

# Datenstrukturen und Algorithmen

Reto Achermann

March 14, 2010

## Part I Basics

### 1 Algorithms

An algorithm is an effective method to solve systematically a problem using an exactly defined and finite sequence of instructions. Characteristics an algorithm must satisfy:

- Determined: If two inputs are equal then the results must be the same.
- Finite: The number of instructions (lines of code) and the required memory at every point of time is limited.
- Termination: An algorithm terminates after a finite number of steps. This holds for every possible input.

### 2 Random Access Machine (RAM)

In order to make a statement about the estimated running time of an algorithm, we have to define a machine. A RAM has following characteristic operations with a cost of one each. (Entity-Cost-Model)

- read and write a value in to the registers
- add / multiply / compare / divide / subtract / etc

### 3 Landau Symbols

Landau symbols are used to describe the asymptotic performance of a function or the running time depended on the dimension of the input. This is a measurement for the required elementary steps, the complexity of a problem. The Landau notation allows its users to simplify functions in order to concentrate on their growth rate.

#### 3.1 Big-O Notation

$$f \in \mathcal{O}(g) := \exists c > 0 \exists n_0 \forall n \leq n_0 : |f(n)| \leq c \cdot |g(n)|$$

In the worst case,  $f$  is bounded above by  $g$ . Therefore the asymptotic runtime of  $f$  is not greater than  $g$  for  $\forall n > n_0$ . Hence we can say that  $f(n) = \mathcal{O}(n^2)$ .

#### 3.2 $\Omega$ - Notation

$$f \in \Omega(g) := \exists c > 0 \exists n_0 \forall n \geq n_0 : |f(n)| \geq c \cdot |g(n)|$$

We can estimate the minimum runtime (best case) and say  $f$  is bounded below  $g$ . It follows that  $f \in \Omega(g) \Leftrightarrow g \in \mathcal{O}(f)$

#### 3.3 $\Theta$ - Notation

$$f \in \Theta(g) := \exists c_0 > 0 \exists c_1 > 0 \exists n_0 \forall n > n_0 : c_0 \cdot |g(n)| \leq |f(n)| \leq c_1 \cdot |g(n)|$$

If  $f \in \mathcal{O}(g)$  and  $f \in \Omega(g)$  then we can say that  $f$  growth asymptotically like  $g$  in every case.

#### 3.4 Runtime Classes

If we have a given function  $f = n^3 + 3n^2 + 3$  then simply say  $f \in \mathcal{O}(n^3)$ . The most important functions for measuring efficiency of algorithms are the following:

1. logarithmic:  $f \in \mathcal{O}(\log n)$
2. linear:  $f \in \mathcal{O}(n)$
3. n-log-n:  $f \in \mathcal{O}(n \log n)$
4. polynomial:  $f \in \mathcal{O}(n^a)$
5. exponential:  $f \in \mathcal{O}(2^n)$
6. factorial:  $f \in \mathcal{O}(n!)$

## 4 Basic Tools

In order to solve some basic problems we can use some of the following tools.

### 4.1 Prefix Sum

The first step of this method is calculating each sum from position 1 to  $n$  and storing it in a array  $s$ . Assume you have to figure out the maximum sum of a sequence in an array. If you first computed the prefix sums you can simply calculate the sum from  $i$  to  $j$  by subtracting:  $sum(i, j) = s(j) - s(i)$ .

Calculating prefix sums costs  $(O)(n)$  additions and  $(O)(n)$  extra storage.

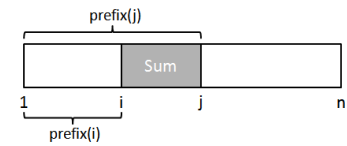


Figure 1: Prefix Sum Scheme

### 4.2 Divide-and-Conquer

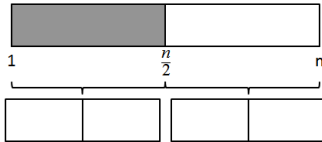


Figure 2: Divide and Conquer

Another way to solve a problem efficiently is by dividing a given set of data in to two subsets. Let  $n = 2^k$ . The input data can now be divided  $k$  times in two equal sized sub arrays. One can now solve the problem recursively which can easily be parallelized. However, solving a problem recursive may cause massive overhead.

