ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Final Exam January 20, 2010

Algorithm

January 20, 2010

- You are only allowed to have an $A4$ page written on both sides.
- Calculators, cell phones, computers, etc... are not allowed.

Family name :

First name :

Section :

Exercise 1 / 10 points	Exercise 2 / 10 points	Exercise 3 / 20 points	Exercise 4 / 20 points	Exercise 5 / 20 points	Exercise 6 / 20 points

Total / 100

Problem 1 [10 points]. Let $f(n)$ be the function given below in pseudocode:

```
Call: $f(n)$
 1: $a \leftarrow 0$
 2: $b \leftarrow \ln(n)$
 3: for $i = 1, \ldots, n$ do
 4:     $a \leftarrow a + b$
 5: end for
 6: for $j = 1, \ldots, n$ do
 7:     for $k = 1, \ldots, j$ do
 8:         for $\ell = j+1, \ldots, j+n$ do
 9:             $a \leftarrow a + b$
10:         end for
11:     end for
12: end for
13: return $a$
```

1. Find a closed-form formula for $f(n)$.
2. Find s and t such that
$$f(n) = \theta\big(n^s \cdot \ln(n)^t\big).$$
3. What is, in θ notation, the running time of this algorithm, given that line 2 runs in time $\theta(1)$?

Corrigé:

1. We have the following:

$$\begin{aligned}
f(n) &= \sum_{i=1}^{n} b + \sum_{j=1}^{n}\sum_{k=1}^{j}\sum_{\ell=j+1}^{j+n} b \\
&= nb + n\sum_{j=1}^{n}\sum_{k=1}^{j} b \\
&= nb + nb\sum_{j=1}^{n} j \\
&= nb + nb\frac{n(n+1)}{2} \\
&= n\ln(n) + \frac{n^2(n+1)\ln(n)}{2}.
\end{aligned}$$

2. The final answer is $f(n) = \theta\big(n^3 \cdot \ln(n)\big)$, that is: $s = 3$ et $t = 1$.
3. In this case, the running time is majorized by the running time of the triple loop in lines 6,7,8, and 9, which is equal to the value of the triple sum in the expression above. Hence, the running time is $\theta\big(n^3\big)$.

Problem 2 [10 points].

Suppose that, given a list of distinct integers and a positive integer i, we wish to find the i largest integers on the list. Consider the following two approaches to solve the problem:

1. Use the MergeSort algorithm to sort the list in increasing order, and pick the last i items from the resulting sequence.

2. Create a heap in a bottom-up fashion, and then obtain the i largest elements by calling the DeleteMax operation i times.

What is the running time of each solution? Which approach is more favorable for finding the 10 largest items on a list of a billion integers?

Corrigé:

Approach 1: According to the description of MergeSort, the running time is $O\big(n\log(n)\big)$;
Approach 2: We need first build up a max-heap and using the DeleteMax operation in order to get a largest number. We build the Max-Heap using a bottom-up approach with $O(n)$ operations. Thereafter, we need i DeleteMax operations, each using $O(\log(n))$ steps. The total running time is therefore $O(n + i\log(n))$ steps.

The second approach is more favorable in this case since $i = 10$ is much smaller than $\log(10,000,000,000)$ which is roughly 33. Moreover, the MergeSort will need additional memory complexity, which is as much as $O\big(n\big)$. It will be very difficult if the cache resource is limited.

Problem 3 [20 points].

Analysis of d-ary heaps: a *d-ary heap* is similar to a binary heap with one exception. The non-leaf nodes have d children instead of 2 children.

1. How would you represent a d-ary heap in an array?
2. What is the height of a d-ary heap of n elements in terms of n and d?
3. Let `EXTRACT-MAX` be an algorithm that returns the maximum element from a d-ary heap and removes it while maintaining the heap property. Give an efficient implementation of `EXTRACT-MAX` for a d-ary heap. Analyze its running time in terms of d and n.
4. Let `INSERT` be an algorithm that inserts an element in a d-ary heap. Give an efficient implementation of `INSERT` for a d-ary heap. Analyze its running time in terms of d and n.

