SOLUTIONS
AA

# Comprehensive Exam: Algorithms and Concrete Mathematics Autumn 2000

1. (*10 points*) *Big-Oh notation, Running times*
   For each of the following functions, circle the best upper bound from among the choices given. You only need to circle one answer in each case. For example, if you say that a function is $O(n)$, you do not need to also say that it is also $O(n^2)$, $O(n \log n)$, and so on.

   (a) (*1 points*) $f(n) = \sqrt{n}$
   Choose: $O(n)$ $O(n^2)$ $O(n^3)$ $O(\log n)$ $O(n \log n)$ $O(2^n)$ none
   **Answer:** $O(n)$

   (b) (*1 points*) $f(n) = (\sqrt{n}+1)(\sqrt{n}-1)\sqrt{n}$
   Choose: $O(n)$ $O(n^2)$ $O(n^3)$ $O(\log n)$ $O(n \log n)$ $O(2^n)$ none
   **Answer:** $O(n^2)$

   (c) (*2 points*) $f(n) = 422n^3 + 5n^2 \log n + 121231234$
   Choose: $O(n)$ $O(n^2)$ $O(n^3)$ $O(\log n)$ $O(n \log n)$ $O(2^n)$ none
   **Answer:** $O(n^3)$

   (d) (*2 points*) $f(n) = n^n$
   Choose: $O(n)$ $O(n^2)$ $O(n^3)$ $O(\log n)$ $O(n \log n)$ $O(2^n)$ none
   **Answer:** None of the above.

   For each of the following programs (or fragments thereof) give a good upper bound on the running time of the algorithm using "big-Oh" notation, as a function of the value of $n$.

   (e) (*2 points*)

   ```
   for $i$ from 1 to $n$ do
       for $j$ from $i$ to $n$ do
           ⟨ something that is $O(\log n)$ ⟩
   ```

   **Answer:** $O(n^2 \log n)$

   (f) (*2 points*)

   ```
   /* print $n$ in binary if $n > 1$; mod runs in constant time */
   bprint($n$)
       if $n > 0$ then
           bprint($\lfloor n/2 \rfloor$);
           print($n$ mod 2);
   ```

**Answer:** $O(\log n)$

2. (*20 points*) *Recurrences.*
Solve each of these three recurrences. You may give an exact solution, or give a good upper-bound using big-oh notation.

   (a) (*5 points*) If $f(n) = f(n/2) + \log n$, with $f(1) = 0$, then find a closed form expression for $f(n)$.
   **Answer:** $f(n) = (\log^2 n + \log n)/2 = O(\log^2 n)$

   (b) (*5 points*) If $g(n) = 2g(n/2) + \sqrt(n)$, with $g(0) = 0$ and $g(1) = 1$, then find a closed form expression for $g(n)$.
   **Answer:** $g(n) = \frac{\sqrt{2}+2}{3}n - \frac{\sqrt{2}+1}{3}\sqrt{n} = O(n)$

   (c) (*10 points*) If $h(n) = h(n/2) + h(n/4)$, with $h(0) = h(1) = 1$, then find a closed-form expression for $h(n)$.
   **Answer:** $h(n) = F_{\log n}$ where $F_n$ is the $n$th Fibonacci number.

3. (*10 points*) *Matching.*
One day, upon returning from the laundromat, I realize that, as usual, I've lost a few socks. In fact, no two of my socks are an exact match! Luckily, though, some of them are close and I have a similarity scoring function: any two socks $i$ and $j$ have an associated score $0 \leq s_{ij} \leq 1$ which is large if they are very similar and small if they are very different. Naturally, $s_{ij} = s_{ji}$ for all $i$ and $j$. I have an even number of socks and decide to try to put my socks together two-by-two so as to maximize the sum of scores.

I decide to use the following greedy algorithm:

```
Repeat until all socks have mates:
  • Pick a random unmated sock $i$.
  • Among the other unmated socks, find the sock $j$ so that $s_{ij}$ is maximum
    (i.e., no other unmated sock $k$ has $s_{ik} > s_{ij}$).
```

Show that this method may produce a poor matching. More specifically, the *score* for a matching is the sum of the similarty scores for the pairs I pick. Show that for any $0 < \delta < 1$, there is a set of socks, and similarity scoring function as described above, so that if I pick socks in a particular order I will get a score less than $\delta$ times the score for the best overall matching of this set of socks.

**Answer:** Consider four socks, $a$, $b$, $c$, and $d$, with $s_{ab} = 1$, $s_{bc} = \frac{\delta}{2}$, and all other scores equal to 0. If the first sock I randomly pick is $c$, I will match it with $b$ for a score of less than $\delta$, where the best possible score (from putting $a$ with $b$ and $c$ with $d$) would be 1.

