

# Introduction to Algorithms and Data Structures

IT University of Copenhagen

7 June 2006

This examination assignment consists of 4 exercises with a total of 12 subexercises. Subexercises are given equal weight in the grading. You have 4 hours to answer all 12 subexercises. Remember to write the page number, your name and your CPR number on each page of your written answer. The complete assignment consists of 13 numbered pages, including this one.

Always read the entire question, before trying to solve it. Write all your answers on solution sheets, not on this printout. Sign all the sheets with your CPR number at least, and *check carefully* whether you are handing in a complete set of pages.

For exercises that ask for efficient algorithms the asymptotic complexity of the specified solution will be taken into account when grading. Exercises asking for time complexity must be answered using  $O$ -notation with the least possible asymptotic growth.

# 1 Algorithm Design

**Question 1.1** The *Dutch flag problem* is to rearrange an array of characters  $R$ ,  $W$ , and  $B$  (red, white, and blue are the colors of the Dutch flag) so that all the  $R$ 's come first, the  $W$ 's come next, and  $B$ 's come last. Design a linear time, constant space algorithm solving this problem. Describe it in natural language (English or Danish) or in pseudocode. What is the lower bound for this problem? Motivate your answer.

**Answer** Count the number of occurrences of letters in the array and generate the array anew, containing letters in the right order. The time needed for counting is linear in the array size. The time needed for generating the ordered array is also linear in the arrays size. The algorithm uses only three additional counters for storing the number of occurrences of each of the letters. The generation of the array can be done in place, as there is no data attached to letters (so the contents of the input array can be destroyed). The lower bound for this problem is  $\Omega(n)$  as one has to read all the array at least once to order elements (there is no cheaper way of figuring out what is the actual contents of the cells, unless more assumptions are made). So the algorithm proposed above is asymptotically optimal.

**Question 1.2** Design an  $O(1)$  space algorithm `HEAP-CHECK`, which checks whether an array  $H[1..n]$  is a min-heap. What is the worst-case running time of your algorithm?

**Answer**

```
HEAP-CHECK( $H, n$ )
1  for  $i \leftarrow 1$  to  $\lfloor \frac{n}{2} \rfloor$ 
2      do if  $H[i] > H[2 * i]$ 
3          then return false
4          if  $2 * i + 1 \leq n$  and  $H[i] > H[2 * i + 1]$ 
5              then return false
6  return true
```

This algorithm runs in  $\Theta(n)$  time (it makes a constant operation for each of the elements in the first half of the array).

Pascal's rule allows computing binomial coefficients  $C_{n,k}$  that are occasionally useful in mathematics. The rule is basically given by the following recurrence for nonnegative  $n$  and  $k$ , such that  $k \leq n$ :

$$\begin{cases} C_{n,k} = C_{n-1,k-1} + C_{n-1,k} \\ C_{n,n} = C_{n,0} = 1 \end{cases}$$

**Question 1.3** Design an efficient algorithm for computing  $C_{n,k}$ , where  $k \leq n$ . Your algorithm should not use more than  $O(nk)$  space. What is the time complexity of your algorithm? **Remark:** You *do not* need to know what is a binomial coefficient and how it is used to answer this question.

**Answer** One should use dynamic programming, which yields an  $O(nk)$  algorithm using  $\Theta(nk)$  space:

BINOMIAL-COEFFICIENT( $n, k$ )

```

1  ▷ Assume both  $n, k \geq 0$  and  $n \leq k$ 
2  create an integer matrix  $C[0 \dots n, 0 \dots k]$ 
3  for  $i \leftarrow 0$  to  $n$ 
4      do  $C[i, 0] \leftarrow 1$ 
5  for  $j \leftarrow 1$  to  $k$ 
6      do  $C[j, j] \leftarrow 1$ 
7  for  $i \leftarrow 2$  to  $n$ 
8      do for  $j \leftarrow 1$  to  $i - 1$ 
9          do  $C[i, j] \leftarrow C[i - 1, j - 1] + C[i - 1, j]$ 
10 return  $C[n, k]$ 

```

## 2 Complexity Analysis

Imagine a variation of binary search trees (BST) called *super trees*. *Super trees* use special insertion and deletion algorithms that maintain the depth of the tree within a constant factor from the optimal (shallowest) regular BST. It is known that any *super tree* is at most 1.51 times deeper than the shallowest BST with the same contents. Otherwise *super trees* use the very same search procedure as BSTs.

