Quiz #1
Introduction to Algorithms
601.433/633
Spring 2020
Mou Zhang

You must submit your solutions by Wednesday, April 8th, 10am. Late submissions will NOT be accepted. You may submit handwritten answers – you will have to scan/photograph them, convert it to a pdf and upload it to Gradescope.

1 Problem 1 (50 points)

For each statement below explain if it is true or false and in a couple of sentences provide an explanation for your answer. Be as mathematically precise as you can in your explanation. The base of $\log$ is 2 unless stated otherwise.

1. $2^{n^2} = \Theta(3^{n+\sqrt{n}})$

   False Since $\lim_{n\to\infty} \frac{3^{n+\sqrt{n}}}{2^{n^2}} = \lim_{n\to\infty} \frac{2^{(n+\sqrt{n})\cdot \log 3}}{2^{n^2}} = \lim_{n\to\infty} 2^{(n+\sqrt{n})\cdot \log 3 - n^2} = 0$, we know that for any positive constant C > 0 and some large n_0, such that $2^{n^2} > C \cdot 3^{n+\sqrt{n}}$ for $n \geq n_0$. Thus $2^{n^2} = \omega(3^{n+\sqrt{n}})$ and $2^{n^2} \neq O(3^{n+\sqrt{n}})$.

2. $n! = \omega(2^n)$

   True Since $\lim_{n\to\infty} \frac{2^n}{n!} = \lim_{n\to\infty} \frac{2}{1} \cdot \frac{2}{2} \cdot \frac{2}{3} \cdot ... \cdot \frac{2}{n} < \lim_{n\to\infty} \frac{2}{1} \cdot \frac{2}{2} \cdot \frac{2}{3} \cdot \frac{2}{3} \cdot \frac{2}{3} \cdot ... \cdot \frac{2}{3} = \lim_{n\to\infty} \frac{2}{1} \cdot \frac{2}{2} \cdot (\frac{2}{3})^{n-2} = 0$, we know that for any positive constant C > 0 and some large n_0, such that $n! > C \cdot 2^n$ for $n \geq n_0$. Thus $n! = \omega(2^n)$.

3. Let f be positive function. Then $f(n) = O((f(n))^2)$.

   False Assume $f(x) = e^{-x}$, which is a positive function. Then $(f(n))^2 = e^{-2x}$. Since $\lim_{n\to\infty} \frac{e^{-2x}}{e^{-x}} = \lim_{n\to\infty} e^{-x} = 0$, we know that for any positive constant C > 0 and some large n0, such that $e^{-x} > C \cdot e^{-2x}$ for $n \geq n_0$.

Thus $e^{-x} = \omega(e^{-2x})$. Then $f(n) = w((f(n))^2)$, which contradicts the statement.

2 Problem 2 (50 points)

Resolve the following recurrences in terms of a big-Θ bound. You may assume that $T(0) = T(1) = 1$. Provide a proof for the bound you give. If appropriate, you may invoke the Master Theorem (and the appropriate case).

1. $T(n) = T(n-1) + 2T(n-2)$

   Solution:
   (1) Adding $T(n-1)$ on the both sides of the equation, we have $T(n) + T(n-1) = 2T(n-1) + 2T(n-2) = 2(T(n-1) + T(n-2))$. Assume $Q(n) = T(n) + T(n-1)(n > 0)$, then we have $Q(n) = 2Q(n-1)$, $Q(1) = 2$.So $Q(n) = 2^n$, which means that $T(n) + T(n-1) = 2^n$
   (2) In the same way, we can figure out that $T(n) - 2T(n-1) = T(n-1) - 2T(n-1) + 2T(n-2) = -(T(n-1) - 2T(n-1))$. Assume $P(n) = T(n) - 2T(n-1)$, then $P(n) = -P(n-1)$ and $P(1) = -1$. Therefore, $P(n) = (-1)^n$, which means that $T(n) - 2T(n-1) = (-1)^n$.
   (3) From the 2 equations above, We can find out $2Q(n) + P(n) = 2T(n) + 2T(n-1) + T(n) - 2T(n-1) = 3T(n) = 2^{n+1} + (-1)^n$, which means that $T(n) = \frac{2^{n+1}+(-1)^n}{3} = \Theta(2^n)$

2. $T(n) = 7T(n/15) + n^5$

   Solution:
   $\forall n \geq n_0$ and large enough c, n_0, $n^{\log_{15} 7+\epsilon} < cn^5$ for $0 < \epsilon < 5 - \log_{15} 7$. So $f(n) = n^5 = \Omega(n^{\log_{15} 7+\epsilon}) = \Omega(n^{\log_b a+\epsilon})$. And for constant $c0 = 0.5 < 1$ and all sufficiently large n, $af(n/b) = 7(\frac{n}{15})^5 \leq c0 \cdot n^5$.Owing to case 3 of the master theorem, $T(n) = \Theta(f(n)) = \Theta(n^5)$.

