**Bo Yang (Aaron) USCID: 7526922531**

# USC CSCI 570 Homework3 Date: 2021-02-22

## For all divide-and-conquer algorithms follow these steps:

1. Describe the steps of your algorithm in plain English.
2. Write a recurrence equation for the runtime complexity.
3. Solve the equation by the master theorem

## For all dynamic programming algorithms follow these steps:

1. Define (in plain English) subproblems to be solved.
2. Write the recurrence relation for subproblems.
3. Write pseudo-code to compute the optimal value
4. Compute its runtime complexity in terms of the input size.

## 1. Solve the following recurrences by giving tight Θ-notation bounds in terms of $n$ for sufficiently large $n$. Assume that $T(\cdot)$ represents the running time of an algorithm, i.e. $T(n)$ is positive and non-decreasing function of $n$ and for small constants $c$ independent of n, $T(c)$ is also a constant independent of $n$. Note that some of these recurrences might be a little challenging to think about at first. Each question has 4 points. For each question, you need to explain how the Master Theorem is applied (2 points) and state your answer (2 points).

- (a) $T(n) = 4 \cdot T(n/2) + n^2 \cdot log(n)$
- (b) $T(n) = 8 \cdot T(n/6) + n \cdot log(n)$
- (c) $T(n) = \sqrt{6006} \cdot T(n/2) + n^{\sqrt{6006}}$
- (d) $T(n) = 10 \cdot T(n/2) + 2^n$
- (e) $T(n) = 2 \cdot T(\sqrt{n}) + log_2 n$

**Solution:**

> for question **(a)**:
>
> > $a = 4, b = 2, c = log_b a = 2, f(n) = n^2 \cdot log(n), f(n) = \Theta(n^2 \cdot log(n))\ and\ k = 1$
> > $since\ k = 1 \geq 0,\ So\ match\ the\ case\ 2, then\ T(n) = \Theta(n^2 \cdot (log(n))^2)$
>
> for question **(b)**:
>
> > $a = 8, b = 6, c = log_b a = log_6 8 > 1, f(n) = n \cdot log(n), f(n) = O(n^{log_6 8 - \epsilon})$
> > $then\ T(n) = \Theta(n^{log_6 8})\ \ for\ some\ 0 < \epsilon < log_6 8 - 1$ matched case 2.
>
> for question **(c)**:
>
> > $a = \sqrt{6006}, b = 2, c = log_b a = log_2 \sqrt{6006} > 1, f(n) = n^{\sqrt{6006}}, f(n) = \Omega(n^{log_2 \sqrt{6006}}).$
> > $Mathced\ case\ 3,\ then\ T(n) = \Theta(n^{\sqrt{6006}})\quad for\ some\ \epsilon > 0$
>
> for question **(d)**:
>
> > $a = 10, b = 2, c = log_b a = log_2 10 > 1, f(n) = 2^n, f(n) = \Omega(n^{c+\epsilon})$
> > $then\ T(n) = \Theta(2^n)\quad for\ any\ \epsilon > 0$ matched case 3.
>
> for question **(e)**:
>
> > Assume $n = 2^m, then\ T(2^m) = 2 \cdot T(2^{m/2}) + m.$ And change the form again.

> > $T(2^m) = F(m)$, so we have $F(m) = 2 \cdot F(m/2) + m$
> > In this case, $a = 2, b = 2, c = log_b a = 1, f(m) = m$
> > $then\ F(m) = \Theta(m \cdot log(m))$

- (1) $T(n) = T(n/2) - n + 10$
- (2) $T(n) = 2^n \cdot T(n/2) + n$
- (3) $T(n) = 2 \cdot T(n/4) + n^{0.51}$
- (4) $T(n) = 0.5 \cdot T(n/2) + 1/n$
- (5) $T(n) = 16 \cdot T(n/4) + n!$

