

Datenstrukturen und Algorithmen

Reto Achermann <acreto@student.ethz.ch>
Christian Reiter <creiter@student.ethz.ch>

March 24, 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 Ω - 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 Θ - 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: $f \in \mathcal{O}(n!)$

3.5 Analysis

If you would like to check whether a function is in a specific class you can use these inequalities:

$$f \in \mathcal{O}(g(n)) \Leftrightarrow 0 \leq \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} < \infty$$

$$f \in \Omega(g(n)) \Leftrightarrow 0 < \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} \leq \infty$$

$$f \in \Theta(g(n)) \Leftrightarrow 0 < \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} < \infty$$

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 an 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 $\mathcal{O}(n)$ additions and $\mathcal{O}(n)$ extra storage.

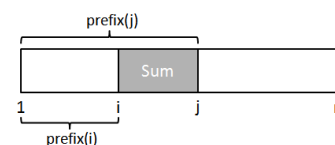


Figure 1: Prefix Sum Scheme

4.2 Divide-and-Conquer

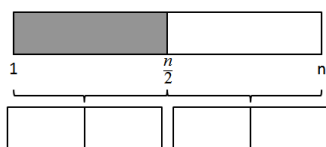


Figure 2: Divide and Conquer

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

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

4.3 Induction

Algorithms based on induction execute the given instructions up to 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 .

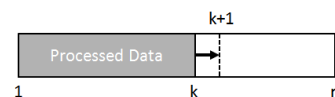


Figure 3: Induction Scheme

4.4 Scan Line Principle

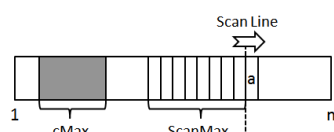


Figure 4: Scan Line Scheme

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 field, 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 $n \times z$ data field results that we have to take $\Theta(n \cdot z)$ steps to fill up the whole field and another $\mathcal{O}(n \cdot z)$ steps to get the solution out of the array.

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

Figure 5: Data field

5 The Knapsack Problem

5.1 Greedy Algorithm

Facing a problem of combinatorial optimization: Imagine you found a treasure and you want to make maximum profit, but you've just a small car with a weight limit of W . So how do you get a combination of

goods with a maximized amount of worth? Each piece a_i has two attributes: a values $v(a_i)$ and a weight $w(a_i)$. Now we say, we want the most valuable pieces with the less weight. We introduce a specific value $q(a_i) = v_i/w_i$. Up on this rating, we order the goods and take the maximum possible until the limit W is reached.

This solution may be very much further than optimal. Imagine, your first piece fill out 51% of your car, and the other two are worth twice the best one.

5.2 PTAS (optimal solution)

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

Figure 6: Backtracking

We can solve the problem using dynamic programming as you learned in the section *Basic Tools*. We have two opportunities to proceed. First by minimizing the weight for a specific (exact) value or by maximizing the value for a specific weight. We take the second for the PTAS solution) So we build up a data field. The columns are labeled by the sequence from 1 to the maximum possible weight W and the rows from 1 to i .

The entries in the data field gives use the maximum value using i things and a maximum weight W . We can fill out the fields by using this recursive formula:

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

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 FPTAS (aproximative solution)

X	1	2	3	4	...	\tilde{V}
1	∞	∞	3	∞	∞	∞
2	0	2	3	∞	∞	4
3	0	2	2	5	∞	6
...	0	1	2	5	9	10
n	0	1	2	5	9	12

Figure 7: FPTAS Scheme

Assuming we just want the 99% solution, we can approximate and get a faster solution. The quality of an approximative solution is given by following formula:

$$\begin{aligned} f_{\Pi}(I, s) &\leq (1 + \epsilon) \cdot OPT & \text{if } \Pi \text{ is a minimization problem} \\ f_{\Pi}(I, s) &\geq (1 - \epsilon) \cdot OPT & \text{if } \Pi \text{ is a maximization problem} \end{aligned}$$

We see that we get close to the OPT solution up to a fixed $\epsilon > 0$. For this we use use the minimum weight for value modus. The idea: We reduce the values of each object by a constant factor K .

