ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Old Exam Questions, Algorithms 2013-2014

This document contains a sample of exam questions. In particular, the problems have previously appeared on a (midterm) exam. The points of the exercises reflect their importance to the final score of the exams, where the total amount of points of an exam is 100. One should note however, that these exams were during 3 hours whereas the mid-term will be during 2 hours. For some exercises, I have also written comments on how related the questions are to this year's version of the course.
Good luck with your preparation!

1 (5 pts) Consider the natural implementation of the three sorting algorithms (Insertion-Sort, Merge-Sort, Heap-Sort) that we studied in class. Which one of these implementations is *not* "in-place"?

Solution: Merge-Sort

2 (5 pts) Simplify and arrange the following functions in increasing order according to asymptotic growth.

$$3^N, \sqrt{4^N}, \log^2 N, \sqrt{N}, N^2, \log N, 20N$$

Solution:

- First the logs: $\log N, \log^2 N$
- Second the polynomials: $\sqrt{N}, 20N, N^2$
- Last the exponential functions: $\sqrt{4^N} = 2^N, 3^N$

3 (10 pts) Let $f(n)$ and $g(n)$ be the functions defined for positive integers as follows:

```
Function f(n):
1: ans ← 0
2: for i = 1, 2, ..., n − 1 do
3:     for j = 1, 2, ..., n − i do
4:         ans ← ans + 1
5:     end for
6: end for
7: return ans
```

```
Function g(n):
1: if n = 1 then
2:     return 1
3: else
4:     return g(⌊n/2⌋) + g(⌊n/2⌋)
5: end if
```

3a (6 pts) What is, in Θ notation, the running time of these algorithms, given that addition runs in time $\Theta(1)$?

Solution: Running time of $f(n)$ is $\Theta(n^2)$. The running time of $g(n)$ is proportional to the recurrence relation $T(n) = 2T(n/2) + \Theta(1)$. Therefore, by the master method, the running time of $g(n)$ is $\Theta(n)$.

3b (4 pts) What is, in Θ notation, the running time of algorithm $g(n)$ if we at line 4 replace $g(\lfloor n/2 \rfloor) + g(\lfloor n/2 \rfloor)$ by $2g(\lfloor n/2 \rfloor)$?

(Assume that multiplication runs in time $\Theta(1)$.)

Solution: Now the running time is proportional to the recurrence $T(n) = T(n/2)+1$. Therefore, by the master method, the running time of the modified version is $\Theta(\log n)$

4 (10 pts) Let $f(n)$ be the function given below in pseudocode:

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

4a Find a closed-form formula for $f(n)$.

Solution: We have the following:

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

4b Find s and t such that

$$f(n) = \theta\big(n^s \cdot \ln(n)^t\big).$$

Solution: The final answer is $f(n) = \theta\big(n^3 \cdot \ln(n)\big)$, that is: $s = 3$ et $t = 1$.

4c What is, in θ notation, the running time of this algorithm, given that line 2 runs in time $\theta(1)$?

Solution: 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 $\theta\big(n^3\big)$.

5 (10 pts)

5a (4 pts) Give a formal specification of the following problem: given an array of integers, and another integer x, determine whether there are two elements in the array that sum up to x. *Note: What formal means is slightly unclear from the perspective of this year's course.*

Solution: The input set I is $(\cup_{n=1}^{\infty}\mathbb{Z}^n)\times\mathbb{Z}$. The output set O is {TRUE,FALSE}. The relation is

$$R = \Big\{((\sigma,x),\text{TRUE}) \;\Big|\; \exists\, 1\le i<j\le|\sigma|\colon\ \sigma_i+\sigma_j=x\Big\}\cup \Big\{((\sigma,x),\text{FALSE}) \;\Big|\; \forall\, 1\le i<j\le|\sigma|\colon\ \sigma_i+\sigma_j\neq x\Big\}$$

where $|\sigma|$ is the length of the sequence σ.

5b (6 pts) Design an algorithm that solves the problem of the previous part in $O(n\log n)$ steps, where n is the length of the array, and a step is either an addition or a comparison of integers.

Solution: Given an array σ consisting of n elements, and the integer x, we first sort σ with say MERGE-SORT using $O(n\log n)$ operations. Next, we initialize two indices i and j to $i=1$ and $j=n$. As long as $i<j$, we perform the following: if $\sigma_i+\sigma_j>x$, then we decrease j by one. If $\sigma_i+\sigma_j<x$, then we increase i by one. We stop and return the value TRUE if we hit (i,j) with $\sigma_i+\sigma_j$. If during the course of the algorithm $i\ge j$, then we return the value FALSE. The running time of this algorithm is obviously $O(n)$, so the total running time is $O(n\log n)$.

