CS 330 – Fall 2018 – Assignment 4 Solutions

Problem 1 (15 pts)**.** *Suppose you are choosing between the following three algorithms, all of which have O(1) base cases for size 1:*

1. *Algorithm A solves problems by dividing them into five subproblems of half the size, recursively solving each subproblem, and then combining the solutions in linear time.*
2. *Algorithm B solves problems of size n by recursively solving one subproblem of size $n/2$, one subproblem of size $2n/3$, and one subproblem of size $3n/4$ and then combining the solutions in linear time.*
3. *Algorithm C solves problems of size n by dividing them into nine subproblems of size $n/3$, recursively solving each subproblem, and then combining the solutions in $O(n^2)$ time.*

What are the running times of each of these algorithms (in asymptotic notation) and which would you choose? You may use the Master Method. Hint: Approach Algorithm B via the substitution method (pp. 211-217) to avoid tough summations. Also, solving that one as O(n^d) for the smallest valid integer d is all I'm looking for.

Algorithm A:
$T(n) = 5T(\frac{n}{2}) + O(n)$
By using the Master Theorem with $a = 5$, $b = 2$ and $c = 1$, we get $O(n^{\log_2 5})$.
Algorithm A runs in time O($n^{2.33}$).

Algorithm C:
$T(n) = 9T(\frac{n}{3}) + O(n^2)$
For Algorithm C, you can also use the Master Theorem $a = 9$, $b = 3$ and $c = 2$ (and $d = 0$).
Since $\log_3 9 = c$, this results to $O(n^2 \log n)$.

Algorithm B:
$T(n) = T(\frac{n}{2}) + T(\frac{2n}{3}) + T(\frac{3n}{4}) + O(n)$.
Algorithm B becomes complicated trying to use the Master Theorem, therefore, we will use substitution instead.
We can observe that $c(\frac{n}{2}) + c(\frac{2n}{3}) + c(\frac{3n}{4}) > cn$.
Similarly, $c(\frac{n}{2})^2 + c(\frac{2n}{3})^2 + c(\frac{3n}{4})^2 = c(\frac{36n^2}{144} + \frac{64n^2}{144} + \frac{81n^2}{144}) = c(\frac{181n^2}{144}) > cn^2$.
But, $c(\frac{n}{2})^3 + c(\frac{2n}{3})^3 + c(\frac{3n}{4})^3 < cn^3$, so we reach $d = 3$ to give the upper bound $O(n^3)$ for Algorithm B.

Therefore, after comparing the three algorithms, Algorithm C results in the fastest running time.

Problem 2 (10 pts)**.** *We are given a $n \times n$ board with one square colored black, all others colored white. Your task is to cover the board with "L" shaped tiles, such that all white squares are covered, the black square remains uncovered and there is no overlap among the L-shapes. An L-shape consists of three out of the 4 squares in a 2×2 square (thus one of the $2 \times 2 = 4$ squares is left uncovered by the shape). Design a divide-and-conquer algorithm to find a tiling of the board. Your algorithm will take as input the size n of one side(!) board and the position of the black square. You may assume that $n \geq 2$ is a power of 2. Prove the correctness of your algorithm and analyze its running time. For full credit you must write the recurrence for the running time in your analysis.*

Algorithm: We can use a divide-and-conquer approach for this problem as well. In the base case we have a 2×2 board with one black square. We can place exactly one L-shape on this board. Now let's look at arbitrary n. At every iteration we place an L-tile in the middle of the board, so that the square that is uncovered by the shape is in the same quadrant as the black square. We can now break up the problem into four smaller subproblems along the four quadrants of the board. One subproblem will be to find a tiling for the quadrant that contains the black square. The other three quadrants will each correspond to a subproblem as well, the square that is covered by the L-shape playing the role of the black square. We recursively call the algorithm on the quadrants, until we reach quadrants of size 2×2.

Observe that in every subproblem, if you know the size of the board and in which quadrant the black square is located, you can exactly **reconstruct the tiling**.

```
 void find_tiling(in, x, y)  // (x,y) are the coordinates of the black square
 {

find_tiling(n/2, x, y)

if !(x>n/2 &&  >n/2): find_tiling(n/2, 0, 0)

if !(x<=n/2 && y>n/2): find_tiling(n/2, n/2, 0)

if !(x>n/2 && y<=n/2): find_tiling(n/2, 0, n/2)

if !(x<=n/2 && y<=n/2): find_tiling(n/2, n/2, n/2)

return
 }
```

Running time: We can write the recurrence $T(n) = 4T(n/2)+O(1)$, hence the running time is $O(n^2)$ by the Master theorem. Observe that the input to the algorithm is n and not n^2!