/1

4. *(10 points)* *Amortized Analysis.*
A bank offers a combined savings/checking account with options to deposit \$1000 to the savings account, deposit \$1000 to checking, withdraw an integer multiple of \$1000 from the savings account, withdraw an integer multiple of \$1000 from checking, or transfer an integer multiple of \$1000 from the savings to the checking account. Each of these operations has a certain overhead cost to the bank, but actually storing the money costs them nothing. (In the event that you attempt to withdraw more money from an account than you actually have in that account, or to transfer more money from savings than you have in savings, the operations fails and costs nothing to the bank.) Below is a table of the operations and the overhead costs to the bank for each of them (assume $\alpha$ and $\beta$ are constants):

| Option | Actual Cost to the Bank |
|---|---|
| Deposit \$1000 to Savings | $\alpha$ |
| Deposit \$1000 to Checking | $\alpha$ |
| Withdraw $k \times$ \$1000 from savings | $\alpha + k\beta$ |
| Withdraw $k \times$ \$1000 from checking | $\alpha + k\beta$ |
| Transfer $k \times$ \$1000 from savings to checking | $\alpha + k\beta$ |

(a) *(5 points)* Say you open an account today with a balance of \$0. Show that the bank can offset the overhead costs from your first $n$ operations by simply charging an $O(1)$ service fee every time you deposit to savings or checking. That is, give an amount the bank could charge so that, regardless of your first $n$ operations, the overhead would not exceed the total service charge, and show why this fee would offset the overhead costs.

**Answer:** Amortized cost of $3\alpha+2\beta)$ for each deposit to savings, and of $2(\alpha+\beta)$ for each deposit to checking. Every thousand dollars deposited to savings costs $\alpha + \beta$ exactly, and then can cost at most twice this much again, if it is transferred to checking and then withdrawn (if it is withdrawn directly from savings, or if it is transferred or withdrawn in along with more money, the individual cost of moving just this thousand dollars is even less than $2(\alpha + \beta)$). Therefore the cost given is sufficient. A similar argument can be made for deposits to checking.

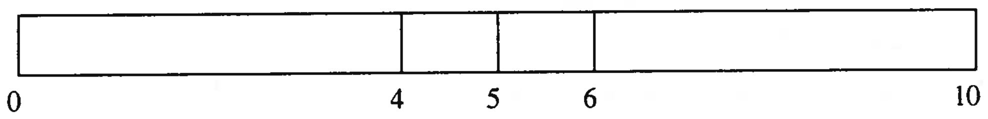
(b) *(5 points)* Now say the bank adds a sixth option: "transfer an integer multiple of \$1000 from *checking* to *savings*," and assume that the bank decides it will still only charge a service fee for deposits, and that this fee will still be constant. Show that, no matter what the service fee is, you could open a new account (also with an initial balance of \$0) and perform a series of operations so that the bank would lose money.

**Answer:** Assume the bank applies a constant fee of $l$ for each deposit to savings, and of $m$ for each deposit to checking. Let $k = \max(l, m)$. Simply deposit one

12

thousand dollars into savings and then transfer it back and forth over an over. If you transfer it back and forth $\frac{2k}{\alpha+\beta}$ times (rounding up to the nearest integer, of course), you will cost the bank $2k$.

5. ~~4~~ **(10 points)** *Dynamic Programming.*
Assume you have an $l$-foot log (i.e., a tree trunk, not a logarithm) that has a few marks spraypainted on it, indicating that you need to saw it at these places. For example, the log may look like the one below:



where the numbers indicate distance from the left end of the log. Assume further that the cost of making a particular cut is the length of the section in which you make the cut. For example, in the diagram, if we cut first at 4, then at 5, then at 6, the cost is $10 + 6 + 5$.

Give an efficient, dynamic-programming algorithm for deciding the cheapest cut order. More precisely, assume you are given a list $L = (l_1, l_2, ..., l_k)$ of cuts, where the $i$th cut $l_i$ is given as the distance between the left end of the log and the place at which that cut is to be made. (The above log would have the list $L = (4, 5, 6)$.) Your algorithm should take such a list and output an ordered list $L'$ that gives the cuts in the cheapest order.

Also, give a big-Oh time bound for your algorithm.

**Answer:** First, let us define $l_0 = 0$ and $l_{k+1} = l$. We will have a $k+1 \times k+1$ array $C$ where the $ij$th entry, $c_{ij}$ gives the cheapest cost of making the cuts *between* $l_i$ and $l_j$ (not including the $i$th and $j$th cuts), when $i \leq j$. Initialize $c_{ii} = 0$, for all $0 \leq i \leq k+1$, $c_{i,i+1} = 0$, for all $0 \leq i \leq k$, $c_{i,i+2} = l_{i+2} - l_i$, for all $0 \leq i \leq k-1$. (If you care, set to zero all entries $c_{ij}$ where $i > j$.)

Next, we will have another array $D$ of the same dimensions, where the $ij$th entry $d_{ij}$ gives the first cut we should make between $l_i$ and $l_j$, given that those two have been made already. Initialize this array to have all $-1$s, if you like.

Now, for each $g$ in $3 \ldots k+1$, letting $i$ range from 0 to $k+1-g$, find $d_{i,i+g}$. How do we do this? Well, $c_{i,i+g} = l_{i+g} - l_i + \min_{i<h<i+g} c_{ih} + c_{h,i+g}$, so, using the array as a lookup table, find the best value of $h$ for this particular $c_{i,i+g}$, and set $c_{i,i+g}$ and $d_{i,i+g}$ accordingly. At the end, $d_{0,k+1}$ will have the first cut, say $l'_1$, that you should make, and you can figure out the next two cuts as $d_{0,l'_1}$ and $d_{l'_1,k}$, and so on.

This algorithm runs in time $O(k^2)$.

13