ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Final Exam, Algorithms 2013-2014

- You are only allowed to have a handwritten A4 page written on both sides.
- Communication, calculators, cell phones, computers, etc... are not allowed.
- Your explanations should be clear enough and in sufficient detail so that a fellow student can understand it. For example, a description of an algorithm should be so that a fellow student can easily implement the algorithm following the description. In particular, do not only write pseudocode without additional explanation.
- **Do not touch until the start of the exam.**

Good luck!

Name: _____ **N° Sciper:** _____

Problem 1 / 15 points	Problem 2 / 20 points	Problem 3 / 15 points	Problem 4 / 20 points	Problem 5 / 10 points	Problem 6 / 20 points

Total / 100

1 *(15 pts)* **Asymptotics and Recursions.**

Suppose you are choosing between the following five Divide-and-Conquer algorithms:

Algorithm A solves problems of size n by dividing (in constant time) them into two subproblems each of size $n/2$, recursively solving each subproblem, and then combining the solutions in $\Theta(n^3)$ time.

Algorithm B solves problems of size n by dividing (in constant time) them into nine subproblems each of size $n/3$, recursively solving each subproblem, and then combining the solutions in $\Theta(n^2)$ time.

Algorithm C solves problems of size n by dividing (in constant time) them into ten subproblems each of size $n/3$, recursively solving each subproblem, and then combining the solutions in $\Theta(n)$ time.

Algorithm D solves problems of size n by dividing (in constant time) them into eight subproblems each of size $n/2$, recursively solving each subproblem, and then combining the solutions in constant time.

Algorithm E solves problems of size n by dividing (in constant time) them into two subproblems each of size $n-1$, recursively solving each subproblem, and then combining the solutions in constant time.

What are the running times of each of these algorithms (in Θ notation), and which would you choose?

Solution:

Algorithm A has running time proportional to the recursion $T(n) = 2T(n/2) + \Theta(n^3)$ which, by the master theorem, is $\Theta(n^3)$.

Algorithm B has running time proportional to the recursion $T(n) = 9T(n/3) + \Theta(n^2)$ which, by the master theorem, is $\Theta(n^2 \lg n)$.

Algorithm C has running time proportional to the recursion $T(n) = 10T(n/3) + \Theta(n)$ which, by the master theorem, is $\Theta(n^{\log_3 10})$.

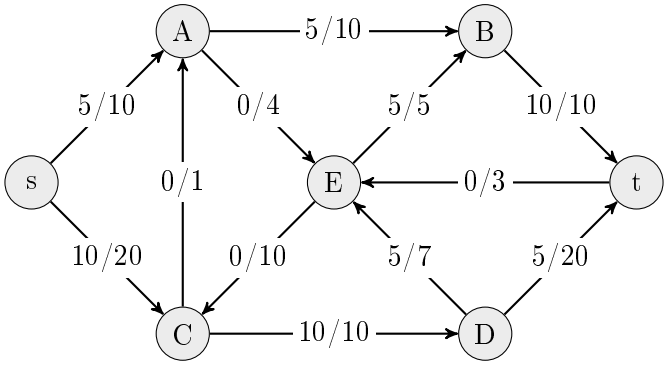
Algorithm D has running time proportional to the recursion $T(n) = 8T(n/2) + \Theta(1)$ which, by the master theorem, is $\Theta(n^3)$.

Algorithm E has running time proportional to the recursion $T(n) = 2T(n-1) + \Theta(1)$ which is $\Theta(2^n)$.

The running time of the algorithms are thus so that $B < C < A = D < E$. Therefore, we choose algorithm B.

(This page is intentionally left blank.)

2 *(20 pts)* **Flows and Cuts.** Assume the following flow network and corresponding flows (the numbers on an edge determine its current flow and its capacity).



2a *(5 pts)* What is the net-flow across the cut $(\{s, A, C, E\}, \{t, B, D\})$? What is the capacity of the same cut?