Corrigé:

1. A *d-ary* heap can be represented by a 1-dimensional array as follows: The root is kept in $A[0]$. The children of the node $A[i]$ are stored in $A[di+1], \dots, A[d(i+1)]$, i.e., the jth child is stored in $A[di+j]$, Therefore, for $i > 0$, the parent of the node stored at $A[i]$ is the node stored at $A[\lceil (i-1)/d \rceil]$.
2. Since each intermediate node has exactly d children, the total number of nodes in a tree of height h is at most $1 + d + d^2 + \cdots + d^h$ (when the tree is a complete d-ary tree) and at least $2 + d + d^2 + \cdots + d^{h-1}$ (when there is exactly one node of depth h and all the other nodes are of depth $h-1$). Therefore, we have
$$1+d+d^2+\cdots+d^{h-1} = \frac{d^h - 1}{d-1} \le 2+d+\cdots+d^{h-1} \le n \le 1+d+d^2+\cdots+d^h = \frac{d^{h+1}-1}{d-1}.$$
This gives us
$$(\log_d(n(d-1)+1)) - 1 \le h \le \log_d(n(d-1)+1)$$
Therefore, $h = \lceil \log_d(n(d-1)+1) \rceil - 1 = \theta(\log_d n)$.
3. The procedure DeleteMax given in the course for binary heaps works for d-ary heaps as well with only a minor change: the SiftDown procedure used in the DeleteMax function should be modified to consider all the d children of a node during the sifting down process, instead of just 2 children. Below you can find the modified algorithms.

   Except for D-SiftDown in line 7 of algorithm 3, all operations in the Extract-Max take $O(1)$. D-SiftDown takes $\theta(d \log_d n)$ operations since it is exactly the SiftDown algorithm mentioned in the course but does d comparisons in each iteration. As a result, the overall running time of Extract-Max is $\Theta(d \log_d n)$.
4. The algorithm SiftUp mentioned in the course for binary heaps works here as well. Thus, the running time is $\theta(h)$, where h is the height of the heap. For the d-ary heaps, as we have shown in part 2 of this problem, $h = \theta(\log_d n)$. Therefore, the running time of Insert is $\theta(\log_d n)$.

Algorithm 1 EXTRACT-MAX(A, d)

Input: An array A containing the d-ary heap
Output: Extracting the maximum element of the heap while preserving the heap structure

```
1: if $length(A) < 1$ then
2:    Error
3: end if
4: $max \leftarrow A[0]$
5: $A[0] \leftarrow A[length(A) - 1]$
6: $length(A) \leftarrow length(A) - 1$
7: D-SIFTDOWN$(A, 0, d)$
8: return $max$
```

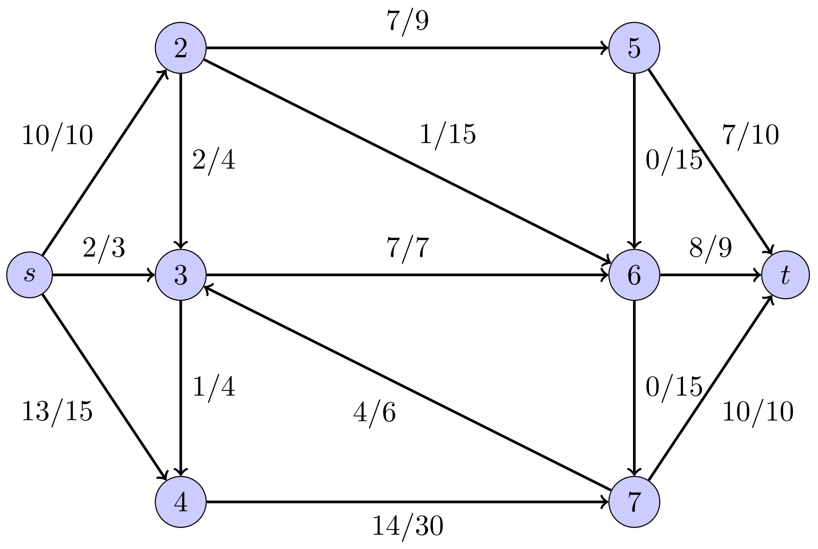
Algorithm 2 D-SIFTDOWN(A, i, d)

