Algorithms and Data Structures (DAT3/SW3)

Exam Assignments

Manfred Jaeger

12 January 2017

Full name:	
Student number:	
E-mail at student.aau.dk:	

This exam consists of three problems and there are three hours to solve them. When answering the questions in problem 1, mark or fill in the boxes on this paper. Remember also to put your name and your student number on any additional sheets of paper you will use for problems 2 and 3.

- *Read* carefully *the text of each problem before solving it!*
- *For all multiple-choice questions in Problem 1: only one of the candidate solutions is correct, and only one box must be checked.*
- *For problems 2 and 3, it is important that your solutions are presented in a readable form. If you don't have enough time to give full solutions for problems 2 and 3, then give a solution outline in a few lines of text.*
- *Make an effort to use a readable handwriting and to present your solutions neatly.*

[ItoA] refers to T.H. Cormen, Ch. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms* 3rd edition.

During the exam you are allowed to consult books, printed lecture slides, and notes. The use of any kind of electronic devices, including calculators, is not permitted.

Problem 1 [50 points in total]

1. (*8 points*)

1.1. $\lg n^n + n \cdot (5 + 3 \cdot \lg n)$ is:

☐ **a)** $\Theta(n)$ ☐ **b)** $\Theta(2^n)$ ☒ **c)** $\Theta(n \lg n)$ ☐ **d)** $\Theta(n^{\lg n})$

1.2. $(\lg n) \cdot (3 \cdot n^2 + \sqrt{n^6} - 18)$ is:

☐ **a)** $\Theta(n^{2.5})$ ☒ **b)** $\Omega(n^3)$ ☐ **c)** $O(n^3)$ ☐ **d)** $O((\lg n)^3)$

2. (*7 points*)
Let $T(n)$ be characterized by the recurrence

$$T(n) = \begin{cases} \Theta(1) & n = 1 \\ 4T(n/2) + \sqrt{n^3} + n & n > 1 \end{cases}$$

Then $T(n)$ is

☒ **a)** $\Theta(n^2)$ ☐ **b)** $\Theta(n^2 \lg n)$ ☐ **c)** $\Theta(n^{\frac{3}{2}})$ ☐ **d)** $\Theta(n^3 \lg n)$

3. (*7 points*)

For the following algorithm we assume that for a stack S we have available the method $S.peek()$, which returns the top element of S without removing it from S.

input : An array A containing integers strictly greater than 0
output: A stack
StackSort(A)

```
1  S= new empty stack
2  for i=1 to A.length do
3      T = new empty stack
4      for j=1 to A.length do
5          if A[j] ≠ 0 and ( T is empty or T.peek() < A[j]) then
6              T.push(A[j])
7              A[j] = 0
8      U= new empty stack
9      while S, T are not both empty do
10         if S and T are not empty and T.peek() ≥ S.peek() then
11             U.push(T.pop())
12         if S and T are not empty and T.peek() < S.peek() then
13             U.push(S.pop())
14         if S is empty and T is not empty then
15             U.push(T.pop())
16         if T is empty and S is not empty then
17             U.push(S.pop())
18     S= new empty stack
19     while U is not empty do
20         S.push(U.pop())
21 return S
```

Which of the following statements is a valid loop-invariant for the **for** loop of lines 2-20?

[X] **a)** The elements in S are sorted in decreasing order (larger elements on top), and S contains at least $i - 1$ elements.

[ ] **b)** The elements in S are sorted in decreasing order (larger elements on top), and S contains at least $2 \cdot (i - 1)$ elements.

[ ] **c)** The elements in S are sorted in increasing order (smaller elements on top), and S contains at least $i - 1$ elements.

[ ] **d)** The elements in S are sorted in increasing order (smaller elements on top), and S contains at least $2 \cdot (i - 1)$ elements.

4. (*7 points*)

Let A be an integer array of length 9 (indexed 1,...,9), and define $A.heap\text{-}size$=9. Which of the following is a possible configuration of A after executing first MAX-HEAPIFY(A,4), and then MAX-HEAPIFY(A,2)?