Problem 3. *(15 pts) A bank has n bank cards which each belong to a bank account. Some cards may belong to the same bank account. It is infeasible to read the account information directly from the cards, but the bank provides a machine which can test whether two cards are registered to the same account.*

a:*(5 pts) Design an algorithm which determines whether there is a set of more than $\frac{n}{2}$ cards which belong to a single account. At worst it should use the machine $O(n \log\ n)$ times. Prove it is correct.*

b:*[10 pts] Now we'll try to solve it faster. Consider the following procedure when n is even: Arbitrarily pair up the cards. Test each pair and if the two elements are different, discard both of them; if they are the same, keep just one of them. Show that this procedure preserves the majority element: if the original batch of cards had a majority element, then the reduced batch of cards has the same majority element. Use this procedure to build a divide-and-conquer algorithm for all n. Analyze its running time by writing a recurrence relation and evaluating it.*

part a: This is the same problem that wasdiscussed in lab 6. Here is the proof from lab: We will use a divide and conquer approach to tackle this problem. The key observation here, is that if there is a majority element that appears in more than $n/2$ locations, then if we split the array into two halves, the same element will be a majority in at least one of the two halves. Note, though, that if an element is a majority in one of the two sides, then it is not necessarily a majority to the whole array. Thus, when a majority element is returned from one of the two halves, we have to compare it against all other elements. The algorithm is as follows:

```
function findMajority(A)
        n = |S|
        If n=2
                If element1 and element2 are equivalent
                        return element1
                else return nothing
        Let S_1 be the set of the first n/2 elements
        Let S_2 be the set of the remaining n/2 elements
        findMajority(S_1)
        If an element is returned
                then test this element against all other elements
```

```
If no element with majority has been found
        findMajority(S_2)
        If an element is returned
                then test this element against all other elements
Return a majority element if one is found
```

Correctness: The correctness of this algorithm follows from the initial observations. If there is a majority element in the array, then one of the two recursive calls, will return this element, and by comparing it with the remaining elements of the array we can make sure that it is indeed the majority element.
Running time: Let $T(n)$ be the maximum number of equivalence tests we need to make to find if there exists a majority element in the array. We do two recursive calls each one with $n/2$ the number of elements. Also, if one of the two recursive calls returns a "candidate" majority element, we need to test this against all other elements. This can be done in linear time $O(n)$. Our base case is when $n = 2$. Hence, the recurrence is the following:
$T(n) = 2T(n/2) + O(n)$, which implies that $T(n) = O(nlogn)$.

part b:
Algorithm: If n = 0 then there is no majority element. If n = 1 then return this element. If n is odd, remove one element from the cards, test if it is a majority and then run the algorithm on the remaining (even number of) cards. When n is even, recursively run the procedure stated in the assignment to get down to a base case. Coming out of the recursion, if an element e is returned, count the appearance of e in the cards and check if it is really a majority element to avoid a false positive. If it is, then return this value, otherwise there was no majority.

```
def majority(N):
        n = |N|;   newN = [];
        if n == 0:   return null
        if n == 1:   return N[0]

        if n is odd:  Remove one element e from N and use n-1
              comparisons to test if e is a majority.  If so, return e.

        Now n is even:     // do reduction procedure
             i = 0
             While i < len(N) - 1:
                    if N[i] == N[i+1]:  newN += [n[i]]
                    i += 2
             e = majority(newN)

        if e == null:  return null

        count = 0
        For i in range(len(N)):   // count # of e's
                if N[i] == e: count ++

        if count > n/2: return e, else return null
```

Show that the stated procedure on even n preserves the majority element:

1). First of all, the majority element must be in the resulting batch of cards: By definition, if n cards have a majority element, then this element occurs at least $n/2 + 1$ times (we assumed n is even). Thus, by pigeonhole principle, at least one pair of cards $(c1, c2)$ are both the majority card. By algorithm, we discard one element of the two, in other words we keep one of the majority element in resulting set.