Assume that  $n$  is the number of elements stored in a tree. We run BST and *super tree* benchmarks on identical machines, without memory caches, and the entire trees always fit in available memory.

**Question 2.1** For each of the following statements state whether it is correct or not. A sentence is *correct* if you can conclude it from the description above. A sentence is *incorrect* if it contradicts the description or it cannot be concluded from the description due to lack of detailed information. *Do not explain your answers.*

You earn a positive point for each correct answer. Each mistake earns you a negative point. No answer earns no points. If the final amount of points for this question is negative, then it will be rounded up to zero.

- (a) If we take any BST and any *super tree* with the same contents and run the search for the same arbitrary element on both, then the search on BST will always finish first.
- (b) If we take any BST and any *super tree* with the same contents and run the search for the same arbitrary element on both, then the search on the *super tree* will always finish first.
- (c) The slowest search on every BST is slower than the slowest search on a *super tree* with the same contents.
- (d) The slowest search on the shallowest BST is at least as fast as the slowest search on a *super tree* with the same contents.
- (e) There exists a BST such that a sequence of searches for all of its  $n$  elements (in any order) takes more time in total, than the same sequence on any *super tree* with the same contents.

**Answer** (a) incorrect, (b) incorrect, (c) incorrect, (d) correct, (e) correct

Consider a programming environment in which the only available dynamic data structure is a stack. The top element of the stack  $s$  can be removed by calling  $x \leftarrow \text{POP}(s)$ . An element  $x$  can be added to the stack  $s$  by calling  $\text{PUSH}(x, s)$ . Both calls take constant time. The whole stack can also be reversed ( $\text{REVERSE}(s)$ ), making its bottom element to be at the top, and the top element to become the bottom one. The reversal takes  $O(n)$  time, where  $n$  is the number of elements on the stack. Such environments are found in functional programming languages.

Consider an implementation of a FIFO (first-in-first-out) queue in such a setup. One way to achieve it is to maintain two stacks:  $in$  and  $out$ . When an element  $x$  is enqueued we push it on the incoming stack  $in$ . When an element needs to be dequeued then we pop it from the outgoing stack  $out$ . If the outgoing stack  $out$  happens to be empty, then we exchange the stacks and reverse the new outgoing stack.

Here is a possible implementation, assuming that  $in$  and  $out$  are global stack variables, initially empty:

ENQUEUE ( $x$ )

1  $\text{PUSH}(x, in)$

DEQUEUE()

1 **if**  $out$  is empty  
2     **then**  $out \leftarrow in$   
3          $\text{REVERSE}(out)$   
4          $in \leftarrow$  empty stack  
5 **return**  $\text{POP}(out)$

**Question 2.2** Prove that the amortized cost of ENQUEUE and DEQUEUE is  $O(1)$  in the above implementation. Assume that you start with an empty queue.

**Answer** For example by accounting method: assigning a cost of one to each PUSH and POP and a cost  $n$  to a reversal of  $n$  element stack. Now for each ENQUEUE we demand 2 credits, spending one on push and assigning the other one to the element in the *in* stack. With each DEQUEUE we demand a new 1 credit, which is used for the POP operation. If we need to reverse the stack then we use the credit stored in the data structure to pay for it. It is always the case that all elements in *out* stack have credit assigned at this point (as the queue was empty initially). Because of that we never get a negative balance. Ultimately this means that a sequence of  $n$  operations can be run only spending constant time on each operation (amortized).

