Analysis of Algorithms - Final (Solutions)

K. Subramani
LCSEE,
West Virginia University,
Morgantown, WV
{ksmani@csee.wvu.edu}

1 Problems

1. **Induction and Recurrences:**

   (a) Professor Rabinowitz claims that the following property is true of all positive integers n: Either n is a power of 2, or there is some number between n and $2 \cdot n$, which is a power of 2. Prove that the Professor is correct.

   (b) Solve the following recurrence (exactly or asymptotically):

$$\begin{aligned} T(n) &= 1, \text{ if n} = 1 \\ &= 2 \cdot T(n-1) + 1, \text{ if n} \geq 2. \end{aligned}$$

   Solution:

   (a) Let $P(n)$ denote the proposition that either n is a power of 2, or there exists some number between n and $2 \cdot n$, which is a power of 2. Observe that $P(1)$ is **true** by inspection. Now assume that $P(n)$ is **true** for some fixed k, where $2 \leq n \leq k$, and consider $n = k+1$. If $k+1$ is a power of 2, we are done. Assuming it is not, we consider the following cases, which are exhaustive and mutually exclusive.

      (i) k is a power of $2 -$ Note that $2 \cdot k$ is also a power of 2 and further that $(k+1) \leq 2 \cdot k \leq 2 \cdot (k+1)$, for $k \geq 1$. It follows that there is a power of 2, between $(k+1)$ and $2 \cdot (k+1)$ and hence $P(k+1)$ is **true**.

      (ii) k is not a power of 2 - By the inductive hypothesis, there must be a power of 2 between k and $2 \cdot k$. Let us call this number l; clearly, $(k+1) < l < 2 \cdot k < 2 \cdot (k+1)$, since we have assumed that $k+1$ is not a power of 2. It follows that $P(k+1)$ is **true**.

   We thus seee that $P(k) \rightarrow P(k+1)$ and applying the first principle of mathematical induction, we can conclude that the Professor is correct.

   (b) (i) Exact solution - Applying the definition, we see that $T(1) =$, $T(2) = 3$, $T(3) = 7$ and thus it seems reasonable to conjecture that $T(n) = 2^n - 1$. We now proceed to prove the correctness of this conjecture using induction. Observe that the basis is **true**, since $T(1) = 1 = 2^1 - 1$. Assume that $T(k) = 2^k - 1$. Now,

$$\begin{aligned} T(k+1) &= 2 \cdot T(k) + 1, \text{ by definition} \\ &= 2 \cdot (2^k - 1) + 1, \text{ by the inductive hypothesis} \\ &= 2^{k+1} - 1 \end{aligned}$$

      Applying the first principle of mathematical induction, we can conclude that our conjecture is correct.

      (ii) We can also solve this problem using the Master Theorem. Substitute $n = \log_2 k$, where we assume without loss of generality that k is a power of 2. It follows that $n = 2^k$ and that $(n-1) = \log_2 \frac{k}{2}$. We can then rewrite the given recurrence as:

$$\begin{aligned} T(\log_2 k) &= 1, \text{ if } \log_2 \text{k} = 1 \\ &= 2 \cdot T(\log_2 \frac{k}{2}) + 1, \text{ if } \log_2 \text{k} \geq 2 \end{aligned}$$

Substituting $G(k) = T(\log_2 k)$, we get,

$$
\begin{aligned}
G(k) &= 1, \text{ if k} = 2 \\
&= 2 \cdot G(\frac{k}{2}) + 1, \text{ if k} \geq 4
\end{aligned}
$$

The recurrence is now in the form required by the Master theorem and applying the theorem, we conclude that $G(k) \in \Theta(k)$, from which it follows that $T(n) \in \Theta(2^n)$.

□

2. **Counterexamples:** In class, we showed that the fractional knapsack problem can be solved efficiently using a greedy approach. Now consider the $0/1$ knapsack problem, in which the objective (maximizing profit) and constraint (respecting knapsack capacity) are identical to the fractional knapsack problem; however, an object is either completely selected or completely discarded, i.e., you cannot select a fraction of an object. Does the greedy strategy discussed in class result in an optimal solution for the $0/1$ knapsack problem? If so, provide a proof of the same; if not, provide a numeric counterexample.