$$\tilde{v}_i = \lfloor \frac{v_i}{K} \rfloor \quad K = \frac{\epsilon \cdot V_{max}}{n} \quad \tilde{V}_{max} = \lfloor \frac{n \cdot V_{max}}{K} \rfloor$$

With the total number of objects n and the maximum value V_{max} occuring. And running the dynamic programming algorithm.

$$A[i][j] = \begin{cases} A[i - 1][j] & \text{if } v_i > j \\ \min(A[i - 1][j], w_i + A[i - 1][j - v_i]) & \text{if } v_i \leq j \end{cases}$$

5.4 Analysis

In fact finding a solution by calculating Analysing the running time we see that this depends on the size of the data field: $\mathcal{O}(n \cdot n \lfloor \frac{V_{max}}{K} \rfloor) = \mathcal{O}(\frac{n^3}{\epsilon})$. We gete a running time which is polynomial in n as well as in ϵ .

Algorithm 1 Knapsack FPTAS

```
1: procedure KNAPSACK( $n, \tilde{v}[], w[], W_{max}, \tilde{V}_{max}$ )
2:   for  $i := 1$  to  $n$  do                                     ▷ initialize first column of A
3:      $A(i)(0) := 0$ 
4:   end for
5:   for  $j := 1$  to  $V_{max}$  do                                   ▷ initialize first row of A
6:     if  $j = \tilde{v}_i$  then
7:        $A[1][j] := w_i$ 
8:     else
9:        $A[1][j] := \infty$ 
10:    end if
11:  end for
12:  for  $i := 2$  to  $n$  do                                       ▷ fill matrix with recursive
13:    for  $j := 1$  to  $V$  do
14:      if  $\tilde{v}_i > j$  then
15:         $A[i][j] := A[i-1][j]$ 
16:      else
17:         $A[i][j] := \min(A[i-1][j], w_i + A[i-1][j - w_i])$ 
18:      end if
19:    end for
20:  end for
21:   $j := V_{max}; found := 0$                                      ▷ find maximum profit
22:  while  $found = 0$  do
23:    if  $A[n][j] \leq W_{max}$  then
24:       $found := 1$ 
25:    else
26:       $j := j - 1$ 
27:    end if
28:  end while
29:  for  $i := n$  to  $2$  do                                       ▷ find set that gives max profit
30:    if  $A[i][j] < A[i-1][j]$  then
31:       $Result[next] := i$ 
32:       $j := j - \tilde{v}_i$ 
33:    end if
34:  end for
35:  if  $i = \tilde{v}_1$  then                                         ▷ Add first element if possible
36:     $Result[next] := 1$ 
37:  end if
38: end procedure
```

Part II

Search Algorithms

1 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. In a randomized array we get an average running time of $\frac{n}{2}$.

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

Algorithm 2 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
```

2 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 procedure again until the sought value found.

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

Algorithm 3 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
```

3 Search by Interpolation

If you are looking for a telephone number of a person starting with P you do not jump to the middle of the telephonebook and then to the middle of the second part. We interpolate the approximate position by selecting the indicated letter on the side of the book.

We can estimate a possible position of a certain value in the array by interpolation from the minimum value to the maximum value. If the key at the estimated position is the one we found, we're finish otherwise we start like binary search from this position.

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

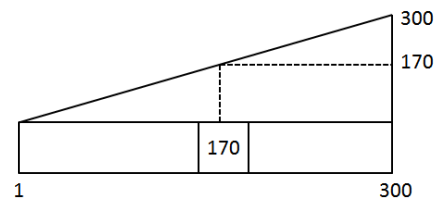


Figure 8: Interpolation Search Scheme

$$Position = \frac{key - A(left)}{A(right) - A(left)}(right - left)$$

Algorithm 4 Interpolation Search

```

1: procedure INTERPOLATIONSEARCH( $k, A$ )
2:    $left := 0, right := length(A)$ 
3:   while  $k > A[left]$  and  $k < A[right]$  do
4:      $diff = A[right] - A[left]$ 
5:      $pos = left + floor((right - left) * (key - A[left]/diffElem))$ 
6:     if  $k > A[pos]$  then
7:        $left = pos + 1$ 
8:     else if  $key < A[pos]$  then
9:        $right = pos - 1$ 
10:    else
11:      return  $pos$ 
12:    end if
13:  end while
14:  return  $-1$ 
15: end procedure