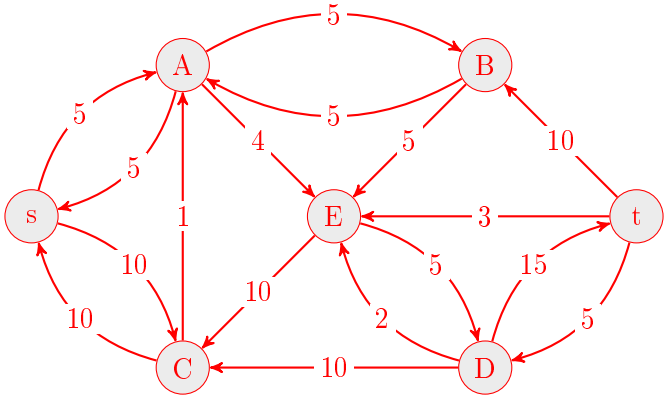
Solution:

- The net flow across the cut is 15.
- The capacity across the cut is 25.

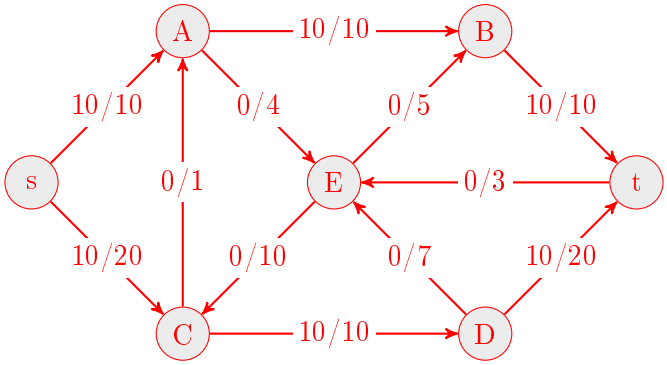
2b *(15 pts)* Starting with the depicted flow, *find a max-flow and a min-cut* by running the Ford-Fulkerson algorithm that in each iteration chooses the fattest augmenting path (the one that can carry the maximum amount of flow). In each iteration draw the residual network and *explain how you found the min-cut.* Finally, *write down the value of the found max-flow and the capacity of the min-cut.*

Solution:

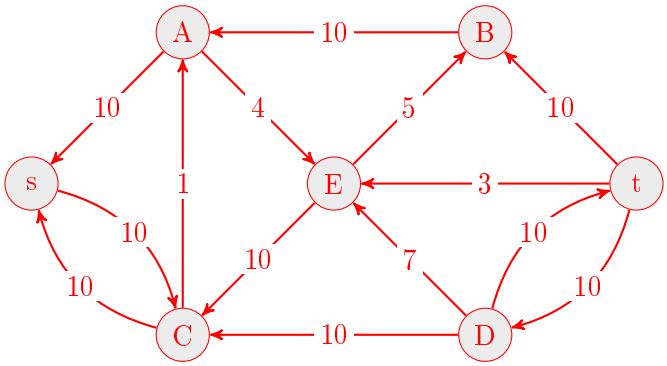
In the first iteration the residual network is:



The fattest path is s, A, B, E, D, t and it has capacity 5. Pushing flow on along this path gives us the flow