**Solution:**

> for question **(1)**:
>
> > $f(n) = 10 - n$, which is not a postive funcion.
> > So cannot apply Master Theorem.
>
> for question **(2)**:
>
> > $a = 2^n$, which is not a constant.
> > So cannot apply Master Theorem.
>
> for question **(3)**:
>
> > $a = 2, b = 4, c = log_b a = 0.5, f(n) = n^{0.51}, f(n) = \Omega(n^{0.5+\epsilon})$ for some $\epsilon > 0$
> > follow the case 3, then $T(n) = \Theta(n^{0.51})$
>
> for question **(4)**:
>
> > $a = 0.5$, which is less than 1.
> > So cannot apply Master Theorem.
>
> for question **(5)**:
>
> > $a = 16, b = 4, c = log_b a = 2, f(n) = n!, f(n) = \Omega(n!)$ for some $\epsilon > 0$
> > follow the case 3, then $T(n) = \Theta(n!)$

## 2. Consider an array $A$ of $n$ numbers with the assurance that $n > 2$, $A_1 \geq A_2$ and $A_n \geq A_{n-1}$. An index $i$ is said to be a local minimum of the array $A$ if it satisfies $1 < i < n$, $A_{i-1} \geq A_i$ and $A_{i+1} \geq A_i$.

```
(a) Prove that there always exists a local minimum for A.
(b) Design an algorithm to compute a local minimum of A.
```

## Your algorithm is allowed to make at most $O(log(n))$ pairwise comparisons between elements of $A$.

**Solution:**

>

> > **(a)Prrof**:
> > I plan to use induction method to prove its existance.
> >
> > *Base case:* when $n = 3$. There are three elements in array $A$. And $A_1 > A_2 < A_3$. So the $A_2$

could be called a local minimum

*Inductive hypothesis:* Assume when $n = k$, there exists a local minimum in array $A$.

*Inductive step:* Prove when $n = k + 1$, array $A$ contains local minimum as well.

Based on the decription above, when $n = k$, we can make sure that $A_k \geq A_{k-1}$, which means the value of element is monotonically increasing. So when we add $A_{k+1}$ into array, we just add one more bigger num, and this does not affect the fact that the existance of local minimun when $n = k$ (Inductive hypothesis). That is also a local minimum for $A[1 : k + 1]$.

**(b)My algorithm**:

1. if $n = 3$, we can directly return $A_2$.
2. if $n > 3$, we need to do futher check.
   2.1 let $k = length/2$, and check $A_k\ and\ A_{k+1}$
   2.2 if $A_k < A_{k+1}$ then call the algorithm recursively on $A[1 : k]$. Since the local minimum won't exist on the right part.
   2.3 if $A_k \geq A_{k+1}$ then then call the algorithm recursively on $A[k : length]$. 2.4 until $A_k$ and $A_{k+1}$ point to same memory. and return $A_k$.

**(b)Recurrence Equation**: $T(n) = T(n/2) + O(1)$
and after invoking *Masters Theorem*, we can get $T(n) = O(log(n))$.

## 3. There are $n$ cities where for each $i < j$, there is a road from city $i$ to city $j$ which takes $T_{i,j}$ time to travel. Two travelers Marco and Polo want to pass through all the cities, in the shortest time possible. In the other words, we want to divide the cities into two sets and assign one set to Marco and assign the other one to Polo such that the sum of the travel time by them is minimized. You can assume they start their travel from the city with smallest index from their set, and they travel cities in the ascending order of index (from smallest index to largest). The time complexity of your algorithm should be $O(n^2)$. Prove your algorithm finds the best answer.

**Solution:**

Pesudo-Code is:

```
def two_travelers(T: list):

    for i in range(n):
        for j in range(i + 1):
            cost[i] += T[j - 1][j]

    for i in range(n):
        for j in range(i + 1, n):
            dp[0][i][j] = min(dp[0][i][j - 1] + cost[j] - cost[i],
                              dp[1][i][j - 1] + cost[j] - cost[i])

            dp[1][i][j] = min(dp[1][i][j - 1] + cost[j] - cost[i],
                               dp[0][i][j - 1] + cost[j]- cost[i])
```

**(a)Explanation**:
Lets denote $dp[0][i][j]$ is the optimal solution which traversal from $i, i + 1, ... j$, also the $j$ city was visited by **Macro**. And $dp[1][i][j]$ is the **Polo's** optimal solution.
Also, need to mention that $cost[i]$ is calculated by $T_{0,1} + T_{1,2} + ... + T_{i-1,i}$
The relationship between two subproblem is to decided whether **Macro** or **Polo** should go to the next city $j$, based on the cost
$min(dp[0][i][j - 1] + cost[j] - cost[i], dp[1][i][j - 1] + cost[j] - cost[i])$