☐ **a)**

2	12	7	4	8	9	3	2	5

☒ **b)**

8	10	12	7	2	1	5	4	3

☐ **c)**

20	15	18	12	9	10	7	13	11

☐ **d)**

15	8	10	9	7	3	6	2	1

5. (*7 points*)

Let T be a hash table of size 8. Write into the boxes below the contents of T after the sequence of keys

$$20, 12, 3, 11, 19$$

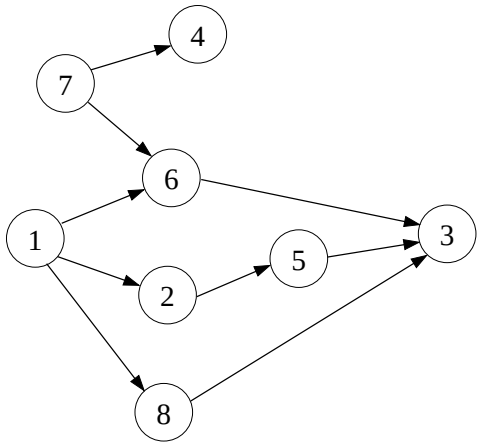
has been inserted using open addressing with the quadratic probing hash function

$$h(k,i) = (k \bmod 8 + \frac{1}{2}(i + i^2)) \bmod 8$$

	19		3	20	12	11	
0	1	2	3	4	5	6	7

6. (*7 points*)

Consider the following graph where nodes are labeled with integer identifiers.



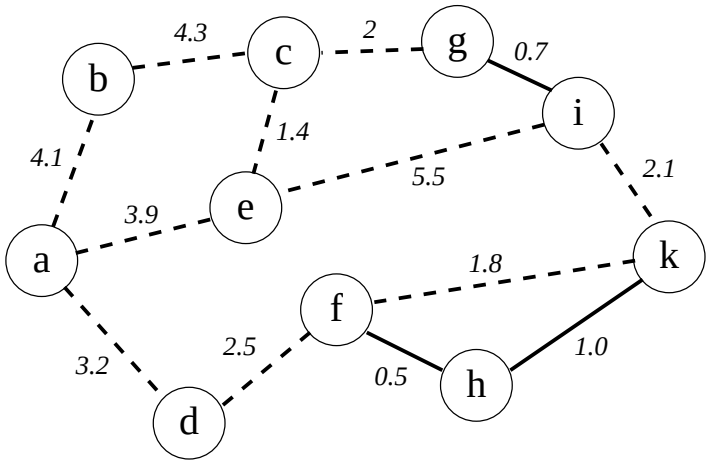
We are performing a TOPOLOGICAL-SORT on this graph ([ItoA p.613]). Assume that both in the enumeration of all nodes in line 1. of DFS, and in the enumeration of adjacent nodes in line 4. of DFS-VISIT, nodes are enumerated in increasing order according to their labels.

Which of the following is the topological sort that will be computed?

☐ **a)** 1,7,6,2,5,8,3,4
☐ **b)** 1,7,2,4,6,8,5,3
☐ **c)** 7,4,1,2,5,6,8,3
☒ **d)** 7,4,1,8,6,2,5,3

7. (*7 points*)

The figure below shows an undirected weighted graph on which Kruskal's Minimum Spanning Tree algorithm ([ItoA], p. 631) is executed. The solid lines are edges that have already been added to the set A. The dotted lines are the remaining edges.



