Design and Analysis of Algorithms, Fall 2012
Exam 18.10.2012: Solutions

---

1 (max 9 points) Give a function $f(n)$ such that $T(n) = \Theta(f(n))$,

(a) when $T(1) = 1$ and $T(n) = 3T(n/3) + n$ for $n > 1$,

(b) when $T(1) = 2$ and $T(n) = 2T(n/3) + \log_2 n$ for $n > 1$.

You may confine your analysis to exact powers of 3.
If using the master method: +3 if cites the theorem correctly; +3 for checking the condition and implications correctly (for both (a) and (b)).
If not using the master method: +1 for showing a correct upper or lower bound; +1 if tight; +1 if proved using induction or careful enough iteration/derivation, for both (a) and (b). In addition, +2 if giving both bounds tight for either (a) or (b), and yet +1 if for both.

The master method gives us a method to solve recurrences of the form

$$T(n) = aT(n/b) + g(n),$$

where $a \geq 1$ and $b \geq 1$ are constants and $g(n)$ is an asymptotically positive function.

In (a) we have $a = 3$, $b = 3$ and $g(n) = n = n^1 = n^{\log_a b} = \Theta(n^{\log_b a})$. This is the second case of the master method, which implies that $T(n) = \Theta(n^{\log_b a} \log n) = \Theta(n \log n)$.

In (b) we have $a = 2$, $b = 3$ and $g(n) = \log_2 n$. Since any logarithm is asymptotically bounded above by any polynomial, we have $g(n) = \log_2 n = O(n^{\log_b a - \varepsilon})$ for example for constant $\varepsilon = (\log_b a)/2 > 0$. As this is the first case of the master method, we have $T(n) = \Theta(n^{\log_b a}) = \Theta(n^{\log_3 2})$.

---

2 (max 9 points) The *segmentation problem* is as follows. For each pair (i, j) of integers, $1 \leq i \leq j \leq n$, we are given a cost $c(i, j) > 0$, and the task is to find an increasing sequence of cutpoints $i_1, i_2, \ldots, i_k \in \{1, 2, \ldots, n-1\}$ so as to minimize the total cost $\sum_{s=0}^{k} c(i_s + 1, i_{s+1})$, where $i_0 = 0$ and $i_{k+1} = n$. Give an algorithm that solves the problem in $O(n^2)$ time.
+4 for correct algorithm for computing the optimal total cost (only +2 if almost correct, and +1 partially correct but some major flaws); +2 for correct analysis of the running time. +1 if dynamic programming recursion correct; +1 if proved, e.g., via the optimal substructure property; +1 if returning an optimal sequence of cutpoints.

Let $i_1, i_2, \ldots, i_k$ be the cutpoints of an optimal segmentation of range $[1, n]$. Now the cutpoints $i_1, i_2, \ldots, i_{k-1}$ form an optimal segmentation of range $[1, i_k]$. To see this, assume that it is not true, that is, there is an increasing sequence of cutpoints $j_1, j_2, \ldots, j_h$ with smaller total cost for range $[1, i_k]$. But now we have $\sum_{s=0}^{h} c(j_s + 1, j_{s+1}) + c(i_k + 1, n) < \sum_{s=0}^{k} c(i_s + 1, i_{s+1})$, where $j_0 = 0$ and $j_{h+1} = i_k$. Thus $i_1, i_2, \ldots, i_k$ cannot be an optimal segmentation, which is a contradiction. This is called the optimal substructure property.