The next residual network is



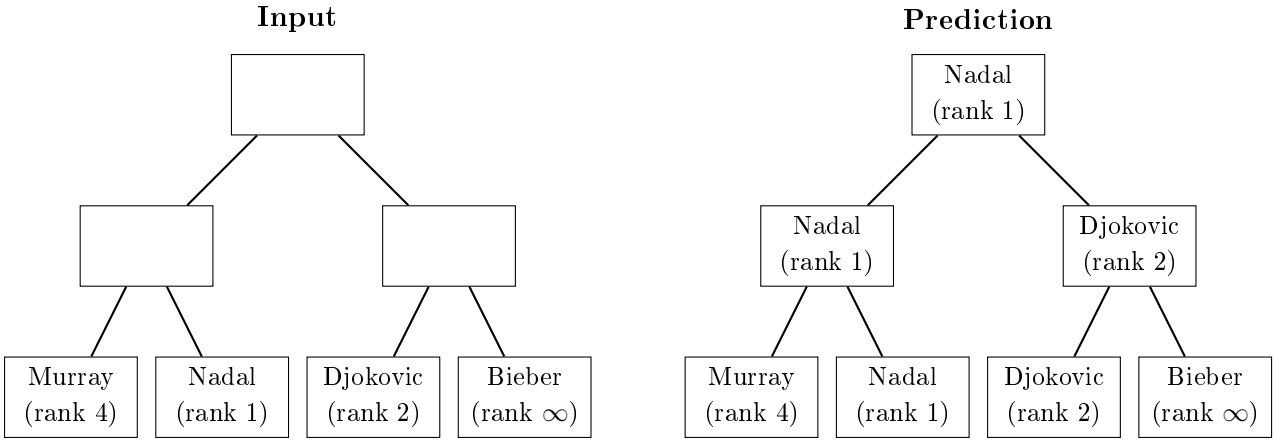
There is no augmenting path as the reachable nodes from the source s is $\{s, C, A, E, B\}$.

The flow has value 20 and the min cut defined by the reachable nodes from the source s has capacity 20.

(This page is intentionally left blank.)

3 *(15 pts)* **Australian Open.** The draw of Australian Open was recently announced. In tennis each match is between two players and the winner progresses to the next round. This naturally leads to a complete binary tree structure of the tournament. At the leaves, we have all the players in the tournament. At the next level, we have those that won their first match, and so on. In particularly, the root of the tree contains the winner of the tournament. We are interested in predicting the outcome of *every match* in Australian Open 2014. To do this we use the following simplifying assumption: a better ranked player always wins over a player with worse rank.

Consider the figure below for an example. We have four players entering the tournament of various rankings. The prediction tree then predicts the winner of each match. For example, as Rafael Nadal is currently ranked number one and Andy Murray is ranked number four, Rafael Nadal wins against Andy Murray. Similarly, we predict that Nadal wins against Djokovic in the final.



Design and analyze an efficient algorithm for the Australian Open prediction problem:

Input: The root of a complete binary tree (the draw) with n players as leaves. Each player/node has a *name* and a *ranking* that is initially empty for nodes that are not leaves. In addition, each node has pointers to its *left child*, its *right child* and *its parent*. Finally, no two players have the same ranking.

Output: A complete binary tree (the prediction) where each node contains the player (his name and rank) that has reached this stage assuming that better ranked players always win over worse ranked players.

Your algorithm should run in *linear time* in the number of players.

Solution:

We shall do an algorithm that is similar to DFS and inspired by Divide-and-Conquer. When we visit a node/player p we first calculate his left subtree rooted at $p.left$ and then his right subtree rooted at $p.right$. After we have calculated both subtrees, we compare the two winning players of them and update p accordingly:

- If $p.left.rank < p.right.rank$ then (left player is winning) set $p.rank = p.left.rank$ and $p.name = p.left.name$
- Else (right player is winning) set $p.rank = p.right.rank$ and $p.name = p.right.name$

The pseudocode of the algorithm PREDICT-TOURNAMENT that takes the root of the tournament as input is as follows:

```
PREDICT-TOURNAMENT(p)
1. if p.left ≠ NIL and p.right ≠ NIL
2.       PREDICT-TOURNAMENT(p.left)
3.       PREDICT-TOURNAMENT(p.right)
4.       if p.left.rank < p.right.rank
5.             p.rank = p.left.rank
6.             p.name = p.left.name
7.       else
8.             p.rank = p.right.rank
9.             p.name = p.right.name
```

Runtime Analyzes: Note that the body of the algorithm takes $\Theta(1)$ time and is executed exactly once for each node in the complete binary tree. As the complete binary tree has n leaves, the total number of nodes it has is $\sum_{i=0}^{\log_2 n} n/2^i$ which is less than $2n$ and greater than n. Therefore the total running time of the algorithm is $\Theta(n)$.