There are  $\log n$  recursive calls unless you have reached the bottom line.

### 4.3 Induction

Algorithms based in induction execute the given instructions up on a given position of data  $k \in [1..n]$ . In order to proceed they take the next piece of data  $k + 1$  and extend the current position by one  $k = k + 1$ . Repeat this iteration unless  $k$  reaches the maximum of data  $n$ .

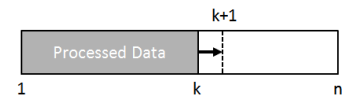


Figure 3: Induction Scheme

### 4.4 Scan Line Principle

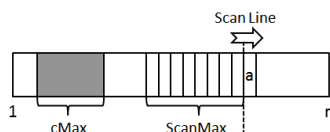


Figure 4: Scan Line Scheme

In order to get a result out of a linear sequence  $Q$ , we can use a so called scanning principle. First of all we initialize two variables  $cMax$  for the current maximum and  $ScanMax$  for the scanned maximum. Now for every element in  $Q$  check if  $ScanMax + a > 0$  then add these two values or else set  $ScanMax = 0$ . If  $max(scanMax, cMax) > cMax$  then update  $cMax$ . Because we are touching every element in  $Q$  exactly once, we get a running time of  $\Theta(n)$ .

### 4.5 Dynamic Programming

To solve a problem with dynamic programming we just build a data field. Rows and columns are labeled with two sequences of  $n$  and  $z$  elements. In the data fields we write the matches between the rows and the columns. The next step is finding a path through this data field to get the result (bottom up). If there is no such path there is no solution for the given input.

The dimension of the  $nxz$  data field results that we have to take  $\Theta(n \cdot z)$  steps to fill up the whole field and another  $O(n \cdot z)$  steps to get the solution out of the array.

		1	2	3	4	5	6
1	X	1	2	3	4	5	6
1	2	0	1	0	0	0	0
4	0	0	0	0	1	0	0
z	6	0	0	0	0	0	1

Figure 5: Data field

## 5 Greedy Algorithm and the Knapsack Problem

Facing a problem of combinatorial optimization: Each piece has a value  $v$  and a weight  $w$ . The task is now to select some pieces so that the sum  $w_i$  is less than a bound  $W$  and the sum of values is maximized.

### 5.1 Greedy Algorithm

We can now calculate the specific value  $v_i/w_i$  and fill up the knapsack sorted by this specific value until the bound is reached. This solution may be very much further than optimal.

### 5.2 Exact Solution

We can solve the problem dynamically with using a data field. The columns is labeled by the sequence from 1 to  $w$  and the rows from 1 to  $i$ . The data is calculated by this formula.

		1	2	3	4	...	W
1	X	1	2	3	4	...	W
1	0	5	5	5	5	5	5
2	3	5	8	8	8	8	8
⋮	3	5	8	9	9	9	12
n	3	5	8	9	10	10	12

Figure 6: Backtracking of exact solution

$$\text{maxValue}(i, w) = \begin{cases} v_1 & \text{if } i = 1 \text{ and } w_1 \leq w \\ 0 & \text{if } i = 1 \text{ and } w_1 > w \\ \text{maxValue}(i-1, w) & \text{if } i > 1 \text{ and } w_i > w \\ \text{maxValue}(i-1, w - w_i) + v_i & \text{if } i > 1 \text{ and } w_i \leq w \end{cases}$$

The last field is the result and we can find a the result by moving up and  $w_i$  left. Because we have to fill out the whole data field, the resulting running time is  $\Theta(n \cdot W)$ .