Input: An array A containing the heap with n elements and an integer i, $0 \leq i < n$ for which $A[i+1], \ldots, A[n-1]$ satisfies the heap property.
Output: Transformation of A such that $A[i], A[i+1], \ldots, A[n-1]$ satisfies the heap property.

```
1: Swapped ← true
2: while Swapped = true and $d.i + 1 < n$ do
3:    Swapped ← false
4:    Find the largest child $A[j]$ of $A[i]$
5:    if $A[j].key > A[i].key$ then
6:       Exchange $A[i]$ and $A[j]$
7:       $i \leftarrow j$
8:       Swapped ← true
9:    end if
10: End
```

Problem 4 [20 points].

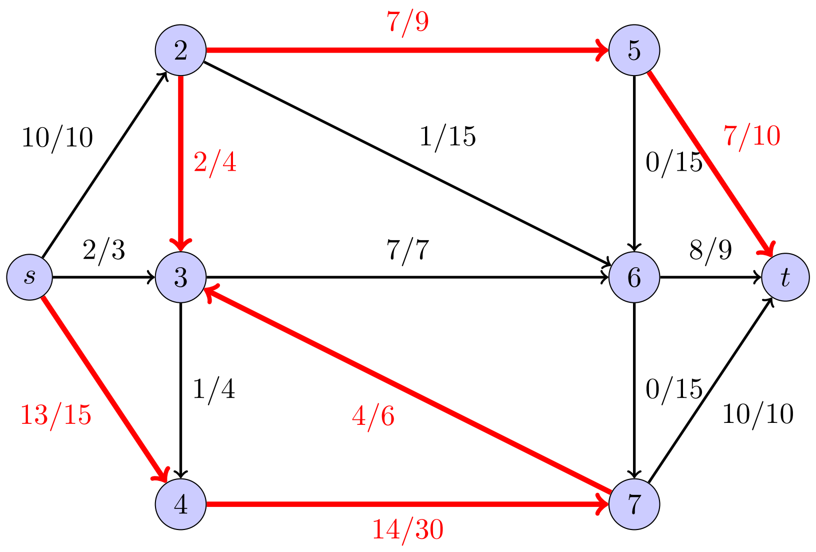
Assume the following graph and corresponding flows (The number on edges determine the capacity and the current flow). Perform one iteration of the Ford-Fulkerson algorithm. Choose the fattest augmenting path, i.e., the path with the highest residual capacity.



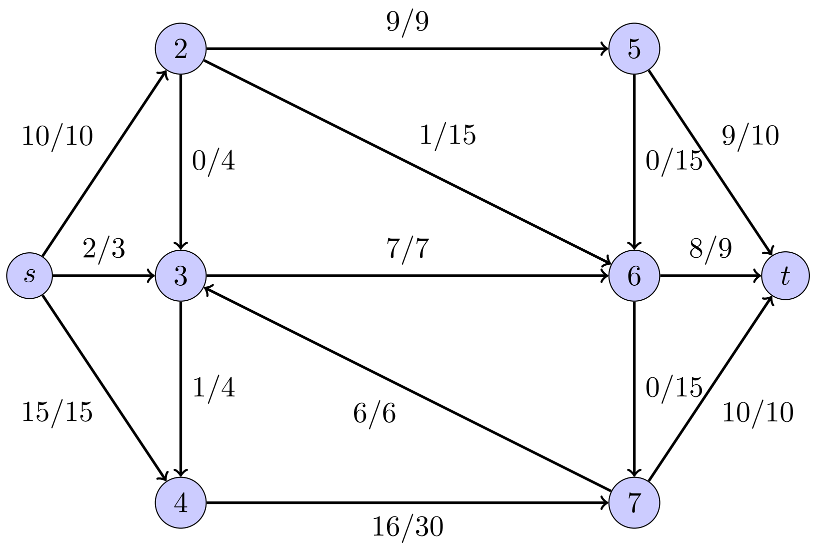
1. Write down the fattest augmenting path.
2. What is the value of the resulting flow?
3. Is the resulting flow optimal? If so, give a min cut whose capacity is equal to the value of the flow. If not, give a fattest augmenting path.