4 *(20 pts)* **Probabilistic analysis**. We shall analyze a randomized procedure RANDOMIZED-SELECT for the select problem: given an array A consisting of n unique integers and an integer $k \leq n$, output the kth smallest integer of A.

For example, if the input is $A =$ | 89 | 14 | 16 | 28 | 51 | 25 | and $k = 3$ then the correct output is 25.

To simplify the description of RANDOMIZED-SELECT, we let $|A|$ denote the length of the array. The pseudocode is as follows:

```
RANDOMIZED-SELECT(A, k)
1. Pick pivot uniformly at random from the numbers in A.
2. Compare each number in A with pivot to obtain arrays S and L:
       S contains all numbers of A strictly smaller than pivot.
       L contains all numbers of A strictly larger than pivot.
3. if |S| = k − 1
4.       return pivot
5. else if |S| ≥ k
5.       return RANDOMIZED-SELECT(S, k)
6. else (we have |S| < k − 1)
7.       return RANDOMIZED-SELECT(L, k − (|S| + 1))
```

The idea of the algorithm is very similar to RANDOMIZED-QUICKSORT that we saw in class. As in that algorithm, we first select a number *pivot* uniformly at random from the numbers in A. We then partition A into two arrays S and L that contain all numbers of A that are strictly smaller and strictly larger than *pivot*, respectively[1]. The time it takes to execute these steps (Steps 1 and 2) of the algorithm is $\Theta(|A|)$ which is also proportional to the number of "$\leq$"-comparisons we make to find S and L. After that, the algorithm recurses on the array where the kth smallest element can be found or simply returns the kth smallest element if the pivot equals it.

We shall now analyze the running time of RANDOMIZED-SELECT.

4a *(4 pts)* Suppose that we are extremely lucky: every time we select a pivot at random, the pivot that *minimizes* the running time is selected. What is the asymptotic running time of RANDOMIZED-SELECT in this lucky case? Motivate your answer.

Solution:

If we have maximum luck the pivot equals the kth smallest element. Therefore the running time of algorithm will be $\Theta(n)$ if $|A| = n$. This is the time it takes to execute Steps 1-4 of the algorithm.

[1] As we saw in class, we can do this without using the extra space S and L. We have presented the algorithm in this way to make the description clearer.

4b *(6 pts)* Suppose that we are extremely unlucky: every time we select a pivot at random, the pivot that *maximizes* the running time is selected. What is the asymptotic running time of RANDOMIZED-SELECT in this unlucky case? Motivate your answer.

Solution:

Let $|A| = n$. Suppose $k \leq n/2$ the argument is symmetric if $k > n/2$. Now if we have bad luck, Step 1 always selects the pivot that equals the largest number. It will then continue to look for the kth smallest element in array S of size $(n-1)$. Thus we are stuck with the following recursion $T(n) = T(n-1) + \Theta(n)$ and $T(k) = 1$. When $k \leq n/2$ this is $\Theta(n^2)$. Note that the maximum bad luck can only be worse so that has also running time $\Theta(n^2)$. That it is $O(n^2)$ follows from that one can see that the running time is upper bounded by $T(n) = T(n-1)+\Theta(n^2)$.

4c We shall now analyze the *expected* running time of RANDOMIZED-SELECT on an array of length n. Similarly to RANDOMIZED-QUICKSORT, the running time is proportional to the total number of comparisons. As we saw in class, if we let X be the random variable that equals the total number of comparisons then

$$\mathbb{E}[X] = \sum_{i=1}^{n} \sum_{j=i+1}^{n} \mathbb{E}[X_{ij}],$$

where X_{ij} is the random indicator variable that takes value 1 if the ith smallest number was compared to the jth smallest number of the array, and 0 otherwise.