**4. Erica is an undergraduate student at USC, and she is preparing for a very important exam. There are $n$ days left for her to review all the lectures. To make sure she can finish all the materials, in every two consecutive days she must go through at least $k$ lectures. For example, if $k = 5$ and she learned 2 lectures yesterday, then she must learn at least 3 today. Also, Erica's attention and stamina for each day is limited, so for $i_{th}$ day if Erica learns more than $a_i$ lectures, she will be exhausted that day.**

**You are asked to help her to make a plan. Design an Dynamic Programming algorithm that output the lectures she should learn each day (lets say $b_i$), so that she can finish the lectures and being as less exhausted as possible (Specifically, minimize the sum of $max(0, b_i - a_i)$). Explain your algorithm and analyze it's complexity.**

**hint:** $k$ **is** $O(n)$.

**Solution:**

Pesudo-Code is:

```
def gen_study_plan(A):

    B = [i for i in A]
    # initialization

    for i in range(1, len(A) - 1):
    # start from the second day

        if sum(B[i - 1:i + 1]) <= k:
        # sum(B[i - 1:i + 1]) means the sum of today's and last day's study ho
urs
            B[i] = max(A[i], k - B[i - 1])

    return B
```

**(a)Explanation**:
we wanna minimize the sum of $max(0, b_i - a_i))$ in the problem. In other word, we need to reduce the difference between array $A$ and array $B$. But what is interesting here is that you won't get any reward or penalty if your study hour less or equal than $a_i$ on $i_{th}$ day. because the problem said $max(0, b_i - a_i))$.
So I initialize the array $B$ based on the array $A$'s value. In this way, we can fully take advantage of Erica's energy and reduce the difference.

So let me restate that $A[i]$ is the energy array that student left at $i_{th}$ day, and $B[i]$ is the minimum hour the student need to finish reviewing work at each day.
The $k$ just the amount of hour needed in every two consecutive days.

To get the minimum value of $max(0, b_i - a_i))$, we can iterate the array $A$ from index $2$.

- If the study hour of today plus the last days, $sum(B[i - 1 : i + 1])$, exceed the number $k$, than we do nothing, since the penalty is zero.
- If the $sum(B[i - 1 : i + 1])$ less than $k$, then we have to add $k - B[i - 1]$ hours to the $B[i]$.

So put it simple, the subproblem is to find the maximum hours that Erica could learn at day $i$, but with least penalty.
The relationship is to define how to minimize the next day's penalty based on the previous day.

**(c)Runtime Complexity**:
The runtime complexity is $O(n)$. there is only iterate array $A$ once.

**5. Due to the pandemic, You decide to stay at home and play a new board game alone. The game consists an array $a$ of $n$ positive integers and a chessman. To begin with, you should put your**

**character in an arbitrary position. In each steps, you gain $a_i$ points, then move your chessman at least $a_i$ positions to the right (that is, $i' >= i + a_i$). The game ends when your chessman moves out of the array.**

**Design an algorithm that cost at most $O(n)$ time to find the maximum points you can get. Explain your algorithm and analyze its complexity.**

**Solution:**

Pesudo-Code is:

```
arr = [1, 3, 2, 4, 6, 5]
  # arr is array representing points

  dp = [0] * len(arr)
  # dp is array which store the maximum point you can get

  dp[-1] = arr[-1]
  # initialization

  for i in range(len(arr) - 2, -1, -1):
      # reverse iteration
      # i is index where you are at

      if i + arr[i] < len(arr):
          dp[i] = max(arr[i] + dp[i + arr[i]], dp[i + 1])
      else:
          dp[i] = max(arr[i], dp[i + 1])

  print(max(dp))
```

**(a)Explanation**:
In order to maximizing points when we are playing the chess in this problem, we need to select some position or index to stop and claim its reward. But it is pretty hard to define the subproblem, since there are many choice or position you can stop. So lets solve problem in a reversed direction.