   Solution: Consider three objects $o =< o_1, o_2, o_3 >$, with respective profit values $p =< 50, 60, 140 >$, respective weights $w =< 5, 10, 20 >$ and knapsack capacity $M = 30$. The greedy approach that succeeded in the fractional case will choose objects o_1 and o_3 (with profit 190), whereas the optimal solution is to choose o_2 and o_3 (with profit 200). It follows that the greedy approach is not optimal for the $0/1$ knapsack problem. □

3. **Graph Theory:** Let $\mathbf{G} =< \mathbf{V}, \mathbf{E} >$ denote an undirected, unweighted graph. The *square* of $\mathbf{G}$ is the graph $\mathbf{G^2} =< \mathbf{V}, \mathbf{E^2} >$ defined as follows: $\mathbf{G^2}$ has the same vertex set as $\mathbf{G}$. (v_i, v_j) is an edge in $\mathbf{G^2}$, if and only if, either (v_i, v_j) is an edge in $\mathbf{G}$ or (inclusively) there exists a vertex v_k, such that (v_i, v_k) and (v_k, v_j) are edges in $\mathbf{G}$. Design an efficient algorithm to compute $\mathbf{G^2}$, given that $\mathbf{G}$ is stored using an adjacency matrix.

   Solution: Consider Algorithm (1.1) which computes the adjacency matrix M_2 of $\mathbf{G^2}$.

Function FIND-SQUARE-GRAPH ($M = \mathbf{G}$)

We assume that $\mathbf{G}$ is given to us an adjacency matrix M of dimensions $n \times n$, where n represents the number of vertices in $\mathbf{G}$.
{We first initialize the M_2 matrix.}
for $(i = 1$ **to** $n)$ **do**
    for $(j = 1$ **to** $n)$ **do**
        $M_2[i][j] = M[i][j]$.
    end for
end for
for $(i = 1$ **to** $n)$ **do**
    for $(j = 1$ **to** $n)$ **do**
        for $(k = 1$ **to** $n)$ **do**
            if $(M[i][k] = 1$ **and** $M[k][j] = 1)$ **then**
                $M_2[i][j] = 1$.
            end if
        end for
    end for
end for

Algorithm 1.1: Computing the Square of a graph

Algorithm (1.1) clearly runs in time $O(n^3)$. □

4. **Sorting and Searching:** Explain briefly how Randomized Quickselect selects the k^{th} smallest element of an n-element integer array, using $O(n)$ comparisons, in the expected case.

Solution: Let $\mathbf{A}[1 \cdot \cdot n]$ denote the array; without loss of generality, assume that the elements of $\mathbf{A}$ are distinct. Recall that the Randomized Quickselect algorithm chooses a random element as the pivot and partitions the elements of $\mathbf{A}$ about this element. Let j be the index at which the pivot element is placed, after the partitioning. If $j = k$, the algorithm returns $\mathbf{A}[j]$. If $k < j$, then the algorithm recurses on a smaller array, by finding the k^{th} smallest element in $\mathbf{A}[1 \cdot \cdot j]$. Finally, if $k > j$, the algorithm recurses by finding the $(k-j)^{th}$ smallest element in $\mathbf{A}[(j+1) \cdot \cdot n]$.

Let $t(n)$ denote the number of comparisons performed by the Randomized Quickselect algorithm on an array of n elements, in the worst case. Likewise, let $b(n)$ denote the worst-case number of comparisons in a single partitioning step and let $g(n)$ denote the worst-case number of recursive invocations that need to be made, before the size of the array drops to $\frac{3 \cdot n}{4}$. It follows that

$$t(n) \leq g(n) \cdot b(n) + t(\frac{3 \cdot n}{4}) \tag{1}$$

In a partitioning step involving n elements, at most n comparisons are made and hence $b(n) \leq n$. Taking expectations on both sides of Equation (1), we get,

$$\mathbf{E}[t(n)] \leq \mathbf{E}[g(n)] \cdot n + \mathbf{E}[t(\frac{3 \cdot n}{4})]$$

If the pivot element is chosen uniformly and at random, with probability at least one half, an element whose rank is between $\frac{1}{4}n$ and $\frac{3}{4}n$ is chosen; any such element immediately results in the size of the array decreasing to at most $\frac{3}{4}n$ and thus $\mathbf{E}[g(n)] = 2$. Setting $T(n) = \mathbf{E}[t(n)]$, we get,

$$T(n) \leq 2 \cdot n + T(\frac{3 \cdot n}{4}) \tag{2}$$

Equation (2) can be solved using the Master Theorem to get $T(n) \in \Theta(n)$; we have thus shown that the Randomized Quickselect algorithm performs $O(n)$ comparisons in the expected case.