Let now $s[j]$ be the total cost of an optimal segmentation of range $[1, j]$. The cost of an optimal solution for a complete problem is $s[n]$. Because of the optimal substructure property, we get the following recurrence:

$$s[j] = \begin{cases} 0 & \text{if } j = 0 \\ \min\limits_{0 \leq i < j} \{s[i] + c(i+1, j)\} & \text{if } j > 0 \end{cases}$$

Using this recurrence, it is now easy to implement a dynamic programming algorithm, which computes $s[j]$ for $j = 0, \ldots, n$ and a backtracking table $b[j]$ which can be used to reconstruct an optimal solution. The algorithm is as follows:

```
PRINT-OPTIMAL-SEGMENTATION(c, n)
    Input: costs c and n
    Output: prints the cutpoints of optimal segmentation
 1  let s[0..n] and b[0..n] be new arrays
 2  s[0] ← 0
 3  for j = 1 to n do
 4      q ← ∞
 5      for i = 1 to j − 1 do
 6          if s[i] + c(i + 1, j) < q then
 7              q ← s[i] + c(i + 1, j)
 8              b[j] ← i
 9      s[j] ← q
10  PRINT-SEGMENTATION-REVERSE(b, n)
```

The following subprocedure is used to reconstruct the solution (in reverse order, but it is easy to reverse the order in linear time if required):

```
PRINT-SEGMENTATION-REVERSE(b, n)
    Input: backtracking array b and n
1  while b[n] > 0 do
2      print b[n]
3      n ← b[n]
```

The main algorithm has two nested loops, both performing at most n iterations. The running time of those is thus $O(n^2)$. PRINT-SEGMENTATION-REVERSE has a loop, the body of which is executed k times, where k is the number of cutpoints in the optimal segmentation. This is always at most n, so the running time of PRINT-SEGMENTATION-REVERSE call is $O(n)$. Therefore, the total time requirement of the algorithm is $O(n^2)$.

---

3 (max 9 points) The *set-partition problem* takes as input a set S of numbers. The question is whether the numbers can be partitioned into two sets A and $S \setminus A$ such that $\sum_{x \in A} x = \sum_{x \in S \setminus A} x$. Show that the set-partition problem is NP-complete. (*Hint:* Reduce from SUBSET-SUM that you may assume is known to be NP-complete).
+3 for showing that the problem is in NP; +3 for trying to find a polynomial-time reduction (even if not correct); +3 if the reduction is correct (only +1 if poly-time but not quite correct, only +1 if correct but not quite poly-time). Example: if only gives a correct reduction but not proving it is poly-time and correct, then 0+2+1=3 points.

First we show that the set-partition problem belongs to NP. Given the set S, our certificate is a set A which is a solution to the problem. The verification algorithm checks that $A \subseteq S$ and that $\sum_{x \in A} x = \sum_{x \in S \setminus A} x$. Clearly this can be done in polynomial time.

To show that the problem is NP-complete we reduce from SUBSET-SUM. Let (S, t) be and instance of SUBSET-SUM. The problem is to determine whether there is a subset $A \subseteq S$ such that $t = \sum_{x \in A} x$. We construct an instance S' of the set-partition problem by setting $S' = S \cup \{r\}$ where $r = 2t - \sum_{x \in S} x$. Clearly this reduction can be done in polynomial time.

Now it remains to show that there is a subset of S whose sum is t if and only if S' can be partitioned into two distinct subsets of equal weight. First, suppose that $A \subseteq S$ and the sum of elements in A is t. But now we have $\sum_{x \in S' \setminus A} x = \sum_{x \in S \setminus A} x + r = 2t - \sum_{x \in A} x = 2t - t = t = \sum_{x \in A} x$. Thus, the partition into A and $S' \setminus A$ is a solution to the set-partitioning problem.

Now, suppose that there exists $A \in S'$ such that $\sum_{x \in A} x = \sum_{x \in S \setminus A} x$. Without loss of generality, assume that $r \notin A$. But now we have $2t = \sum_{x \in S} x + r = \sum_{x \in S'} x = \sum_{x \in A} x + \sum_{x \in S \setminus A} x = 2 \sum_{x \in A} x$. Thus $\sum_{x \in A} x = t$ and A is a solution to the subset-sum problem.