6 (10 pts) 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?

Solution: 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/memory resource is limited.

7 (15 pts) Suppose that you are given a sorted sequence of n *distinct* integers $\{a_1,\ldots,a_n\}$. Give an algorithm to determine whether there exists an index i such that $a_i=i$, outputting one i if such an i exists, and which uses $O(\log(n))$ steps. For example, in $\{-10,-3,3,5,7\}$, $a_3=3$, whereas $\{2,3,4,5,6,7\}$ has no such i.

Solution: The key is the following realization: if $a_i < i$, then for all $\ell \leq i$ we have $a_\ell < \ell$, and if $a_i > i$, then for all $\ell \geq i$ we have $a_\ell > \ell$. To see this, note that in the former case, $a_{i-\ell} \leq a_i - \ell \leq i - \ell$ since the a_i's are distinct and integers. In the latter case, $a_\ell \geq a_i + (\ell - i) \geq \ell$, again since the a_i's are distinct.

Now we use binary search on the sequence $(a_i - i | 1 \leq i \leq n)$ to find a zero of this sequence if it exists.

8 (15 pts) Given an $O(n \log(k))$-algorithm that merges k sorted list of integers with a total of n elements into one sorted list. *(Hint: use a heap of size k)*

Solution: *Note: in this solution the array of ℓ elements is indexed $0, 1, \ldots, \ell - 1$ instead of $1, 2, \ldots, \ell$ as we have been doing in this year's course.*

Let the arrays be $A_1, \ldots, A_k$ and assume that they are sorted in an ascending way. We will create sorted array S consisting of the union of the A_i's.

```
Create min-heap for elements (1,A_1[0]),...,(k,A_k[0]) where
the minimization is with respect to the second variable.
i = 0;
while ( i < n ) do
  (j,A_j[l]) = DeleteMin of the Heap;
  S[i] = A_j[l];
  i = i+1;
  if (l is not the length of A_j ) then
     Insert A_j[l+1] into the min-heap.
```

For every entry of S we need to do one deletemin operation and at most one insertion operation into the heap. The heap size is at most k, so these operations cost $O(\log(k))$. In total we will have an $O(n \log(k))$-algorithm.

9 (20 pts) **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.

9a How would you represent a d-ary heap in an array?

Solution: 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 $A[di+1], \ldots, A[d(i+1)]$, 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 $A[\lceil (i-1)/d \rceil]$.

9b What is the height of a d-ary heap of n elements in terms of n and d?

Solution: 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

$$1 + d + d^2 + \cdots + d^{h-1} = \frac{d^h - 1}{d-1} \leq 2 + d + \cdots + d^{h-1} \leq n \leq 1 + d + d^2 + \cdots + d^h = \frac{d^{h+1} - 1}{d-1}.$$

This gives us

$$(\log_d(n(d-1)+1)) - 1 \leq h \leq \log_d(n(d-1)+1)$$

Therefore, $h = \lceil \log_d(n(d-1)+1) \rceil - 1 = \theta(\log_d n)$.

9c 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.

Solution: The procedure EXTRACT-MAX given in the course for binary heaps works for d-ary heaps as well with only a minor change: the MAX-HEAPIFY procedure used should be modified to consider all the d children of a node during the process, instead of just 2 children. For a complete solution, you would need to explain the procedure in detail. One way would be to give the pseudo-code but I leave this as an exercise.

9d 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.

Solution: Again the insert procedure given in class works well; for a complete solution, you would need to describe it in detail with pseudo-code for example As the procedure "walks" up the heap, 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 $\theta(\log_d n)$.

10 (20 pts) 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.

Solution: 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 postfix 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 prefix of $s_1, \ldots, s_i$ for some i. This gives the following algorithm.