$\square$

5. **Greedy Strategy:** The problem of *Coin changing* is concerned with making change for a specified coin value using the fewest number of coins, with respect to the given coin denominations. Assume that your coin denominations are quarters (25 cents), dimes (10 cents), nickels (5 cents) and pennies (1 cent) and that you have an infinite supply of the same. For instance, given a coin value of 31 cents, the optimal change value is 3 coins, viz., 1 quarter, 1 nickel and 1 penny. (Any other breakup will result in more than 3 coins!) Design a greedy algorithm for the Coin changing problem and argue its correctness.

Solution:

Algorithm (1.2) represents our greedy strategy.

Function OPT-COIN-CHANGE (n)

We assume that n represents the coin value that needs to be converted into quarters, dimes, nickels and pennies.
$c_q = \lfloor \frac{n}{25} \rfloor$; $n = n - c_q \cdot 25$.
$c_d = \lfloor \frac{n}{10} \rfloor$; $n = n - c_d \cdot 10$.
$c_n = \lfloor \frac{n}{5} \rfloor$; $c_p = n - c_n \cdot 5$.
return($c_d + c_d + c_n + c_p$)

Algorithm 1.2: Optimal Coin changing

In effect, Algorithm (1.2) converts as much of n into quarters. This is followed by converting as much of what is left over into dimes. Likewise, as much of the remainder that can be converted into nickels is so converted, with the remaining value represented by pennies. We need to argue that this strategy converts n into change, using the fewest number of coins.

Let $c =< c_q, c_d, c_n, c_p >$ represent the greedy solution and let $c' =< c'_q, c'_d, c'_n, c'_p >$ represent the output of a supposedly optimal algorithm (called $\mathcal{A}$), such that $c'_q + c'_d + c'_n + c'_p < c_q + c_d + c_n + c_p$.

We make the following crucial observation. Any optimal solution to the Coin changing problem must satisfy the following criteria (Why?):

(i) The number of pennies is at most 4,

(ii) The number of nickels is at most 1, and

(iii) The number of dimes is at most 2.

We refer to the above criteria as the *optimality conditions*; it is clear that the greedy strategy satisfies the optimality conditions.

Observe that $c'_q \leq c_q$, since $c_q = \lfloor \frac{n}{25} \rfloor$ represents the maximum amount that can be converted into quarters. If $c'_q < c_q$, we can convert $(c_q - c'_q) \cdot 25$ worth of coin value into quarters; doing so does not increase c'_d, c'_n or c'_p. To see this, note that each quarter that is missing in c'_q has been distributed among its dimes and nickels. Further as per the optimality conditions, a quarter can only be represented as 2 dimes and a nickel; replacing these three coins by a single coin clearly decreases the number of coins in $\mathcal{A}$'s solution, thereby contradicting its optimality. In other words, if $\mathcal{A}$ is optimal, $c_q = c'_q$. Likewise, given that $c_q = c'_q$, we must have $c'_d \leq c_d$. If $c'_d < c_d$, then each dime that is missing in c'_d is distributed among the nickels of $\mathcal{A}$'s output; however, since a single dime corresponds to two nickels, we can always increase c'_d to c_d while decreasing c'_n and not increasing c'_p. Thus, if $\mathcal{A}$ is to be optimal, we must have $c_d = c'_d$. An identical argument establishes that c_n must equal c'_n, from which it follows that c_p is equal to c'_p. In other words, the hypothesis that the number of coins output by $\mathcal{A}$ is less than the number of coins output by the greedy algorithm is false. We can therefore conclude that the greedy algorithm is optimal.

$\square$