3 Problem 3 (50 points)

A sequence $a_1, a_2, \ldots, a_n$ has a special element if more than half of the elements in the sequence are the same. For example, 3 is a special element in the sequence $7, 3, 3, 3, 1, 3, 3, 4, 5, 3$. On the other hand, the sequence $5, 4, 1, 1, 2, 3, 2, 3, 6$ has no special element. Give a divide and conquer algorithm that runs in time $O(n \log n)$ and returns a special element in a sequence of n numbers or returns $None$ if no such element exists. Prove the correctness of your algorithm and prove that its runtime is $O(n \log n)$. (Note: there exists an $O(n)$ algorithm to solve this problem that doesn't make use of divide and conquer– if you figure it out, you may prove its correctness and runtime instead.)

Solution:
You can see the solution in the pseudocode below.

Proof of correctness:

Proof. We gonna prove the correctness by induction. We assume that the special element exist in the proof below. If the number we find is not the special number, we gonna check it out in the last for loop. We gonna prove we can find the special element if it exists for different length of array of elements.

1. Base Case: length of array =1, It is obvious that the only element is special element. After we go through the whole algorithm, it will return the special element.

2. Induction Hypothesis: length of array = k, assume that for all length ¡=k, the final value returned after completing the loop is majority element.

3. Induction Step: length = k+1, there are two cases:

   (a) case 1: The first element is not the special element. Then at some time of our algorithm, the counter must become 0. Let's assume that there are p elements remain at this moment. The circumstance at this moment is the same as we have a array of p elements and try to find the special elements of these elements. Besides, the special elements in the remaining array is obviously the special elements in the origin array. Assume that the special element appears total $q(q > \frac{n}{2})$ times in the original array of n elements. Then it at most appears $\frac{n-p}{2}$ in the scanned

Algorithm 1 Find special element

```
1: procedure FIND SPECIAL ELEMENT(A)
2:     number, counter ← 0
3:     for i ← 1, n do
4:         if a_i ≠ number then
5:             if counter > 0 then
6:                 counter ← counter − 1
7:             else
8:                 number = a_i
9:                 counter ← 1
10:            end if
11:        else
12:            counter ← counter + 1
13:        end if
14:    end for
15:    counter ← 0
16:    for i ← 1, n do
17:        if a_i == number then
18:            counter ← counter + 1
19:        end if
20:    end for
21:    if counter > n/2 then
22:        return number
23:    else
24:        return None
25:    end if
26: end procedure
```

first $n - p$ elements. So in the rest p elements, the special element still appears $q - \frac{n-p}{2}$ times. It is obvious that $q - \frac{n-p}{2} > \frac{n}{2} - \frac{n-p}{2} = \frac{q}{2}$. So the origin special elements must be the special element in the new q elements. Because in the Induction Hypothesis we known that we can find the special element in q elements, so we know that we can find the special element in n = k+1 elements. True.

(b) case 2: The first element is the special element. Then there are two cases: 1) the first element go through the whole algorithm. Under this condition, it is obvious that we find the special element. 2) At some time the counter goes to 0. This circumstance is the same as case 1 above, so we can also find the special elements by this algorithm. True.

4. To sum up, we can find the special element in k+1 elements if the special element exists. The Induction is right.

5. It is easy to check whether it exists. We just need to go through the origin array and check out.

□

Proof of running time:

Proof. There are 2 for loops in the algorithm. The first for loops in lines 3-14 has on inner for loops and go from 1 to n, so the running time is O(n). The second for loops in lines 16-20 is also very simple and go from 1 to n, so the running time is O(n). The if parts only cost O(1). So the total Running time is O(n) + O(n) + O(1) = O(n). □

4 Problem 4 (50 points)

You are given n tasks for a machine. Task i is described by $l_i = [a_i, b_i]$ on the real line, where a_i, b_i are real numbers, $a_i \leq b_i$ and $1 \leq i \leq n$. Give an algorithm that computes the total times of this set of tasks, that is, the length of $\cup_{i=1}^{n} l_i$ in $O(n \log n)$ time.

For example, for the set of tasks $\{[1,3],[2,4.5],[6,9],[7,8]\}$, the total time is $(4.5-1)+(9-6)=6.5$.

Make sure to prove the correctness and running time of your algorithm.