A Write in the boxes below the next three edges that will be added to A by the algorithm, in the order in which they are added:

E - C
C - G
I - K

B Write in the box below the number of iterations of the **for** loop of lines 5-8 that, starting from the configuration shown in the figure, are performed until the third additional edge is added to A (including the iteration in which the third edge is added):

4

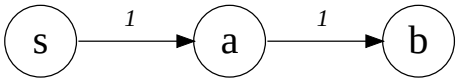
Problem 2 [25 points]

Let G be a weighted directed graph with non-negative edge weights and a distinguished source node s. We assume that in G

1. every node v is reachable by a path from s

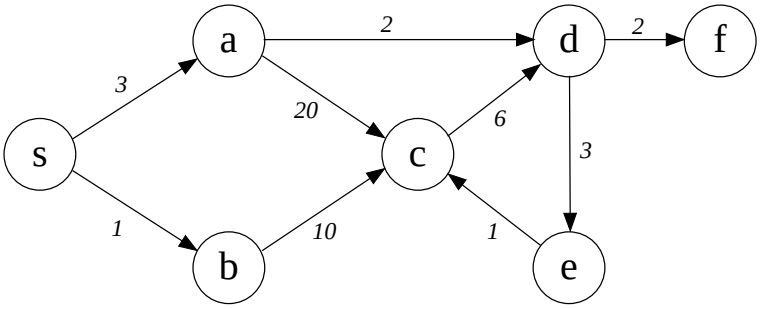
2. shortest paths from s to v are unique, i.e., there do not exist two different shortest paths from s to v.

We define the *Bottleneck Value* of nodes as follows: the bottleneck value $B(u)$ of a node u is equal to the number of nodes v, for which u lies on the shortest path from s to v. Example: in the simple three node graph



we have $B(b) = 1$, because b lies on the shortest path from s to b, $B(a) = 2$, because a lies on the shortest path from s to a, and on the shortest path from s to b, and $B(s) = 3$, because s lies on the shortest paths from s to s, a and b.

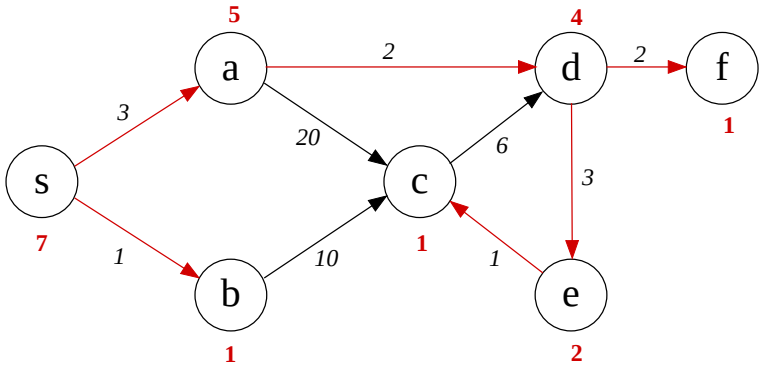
a.) Determine the bottleneck values for all nodes in the following graph:



b.) Describe in pseudo-code an efficient algorithm for computing the bottleneck values of all nodes in a graph G satisfying assumptions 1. and 2. above.

c.) What is the worst-case complexity of your algorithm?

Solution: a.) The red numbers next to the nodes are the bottleneck values:



The red edges are the ones that are used on shortest paths, i.e., these are the edges of the form $(v.\pi, v)$ at the end of Dijkstra's or the Bellman-Ford algorithm.

b. We can consider the shortest path tree consisting of the nodes of G and the edges $(v.\pi, v)$. We then have that $B(v)$ is equal to the number of nodes in the sub-tree of the shortest path tree that is rooted at v. Expressed as a formula, this means:

$$B(v) = 1 + \sum_{u:\, v=u.\pi} B(u).$$

We can quite easily adapt Dijkstra's algorithm to compute the $B(u)$ values. To do this, we first add to the INITIALIZE-SINGLE-SOURCE method [ItoA, p.648] one line to initialize an additional B value for all nodes:

2a $\quad v.B = 0$

We then use Dijkstra's algorithm to compute the π pointers defining the shortest path tree. Since we later on want to propagate the $B(v)$ values up in the tree starting from the leaves, we make sure that we can easily retrieve the nodes in the right order. This is easily done by implementing the set S in Dijkstra's algorithm as a stack. Thus we change two lines of Dijkstra's algorithm [ItoA, p.658]:

2new $\quad$ let S be a new empty stack
6new $\quad\quad$ $S.push(u)$

At the end of Dijkstra's algorithm S contains the nodes in decreasing distance to the source. In other words, a node v always lies above $v.\pi$ in the stack. We can therefore compute $B(v)$ simply by appending the following lines to Dijkstra's algorithm

9 $\quad$ **while** S is not empty
10 $\quad\quad$ $u = S.pop()$
11 $\quad\quad$ $u.B = u.B + 1$
12 $\quad\quad$ $u.\pi.B = u.\pi.B + u.B$

c. The modifications at lines 2 and 6 have no impact on the complexity, and the additional time required by lines 9-12 is $O(|V|)$, which is dominated by the

complexity of the original part of the algorithm in lines 1-8. Using a min-heap implementation of the priority queue, this gives us a complexity of $O(|E|\lg|V|)$.

Problem 3 [25 points]

Lars is a freelance employee of a consulting company WeWillHelpU. Lars can freely decide on which days to work for WeWillHelpU (work is always full days). In order to encourage longer continuous periods of work, WeWillHelpU has the following salary scheme for its freelance employees: for the first day that an employee comes to work after a period of absence, the salary is 100. For the second, third, fourth, ... day of a continuous period of work the salary then is 110, 120, 130, ..., i.e., for each additional day the salary is increased by 10. Precisely, the total salary for a continuous period of k days of work then is

$$S(k) = \sum_{i=1}^{k} (100 + 10(i-1)).$$

Since WeWillHelpU provides support around the clock, Saturday, Sunday and holidays are not treated any different from ordinary working days.

Lars still likes to take days off once in a while. In order to plan his work schedule, he assigns a *vacation value* to each of the next n days. We call n the *planning horizon*. Vacation values represent the benefit obtained from taking the day off. They are measured on the same scale (or currency) as the salary, and depend on factors like special events that Lars might want to attend, the weather forecast, etc. Thus, if Lars assigns a vacation value of 100 to a certain day, that means that for him taking that day off is just as good as earning 100 from work. For example, for a planning horizon of 4 days, Lars might assign the following vacation values for the coming 4 days:

Day 1	105
Day 2	140
Day 3	100
Day 4	115

Given a planning horizon, and vacation values for all days within that horizon, Lars needs to figure out when to work and when to take days off, in order to mazimize his overall benefit, which is measured as the sum of the salary amount and vacation values obtained in the period.

a. Find the optimal schedule for Lars given the planning horizon of 4 days, and the vacation values specified above. What is the benefit obtained with this schedule?

b. Formalize Lars's problem as an optimization problem by giving a precise specification of a solution space and a value function.

c. Describe an efficient algorithm that solves Lars's problem for arbitrary planning horizons n and vacation value tables. The description should be precise, for example using pseudo-code (but a sufficiently precise description using other means also is o.k.). What is the complexity of your algorithm as a function of n?

Solution

a. There are multiple equally good plans, all giving a benefit of 460: work all 4 days, or first taking 2 days of vacation, then work a day, and then vacation again, or taking vacation all the time.

b. The solution space for a work schedule problem given by a planning horizon n and a vacation value table VVT is a mapping that assigns a *work* or *vacation* label to the sequence of the next n days. Formally, a solution s is a function:

$$s : \{1, \ldots, n\} \to \{w, v\}$$

For a solution s, we can first define the *benefit* earned on day i as

$$b(s,i) = \begin{cases} VVT(i) & \text{if } s(i) = v \\ 100 & \text{if } s(i) = w \text{ and } i = 1 \text{ or } s(i-1) = v \\ b(s, i-1) + 10 & \text{if } s(i) = w \text{ and } s(i-1) = w \end{cases}$$

The value of a solution then simply is the sum of day benefits:

$$b(s) = \sum_{i=1}^{n} b(s,i)$$

c. Lars's problem is closely related to the rod cutting problem, and can be solved using a similar dynamic programming approach.