Corrigé:

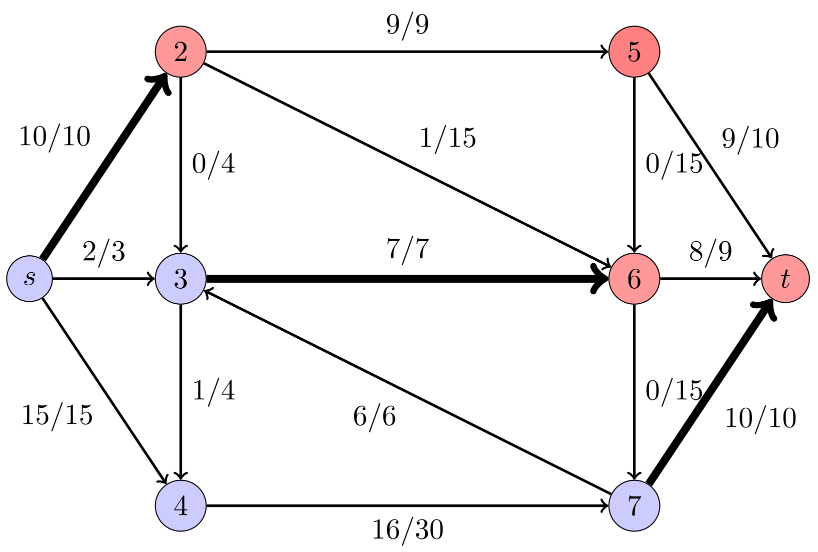
1. The fattest augmenting path is $s - 4 - 7 - 3 - 2 - 5 - t$ which increases the flow by 2. No other path can increase the flow more than this. Because it has to pass node 3 at the beginning which increases the flow only by 1.



2. The value of the resulting flow is 27.



3. The resulting flow is optimal. Because there exists a cut with cost $10 + 7 + 10 = 27$, namely , $\{s, 3, 4, 7\}$ and $\{2, 5, 6, t\}$.


9/9
10/10
0/4
1/15
0/15
9/10
2/3
7/7
8/9
s
2
3
5
6
t
4
7
1/4
0/15
15/15
6/6
10/10
16/30

Problem 5 [20 points].

A *contiguous subsequence* of a sequence S is a subsequence consisting of consecutive elements of S. For example, we might have

$$S = (2, -5, 10, 4, -12, 5, 0, 1),$$

for which $(10, 4, -12)$ is a contiguous subsequence but $(2, 4, 5, 1)$ and $(-12, 4)$ are not.

Using dynamic programming, design a linear time algorithm that, given a sequence $S = (s_1, \ldots, s_n)$ of integers, finds a contiguous subsequence of S with maximum sum. That is, a contiguous subsequence $(s_i, s_{i+1}, ..., s_j)$ of S for which the summation of all entries $(s_i + s_{i+1} + \cdots + s_j)$ is maximum.

Hint: for each $j \in \{1, ..., n\}$, consider the subproblem of finding the optimal contiguous subsequence within the first j elements of S.

Corrigé:

Denote the sequence by $S = (s_1, \ldots, s_n)$. For $i \geq 0$ let $Q[i]$ be the index ℓ in $\{1, \ldots, i+1\}$ for which $s_\ell + s_{\ell+1} + \cdots + s_i$ is maximized, and denote by $P[i]$ the sum $s_{Q[i]} + \cdots + s_i$. We call $P[i]$ a maximum-sum suffix of $S_i = (s_1, \ldots, s_i)$.