```

▷ Returns -1 if key not found

Part III

Sort Algorithms

1 Selection Sort

Sorting by selection is based on induction. Assuming we have an already sorted part from 1 to $i - 1$ in an array A . The next step is to figure out what is the correct value for position i . We walk through the array from i to n and find the minimum key. Then we swap position i with the position of the minimum key. Then we increase i and repeat the steps until we reach the end of the array. On each step there is only a constant exchange of element needed. Where as on each step the whole rest of the array needs to be scanned. We can divide the array into two lists and solve the problem by conquer.

Runningtime		Memory	
Worst Case:	$\mathcal{O}(n^2)$	Extra Storage	∞
Average Case:	$\mathcal{O}(n^2)$	Datastructure	Array
Best Case:	$\mathcal{O}(n^2)$	Space Complexity	$\mathcal{O}(n)$

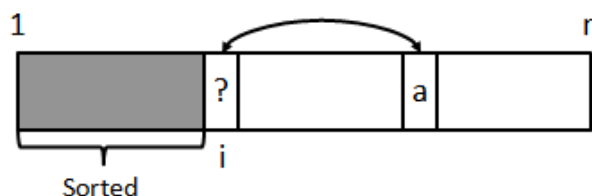


Figure 9: Selection Sort Scheme

Algorithm 5 Selection Sort

```

1: procedure SELECTIONSORT( $A$ )
2:    $n := \text{lenght}(A)$ 
3:   for  $i := 1$  to  $n$  do
4:      $min := i$ 
5:     for  $j := i$  to  $n$  do ▷ find the minimum value of the rest
6:       if  $A[j] < A[min]$  then
7:          $min := j$ 
8:       end if
9:        $swap(i, min)$  ▷ swap the minimum value with the current one
10:    end for
11:  end for
12: end procedure

```

2 Insertion Sort

In comparison to selection sort, this sort algorithm takes one after the other element in array A and moves it to the correct position. This happens by storing the current *value* temporarily and moving the already sorted elements right until *value* is not smaller than the current element anymore. In the end copying *value* at the current position. Analyzing the running time, we see that have to compare each element of A with each other element which gives us a maximum complexity of $\Theta(n^2)$ for key comparisons. Looking at extra memory needed, see that there are only 3 additional fields, wich means a constant. But this algorithm is poor in the number of element movements: In the worst case we have to switch every element by $i + 1$ wich also gives a $\Theta(n^2)$.

Runningtime		Memory	
Worst Case:	$\mathcal{O}(n^2)$	Extra Storage	$\mathcal{O}(1)$
Average Case:	$\mathcal{O}(n^2)$	Datastructure	Array
Best Case:	$\mathcal{O}(n)$	Space Complexity	$\mathcal{O}(n^2)$

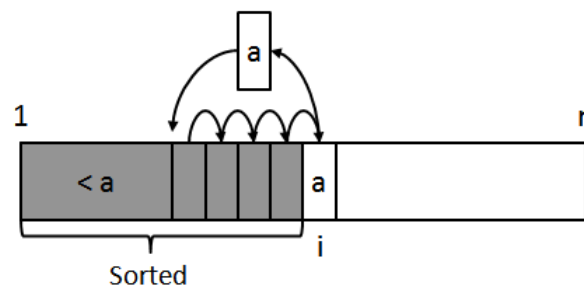


Figure 10: Insertion Sort Scheme

3 Bubble Sort

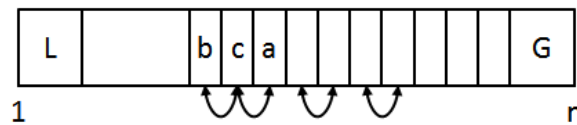
This simple sort algorithm works as follows: Go through the list and swap the two neighbour elements if the left one is greater than the right one and proceed to the next element and do the same. If you reached the end, iterate through the list again and again until there are no swappings of elements in the list. With each iteration we can verify that the greatest element of the step is at the very right side. Generally smaller elements are moving left and greater elements are moving right.