4 (max 9 points) Show that the following algorithm is correct and runs in $O(n^2)$ time:

```
Input: An undirected graph G in adjacency list representation,
       vertices labeled by 1, 2, ..., n.
Output: "YES" if there is a simple cycle of four edges in G,
       and "NO" otherwise.
 1. Initialize an array C[1..n, 1..n] to all-zeros.
 2. For i = 1 to n
 3.   For each pair (j, k) of distinct neighbors of i in G
 4.         if C[j, k] == 1
 5.           Return "YES"
 6.         else
 7.           C[j, k] = 1
 8. Return "NO"
```

+5 for showing correctness (+3 if only one direction of "iff" is shown); +4 for correct analysis of the running time (+3 if does not note the importance of the assumed adjacency list representation compared to adjacency matrix representation, or otherwise note that the pairs of neighbors have to be traversed through efficiently enough).

Correctness: We need to show that the algorithm returns "YES" if and only if there is is a simple cycle of four edges in G. First assume that such cycle exists. Denote the four nodes of the cycle by a, b, c and d such that the cycle contains edges ab, bc, cd and da. Without loss of generality, assume that $a < c$. Once the outer loop reaches ath iteration, that is $i = a$, the algorithm either returns "YES" or sets $C[b,d] = 1$ as (b,d) is a pair of distinct neighbors of a. (It is also possible, that the algorithm returns "YES" before this happens.) Now, unless not returned "YES" before, when the outer loop reaches cth iteration, the algorithm returns "YES" since the pair b and d are also neighbors of c and $C[b,d] = 1$.

Now assume that the algorithm returns "YES". This means that at that the nodes $i = a$, j and k are distinct, there are edges aj and ak in G and at that point $C[j,k] = 1$. The last fact implies, that the algorithm has set $C[j,k] = 1$ earlier at some point for some $i = b < a$. Thus, there must also be edges bj and bk in G. But this means, that there is a simple cycle of four edges in G.

Running time: Line 1 takes $O(n^2)$ time since the array C contains n^2 elements. Line 2 is executed n times and each takes a constant time. As the algorithm is using adjacency list representation, traversing through the distinct pairs of neighbors on line 3 can be done in constant time per pair. (For example, by using two nested loops, one for j and the other for k.) Lines 4–7 take also constant time per pair. The running time of lines 2–7 is thus dominated by the number of times the inner loop body is executed. On each execution of the inner loop body, the algorithm either terminates or changes $C[i,j]$ from 0 to 1 for some pair (i,j). As each $C[i,j]$ can be set to 1 only once and there are $O(n^2)$ distinct pairs (i,j), the inner loop is executed $O(n^2)$ times. Therefore, the algorithm runs in $O(n^2)$ time.

---

5 (max 9 points) Show that there is a polynomial-time randomized algorithm that given a 3-CNF-SAT formula ϕ as input, outputs a truth assignment that, in expectation, satisfies at least $7/8$ of the clauses of ϕ. You may assume that each clause in the formula consists of exactly three distinct literals. You may also assume the availability of a routine RANDOM(0, 1) that returns 0 with probability $1/2$ and 1 with probability $1/2$. (*Hint:* Consider a truth assignment drawn uniformly at random.)
+3 for understanding the problem (what a clause is, what a truth assignment is, what polynomial-time means); +2 for suggesting a correct algorithm and observing its running time; +4 for correct calculation of the expectation (+3 if not handling the issue of having a clause that contains both a variable and its negation).

We consider an algorithm that draws a truth assignment uniformly at random, that is, for each variable x_i we call RANDOM(0,1) set x_i to the returned value. Because RANDOM(0,1) runs in polynomial time, this is a polynomial-time algorithm.

Let m be the number of clauses in ϕ. For each $j = 1,\ldots,m$ we define an indicator random variable X_j, which gets value 1 if the clause ϕ_j is satisfied by the assignment returned by the algorithm, and 0 otherwise.

