Chapter 29

Searching

We present some algorithms for searching into data collections.

29.1 Aim

We want to see whether an data collection, e.g., an array contains a certain data element and if so, where it is located. We want to analyse the search algorithms on their relative speed.

29.2 Means

Several *algorithms* exist to search into (large) date collections. Which one to choose depends on the structure of the data and what we can assume about it.

29.2.1 Linear search in an array

A simple structure to store data is the *array*. When we want to find an element in the array and we don't know anything about the way the elements are stored, the only option is to inspect every element until the requested value has been found. Usually the elements are inspected in the order they are found in the array, leading to a *linear* or *sequential* search algorithm.

The following method searches an integer array arr for the value key. If the array contains the key value, it returns *an* index of the value, if it doesn't contain the key, it returns a negative number.

```
1  final static NOT_FOUND = −1;

3  int linearSearch(int[] arr, int key) {
       int i = arr.length−1;
5      while (i >=0 && arr[i] != key) {
           i++;
7      }
       return i;
9  }
```

In this program, we use a property of the &&-operator called *non-strictness* or the *short-circuit* property. It means that the second operand is not evaluated when the first one turns out to be false. Similarly, the second operand of the ||-operator will not be evaluated when the first operand turns out to be true. This is very convenient in cases like this, where array indexing, or other expressions that can be undefined is part

of a boolean expression. In our program, the expression `arr[i] != key` is undefined when `i` is outside the index range of the array, in particular, when `i<0`. This could occur when the array doesn't contain the searched key value. In that case, evaluation of this expression would lead to an exception and possibly abortion of the program. Putting the expression `i < 0` as the first operand of the conjunction avoids this unwanted behavior.

29.2.2 Analysis of linear search

Searching is a very common task for computer programs and often the data volume is very large, such as in search engines that search large portions of the World Wide Web. It is therefore important to analyse an algorithm on how long the search will take, in the number of array inspections. We give three important cases for this analysis. Take n as the number of elements in the array.

- *best case*: 1; when the first element in the array is the one we are looking for, the loop terminates immediately.
- *worst case*: n; when only the last element in the array is the one we are looking for, or when the array doesn't contain the key at all, it takes n array inspections;
- *average case*: $\frac{1}{2}n$: when the array contains exactly one key and the elements are randomly ordered, the search will take on average $\frac{1}{2}n$ steps.

The best case is not very informative, but often easy to calculate. Worst case is what is mentioned often, since it gives an upper bound to the execution time and it is easier to calculate than average case, which also requires knowledge or assumptions about the distribution of the data.

In comparing algorithms by efficiency, we are interested in how fast the execution time grows with the size of the input. Here, the size of the input is the length of the array, called n, and we see that the best case performance doesn't depend on n. We say that best case performance is of *constant* time complexity, denoted by $\Theta(1)$. The worst case execution time grows linearly with n and we say that this has *linear* time complexity, denoted by $\Theta(n)$. Although average case execution time is less than worst case execution time, but it also grows linearly with n and we also denote the complexity by $\Theta(n)$. We will see a definition of this concept later.

29.2.3 Binary search

When the array is sorted, a more efficient search algorithm can be used, called *binary search*. The reason why this is possible is the following. When an array a is sorted in ascending manner and it turns out that the $a[i] < key$, we know that all elements before i are also smaller than the key, so this whole part of the array can be discarded for the search. And the same holds for part beyond i, if $key < a[i]$. So if we inspect the middle element of the array (or an element close to the middle), we can always discard about half the elements of the array (either the left part or the right part), or stop, because we have found the element. Then we can repeat with the part that remains, etc. At every step the part of the array that has to be searched will be halved. If the array has length 2^k, the number of loop executions will be a little bit less than k, in the worst case. So the worst case time complexity is $\Theta(^2\log n)$.

The strategy of binary search is that of *divide-and-conquer*. The problem is divided in smaller parts, in this case in about equal halves, the parts are solved and their solutions are combined into a solution to the original problem. In this case, the solution of one of the parts is trivial: we see immediately that the sought for key can not be in that part of the array. And then the combination of the two is trivial also: we simply take the solution of the other half.

Strategies of divide-and-conquer are often natural to implement with *recursion*. We also give a recursive program here, although in practice usually an iterative program is used, since this is slightly more efficient.

```
7       /**
          * @pre a is ascending
9         * @post values a unchanged
          * @post \result  0 => 0  \result < a.length && a[\result] == key
11        * @post \result < 0 => −\result−1 is the insertion point ip
          * i.e., \foralll 0i<ip: a[i] < key && \forall ip<i<a.length: key < a[i]
13        */
        int binarySearch(int[] a, int key) {
15          int lo = −1;
            int hi = a.length;
17
            // imagine a[−1] == −infinity and a[a.length] = +infinity
19          //@inv a[lo]  key < a[hi] && −1 <= lo < hi <= a.length

21          while (lo+1 < hi) {
                int mid = (lo+hi)/2;
23              //assert 0 <= mid < a.length;
                if (a[mid] <= key) {
25                  lo = mid;
                } else { assert key < a[mid];
27                  hi = mid;
                }
29          }
            //assert a[lo] <= key < a[hi];
31          if (a[lo] == key) {
                return lo;
33          } else { // key not in a
                return −hi−1;
35          }
        }
37  }
```

29.3 Complexity Analysis

We call the analysis of the efficiency of an algorithm in quantitative terms of the amount of time (sometimes also memory) used *complexity analysis*. We don't measure in actual time used or CPU cycles because depends on the actual computer that is used, the operating system, etc. Therefore, we measure by determining the number of times some basic operation is performed as a function of the size of the input.

29.3.1 Order of complexity

We introduce the following mathematical notions.

Let functions $f, g : N \to N$ be given.

We define $f \in \mathcal{O}(g(n))$ (pronounced as "big-oh") if there exist c and n_0 such that $0 \leq f(n) \leq c \cdot g(n)$ for all $n > n_0$.

For example, $2n^2 - 2n + 12 \in \mathcal{O}(n^2)$, $2n^2 - 2n + 12 \in \mathcal{O}(n^3)$, $2n^2 - 2n + 12 \notin \mathcal{O}(n)$.

So $\mathcal{O}$ gives an upper bound to the rate of growth of a function. E.g., we say that the worst case time complexity of linear search is $\mathcal{O}(n)$. This means that for any implementation of linear search there is a constant c such that the time it takes, in worst case, to execute it on an array of n elements, is less than $c \cdot n$, for large n. So the timing behaviour of linear search linear in n, or better.

We define a similar notion to express that a function is at least as worse as another function.

We define $f \in \Omega(g(n))$ (other notation is $\emptyset(g(n))$) if there exist c and n_0 such that $0 \leq f(n) \geq c \cdot g(n)$ for all $n > n_0$.

We define $f \in \Theta(g(n))$ (Greek capital letter theta) if both $f \in \mathcal{O}(g(n))$ and $f \in \Omega(g(n))$.

For example, $2n^2 - 2n + 12 \in \mathcal{O}(n^2)$, $2n^2 - 2n + 12 \notin \mathcal{O}(n^3)$, n.

Θ is the most informative of these three notions. It defines complexity classes for algorithms. When two algorithms are in $\mathcal{O}(n)$, it means that they are of the same efficiency. Execution times may differ in actual measurements, due to differences in computers or the actual way it is implemented. An algorithm that is in a lower class, however, e.g., $\mathcal{O}(\log n)$, will always be faster in the end. No matter how slow the computer is that is running on, there will be an input size on which it will be faster, and it will be faster on all inputs larger than that.