Algorithm 6 Insertion Sort

```
1: procedure INSERTIONSORT( $A$ )
2:   for  $i := 2$  to  $length(A)$  do
3:      $value := A[i]$ 
4:      $j := i$ 
5:     while  $A[j - 1] > value$  do
6:        $A[j] := A[j - 1]$  ▷ move elements right
7:        $j := j - 1$ 
8:     end while
9:      $A[j] := value$  ▷ insert value
10:  end for
11: end procedure
```

With each iteration we do not have to walk to the end of the array, we can stop at the $n - (i - 1)$ th position. This algorithm can be used to simply verify that a list is already sorted. Because there we have to start n times from the beginning of the list and go upto the maximum we get a running time of $\mathcal{O}(n)$. In the best case, which means the list is already sorted, we have to go through the list exactly one times which gives us a $\Omega(n)$.

Runningtime		Memory	
Worst Case:	$\mathcal{O}(n^2)$	Extra Storage	$\mathcal{O}(1)$
Average Case:	$\mathcal{O}(n^2)$	Datastructure	Array
Best Case:	$\mathcal{O}(n)$	Space Complexity	$\mathcal{O}(n^2)$



Algorithm 7 Bubblesort

```
1: procedure BUBBLESORT( $A$ )
2:    $notSwapped := true$ 
3:   repeat
4:     for  $i := 1$  to  $length(A) - 1$  do
5:       if  $A(i) > A(i + 1)$  then
6:          $swap(i, i + 1)$ 
7:          $notSwapped := false$ 
8:       end if
9:     end for
10:  until  $notSwapped$ 
11: end procedure
```

Figure 11: Bubble Sort Scheme

3.1 Odd-Even Sort

If we modify this algorithm by parallelisation, we can divide the whole operation into two steps:

1. We let each processor compare an odd position
2. We let each processor compare an even position

4 Merge Sort

The name of this algorithm comes from its operator mode. Merge Sort follows like Quicksort the divide and conquer principle. The sorting process works by merging two already sorted lists into a new sorted one. The merging process compares in each iteration the two smallest elements of each list and then putting the smaller one in the new list and then repeat.

The function merge is the part where the real work is done and it is more or less a finger exercise to implement it. But now we can ask if it is possible to omit the recursion. The answer is yes, we can.

4.1 Straight 2-way merge sort

The new algorithm, which is called straight 2-way merge sort works by looking each number as a list with

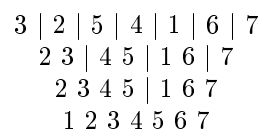


Figure 12: Merge Sort Scheme

Algorithm 8 Merge Sort TODO VERIFY

```
1: procedure MERGESORT( $A, left, right$ )
2:   if  $l < r$  then
3:      $middle := (l + r)/2$  ▷ find the middle
4:      $mergesort(A, left, middle)$ 
5:      $mergesort(A, middle + 1, right)$ 
6:      $merge(A, left, middle, right)$ 
7:   end if
8: end procedure
```

one element and then merging two adjacent lists in each iteration.

4.2 Natural 2-way merge sort

Now we can even improve this algorithm to the so called natural 2-way merge sort. The clue is to use already sorted lists. In our above example we can do better.

```
3 | 2 5 | 4 | 1 6 7
2 3 5 | 1 4 6 7
1 2 3 4 5 6 7
```

5 Heap Sort

TODO: Insert Text here

6 Quick Sort

Quicksort is a quite fast recursive sort algorithm which follows the divide and conquer principle. Therefore it can be simply parallelized. The benefits of this algorithm is the inner loop, which can be efficiently implemented. There exists different versions of quicksort implementations.