The number of satisfied clauses is then given by $X = \sum_{j=1}^{m} X_j$. Now, for any j, the clause ϕ_j either contains three distinct variables or contains both a variable and its negation. In the first case there are 8 possible truth assignment for the tree variables, each having equal probability. As only one of those assignment does not satisfy the clause, we have $\mathbf{E}[X_j] = \Pr(X_j = 1) = 7/8$. In the second case, when the clause contains both a variable and its negation, the clause is always satisfied and thus we have $\mathbf{E}[X_j] = 1$. Thus, in any case $\mathbf{E}[X_j] \geq 7/8$. By linearity of expectation the expected number of satisfied clauses is now

$$\mathbf{E}[X] = \mathbf{E}\left[\sum_{j=1}^{m} X_j\right] = \sum_{j=1}^{m} \mathbf{E}[X_j] \geq \sum_{j=1}^{m} \frac{7}{8} = \frac{7}{8}m.$$

---

6 (max 9 points) Suppose somebody gives you an algorithm $\mathcal{A}$ that, given an n-vertex undirected graph and one of its vertex s as input, *decides* in $O(\beta^n)$ time whether the graph contains a hamiltonian path that starts from s. Here $\beta > 1$ is a constant independent of n. Using $\mathcal{A}$ as subroutine, give another algorithm that, given the graph and the vertex s as input, *constructs* such a hamiltonian path when one exists. The algorithm should be as fast as possible. (Time bound $O(\beta^n n^k)$, with some constant k, yields 5 points; $O(\beta^n n)$ yields 7 points; $O(\beta^n \log n)$ yields full points.)
Points as described above. If justification of the running time bound is missing or the bound not shown at all, then give 2 and 3 points less, respectively. Example: if a correct and fastest algorithm is described but running time bounds not given, then give 9-3=6 points.

Let G be the input graph with n vertices. If G contains only one vertex, namely s, the problem is trivial. Otherwise, let $N(s)$ be the set of neighbors of s in the graph. If $N(s) > 1$, then the algorithm partitions $N(s)$ into two (almost) equally sized subsets A and B such that $|A| = \lceil N(s)/2 \rceil$ and $|B| = \lfloor N(s)/2 \rfloor$. Let G_A denote the modified graph, from which the edges between s and B are removed, and G_B the graph, from which the edges between s and A are removed. Since G contains at least one hamiltonian path starting from s, also either G_A or G_B contains the same path. In addition, any hamiltonian path in G_A or in G_B is also a hamiltonian path in G. The algorithm calls $\mathcal{A}$ with G_A and s as input. If $\mathcal{A}$ returns true, then G_A contains a hamiltonian path starting from s and the algorithm replaces G by G_A. Otherwise G_B must contain a hamiltonian path starting from s and the algorithm replaces G by G_B. The algorithm continues this way halving $N(s)$ until it contains only one vertex, say u.

At this point we know, that u must be the next vertex on any hamiltonian path starting from s. Let now $G \setminus s$ denote the subgraph that results when s and the related edges (in our case only edge between s and u) are removed from G. The subgraph $G \setminus s$ contains a hamiltonian path starting from u if and only if G contains a hamiltonian path starting from s. And from any such hamiltonian path in $G \setminus s$ we can construct a hamiltonian path in G by adding the vertex s before u. The algorithm now recursively constructs a hamiltonian path starting from u in $G \setminus s$ and concatenates s in the beginning to get a hamiltonian path in G.

The algorithm performs n iterations. On each iteration the number of vertices is reduced by one. Let $n_i = n - i + 1$ denote number of vertices left on ith iteration. On the beginning of ith iteration, the number of neighbors of s is $|N(s)| \leq n_i \leq n$. Thus the number of halving steps needed until $|N(s)| = 1$ is $O(\log n)$. (It is easy to prove this by induction or by using the master method.) The subgraphs G_A and G_B can be constructed in time polynomial in n_i, say $O(n_i^c)$ where c is a constant. As $\mathcal{A}$ is called once per halving step and each call takes $O(\beta^{n_i})$ time, the time requirement of ith iteration is $O((\beta^{n_i} + n_i^c)\log n) = O(\beta^{n_i} \log n)$. The total time requirement of the algorithm is therefore

$$\sum_{i=1}^{n} O(\beta^{n_i} \log n) = O\left(\log n \cdot \sum_{i=1}^{n} \beta^i\right) = O\left(\log n \cdot \beta \frac{1-\beta^n}{1-\beta}\right) = O(\beta^n \log n).$$