Since $Q[0] = 1$, we have $P[0] = 0$. Our first goal is to find a recursion for $P[i+1]$ in terms of $P[i]$. If $Q[i] \leq i$, then $P[i] + s_{i+1} = s_{Q[i]}, \ldots, s_i + s_{i+1}$ is no smaller than $s_\ell + \cdots + s_i + s_{i+1}$ for any $1 \leq \ell \leq i$, hence $Q[i+1] = Q[i]$ if $P[i] + s_{i+1} \geq 0$, and $Q[i+1] = i+2$ (and hence $P[i+1] = 0$) otherwise. The same conclusion holds if $Q[i] = i+1$. Therefore, we have $P[i+1] = \max(P[i] + s_{i+1}, 0)$.

Coming back to our problem, note that the maximum-sum contiguous subsequence is the maximum-sum suffix of $s_1, \ldots, s_i$ for some i. This gives the following algorithm.

```
Maximum Sum Contiguous Subsequence: $(s, n)$
 1: $P[0] \leftarrow 0, \quad Q[0] \leftarrow 1$
 2: for $i = 0, \ldots, n-1$ do
 3:    if $P[i] + s_{i+1} > 0$ then
 4:       $P[i+1] \leftarrow P[i] + s_{i+1}, \quad Q[i+1] \leftarrow Q[i]$
 5:    else
 6:       $P[i+1] \leftarrow 0, \quad Q[i+1] \leftarrow i+2$
 7:    end if
 8: end for
 9: $i \leftarrow \arg\max_{j=1}^{n} P[j]$
10: return $s_{Q[i]}, \ldots, s_i$
```

Problem 6 [20 points].

Consider the following variation of SAT, called SAT(5): It is defined as the set of satisfiable Boolean CNF formulas in which each variable appears (either in positive or negated form) in exactly 5 clauses. Show that SAT(5) is NP-complete.

Hint: Introduce auxiliary variables in the formula.

Corrigé:

First, observe that SAT(5) is a special case of SAT which we know is in NP and therefore SAT(5) is also in NP.

It remains to prove that SAT(5) is NP-hard. We do so by providing a polynomial time reduction from SAT to SAT(5). Given a SAT formula φ we obtain a formula ψ of SAT(5) as follows: for each variable x in φ we replace its first appearance by the auxiliary variable x_1, its second appearance by x_2 and so on, replacing each of its k appearances with a different new auxiliary variable. In addition, we add two copies of the *consistency clauses*

$$(\bar{x}_1 \vee x_2) \wedge (\bar{x}_2 \vee x_3) \wedge \cdots \wedge (\bar{x}_k \vee x_1).$$

Note that the construction of ψ can clearly be done in polynomial time as required. Furthermore, since we added two copies of the consistency clauses each variable of ψ appears in exactly 5 clauses and ψ is therefore a formula of SAT(5).

It remains to prove that the reduction preserves satisfiability, i.e., that φ is satisfiable if and only if ψ is satisfiable. The key insight is that the consistency clauses $(\bar{x}_1 \vee x_2) \wedge (\bar{x}_2 \vee x_3) \wedge \cdots \wedge (\bar{x}_k \vee x_1)$ of the auxiliary variables that replaced the occurrences of variable x implies that a satisfying assignment of ψ must assign the same truth value to all of them. To see this, assume toward contradiction that a satisfying assignment of ψ assigned different truth values to the variables $x_1, \dots, x_k$. Then there must exist an i so that x_i and x_{i+1} (or x_k and x_1) are assigned values true and false, respectively. However, this contradicts that the clause $\bar{x}_i \vee x_{i+1}$ is satisfied.

As any satisfying assignment to ψ must assign the same truth value to the auxiliary variables $x_1, \dots, x_k$ in ψ, they behave identically to the original variable x in φ that they replaced. We can therefore conclude that φ is satisfiable if and only if ψ is satisfiable.