DELFT UNIVERSITY OF TECHNOLOGY
Faculty of Electrical Engineering, Mathematics and Computer Science

TI2306 Algorithm Design
Digital test – part 2 of 2

January 29, 2019, 13.30-15.30

- Usage of the book, notes or a calculator during this test is not allowed.
- This digital test contains 3 questions (worth a total of 10 points) contributing 10/20 to your grade for the digital test. The other digital test contributes the other 10/20 to your grade. Note that the final mark for this course also consists for $\frac{2}{3}$ of the unrounded score for the written exam (if both ≥ 5.0).
- The storyline throughout the questions can be ignored: each question can be answered independently of the others and in any order.
- The spec tests in WebLab **do not represent your grade**.
- There is an API specification of the Java Platform (Standard Edition 8) available at `https://weblab.tudelft.nl/java8/api`.
- When an implementation is asked, please provide the **most efficient** implementation in terms of asymptotic time complexity. Providing a suboptimal implementation can lead to a subtraction of points.
- The material for this tests consists of module 3 (Dynamic Programming) and 4 (Network Flow) of the course and the corresponding book chapters and lectures.
- Total number of pages (without this preamble): 4.

Learning goals coverage of the *digital test*–part 2, based on the learning objectives on BS:

Goal	digital test 2018-2019	
Memoization		
Weighted interval scheduling		
Segmented least squares	1	
Subset-sum		
Knapsack	2	
RNA		
Sequence alignment		
Space-efficient alignment		
All-pairs shortest path		
A new DP algorithm	1	
Flows, residual graphs, augmenting paths		
Cuts, capacity of a cut		
Demand and circulation with lower bounds		
(Scaling-)Ford-Fulkerson		
Maximum bipartite matching		
Maximum Edge-Disjoint paths		
Survey Design		
Image Segmentation		
Project Selection	3	
Baseball elimination		
Max-flow min-cut	3	
New network flow model	3	

In general, the digital test is about designing, implementing, applying and analyzing a new algorithm that is similar to one or more of the algorithms listed above.

1. (3 points) A company behind an operating system supplies updates (e.g., with security-related bug fixes) via the Internet. At a certain moment in time, a number of updates are in development, and for each update we know the exact future date when it is ready. In order of these dates, the updates are denoted by their index in the sequence $1, 2, \ldots, n$.

   Shipping an update comes with some constant (and known) costs c. To minimise these shipping costs, the board of directors of the company are asking you to explore the possibility of *bundling* updates. A *bundle* is a series of consecutive updates $i, i+1, \ldots, j$ that is shipped in one go.

   The total costs of a bundle consists of shipping costs and risk-related costs: shipping one complete bundle is equal to the cost of a single update (c). However, postponing security updates until the complete bundle of updates is ready means users are at risk longer. These (estimated) costs of a bundle $i, i+1, \ldots, j$ are given to us, denoted by `costs[i][j]` for all $1 \leq i \leq j \leq n$.[1] The board would like to know how to bundle updates, so that the sum of the costs of all bundles is minimised.

   Implement an efficient *iterative* dynamic programming algorithm to compute the minimal costs of bundling. You are given the following input: the number of updates n, the shipping costs c, and the table `costs` with dimensions $(n+1) \times (n+1)$.

   Solution: This is equivalent to a segmented least squares problem (group items with the trade-off between constant costs and the cost of a group).

   $$OPT(j) = \begin{cases} 0 & \text{if } j = 0 \\ \min_{1 \leq i \leq j} \{c + \texttt{costs[i][j]} + OPT(i-1)\} & \text{otherwise} \end{cases}$$

   The idea is as follows. The value $OPT(j)$ are the minimum costs of bundling updates $1, \ldots, j$ in an optimal way.

   So a one-dimensional array can be used. The value $OPT(j)$ uses $OPT(i)$ with $0 \leq i \leq j-1$, so the algorithm should compute the values in increasing j. There are two loops because for each j, $OPT(j)$ is computed as the minimum over the values $1, \ldots, j$.

   $M \leftarrow$ Array$[0, \ldots, m]$
   $M[0] \leftarrow 0$
   for $j = 1, \ldots, m$ **do**
   $min \leftarrow \infty$
   for $i = 1, \ldots, j$ **do**
   $min \leftarrow \min(min, c + \texttt{costs[i][j]} + M[i-1])$
   end for
   $M[j] \leftarrow min$
   end for
   return $M[m]$

   Rubric:

   - There is an array that is being used to store smaller subproblems that are re-used: 1.0
   - The minimum is taken over $\Theta(n)$ options in the biggest iteration: 1.0
   - The size of the array is $\Theta(n)$: 0.5
   - All answers to test cases are correct: 0.5