Running quicksort, we first pick a pivot element out of the list. Then we reorder the list by moving every element smaller than then pivot element to the left and every element greater to the right (Partitioning). The last step is running quicksort recursively on each of the two lists.

Storing each pivot element while calling recursively which results a space complexity of $\mathcal{O}(n)$. Analysis of the running time we get an recursive form of $T(n) = \Theta(n) + 2T(\frac{n}{2})$ which calculated gives us a $\mathcal{O}n \log n$ runningtime by average.

7 Comparison

We now have seen many different Algorithms for sorting. But where do they differ and when to use a specific algorithm? First of all we can compare the run times in different cases to get a basic idea where to use a algorithm.

Algorithm	Best	Average	Worst
Selection Sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$
Insertion Sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$
Bubble Sort	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$
Merge Sort	$\Theta(n)$	$\Theta(n \log n)$	$\Theta(n \log n)$
Heap Sort	-1	-1	$\mathcal{O}(n \log n)$
Quick Sort	$\Theta(n \log n)$	$\mathcal{O}(n \log n)$	$\Theta(n^2)$

As you can see **Bubble Sort** is really good in traversing the elements. This can easily be used to determine if a sequence is in order or not and it takes Bubble Sort a maximum of n compares to do so.

Selection Sort is really strong if compares are cheap and moves are expensive.

	Runningtime		Memory
Worst Case:	$\Theta(n^2)$	Extra Storage	$\Theta(n)$
Average Case:	$\mathcal{O}(n \log n)$	Datastructure	Array
Best Case:	$\Theta(n \log n)$	Space Complexity	$\mathcal{O}(n)$

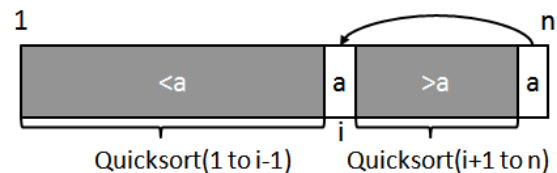


Figure 13: Quicksort Scheme

Algorithm 9 Quicksort TODO VERIFY

```
1: procedure QUICKSORT( $A, left, right$ )
2:   if  $left < right$  then
3:      $pivot := Random(left, right)$  ▷ Select a pivot element by random
4:      $pval := A(pivot)$ 
5:      $swap(pivot, right)$  ▷ move pivot to the right
6:      $index := left$ 
7:     for  $i := left$  to  $right - 1$  do
8:       if  $A(i) \leq pval$  then
9:          $swap(i, index)$ 
10:         $index = index + 1$ 
11:      end if
12:    end for
13:     $swap(index, right)$ 
14:     $quicksort(A, left, p - 1)$ 
15:     $quicksort(A, p + 1, right)$ 
16:  end if
17: end procedure
```

Insertion Sort is fast if it comes down to small data sets.

Quick Sort is, although it has a worst case of $\mathcal{O}(n \log n)$, one of the fastest algorithm (based on experience) for sorting.

Heap Sort is our first algorithm which is even in worst case in $\Theta(n \log n)$. Even though that's pretty good it is in practise slower than a good implementation of Quick Sort but in cases where it is urgent to have a maximum runtime of $\mathcal{O}(n \log n)$ Heap Sort is preferred to Quick Sort.

Merge Sort is another algorithm with a runtime of $\mathcal{O}(n \log n)$ but in comparison to Heap Sort it needs linear extra storage, whereas Heap Sort just needs a constant amount. Another important point is that Merge Sort is stable and Heap Sort is not.

So we now know that there is not a perfect algorithm for sorting. You have to choose one for your needs. But actually nobody forbids you to use more than one of this algorithms. For example you can use Heap Sort for big data sets (i.e Quick Sort would be too deep) than you can use Quick Sort. This technique of using these algorithms together is known as Introsort. You could even go any further and use for the small sets Insertion Sort. This would result in a general purpose Sorting Algorithm which does pretty well in many day by day scenarios.

Part IV

Data Structures

- 1 Arrays
- 2 Linked List
- 3 Heap