Solution:

Algorithm 2 Compute total time

```
1:  procedure COMPUTE TOTAL TIME(L)
2:      QuickSort all tasks l_i in L by a_i increasingly
3:      starttime, endtime, ans ← 0
4:      for i ← 1, n do
5:          if a_i < endtime then
6:              endtime ← max(endtime, b_i)
7:          else
8:              ans ← ans + (endtime − starttime)
9:              starttime ← a_i
10:             endtime ← b_i
11:         end if
12:     end for
13:     ans ← ans + (endtime − starttime)
14:     return ans
15: end procedure
```

What we do is to sort all tasks in increasing order of their start time. After that we can go through every task one by one. If the start time of ith task a_i is bigger than all endtime before task i, than it is simple that every task after task i (including task i) has no sharing time with task before task i. Otherwise task i has sharing time with task before, and we shall calculating their sharing starttime and endtime. If they has no sharing time then we can calculate the running time for tasks before task i by calculating their working time as their $endtime$ - their $starttime$.

Proof of correctness:

Proof. We gonna use Induction to prove the correctness. In this algorithm, I calculate the time by sets of tasks. I separate all tasks into different sets of tasks and sum up the time of each set of task to get the final answer. Tasks in the same set means that they have sharing time. This means that each task in the set has some sharing working time with at least one other task in the same set. The ans saves the total time of previous finished sets of tasks, which has no sharing time with any task in the current set of tasks. The starttime saves the earliest start time of current set of tasks that have sharing time with current task. The endtime saves the latest end time of current set of tasks that have sharing time with current task.

Base Case: At the beginning, there are no task, so the ans is 0 and starttime and endtime are both 0.

Induction Hypothesis: Assume that for task k, the ans saves the total time of previous finished sets of tasks, which has no sharing time with any task in the current set of tasks. The starttime saves the earliest start time of current set of tasks that have sharing time with current task k. The endtime saves the latest end time of current set of tasks that have sharing time with current task k. What we need to prove is that after step k+1, ans, starttime and endtime still keeps what they should keep, which is described above.

Induction Step: For task k+1, there are 2 cases:

Case 1: This task has no sharing time with tasks before. In other words, this k+1 task belongs to a new set of tasks. Since We have sort the tasks in order of their start time, this also means that none of the task after this task has sharing time with tasks before. So the working time of set of tasks before this task can be calulated only use information before, which is endtime - starttime. This is because that the starttime saves the earliest start time of previous set of tasks and the endtime saves the latest end time of previous set of tasks. The difference of them is the total time used by the previous set of tasks. ans should be euqal to ans + endtime - starttime, which shows that ans countains all time used by previous sets of task. After that, starttime should be a_{k+1} and endtime should be b_{k+1} becuase task k+1 is the only task in the new set now. After this performancde, ans, starttime and endtime still keeps what should be kept in them. True.

Case 2: This task has sharing time with current set of tasks. This proves that task k+1 is still in the current set of tasks, together with task k. So this set of tasks is not finished and we should not calculate its usage of time right now. What we only need to do is to update starttime and endtime to keep them representing the

start time and end time of current set of tasks. In fact, we don't need to update the starttime because it is sorted in increasing order. After this operation, ans, starttime and endtime still keeps what should be kept in them. True.

In summary, ans, starttime and endtime always keep what should be kept, our induction is right.

After going through the whole algorithm, the ans saves the total time of previous finished sets of tasks(not including the last set). The starttime saves the earliest start time of last set of tasks The endtime saves the latest end time of last set of tasks. So the final answer is ans + (endtime - starttime). □

Proof of running time:

Proof. For line 2 the sorting part, the time complexity for quick sort is $\Theta(nlogn)$, which is proved in class. For lines 3-12, what I do is to go through each tasks and calculating the result, and time complexity for a for loop is $\Theta(n)$. So the total running time is $\Theta(n) + \Theta(nlogn) = \Theta(nlogn)$ □

5 Problem 5 (50 points)

Suppose that you have a set of n integers, $A = \{a_1, ..., a_n\}$, each of them is between 0 and K (inclusive). Your goal is to find a partition of A into two sets S_1 and S_2 (so $S_1 \cup S_2 = A$ and $S_1 \cap S_2 = \emptyset$) that minimizes $|W(S_1) - W(S_2)|$ where $W(S)$ denote the sum of integers in S. Your algorithm's running time should be polynomial in n and K.

Make sure to prove the correctness (mainly the optimal substructure property) and running time.