Maximum Sum Contiguous Subsequence: (s, n)

```
1: P[0] ← 0,  Q[0] ← 1
2: for i = 0, ..., n − 1 do
3:    if P[i] + s_{i+1} > 0 then
4:       P[i + 1] ← P[i] + s_{i+1},  Q[i + 1] ← Q[i]
5:    else
6:       P[i + 1] ← 0,  Q[i + 1] ← i + 2
7:    end if
8: end for
9: i ← arg max_{j=1}^{n} P[j]
10: return s_{Q[i]}, ..., s_i
```

11 (20 pts) Let $x_1...x_m$ and $y_1...y_n$ be two strings. For $0 \leq i \leq m$ and $0 \leq j \leq n$, let $c[i, j]$ be the length of the longest common subsequence of $x_1...x_i$ (or the empty string if $i = 0$) and $y_1...y_j$ (or the empty string if $j = 0$). In other words, $c[i, j]$ is defined to be the maximum k such that there exist indices $1 \leq i_1 < i_2 < \cdots < i_k \leq i$ and $1 \leq j_1 < j_2 < \cdots < j_k \leq j$ satisfying $x_{i_\ell} = y_{j_\ell}$ for all $\ell = 1, \ldots, k$.

11a (10 pts) Complete the recurrence relation for $c[i, j]$ that can be used for dynamic programming:

$$c[i,j] = \begin{cases} \_\_\_\_\_ & \text{if } i = 0 \text{ or } j = 0 \\ \_\_\_\_\_ & \text{if } i, j > 0 \text{ and } x_i = y_j \\ \_\_\_\_\_ & \text{if } i, j > 0 \text{ and } x_i \neq y_i \end{cases}$$

Solution:

$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ c[i-1, j-1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j \\ \max(c[i, j-1], c[i-1, j]) & \text{if } i, j > 0 \text{ and } x_i \neq y_i \end{cases}$$

11b (10 pts) Use the recurrence relation to return the length of the longest common subsequence of the two strings BABDBA and DACBCBA by filling in the table of $c[i, j]$ values below (in a bottom-up dynamic programming fashion).

	j	0	1	2	3	4	5	6	7
i		y_j	D	A	C	B	C	B	A
0	x_i								
1	B								
2	A								
3	B								
4	D								
5	B								
6	A								

Solution:

	j	0	1	2	3	4	5	6	7
i		y_j	D	A	C	B	C	B	A
0	x_i	0	0	0	0	0	0	0	0
1	B	0	0	0	0	1	1	1	1
2	A	0	0	1	1	1	1	1	2
3	B	0	0	1	1	2	2	2	2
4	D	0	1	1	1	2	2	2	2
5	B	0	1	1	1	2	2	3	3
6	A	0	1	2	2	2	2	3	4

Longes commong subsequence is 4 (it is ABBA)

12 (20 pts) A max-min algorithm finds both the largest and the smallest elements in an array of n integers. Design and analyze a divide-and-conquer max-min algorithm that uses $\lceil 3n/2 \rceil - 2$ comparisons for any integer n. (You will receive 10 points if you can show this for the case when n is a power of 2, and an additional 10 points if you can prove it for general n.)

Hint: If $T(n)$ denotes the number of comparisons of your algorithm, try to find a recursion relating $T(n+m)$ to $T(n)$ and $T(m)$. Then study first the case where n is a power of 2.

Solution:

a. First, we will show how to solve this problem when n is a power of 2. We will use the procedure Max-min(A, n), which receives as input an array A of length n and returns (a, b) where a and b are the maximum and minimum elements of A. The algorithm works by recursively dividing A into smaller sub-arrays of approximately half the size of A.

```
Max-min: (A, n)
  if n == 1 then
    return (A[0], A[0])
  end if
  if n == 2 then
    if A[0] ≥ A[1] then
      return (A[0], A[1])
    else
      return (A[1], A[0])
    end if
  end if
  if n > 2 then
    (a_1, b_1) ←Max-min(A[0 : ⌈n/2⌉ − 1], ⌈n/2⌉)
    (a_2, b_2) ←Max-min(A[⌈n/2⌉ : n − 1], n − ⌈n/2⌉)
    a ← max{a_1, a_2}
    b ← min{b_1, b_2}
    return (a, b)
  end if
```

Let us now analyze how many comparisons this algorithm uses: clearly, each max or min operation requires only one comparison. Therefore, the recurrence relation describing how many comparisons we use is the following:

$$T(n) = T(\lceil \frac{n}{2} \rceil) + T(\lfloor \frac{n}{2} \rfloor) + 2$$

and $T(1) = 0$, $T(2) = 1$.