The optimal solutions for Lars's problem have the following optimal substructure property: if $s^{opt}(n, VVT)$ is an optimal solution for the problem given by n and VVT, and s^{opt} starts with $s^{opt}(1) = w$, and j is the first day for which $s^{opt}(j) = v$, then

$$b(s^{opt}) = \sum_{i=1}^{j-1} 100 + 10 \cdot (i-1) + b(s^{opt}(n-j+1, VVT[j \ldots n]))$$

where $VVT[j \ldots n]$ is the vacation value table for days $j, \ldots, n$ in the original problem.

Similarly, if s^{opt} starts with $j-1$ vacation days, followed by a first working day, then we have

$$b(s^{opt}) = \sum_{i=1}^{j-1} VVT(i) + b(s^{opt}(n-j+1, VVT[j \ldots n]))$$

Moreover, we already know that in the first case $s^{opt}(n-j+1, VVT[j \ldots n])$ starts with a v, and in the second case with a w.

We can compute in a bottom-up dynamic programming approach the values $s^{opt}(n-k+1, VVT[k \ldots n])$ for $k = n \ldots 1$ and the two cases that s^{opt} starts with w or v. At the same time we have to remember how long the initial sequence of w's or v's is in the optimal solution. Therefore, we construct 4 arrays of length n:

- *Best_w*, so that $Best\_w[k]$ stores the value of the optimal solution for the subproblem given by $n-k+1$ and $VVT[k \ldots n]$, among all solutions that start with w.
- *Length_w*, so that $Length\_w[k]$ stores the number of initial w's in the optimal solution for the subproblem given by $n-k+1$ and $VVT[k \ldots n]$, among all solutions that start with w.
- *Best_v*, so that $Best\_v[k]$ stores the value of the optimal solution for the subproblem given by $n-k+1$ and $VVT[k \ldots n]$, among all solutions that start with v.
- *Length_v*, so that $Length\_v[k]$ stores the number of initial v's in the optimal solution for the subproblem given by $n-k+1$ and $VVT[k \ldots n]$, among all solutions that start with v.

We then obtain from the above the following computation rules:

$$\begin{aligned}
Best_w[k] &= max_{j=1,\ldots,n-k+1} \textstyle\sum_{i=1}^{j} 100 + 10 \cdot (i-1) + Best_v[k+j] \\
Length_w[k] &= argmax_{j=1,\ldots,n-k+1} \textstyle\sum_{i=1}^{j} 100 + 10 \cdot (i-1) + Best_v[k+j] \\
Best_v[k] &= max_{j=1,\ldots,n-k+1} \textstyle\sum_{i=1}^{j} VVT[n-j+k] + Best_w[k+j] \\
Length_v[k] &= argmax_{j=1,\ldots,n-k+1} \textstyle\sum_{i=1}^{j} 100 + 10 \cdot (i-1) + Best_w[k+j]
\end{aligned}$$

For our example, the filled-in arrays for our small example would be

k	1	2	3	4
VVT	105	140	100	115
$Best\_v$	460	355	215	115
$Length\_v$	2	3	2	1
$Best\_w$	460	330	215	100
$Length\_w$	4	3	1	1

Comparing $Best\_w[1]$ with $Best\_v$ one finds that the two policies of working for 4 days, or taking vacation for 2 days, then work 1 day, and then take vacation again, leads to the optimal benefit of 460. The third policy, taking vacation all 4 days, is not visible from this table. This is because here it is assumed that when $Best\_v$ or $Best\_w$ are achieved by multiple different j, we just pick the first for the *argmax* (any other rule for breaking ties is o.k.).

The complexity of the algorithm is $\Theta(n^2)$: the computation time for calculating the kth entries for the four arrays is $\Theta(n-k)$, and summation over all k (the usual arithmetic series) then gives $\Theta(n^2)$.