Consider the following algorithm:

PUZZLE( $A$  :array of real numbers,  $l$  :integer,  $r$  :integer)

```
1  ▷ Assume  $l, r > 0$  and  $l \leq r$ 
2  if  $l = r$ 
3      then return  $A[l]$ 
4   $temp_1 \leftarrow$  PUZZLE( $A, l, \lfloor \frac{l+r}{2} \rfloor$ )
5   $temp_2 \leftarrow$  PUZZLE( $A, \lfloor \frac{l+r}{2} + 1 \rfloor, r$ )
6  if  $temp_1 < temp_2$ 
7      then return  $temp_1$ 
8  else return  $temp_2$ 
```

**Question 2.3** (a) Describe in natural language (English or Danish) the problem solved by this algorithm. (b) Formulate a recurrence describing the cost of running this algorithm on  $n$  element array: PUZZLE( $A, 1, n$ ). Solve the recurrence and give the resulting running time of the algorithm.

**Answer** (a) The above algorithm computes minimum value out of real numbers stored in the array  $A$ , on positions from  $l$  to  $r$ .

(b) The algorithm divides the problem into two parts one roughly half of the size of the original problem. Then it combines the two parts using a constant time operation. If  $T_k$  denotes the cost of running the algorithm on the interval containing  $k$  cells, then the required recurrence is:

$$T_1 = O(1)$$
$$T_k = T_{\lfloor \frac{k}{2} \rfloor} + T_{\lceil \frac{k}{2} \rceil} + O(1) .$$

We could write it as:

$$T_1 = c_1$$
$$T_k = T_{\lfloor \frac{k}{2} \rfloor} + T_{\lceil \frac{k}{2} \rceil} + c_2 ,$$

where  $c_1, c_2 > 0$ .

The inductive hypothesis needs to be strengthened for this recurrence and it could be phrased as:

$$T_k \leq ck - c_2 ,$$

for some positive  $c$  and all  $k \geq 1$ .

Base step:  $T_1 = c_1 < c - c_2$  holds for a sufficiently big value of  $c$  ( $c > c_1 + c_2$ ).

Inductive step:  $T_{k+1} = T_{\lfloor \frac{k+1}{2} \rfloor} + T_{\lceil \frac{k+1}{2} \rceil} + c_2 \leq c \lfloor \frac{k+1}{2} \rfloor - c_2 + c \lceil \frac{k+1}{2} \rceil - c_2 + c_2 = c \lfloor \frac{k+1}{2} \rfloor - c_2 + c \lceil \frac{k+1}{2} \rceil = c(k+1) - c_2$ . By the principle of mathematical induction there exists a positive  $c$  such that  $T_k < ck - c_2 < ck$  for all  $k$ , so the algorithm runs in linear time.

### 3 Search Data Structure

In this problem we consider a new set data structure, that can be used as an alternative to search trees. The data structure for  $n$  elements is a square matrix of size  $\sqrt{n} \times \sqrt{n}$ . Assume that  $\sqrt{n}$  is an integer. Each row in the matrix is sorted in ascending order and each column is sorted in ascending order. See figure for a small example. Note that the smallest element is stored in the top left corner and the largest element is stored in the bottom right corner.

<b>M</b>	1	2	3	4	5
1	2	4	5	7	10
2	3	6	8	15	17
3	14	19	25	32	39
4	18	26	34	41	47
5	21	28	36	45	52

The following is the pseudocode for searching for an element with key  $k$  in the data structure  $M$ . To simplify we assume that we only store the keys and no associated information, i.e. an element consists only of an integer. We denote by  $l = \sqrt{n}$  the number of rows and columns in the matrix.  $M[r, c]$  refers to the entry in matrix  $M$  with row number  $r$  and column number  $c$ .

SEARCH-MATRIX( $M$  : integer matrix,  $k$  : integer) : boolean

```
1   $r \leftarrow l$ 
2  for  $c \leftarrow 1$  to  $l$ 
3      do while  $M[r, c] > k$  AND  $r > 1$ 
4          do  $r \leftarrow r - 1$ 
5          if  $M[r, c] = k$ 
6              then return TRUE
7  return FALSE
```

**Question 3.1** Illustrate a search in the data structure by showing a concrete search for key 15 in the above example data structure.