### 5.3 Approximative Solution (FPTAS)

Assuming we just want the 99% solution, we can approximate and get a faster solution. We reduce the number of values by a constant factor  $K$

$$\tilde{v}_i = \lfloor \frac{v_i}{K} \rfloor \quad \text{e.g. } v_i = \{1, 2, 3, 4, 5\} \text{ and } K = 10 \Rightarrow \tilde{v}_i = \{0\}$$

We build a new data field with columns from 1 to  $\tilde{V} = n \cdot \lfloor \frac{V_{max}}{K} \rfloor$  and rows from 1 to  $i \in \tilde{I}$ . Choose first an  $\epsilon > 0$  in order to get 99% of the ideal solution. Then calculate  $K = \frac{\epsilon \cdot V_{max}}{n}$

Analysing the running time we see that this depends on the size of the data field:  $\mathcal{O}(n \cdot n \lfloor \frac{V_{max}}{K} \rfloor) = \mathcal{O}(\frac{n^3}{\epsilon})$  (TODO: Extend)

## Part II

# Such Algorithmen

## 6 Linear Search

Using a linear search algorithm is the simplest way to find the first occurrence of a value  $k$  in an array  $a$  by walking through the array from the beginning until the end or the key value. Straight forward we get a maximum running time of  $\mathcal{O}(n)$  if the demanded value is at the end of the array and a minimum running time of  $\Omega(1)$  if the element is at the beginning. of  $\frac{n}{2}$ .

Runningtime		Memory	
Worst Case:	$\mathcal{O}(n)$	Extra Storage	0
Average Case:	$\mathcal{O}(n)$	Datastructure	Array
Best Case:	$\mathcal{O}(1)$		

In a randomized array we get an average running time

---

### Algorithm 1 Linear Search

---

```

1: procedure LINEARSEARCH(k, a)
2:   keyFound := false
3:   for  $i := 1$  to length(a) or keyFound = true do
4:     if  $a(i) = k$  then
5:       keyFound := true
6:     end if
7:   end for
8:   if keyFound = true then
9:     return  $i$ 
10:  else
11:    return  $-1$ 
12:  end if
13: end procedure

```

---

## 7 Binary Search

Using a binary search algorithm requires that the underlying datastructure is an ordered list. First we pick the middle element of the array  $a$  and compare its value with the sought one. If they are equal we are done. Otherwise if the sought value is greater we jump to the middle of the upper half or if smaller to the middle of the lower half and repeat the whole

Runningtime		Memory	
Worst Case:	$\mathcal{O}(\log n)$	Extra Storage	$\mathcal{O}(1)$
Average Case:	$\mathcal{O}(\log n)$	Datastructure	Array
Best Case:	$\mathcal{O}(1)$		

procedure again until the sought value found.

---

**Algorithm 2** Binary Search (iterative)

---

```
1: procedure BINARYSEARCH(k, a)
2:   min := 1
3:   max := length(a)
4:   repeat
5:     mid := (min+max) div 2
6:     if x > a(mid) then
7:       min := mid + 1
8:     else
9:       max := mid - 1
10:    end if
11:  until (a(mid) = k) or (min > max)
12: end procedure
```

---

## Part III

# Sortier Algorithmen

## 8 Insertion Sort (Suchen durch Einfügen)

## 9 Odd-Even-Sort

## 10 Heap Sort

## 11 Selection Sort (Sortieren durch Auswählen)

## 12 Bubblesort

### 12.1 Funktionsweise

asdfsdfasdfsdf asdfsdf asdfsdf asdfsdf

### 12.2 Performance

Laufzeit		Speicher	
Worst Case:	$\mathcal{O}(n^2)$	Extra Platz	$\mathcal{O}(1)$
Average Case:	$\mathcal{O}(n^2)$	Datenstruktur	Array
Best Case:	$\mathcal{O}(n)$		

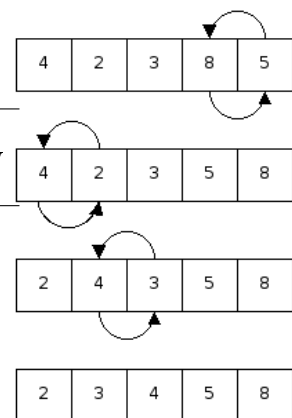


Figure 7: Bubble Sort Scheme

### 12.3 Pseudo Code Implementation

```
procedure bubbleSort( A : list of sortable items ) defined as:
do
  swapped := false
  for each i in 0 to length(A) - 2 inclusive do:
    if A[i] > A[i+1] then
      swap( A[i], A[i+1] )
      swapped := true
    end if
  end for
  while swapped
end procedure
```

## 13 Quicksort

## 14 Polyphase Merge Sort

## 15 Radix Sort

## 16 Topological Sort