Solution:

Algorithm 3 Split golden coins

```
1: procedure SPLIT GOLDEN COINS(A)
2:     Initialization case j = 0 or i = 0, calculating T
3:     for i ← 1, n do
4:         for j ← 1, T/2 do
5:             DP[i, j] ← DP[i − 1, j]
6:             if g ≥ a_i then
7:                 DP[i, j] ← DP[i, j] OR DP[i − 1, j]
8:             end if
9:         end for
10:    end for
11:    for i ← 1, T/2 do
12:        if DP[n][i] is True then
13:            max_half_sum ← i
14:        end if
15:    end for
16:    return (T − max_half_sum) − max_half_sum
17: end procedure
```

Assume $T = \sum a_i$. The original problem is equal to the following statement: If such S_1, S_2 exist, such that $\sum_{i \in S_1} a_i - \sum_{i \in S_2} a_i$ is minimal(Let's assume that $W(S1) > W(S2)$). Since $W(S1) > W(S2)$, $W(S2) < T/2$, then to minimize $W(S1) - W(S2)$ is to find the largest W$(S2)$ such that $W(S2) = \sum_{i \in S_2} a_i$ is the maximum achievable value smaller than T/2. So I treat it as a backpack problem with capacity T/2.

1. Define the subproblems (and write down what it means). If not for anything but to help you define the algorithm.

   - DP[i, j] = If using the first i goods to get j total values is possible, which is the same as $\exists S_2$, s.t. $\sum_{i \in S_2} a_i = j$.(Presented by True/False)

2. Write down a recurrence showing how the larger subproblem can be solved in terms of the smaller ones.

   - $$DP[i,j] = \begin{cases} True & j = 0 \\ False & i = 0 \; and \; j \neq 0 \\ DP[i-1,j] & i \geq 0 \; and \; j < a_i \\ DP[i-1,j] \; OR \; DP[i-1,j-a_i] & i \geq 0 \; and \; j \geq a_i \end{cases}$$

3. Write down which subproblem gives the answer to the original question.

   - Output $(T - max\_half\_sum) - max\_half\_sum$.
   - Since W(S2) is the maximum achievable value smaller than T/2, We can simply use a for loop to find the maximum achievable value smaller than T/2, which is max_half_sum.
   - max_half_sum is W(S2) and $(T - max\_half\_sum)$ is W(S1). So the answer is W(S1) - W(S2).

4. Prove correctness by proving the "optimal substructure property". I.e. The recurrence relation defined by 2. is the solution for the subproblem as defined by 1.

   - We want to prove that the recurrence relation for DP[i, j] in 2. is the correct True/False of whether using some of the first i elements to get the total value j is possible.
   - To do so, we will induct on (i, j).
     (a) Base Case (i=0 or j = 0), the recurrence relation is correct. It is because when j = 0, there is always a way to get 0 value - to take nothing. When i = 0 and j $\neq$ 0, since there's no good, getting value of j is impossible.
     (b) IH : for i $\leq$ k1 and j $\leq$ k2, DP[i, j] is the the True/False of whether using some of the first i elements to get the total value j is possible.
     (c) Induction Step: For i = k1 + 1 and j = k2 + 1, there are two possible cases —

- Case 1 : take the i_{th} good in S_2. In this case, the True/False state of DP[i,j] is equal to the True/False state of using the first i - 1 goods to get a total value of j - a_i, which can be presented as DP[i, j] = DP[i - 1, j - a_i]. (This case only happens when $j > a_i$ because if $j < a_i$, the sum of the value when taking a_i must be bigger than j)
- Case 2 : don't take the i_{th} good in S_2. In this case, since we don't use the i_{th} good, the problem is equal to use the first i - 1 goods to get total value j. This can be presented as DP[i, j] = DP[i - 1, j]
- Since there are only 2 cases and either case above is true will make DP[i, j] becomes True, the maximum revenue must be the bigger one of them. So

$$DP[i,j] = \begin{cases} DP[i-1,j] & i \geq 0 \; and \; j < a_i \\ DP[i-1,j] \; OR \; DP[i-1,j-a_i] & i \geq 0 \; and \; j \geq a_i \end{cases}$$

  is the correct True/False state of whether using some of the first i elements to get the total value j is possible. True.

5. Prove runtime by counting the number of subproblems.

   - There are at most (n * $\frac{T}{2}$) unique subproblems. Using memoization, and by the definition of the algorithm in 2., each subproblem only calls other subproblems at most twice and since we do O(1) in each subproblem, the runtime of the algorithm is O(n * $\frac{T}{2}$) = O(n * n * k / 2) = O(n^2k). The initialization part takes O(n) time. The searching for max_half_sum in lines 11-15 takes O(nk) time. So the total time complexity is O(n^2k) + O(n) + O(nk)= O(n^2k).