { return s.compareTo(((OrderedString)t).s) < 0; }
}
```

One can think of `OrderedString` as an improved `String` that comes with an ordering `less`. Since `OrderedString` has a method `public boolean less(Ordered t)` as required by the interface `Ordered`, it is correct to say that `OrderedString implements Ordered`.

It would have been more elegant to make class `OrderedString` a subclass of `String`, but that is not allowed in Java: the `String` class is final and therefore cannot be subclassed.

In method `less` it is necessary to cast the argument `t` to the `OrderedString` class for the use of `compareTo` to make sense. The method `less` cannot just take a `String` as argument: according to the interface `Ordered` it must take an object `t` of type `Ordered`.

The sorting of text lines from Section 4.7 can now be performed by a main method of this form:

```
public static void main(String[] args)
     throws FileNotFoundException, IOException
{
  OrderedString[] lines = new OrderedString[100];
  int n = readfile(lines, "addrlist.txt");
  Objsort.selsort(lines, n);
  printout(lines, n);
}
```

where `lines` is an array with elements of type `OrderedString` instead of `String`. A possible method `readfile` for reading strings from a file is this:

```
public static int readfile(OrderedString[] arr, String filename)
     throws FileNotFoundException, IOException
{
  int n = 0;
  Reader inp = new FileReader(filename);
  StreamTokenizer tstream = new StreamTokenizer(inp);
  tstream.wordChars(' ', ' ');
  tstream.parseNumbers();
  tstream.nextToken();
  while (n < arr.length && tstream.ttype != StreamTokenizer.TT_EOF)
    {
      arr[n] = new OrderedString(tstream.sval);
      tstream.nextToken();
      n++;
    }
  return n;
}
```

In the method header `String[]` has been replaced by `OrderedString[]`. Also, we need to allocate a new `OrderedString` for every text line read from the file; that is done in the first line of the `while` loop.

8.4 Example: Sorting records

In Section 4.8 we sorted records containing name and date of birth for one person. Now we shall see how this is done using the general sorting methods. Recall that a `Person` is a simple object with two fields: `date` and `name`:

```
class Person {
  int date;
  String name;
}
```

We can make the records sortable by defining a subclass of `Person` which implements `Ordered`. As we intend to declare (at least) two subclasses, corresponding to different sorting criteria, we define a common abstract superclass `OrderedPerson`:

```
abstract class OrderedPerson extends Person implements Ordered {
  abstract public boolean less(Ordered x);
}
```

This looks pretty hairy, but the idea is straightforward: To be an OrderedPerson an object must be a Person as well as Ordered, so the class must extend Person and implement Ordered.

Now we can define various subclasses of OrderedPerson. To sort by date of birth (age) we use the subclass OrderedPerson whose method less compares the date fields of two objects of class Person:

```
class OrderedPerson1 extends OrderedPerson {

  public boolean less(Ordered f)
  { return date < ((OrderedPerson1)f).date; }
}
```

To sort by name we use the subclass OrderedPerson2 whose method less compares the name fields of two objects of class Person:

```
class OrderedPerson2 extends OrderedPerson {

  public boolean less(Ordered f)
  { return name.compareTo(((OrderedPerson2)f).name) < 0; }
}
```

To read records from a file and sort them by date of birth, one must create them as objects of class OrderedPerson1:

```
public static void main(String[] args)
     throws FileNotFoundException, IOException
{
  OrderedPerson[] people = new OrderedPerson[100];
  for (int i=0; i<people.length; i++)
    people[i] = new OrderedPerson1();
  int n = readfile(people, "birthday.txt");
  Objsort.selsort(people, n);
  printout(people, n);
 }
```

Sorting by name could be achieved by creating the records as objects of class OrderedPerson2 instead.

Note that the input and output methods readfile and printout from Section 4.8 can be used without any changes whatsoever. They handle objects of class Person, and since OrderedPerson1 is a subclass of Person, they automatically handle objects of class OrderedPerson1 also. The same holds for OrderedPerson2.

8.5 Determining the sorting criterion for a class of objects

Although the above design (using `OrderedPerson1` and `OrderedPerson2`) may seem clever, it is impractical. If one needs to sort the same data twice, first by name and then by date of birth, one has to duplicate all data records, both as `OrderedPerson1` and as `OrderedPerson2` objects. A more flexible solution is to create yet a third kind of ordered person, in which a common (static) field determines the sorting criterion for *all* objects of that class:

```
class OrderedPerson3 extends OrderedPerson {

  public static boolean datesort;

  public boolean less(Ordered f)
  {
    if (datesort)
      return date < ((OrderedPerson3)f).date;
    else
      return name.compareTo(((OrderedPerson3)f).name) < 0;
  }
}
```

If the field `datesort` is `true`, then the records will be sorted by date, otherwise by name.

8.6 Exercises

1. Create a general version of Quicksort from Section 5. It should take as parameter an array of `Ordered` objects and sort them by their `less` relation. Use it for sorting and printing the list of birthdays (from Section 4.8) by `date` and by `name` in a single program.
2. Define a class `StringIgnoreCase` of strings, which implements `Ordered`. The sorting criterion `less` should be lexicographic ordering, with no distinction between upper case and lower case letters. (Hint: The simplest — but not the most efficient — way to do this involves the `String` method `toLowerCase`). Try it with one of the general sorting routines and a suitable input file.
3. Create a general version of heap sort from Section 6. It should take as parameter an array of `Ordered` objects and sort them by their `less` relation. Use it for sorting and printing the list of birthdays (from Section 4.8) both by `date` and `name`.
4. The binary search method from Section 1.2 can be generalized as well, so that it searches an array whose elements have class `Ordered`. Program and test a generalized binary search.
5. Assume there is a file `grades.dat` of grades and course subjects in this format:

```
03 Oldgræsk
11 Programmering
7 Ukrudtslære
10 Nyere fransk litteratur
9 Limnologisk taksonomi
8 Matematisk grundkursus
6 Programmeringsteori
9 Levnedsmiddelanalyse
...
```

Define a suitable class for representing such data, and write a program to input, sort, and print lists of grades.

6. Generalize the above program so that the first line of the file determines the sorting criterion. If the first two characters of the first line is '`G+`', then the output should be sorted by increasing grades. If the first two characters of the first line is '`G-`', then the output should be sorted by decreasing grades. Similarly, '`S+`' determines that the output should be sorted alphabetically by course subject, and '`S-`' that it should be inversely sorted by course subject.
7. Modify the program from Exercise 5 so that it finds the courses with the five *highest* grades, in such a way that the program performs no unnecessary work. What algorithm is best for this purpose?
8. The k percent fractile (k-percentile) in a data set is an observation x for which it holds that k percent of the values are less than or equal to x. Describe how one can find the k-percentile in a data set. Write a program to do this. Let k (between 0 and 100) be the first value read in by the program.
9. To test whether a data set is normally distributed (that is, could stem from a normal or Gaussian distribution), one may compute the *empirical distribution function* for the data set and compare it with the normal distribution function.
   Assume the given data set has n observations. To find the empirical distribution function, one sorts the data set to obtain ordered observations $x_0, x_1, \ldots, x_{n-1}$.
   Now the empirical distribution function $F_e(x)$ is defined as follows:

$$F_e(x) = \begin{cases} 0 & \text{if } x < x_0 \\ \frac{i}{n} & \text{if } x_{i-1} \leq x < x_i \text{ where } 1 \leq i \leq n-1 \\ 1 & \text{if } x_{n-1} \leq x \end{cases}$$

   Write a Java program that reads a data set from a file and computes some values of the empirical distribution function, e.g. at $x = 0$, $x = 5$, and $x = 10$. Hint: this exercise requires sorting as well as searching.