CSCI 570 - Spring 2021 - HW 3 Rubric

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) $T(n) = 4T(n/2) + n^2 \log n$.
- (b) $T(n) = 8T(n/6) + n \log n$.
- (c) $T(n) = \sqrt{6006}T(n/2) + n^{\sqrt{6006}}$.
- (d) $T(n) = 10T(n/2) + 2^n$.
- (e) $T(n) = 2T(\sqrt{n}) + log_2 n$.
- (f) $T(n) = T(n/2) - n + 10$
- (g) $T(n) = 2^n T(n/2) + n$
- (h) $T(n) = 2T(n/4) + n^{0.51}$
- (i) $T(n) = 0.5T(n/2) + 1/n$
- (j) $T(n) = 16T(n/4) + n!$

Solution: In some cases, we shall need to invoke the Master Theorem with one generalization as described next. If the recurrence $T(n) = aT(n/b) + f(n)$ is satisfied with $f(n) = \Theta(n^{\log_b a} \log^k n)$ for some $k \geq 0$, then $T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$.

[Rubic (40 points, 4 points for each)
– 2 points for correctly explaining how to apply the Master theorem in this question
– 2 point for the correct answer

- (a) Observe that $f(n) = n^2 \log n$ and $n^{\log_b a} = n^2$, so applying the generalized Master theorem, $T(n) = \Theta(n^2 \log^2 n)$
- (b) Observe that $n^{\log_b a} = n^{log_6 8}$ and $f(n) = n \log(n) = O(n^{log_6 8 - \epsilon})$ for any $o < \epsilon < \log_6^8 - 1$. Thus, invoking Master theorem gives $T(n) = \Theta(n^{\log_b a}) = \Theta(n^{\log_6 8})$.
- (c) We have $n^{log_b a} = n^{\log_2 \sqrt{6006}} = n^{0.5 \log_2 6006} = O(n^{0.5 \log_2 8192}) = O(n^{13/2})$ and $f(n) = n^{\sqrt{6006}} = \Omega(n^{\sqrt{4900}}) = \Omega(n^{70}) = \Omega(n^{13/2+\epsilon})$ for any $0 < \epsilon < 63.5$. Thus from Master's Theorem $T(n) = \Theta(f(n)) = \Theta(n^{\sqrt{6006}})$.
- (d) We have $n^{\log_b a} = n^{\log_2 10}$ and $f(n) = 2^n = \Omega(n^{\log_2 10 + \epsilon})$ for any $\epsilon > 0$. Therefore Master theorem implies $T(n) = \Theta(f(n)) = \Theta(2^n)$
- (e) Use the change of variables $n = 2^m$ to get $T(2^m) = 2T(2^{m/2}) + m$. Next, denoting $S(m) = T(2^m)$ implies that we have the recurrence $S(m) = 2S(m/2) + m$. Note that $S(\cdot)$ is a positive function due to the monotonicity of the increasing map $x \to 2^x$ and the positivity of $T(\cdot)$. All conditions for applicability of Master Theorem are satisfied and using the generalized version gives $S(m) = \Theta(m \log m)$ on observing that $f(m) = m$ and $mlog_b a = m$. We express the solution in terms of $T(n)$ by

  $$T(n) = T(2^m) = S(m) = m \log(m) = log_2 n \log \log_2 n$$

  , for large enough n so that the growth expression above is positive.
- (f) We cannot apply master Theorem on this, because the $f(n)$ could not be negative (we cannot perform negative number of operations in the recursion).
- (g) We cannot apply master Theorem on this, since a is not a constant. The number of sub-problems at each recursion step should be fixed.
- (h) We have $n^{\log_b^a} = n^{\log_4^2} = n^{0.5}$ and $f(n) = n^{0.51} = \Omega(n^{0.5+\epsilon})$ for any $0 < \epsilon < 0.01$. Based on case 3 of master theorem we have $T(n) = \Theta(f(n)) = \Theta(n^{0.51})$
- (i) We cannot apply master Theorem on this, since $a < 1$. We cannot have less than one sub problem
- (j) We have $n^{\log_b^a} = n^{\log_4^{16}} = n^2$, and $f(n) = n! = \Omega(n^{2+\epsilon})$ for any $\epsilon > 0$. Based on case 3 of master theorem we have $T(n) = \Theta(f(n)) = \Theta(n!)$

]

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 $O(logn)$ pairwise comparisons between elements of A.