2. (3 points) When you present your solution to the board members, one of them explains that the content of updates come from code changes that either resolve bugs or include new features. We call these code changes "jobs". Each job $i \in \{1, 2, \ldots, n\}$ has some (estimated) costs `costs[i]`, representing the number of hours required for a developer, and some (estimated) benefits `benefits[i]` to end-users. There is a dynamic program in place that finds the maximal total benefit of any list of selected jobs given a (monthly) cost budget of C: this program fills an array M with at position $M[i][c]$ for $i \in \{1, 2, \ldots, n\}$

[1] Where `costs[i][j]` $= 0$ for all $i \geq j$.

and $0 \leq c \leq C$ the benefit of the optimal selection of jobs from $\{1, \ldots, i\}$ such that the total costs of the included jobs is not more than the budget c, i.e., this array contains the results of the following recursive function:

$$\mathsf{M}(i,c) = \begin{cases} 0 & \text{if } i < 1 \\ \mathsf{M}(i-1,c) & \text{if } \texttt{costs[i]} > c \\ \max\{\mathsf{M}(i-1,c),\ \texttt{benefits[i]} + \mathsf{M}(i-1, c-\texttt{costs[i]})\} & \text{otherwise} \end{cases}$$

Implement an algorithm that determines which jobs together have the maximum total benefit *using the table* M. As input you get: the number of jobs n, the total cost budget C, the array `benefits` of dimension $n+1$, the array `costs` of dimension $n+1$, and the array `M` filled as described in the text above, with dimensions $(n+1) \times (C+1)$. The algorithm should return a list of the indices of the included jobs *in increasing order*. This algorithm should run in linear time.

Solution: This can be done recursively. When considering updates $1, \ldots, i$, consider whether or not i is included. Call with FIND-SOLUTION(n, C).

```
function FIND-SOLUTION(i, c)
    if i > 0 then
        if M[i, c] = M[i − 1, c] then
            FIND-SOLUTION(i − 1, c)
        else
            FIND-SOLUTION(i − 1, c − costs[i])
            append i to the global list.
        end if
    end if
end function
```

Alternatively,

```
function FIND-SOLUTION(i, c)
    while i > 0 do
        if M[i, c] = M[i − 1, c] then
            i ← i − 1
        else
            append i to the global list
            i ← i − 1
            c ← c − costs[i]
        end if
    end while
    reverse list
end function
```

Rubric:

- Add to list if $M[i][C] > M[i-1][C]$ or if ($costs[i] \leq C$ and $M[i][C] \leq benefits[i] + M[i-1][C - costs[i]]$): 1.5
- Update C correctly: 1.0
- Correct answer on all instances: 0.5
- Deduction if runtime is not $O(n)$: -0.5

3. (4 points) When you bring your solution to the attention of the developers of the operating system, it appears that the board of directors (who gave you the assignment in the first place) completely misunderstood how such a selection of code changes ("jobs") is made:

    1. Jobs are sometimes dependent on each other: there are a number of pairs of jobs (i, j) for which it holds that if i is included, then also j must be included in the new release.

2. There is no exact limit on the budget: when it is clear that the benefits outweigh the costs (e.g., with high-risk security bugs), developers from other departments jump in to help, or implementing all the jobs just takes longer (e.g., than a month).

You are asked to implement an algorithm to find a subset of jobs that maximises the net profit (total benefit minus total costs) under these (new) conditions.

You are again given a list of n jobs, with for each job $1 \leq i \leq n$ the `costs[i]` and the `benefits[i]` in arrays as for question 2. Also now you are given a list of pairs of code changes (i, j) where i is dependent on j. These are represented as a 2D-array `dependencies` of dimension $(n+1) \times (n+1)$ where `dependencies[i][j]` $= 1$ iff i depends on j and 0 otherwise.

Create a flow network and use the given implementation of Ford-Fulkerson to find the subset of jobs with maximal net profit (to include in the next release). Output only the (net) value of this release, i.e., the benefit minus the costs of the included jobs.

Solution: We expect a model like this:

- Add a node for every code change i.
- Add a source and sink node.
- Add an edge from the source to every job with capacity `benefits[i]`.
- Add an edge from every job to the sink with capacity `costs[i]`.
- Add an edge from job i to j if i depends on j, with capacity ∞.

Then find the maximum flow with Ford-Fulkerson. The value of this flow is also the capacity of the minimum cut. This represents the costs of jobs selected and the benefit of jobs not selected for the release. The (net) value can then be computed by taking the sum of the benefits of all jobs and subtracting this capacity (value of max flow).

Rubric:

- Node for every job (code change): 0.5
- Dependencies modeled for right i and j (independent of direction): 0.5
- Dependencies modeled correctly, with capacity $\geq$ max possible flow: 0.5
- Benefits and costs modeled correctly: 1.0
- Iteration over the correct edges in attempt to find the max flow, or identifying selected jobs A in min cut (A,B) correctly: 0.5
- Correctly subtracted flow from sum of all benefits (if benefits on edges from source), or summing profit-costs of jobs in A: 1.0

End of exam questions