Let denote $dp[i]$ is the maximum points gained when you at index $i$ position.
I use array $[1, 3, 2, 4, 6, 5]$ to support my explanation.
At the very end of array $a$, if there is a solution reach the last index, their score need to add $a_i$ that is $(5, i = 5)$, since it is the biggest in $dp[5] = 5, (i = 5)$, if there has a soultion stop at the second last index, the largest point becomes $dp[4] = 6, (i = 4)$. since when it can reach the last second position, it can reach the last position as well. And the reward in $a_4$ is bigger than $dp[5]$.
$arr: \quad [1, 3, 2, 4, 6, 5]$
$i: \quad [0, 1, 2, 3, 4, 5]$
$dp: \quad [0, 0, 0, 0, 6, 5]$

So anyone solution can reach the last two index, they don't need to consider to stop at the last index.
Like when $i = 2$, the maximum point it can get is $(arr[i] + dp[i + arr[i]] = 2 + 6 = 8, i = 2)$
Of course it can reach both last two positions, but if it moves to the last position directly, the reward is only $dp[5] = 5$, which is less than $dp[4] = 6$.
$arr: \quad [1, 3, 2, 4, 6, 5]$
$i: \quad [0, 1, 2, 3, 4, 5]$
$dp: \quad [0, 0, 8, 6, 6, 5]$

and keep moving forward, we can know the max reward we can get in every position. And the result will be $dp[0]$.

So put it simple, **the subproblem** $dp[i]$ is local maximum reward you find from back to fornt at position $i$.
The **relationship** is to define how to compute the front maximum based on the back parts.
The basic rule is if the upper part index can reach the lower part and doesn't out of index, we can assign the bigger number to the front index $max(arr[i] + dp[i + arr[i]], dp[i + 1])$
If it out of index, it means it will be the last stroke of chess game, so we just compare $arr[i]$ and $dp[i + 1]$.

**(b)Compute Process**:

$$\begin{array}{ll} arr: & [1,3,2,4,6,5] \\ i: & [0,1,2,3,4,5] \\ dp: & [10,9,8,6,6,5] \end{array}$$

**(c)Runtime Complexity**:
Obviously, the runtime complexity is $O(n)$. there is only one loop.

## 6. Joseph recently received some strings as his birthday gift from Chris. He is interested in the similarity between those strings. He thinks that two strings $a$ and $b$ are considered J-similar to each other in one of these two cases:

1. $a$ is equal to $b$.
2. he can cut $a$ into two substrings $a_1, a_2$ of the same length, and cut $b$ in the same way, then one of following is correct:

> (a) $a_1$ is J-similar to $b_1$, and $a_2$ is J-similar to $b_2$.
> (b) $a_2$ is J-similar to $b_1$, and $a_1$ is J-similar to $b_2$.

## Caution: the second case is not applied to strings of odd length.

## He ask you to help him sort this out. Please prove that only strings having the same length can be J-similar to each other, then design an algorithm to determine if two strings are J-similar within $O(n \cdot log(n))$ time (where $n$ is the length of strings).

**Solution:**

Basically, the question is defined such a function *j_sim*:

```
def j_sim(a, b):
    if a == b:
        return true
    elif len(a) % 2 == 0 and len(b) % 2== 0:
        a1, a2 = split(a)
        b1, b2 = split(b)
        if (j_sim(a1, b1) and j_sim(a2, b2)) or (j_sim(a1, b2) and j_sim(a2, b1
)):
            return true

    return false
```

**(a)Prrof**:
if any string pair, $a, b$ could be called **J-similar**, they need to meet at least thses requirements:

1. if their length are odd, they can follow the *case 1* -> $a = b$, only in this way, they are **J-similar**. Since the *case 2* is not applied to the odd length string. When $a = b$, they have same length for sure.
2. when these have even length and the length are same, the alogirthm can split recurively and check if they are **J-similar**. And if they have even length but differnt length, like one has 6 characters, and another has 8 characters. they cannot follow the *case 1* firstly. And also when these two string are splited, they cannot equal to each other, since their substring length is 3 and 4.

**(b)My algorithm**:

1. if string $a$ equals to string $b$, we can directly return true.
2. if they are not equal but with same even length,
   2.1 algorithm split each string fifty-fifty => $a_1, a_2$ and $b_1, b_2$, which take $O(n)$ time.
   2.2 and then call the algorithm *J_sim* recursively.
   2.3 to check $a_1, b_1$ and $a_2, b_2$
   2.4 or to check $a_1, b_2$ and $a_2, b_1$
   2.5 either one is true, we can return true.