Give a tight asymptotic analysis of the *expected* running time of RANDOMIZED-SELECT by analyzing the above expression.

(Hint: Distinguish between the following three cases: $i < j \leq k$, $k \leq i < j$, and $i < k < j$.)

Solution:

We start by noting that

$$\mathbb{E}[X_{i,j}] = \Pr[i\text{th smallest number was compared to the } j\text{th smallest number}]$$

and that two numbers are compared only if one is a pivot. We now do case distinction as in the hint.

Case $i < j \leq k$: Suppose that the ℓth smallest number was the first pivot such that $i \leq \ell \leq k$. i and j are compared if $\ell = i$ or $\ell = j$. We shall show that they will never be compared if $\ell \neq i$ and $\ell \neq j$. To see this suppose first that $i < \ell < j$ but then i will be put in S and j in L so they will never be compared. On the other hand, if $j < \ell \geq k$ then $i, j \in S$ but $k \in L$ so the algorithm will recurse on L. Therefore, the probability that i and j will be compared in this case is $\frac{2}{k-i+1}$.

Case $k \leq i < j$: By the same arguments as in the case above, i and j will be compared with probability $\frac{2}{j-k+1}$.

Case $i < k < j$: In this case we only compare i and j if no pivot is chosen between them before anyone of them is chosen as a pivot (same as in the quicksort analysis). The probability that they are compared is thus $\frac{2}{j-i+1}$.

Having analyzed $\mathbb{E}[X_{ij}]$, we turn our attention to the sum. Note that it can be written as

$$\begin{aligned}
\mathbb{E}[X] &= \sum_{i=1}^{k-1}\sum_{j=i+1}^{k}\mathbb{E}[X_{ij}] + \sum_{j=k+1}^{n}\sum_{i=k}^{j-1}\mathbb{E}[X_{ij}] + \sum_{i=1}^{k-1}\sum_{j=k+1}^{n}\mathbb{E}[X_{ij}] \\
&= \sum_{i=1}^{k-1}\sum_{j=i+1}^{k}\frac{2}{k-i+1} + \sum_{j=k+1}^{n}\sum_{i=k}^{j-1}\frac{2}{j-k+1} + \sum_{i=1}^{k-1}\sum_{j=k+1}^{n}\frac{2}{j-i+1} \\
&= \sum_{i=1}^{k-1}(k-i)\frac{2}{k-i+1} + \sum_{j=k+1}^{n}(j-k)\frac{2}{j-k+1} + \sum_{i=1}^{k-1}\sum_{j=k+1}^{n}\frac{2}{j-i+1} \\
&\leq 2n + \sum_{i=1}^{k-1}\sum_{j=k+1}^{n}\frac{2}{j-i+1} \\
&= 2n + \sum_{l=1}^{n-1}\sum_{i=k-\ell+1}^{k-1}\frac{2}{\ell+1} \\
&\leq 2n + 2n = 4n.
\end{aligned}$$

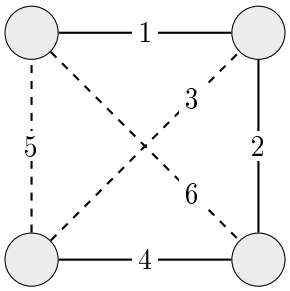
We have thus proved that in expectation 4n comparisons are made. Therefore, RANDOM-SELECT runs in expected linear time.

(This page is intentionally left blank.)

5 *(10 pts)* **Spanning trees.** Consider three undirected edge-weighted connected graphs $G_1 = (V, E_1), G_2 = (V, E_2)$, and $H = (V, E_1 \cup E_2)$ with non-negative weights $w : E_1 \cup E_2 \to \mathbb{R}_+$ on the edges. Note that they are all graphs on the same vertex set but their edges differ: G_1 has only the edges in E_1, G_2 has only the edges in E_2, and H has all the edges $(E_1 \cup E_2)$.