Describe also in natural language (English or Danish) how the search procedure SEARCH-MATRIX works (in general, not only for the example).

**Answer** When searching for 15 the algorithm compares 15 to 21, 18, 14, 19, 6, 8, 15, and returns TRUE.

The algorithm starts the search in the bottom left corner of the matrix and walks up until an entry smaller than key  $k$  is found, then it walks right until a larger entry than key  $k$  is found. This is repeated until the top limit or right limit of the matrix is reached or until the element is found.

**Question 3.2** What is the time complexity of SEARCH-MATRIX on a data structure containing  $n$  elements? Give the time complexity in  $O$ -notation and argue for the correctness of your answer. Compare the time complexity to the time complexity for searching in a balanced search tree (e.g. an AVL tree) and state which is better.

**Answer** The time complexity is  $O(\sqrt{n})$ , since the algorithm only walks up and to the right in the matrix. Hence there are at most  $\sqrt{n}$  steps up and  $\sqrt{n}$  steps to the right. For each step only one comparison is done, i.e. constant time per step.

The time to search this data structure is worse than searching in an AVL tree, since  $\sqrt{n}$  grows faster than  $\log n$ .

**Question 3.3** Exchange line 3 to 4 in the above code for SEARCH-MATRIX with the following:

```
do  $r \leftarrow \text{BIN-SEARCH}(M, 1, r, c, k)$ 
```

$\text{BIN-SEARCH}(M : \text{integer matrix}, \text{min}, \text{max}, c, k : \text{integer}) : \text{integer}$

```
1  if  $\text{min} = \text{max}$   
2    then return  $\text{min}$   
3   $\text{mid} \leftarrow \lceil (\text{min} + \text{max})/2 \rceil$   
4  if  $M[\text{mid}, c] > k$   
5    then return  $\text{BIN-SEARCH}(M, \text{min}, \text{mid} - 1, c, k)$   
6  else return  $\text{BIN-SEARCH}(M, \text{mid}, \text{max}, c, k)$ 
```

In what way does this change the way the algorithm works? Does this change the worst case time complexity of the algorithm? In that case, what is the new time complexity?

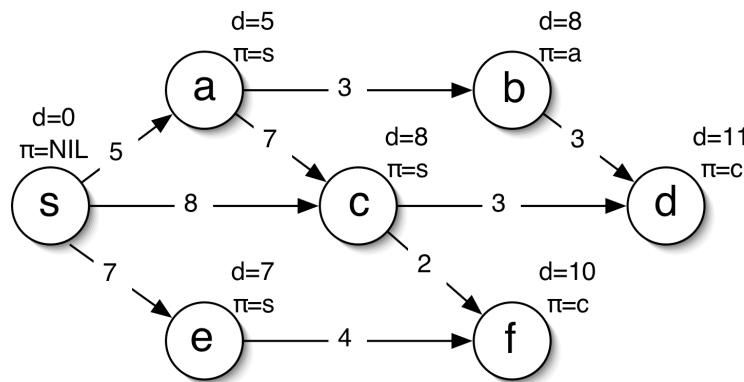
**Answer** The new algorithm does binary search in a column to find the matching element or the place in the column where all elements above are smaller than the key  $k$ .

The time changes. For each column a binary search is performed, which takes  $O(\log \sqrt{n}) = O(\log n)$  in the worst case. The worst case can happen each time, e.g. when the key one search for is larger than all elements in the matrix. The total time is  $O(\sqrt{n} \log n)$ , which is worse than just doing linear search.

## 4 Single Source Shortest Path

In this problem weighted graphs with small weights are considered. As usual we denote a graph by  $G = (V, E)$ , where  $V$  is the set of vertices and  $E$  is the set of edges in the graph. The weight of an edge  $e \in E$  is denoted  $w(e)$ .

**Question 4.1** Given the following graph, with edge weights,  $\pi$ -values and  $d$ -values, is this a correct result of running Dijkstra's algorithm on the graph? If no, why not? If yes, is it the only possible result?