2). Second, the majority element is still majority in resulting batch: Consider the counts of majority and non-majority elements as we perform comparisons in the reduction operation. Of the $n/2$ pairings, let's say that a have

two majority elements, b have exactly one majority element, and c have none. By the algorithm, this will result in exactly a copies of the majority element and at most c non-majority elements (if all the pairs in c happen to match, fewer if not) after the reduction operation. To add up to $n/2$ comparisons, $a + b + c = n/2$. Also, $2a + b > n/2$ (to ensure majority). Subtracting equation 1 from equation 2, gives us $a - c > 0$ or $a > c$. Therefore, majority elements outweigh non-majority after the procedure.

Correctness: At the end, the algorithm only returns a majority if it is brute-force tested against all elements. Therefore, there are no false positives. By the argument above, if a majority exists, the procedure preserves it, so it will be returned correctly.

Running time: The size of input is at least cut in half in majority(), and every time we compare or remove take O(1) time. Checking for an odd element takes O(n) time. Checking for false positive result at the end takes O(n) time, additive. Thus $T(n) = T(n/2) + O(n)$, with an O(1) base case. Using Master theorem, this is O(n) time.

Problem 4 (10 pts)**.** *Chapter 6, exercise 2, page 313.*

(a) Here's a counterexample. Low-stress payouts by week: 1, 10, 6. High-stress payouts by week: 1, 12, 1000. The proposed algorithm greedily looks only two weeks out and takes the high-stress job in week 2, and is forced to do the low-stress job in week 3. Clearly, it is better to save energy to do the stressful job in week 3.

(b) Define OPT(i) to be the optimal payout that can be achieved up through week i. Base cases: OPT(0) = 0, and OPT(1) = $\max(\ell_i, h_i)$, since the problem statement allowed you to do either job in week 1. In the general case, week i is a 2-way choice: either do the low-stress job and recurse on OPT($i-1$), or do the high-stress job, take the prior week off, and recurse on OPT($i-2$). Therefore:

$$\text{OPT}(i) = \max\,(\text{OPT}(i-1) + \ell_i, \text{OPT}(i-2) + h_i), \text{for } i > 1.$$

Assuming we are given low-stress and high-stress payouts as input, the optimal payouts can be computed iteratively from this recurrence:

```
DP (LowStress, HighStress, n)
{
   M[0] = 0,  M[1] = LowStress[1];   // base cases

   for (i = 2;  i <= n; i++)
   {
      low-stress-option = M[i-1] + LowStress[i]
      high-stress-option = M[i-2] + HighStress[i]

      M[i] = max(low-stress-option, high-stress-option)
   }
}
```

The value of the optimal solution, $M[n]$, is computed in O(n) time, since the loop iterates $n-2$ times and does O(1) work per iteration.

Problem 5 (10 pts)**.** *Chapter 6, exercise 9.*

example: There are many examples possible. Here is one: $s_1 = 32, s_2 = 16, s_3 = 8, s_4 = 4, s_5 = 2, s_6 = 1$ and $x_i = 40$. In this example you want to reboot every even day (except the last).

solution: Let $OPT(j)$ denote the maximum terrabytes we can process up until day j. Assume that the last reboot happened i days before j. Then on day j the amount we can process is $\min\{s_i, x_j\}$. The recursive formula we use to compute $OPT(j)$ takes the minimum over the possible days that the last reboot before j happened. Note that the revenue on day i is zero in this case.

$$OPT(j) = \begin{cases} 0, & \text{if } j = 0 \\ \min\limits_{i<j}[OPT(i-1) + \sum\limits_{k=i+1}^{j} \min\{s_k, x_k\}], & \text{otherwise} \end{cases}$$

Observe that $OPT(j)$ has only one variable, hence we can use a 1-dimensional array for the purpose of memoization.

```
RebootSchedule(x_1, x_2, ..., x_n, s_1, s_2, ..., s_n ):
 M = [ ]
 M[0] = 0

 For j = 1 to n:
      M[j] = 0
      for i = 1 to j:
         M[j] = min( M[j], M[i-1] + sum_{k=i+1...j} min(s_i,x_i) )

return M[n]
```

Observe that this solution only provides the total number of TB processed. In order to find the actual reboot days we need to backtrack the solution. For this we need to keep track for every j what the optimal i is.

running time: The outer for loop has n iterations, the inner loop has $O(n)$ iterations and each iteration contains an $O(n)$ summation. Hence, the total running time is $O(n^3)$. (Another way to look at the inner for loop is to say that in iteration i the loop computes the sum of i-1 items. Hence the total number of items added over all iterations is $\sum_{k=1}^{j} k = \frac{j(j+1)}{2} = O(j^2)$)