Let T, T_1, T_2 be minimum spanning trees of H, G_1, and G_2, respectively. Assuming that the weights of the edges are unique, i.e., no two edges have the same weight, *prove that* $T \subseteq T_1 \cup T_2$.

For an example of the statement see the figure below. The solid edges are E_1 and the dashed edges are E_2. Note that the minimum spanning tree of G_1 is $T_1 = \{1, 2, 4\}$, the minimum spanning tree of G_2 is $T_2 = \{3, 5, 6\}$, and the minimum spanning tree of H is $T = \{1, 2, 3\}$. We have thus that $T \subseteq T_1 \cup T_2$ in this case. You should prove that it holds in general.



Solution:

Let e be an arbitrary edge in T. Consider the cut defined by the two connected components of $T \setminus \{e\}$. Then e is the minimum-cost edge in this cut. (*Proof.* Suppose $e' \neq e$ is the minimum-cost edge in the cut (A, B). Then $T \setminus \{e\} \cup \{e'\}$ is a spanning tree of strictly better cost, contradicting the assumption that T is a minimum spanning tree.)

Since $e \in E_1 \cup E_2$, $e \in E_i$ for some $i \in \{1, 2\}$, and e is the minimum-cost edge in (A, B) in G_i; thus, $e \in T_i$ from the cut property.

Alternative proof: Suppose that we run Kruskal's algorithm on H, G_1, G_2 to find T, T_1, T_2. Let $m = |E_1 \cup E_2|$.

Suppose toward contradiction that there exists an edge that Kruskal's algorithm adds to T but not to T_1 or T_2. Let $e = \{u, v\}$ be the first such edge. Suppose that $e \in E_1$ (the case $e \in E_2$ is symmetric). If Kruskal's algorithm don't add e to T_1, T_1 already contains a path between u and v that consists of edges of strictly smaller weight than e. However, as Kruskal's algorithm greedily adds edges of smallest edge first, we have that there must be a path between u and v in T as well, which contradicts that e is added to T.

Alternative proof: Consider the edges in $E_1 \cup E_2$. Let $m = |E_1 \cup E_2|$ and suppose that they are ordered $e_1, e_2, \ldots, e_m$ such that $w(e_1) < w(e_2) < \cdots < w(e_m)$. Now note that if we run Kruskal's algorithm on H then an edge $e_i = \{u, v\}$ is added to the tree T if and only if the vertices u and v are in different components in the graph with edges $E_{<i} = \{e_1, e_2, \ldots, e_{i-1}\}$. Therefore if e_i is added to the tree T of H it is also clearly added by Kruskal's algorithm to the tree T_1 of G_1 if $e_i \in E_1$ or to the tree T_2 of G_2 if $e_i \in E_2$. To see this note that since u and v are in different components in the graph with edges $E_{<i}$, they are also in different components in the graph with edges $E_{<i} \cap E_1$ and in the graph with edges $E_{<i} \cap E_2$.

6 *(20 pts)* **Dynamic programming.** The capacitated increasing subsequence problem is defined as follows:

Input: a sequence of integers $a_1, a_2, \ldots, a_n$, sizes $s_1, s_2, \ldots, s_n$ that are either $1, 2$, or 3, and an integral capacity C.

Output: the length of a longest subsequence $a_{i_1}, a_{i_2}, \ldots, a_{i_k}$ satisfying:

- It is an increasing subsequence: $1 \leq i_1 < i_2 < \cdots < i_k \leq n$ and $a_{i_1} < a_{i_2} < \cdots < a_{i_k}$.
- It satisfies the capacity: $s_{i_1} + s_{i_2} + \cdots + s_{i_k} \leq C$.