Since n is even, we have

$$T(n) = T(\frac{n}{2}) + T(\frac{n}{2}) + 2$$

Next, in order to solve our recurrence relation, we will use the substitution method; specifically, we will use $T(k) \leq \lceil \frac{3k}{2} \rceil - 2$ as our induction hypothesis. Notice it holds for $k = 1, 2$, since $T(1) = 0, T(2) = 1$. We will assume the induction hypothesis holds for all $k \leq n - 1$, where k is a power of 2, and prove it holds for $k = n$. We have

$$T(n) = T(\frac{n}{2}) + T(\frac{n}{2}) + 2 = 2T(\frac{n}{2}) + 2 \leq 2(\frac{3n}{4} - 2) + 2 = \frac{3n}{2} - 2 = \lceil \frac{3n}{2} \rceil - 2$$

which concludes the proof.

We saw how to solve this problem when n is a power of 2; our new algorithm will use this fact, by dividing an array A of n integers into two arrays, the larger of which has size equal to the largest power of 2 which is less than n. The procedure we use is Max-min-general(A, n).

```
Max-min-general: (A, n)
  if n == 1 then
    return (A[0], A[0])
  end if
  if n == 2 then
    if A[0] ≥ A[1] then
      return (A[0], A[1])
    else
      return (A[1], A[0])
    end if
  end if
  if n > 2 then
    k ←largest power of 2 which is less than n
    (a_1, b_1) ←Max-min-general(A[0 : k − 1], k)
    (a_2, b_2) ←Max-min-general(A[k, n − 1], n − k)
    a ← max{a_1, a_2}
    b ← min{b_1, b_2}
    return (a, b)
  end if
```

Let us analyze how many comparisons this algorithm uses. Let $f(n)$ be the largest power of 2 less than n. The recurrence relation we want to solve is

$$T(n) = T(f(n)) + T(n - f(n)) + 2$$

In order to take a deeper look at the above relation, consider the binary representation of $n = \sum_{i=0}^{\infty} a_i^n 2^i$, where $a_i^n \in \{0, 1\}$, and let $g(n)$ be the number of non-zero a_i^n-s. Expanding $T(n)$ we get

$$T(n) = 2(g(n) - 1) + \sum_{i=0}^{\infty} a_i^n T(2^i)$$

Let $T(k) \leq \lceil \frac{3k}{2} \rceil - 2$ be our inductive hypothesis, and let it hold for all $k \leq n - 1$. We will prove that it holds for $k = n$. We will distinguish two cases:

- n is even: We can assume $n > 2$, and since we have already done the analysis for the case n being a power of 2, we can also assume that n is not a power of 2; therefore, we can write

  $$T(n) = T(x) + T(y) + 2$$

  where x is the highest power of two which is less than n and $y = n - x$. Then by the inductive hypothesis and since x, y are even

  $$T(n) \leq \frac{3x}{2} - 2 + \frac{3y}{2} - 2 + 2 = \frac{3n}{2} - 2 = \lceil \frac{3n}{2} \rceil - 2$$

- n is odd: Again, we can write

  $$T(n) = T(x) + T(y) + 2$$

  where x is the highest power of two which is less than n and $y = n - x$. y is odd and by the inductive hypothesis we have

  $$T(n) \leq \frac{3x}{2} - 2 + \lceil \frac{3y}{2} \rceil - 2 + 2 = \frac{3x}{2} + \frac{3y + 1}{2} - 2 = \lceil \frac{3n}{2} \rceil - 2$$

Since $T(1) = 0$ and $T(2) = 1$, the induction base is also true and our proof is concluded.

13 (20 pts, *Note: this one is quite tricky and was a bonus problem*) Consider a binary heap containing n numbers where the root stores the largest number. Let $k < n$ be a positive integer, and x be another integer. Design an algorithm that determines whether the kth largest element of the heap is greater than x or not. The algorithm should take $O(k)$ time and may use $O(k)$ additional storage.

Hint: don't try to find *the kth largest element.*

Solution: Starting from the root of the tree, traverse the children of a node until the value of the node is strictly smaller than x. Put the index of this node into an array of length $2k$. If there is an overflow, then state that the kth largest element is strictly larger than x. *Claim: if the k-th largest element is smaller than or equal to x, then the array will contain at most $2k$ entries.* Proof. For every node in the array, the value of the ancestor is greater than or equal to x. There are thus at most k such ancestors. Each ancestor has at most 2 children, hence the number of entries in the array is at most $2k$. *Claim: running time is $O(k)$.* Proof. The number of nodes traversed is at most the number of direct descendants of nodes whose values are greater than or equal to x, hence at most $2k$.

Now we will use the new array to see whether the kth largest element of the array is at least x. We do this by doing one pass through the array counting every element that is greater than x.