**(c)Recurrence Equation**: $T(n) = 2 \cdot T(n/2) + n$
and after invoking *Masters Theorem*, we can get $T(n) = O(n \cdot log(n))$.

**7. Chris recently received an array $p$ as his birthday gift from Joseph, whose elements are either 0 or 1. He wants to use it to generate an infinite long superarray. Here is his strategy: each time, he inverts his array by bits, changing all 0 to 1 and all 1 to 0 to get another array, then concatenate the original array and the inverted array together. For example, if the original array is $[0, 1, 1, 0]$, then the inverted array will be $[1, 0, 0, 1]$ and the new array will be $[0, 1, 1, 0, 1, 0, 0, 1]$. He wonders what the array will look like after he repeat this many many times.**

**He ask you to help him sort this out. Given the original array $p$ of length $n$ and two indices $a, b$ $(n \ll a \ll b, \quad \ll$ means much less than) Design an algorithm to calculate the sum of elements between $a$ and $b$ of the generated infinite array $\hat{p}$, specifically, $\sum_{a \leq i \leq b} \hat{p}_i$. He also wants you to do it real fast, so make sure your algorithm runs less than $O(b)$ time. Explain your algorithm and analyze its complexity.**

**Solution:**

Basically, the question can be solved by simply find four useful indexes.
As it described in the question, there is an array $p$ whose length is $n$.
let's denote the $q$ is the corresponding array that has the same length and its elements are inverted by bits, compared with the array $p$.

So when the strategy executed once, we got a new array $pq$.

the second time, we got $pqqp$.

the third time, we got $pqqp\ \ qppq$

after the fourth time, the array will be $pqqp\ \ qppq\ \ qppq\ \ pqqp$

... we found that every 4 block of ($p$ or $q$) has 2 $\cdot p$ and 2$\cdot q$, which means the num of $1$ is fixed.

After the strategy executed many many times, the array might look like this:

$$\begin{array}{l} \quad\quad a \quad\quad\quad\quad\quad\quad\quad\quad\quad\quad b \\ \quad\quad | \quad\quad\quad\quad\quad\quad\quad\quad\quad\quad | \\ pqqp\ \ qppq\ \ qppq\ \ pqqp\ \ldots\ pqqp\ \ qppq\ \ qppq\ \ pqqp \end{array}$$

we can use algorithm to find $i$ and $j$, and split the whole problem into three pieces.

$$\begin{array}{l} \quad\quad a \quad j \quad\quad\quad\quad\quad\quad\quad\quad k \quad b \\ \quad\quad | \quad | \quad\quad\quad\quad\quad\quad\quad\quad | \quad | \\ pqqp\ \ qppq\ \ qppq\ \ pqqp\ \ldots\ pqqp\ \ qppq\ \ qppq\ \ pqqp \end{array}$$

**My algorithm**:

1. divide $\sum_{a \leq i \leq b} \hat{p_i}$ into three parts. $\sum_{a \leq i \leq j} \hat{p_i}$, $\sum_{j \leq i \leq k} \hat{p_i}$ and $\sum_{k \leq i \leq b} \hat{p_i}$
2. based on value of index $a$ and $b$, compute the index $j$ and $k$.
   2.1 similar to the linear probing technique, the first index meets $(index\%(4 \cdot n) = 0)$ is what we are looking for.
   2.2 about $j$, we increase value of index $a$ one by one, to test that formula.
   2.3 about $k$, we decrease value of index $b$ one by one, to test that formula.
   2.4 both step 2.2 and 2.3 will spend $(4 \cdot n)$ times at most.
3. after we have four indexes, we can iterate all elements in the first part and third part, and count how many $1$ in there. About the second part, we can simply compute $((k - j)/4/n) \cdot ($ the num of $1$s in $pqqp$ or $qppq$)). This takes $O(1)$ time.

**Runtime Complexity**: So, overall, the alogrithm only take $O(n)$ time, since $(n \ll a \ll b)$.
So the algorithm runs less than $O(b)$ time for sure.