For example, consider the following input: $a_1 = 8, a_2 = 7, a_3 = 9$ and $s_1 = 1, s_2 = 2, s_3 = 1$ and capacity $C = 2$. The correct output is 2 as a capacitated increasing subsequence of maximum length is a_1, a_3 which has size $s_1 + s_3 = 2 = C$.

6a *(10 pts)* Let $R(j, c)$ = "length of longest increasing subsequence which ends in a_j and has capacity at most c". If $s_j > c$, we let $R(j, c) = -\infty$. Find a recursive formulation of $R(j, c)$.

Solution: Note that if we have an maximum length increasing subsequence that ends in a_j of capacity at most c. Then there are 3 cases

- it is invalid: $s_j > c$ then $R(j, c) = -\infty$ by definition.
- It is the only number in the sequence. Then $R(j, c) = 1$. This happens if $j = 1$ and $s_j \leq c$ or if $a_i \geq a_j$ or $s_i > c - s_j$ for all $i : 1 \leq i < j$.
- It is the last number of a longer subsequence. Then $R(j, c)$ is equal to 1 + the maximum length of a subsequence of capacity $c - s_j$ ending in some $i : 1 \leq i < j$ with $a_i < a_j$. Therefore we get in this case that $R(j, c) = \max_{1 \leq i < j}\{1 + R(i, c - s_j) | a_i < a_j\}$.

One can formulate this as (at least) two different recurrences:

$$R(j, c) = \begin{cases} -\infty, & \text{if } s_j > c \\ 1, & \text{if } j = 1 \text{ and } s_j \leq c \\ \max\left[1, \max_{1 \leq i < j}\{1 + R(i, c - s_j) | a_i < a_j\}\right], & \text{if } j > 1 \text{ and } s_j \leq c \end{cases}$$

Alternative: Let $R(j, c) = -\infty$ if $s_j > c$ and let $R(1, c) = 1$ if $s_1 \leq c$. For $j > 1$ and $s_j \leq c$,

$$R(j, c) = \begin{cases} 1, & \text{if } a_i \geq a_j \text{ or } s_i > c - s_j \text{ for all } 1 \leq i < j \\ \max_{1 \leq i < j}\{1 + R(i, c - s_j) | a_i < a_j\}, & \text{otherwise} \end{cases}$$

6b *(10 pts) Design and analyze* an efficient algorithm for solving the capacitated increasing subsequence problem.

Solution:

We use the bottom-up-approach starting from the following recursion:

$$R(j,c) = \begin{cases} -\infty, & \text{if } s_j > c \\ 1, & \text{if } j = 1 \text{ and } s_j \leq c \\ \max\left[1, \max_{1 \leq i < j}\{1 + R(i, c - s_j) | a_i < a_j\}\right], & \text{if } j > 1 \text{ and } s_j \leq c \end{cases}$$

The pseudocode is as follows

```
BOTTOM-UP-INCREASING-SUBSEQUENCE(a, s, C, n)
1. Let r[·][·] be a new matrix of dimension n × C.
2. for j = 1, ..., n
3.     for c = 1, ..., C
4.         if s_j > c
5.             r[j][c] = −∞
6.         else if j = 1
7.             r[j][c] = 1
8.         else
9.             r[j][c] = max[1, max_{1≤i<j}{1 + R(i, c − s_j) | a_i < a_j}]
10. return max_{1≤j≤n} r[j][C]
```

Run time analysis:

Note that the table r has $n \cdot C$ cells. Each cell takes $O(n)$ time to fill in (line 9 is dominating). Therefore, the total running time is $O(n^2C)$. To see that it is $\Omega(n^2C)$. Note that line 9 takes time $\Omega(n)$ if j is greater than $n/2$ and that step 9 is executed $n/2 \cdot C$ times when $j \geq n/2$. This leads to a running time of $\Omega(n^2C)$ and hence the total running time is $\Theta(n^2C)$. This is efficient ($\Theta(n^3)$ time) since we may assume that $C \leq 3n$ because sizes where at most 3.