**Answer** This is a correct result of running Dijkstra's algorithm on the graph. There is one more possible result, namely that vertex  $d$  has  $\pi$ -value  $b$ . This is because  $b$  and  $c$  have the same  $d$ -value, and hence they were both in the priority queue, with the same priority when  $c$  was returned by the EXTRACT-MIN call, and therefore  $b$  could as well have been returned.

In this problem we will assume that all edge weights in the graph are small positive integers. Let us assume that all weights are integers in the range from 1 to 100.

In Dijkstra's algorithm a priority queue is used to find the next vertex for which the outgoing edges are relaxed. A Fibonacci heap implements a priority queue such that the amortized cost of DECREASE-KEY is  $O(1)$  and EXTRACT-MIN runs in  $O(\log n)$  time, when there are  $n$  elements in the priority queue. Using a Fibonacci heap, Dijkstra's algorithm runs in  $O(V \log V + E)$  time.

In the special case where the number of *different* priorities of the elements represented in the priority queue *at the same time*, is limited to a constant a more efficient priority queue exists. Both DECREASE-KEY and EXTRACT-MIN can in this case be implemented to run in  $O(1)$  time, worst case.

**Question 4.2** Consider using the priority described above in Dijkstra's algorithm. Argue for that at any time during the algorithm only a constant number of different priorities are present in the priority queue. Also give the time complexity of the algorithm when using this more efficient priority queue.

**Answer** The time for the algorithm becomes  $O(V + E)$ , since now the  $V$  EXTRACT-MIN operations take time  $O(V)$  in total instead of  $O(V \log V)$ . To show correctness one has to show that the elements in the priority queue only have a constant number of different priorities at any point of time during the algorithm. Note however that the total number of different priorities during the whole algorithm may be more than a constant. To start with there are two different priorities, 0 for the start vertex and infinity for all other vertices. Say that we are at a point in the algorithm where the last EXTRACT-MIN returned an element with priority  $x$ , i.e. the  $d$ -value to the vertex is  $x$ . No vertex in the priority queue has a smaller  $d$ -value/priority queue. For those elements with  $d$ -value less than infinity we want to show that the  $d$ -values is between  $x$  and  $x + 100$  when 100 is the largest weight of an edge. If this is true, then we only have a constant number of different priorities. A vertex  $b$ 's  $d$ -value changes when an edge  $(a, b)$  is relaxed. An edge  $(a, b)$  is only relaxed if  $a$  has the minimum  $d$ -value, hence  $a$ 's  $d$ -value was at most  $x$  when  $(a, b)$  was relaxed, and therefor  $b$ 's new  $d$ -value can not be more than  $x + 100$ .

**Question 4.3** Describe an implementation of a priority queue with constant time DECREASE-KEY and EXTRACT-MIN operations in this setting where only a constant number of different priorities are present at the same time. Note that the data structure should support that the set of different priorities change over time, as long as it keeps its constant size. Describe both the data structure and the implementation of the two operations. Include correctness and time analysis.

**Answer** Store all different priorities in sorted order in a doubly linked list, PrList. For each entry in the list a doubly linked list of elements with this priority is stored, List( $x$ ) for priority  $x$ . When a list List( $x$ ) becomes empty, the entry  $x$  in PrList is removed from PrList. When a new priority  $y$  not in PrList appears a new element in PrList is inserted in the sorted order and a new list with one element, List( $y$ ), is created. Since there are only a constant number of different priorities, this list has constant length, so finding the right position in sorted order takes constant time.

DECREASE-KEY( $k$ ) removes element  $k$  from its list List( $old$ ), if List( $old$ ) becomes empty, then the element  $old$  in PrList is also removed.  $k$  is inserted in the list List( $new$ ) if this list exists. Finding the correct list takes constant time. If it does not exist a new entry  $new$  is created in PrList and a list List( $new$ ) with one element  $k$  is created.  $O(1)$  time.

EXTRACT-MIN returns the first element in the first list in PrList, and removes the element from the list List().  $O(1)$  time. In the same way as above, an entry in PrList and List() is be removed if necessary.