Solution:

(a) We use induction. For $n = 3$, since we know $A_1 \geq A_2$ and $A_3 \geq A_2$, so A_2 is a local minimum. The induction hypothesis is for $k < n$: for array A with $A_1 \geq A_2$ and $A_k \geq A_{k-1}$ there exists local minimum. We want to prove for n. There are two cases: 1) $A_{n-2} \geq A_{n-1}$ or 2) $A_{n-1} > A_{n-2}$. Since we know $A_n \geq A_{n-1}$, in the first case, A_{n-1} is local minimum. In the second case, we know $A_1 \geq A_2$ and $A_{n-1} \geq A_{n-2}$. Based on induction hypothesis, there exists a local minimum in the array $[A_1, A_2, ..., A_{n-1}]$.

(b) Similar to binary search call function $recur(1, n)$ to find local minimum of the array $A_1, A_2, ..., A_n$. In $recur(s, e)$ function, define mid as $\lfloor \frac{s+e}{2} \rfloor$. Then:

1. if $e - s$ is 2: return $s + 1$ (base case)
2. if $A_{mid-1} \geq A_{mid}$ and $A_{mid+1} \geq A_{mid}$: return mid as local minimum
3. else if $A_{mid-1} < A_{mid}$: return $recur(s, mid)$
4. else if $A_{mid} > A_{mid+1}$: return $recur(mid, e)$

Complexity: If the running time of the algorithm on an input of size n is $T(n)$, then it involves a constant number of comparisons and assignments and a recursive function call on either $A_1, ..., A_{\lfloor \frac{n+1}{2} \rfloor}$ or $A_{\lfloor \frac{n+1}{2} \rfloor}, ..., A_n$. Therefore, $T(n) \leq T(\lfloor \frac{n+1}{2} \rfloor) + \theta(1)$. Assuming n to be a power of 2 (minus 1), this recurrence simplifies to $T(n) \leq T(\frac{n}{2}) + \theta(1)$ and invoking Master's Theorem gives $T(n) = O(logn)$.

Correctness: Similar to part (a), we can prove the correctness of the algorithm by induction. The base case (1) is clear. The induction hypothesis is that the algorithm return the correct local minimum for any array of size less than n. In case of event 2, we know $A_{mid-1} \geq A_{mid}$ and $A_{mid+1} \geq A_{mid}$, so the mid is local minimum. Otherwise, by moving to the appropriate sub-problem in cases 3 and 4 and based on induction hypothesis, the algorithm returns the correct answer.

[Rubic (15 points):

- [5 points]: part a (1 for base case, 4 for induction step: 2 point for each case)
- [4 points]: algorithm of part b (1 for each case).
- [1 points]: Proof of correctness.
- [5 points]: Computing time complexity.

]

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 $O(n^2)$.

Solution: We want to use dynamic programming. First, we are going to use array s (as a pre-processing step) to speedup our computations. Define $s_i = \Sigma_{j=i}^{n-1} T_{j,j+1}$. Array s could be easily computed in $O(n)$ time.

1. Defining sub-problem: Let's define $c_{i,j}$ for each $i < j$ as the minimum possible time to travel through all the cities from i to n, while Marco starts from city i and Polo starts from city j.

2. Recurrence Relation: The base case of the dynamic Programming is for $c_{n-1,n} = 0$. Now, we want to find the best answer for $c_{i,j}$. First we know if $i \neq j-1$ Marco should travel cities i to $j-1$ (because Polo cannot travel backwards). So, in that case $c_{i,j} = s_i - s_j + c_{j-1,j}$. So now we should compute $c_{j-1,j}$. There are 4 different possibilities to assign cities $j-1$ and j to Marco and Polo. The value of $c_{j-1,j}$ is minimum of these 4 cases. Here k is all the possible cities after city $j+1$:

1. Travel city $j-1$ with Marco and give the rest to Polo: s_j
2. Travel city j with Polo and give the rest to Marco: $T_{j-1,j+1} + s_{j+1}$
3. Add city $j+1$ to Marco: $\min\{T_{j-1,j+1} + T_{j,k} + c_{j+1,k} | \forall k > j+1\}$
4. Add city $j+1$ to Polo: $\min\{T_{j-1,k} + T_{j,j+1} + c_{j+1,k} | \forall k > j+1\}$

The final answer to the problem is $\min\{c_{1,j} | 2 \leq j \leq n\}$.

3. Pseudo Code:

```
s[n] = 0
for i=n-1 to 1:
  s[i] = T[i][i + 1] + s[i + 1]
c[n-1][n] = 0 # Base case
for i=1 to n-2:
    c[i][n] = s[i] - s[n - 1]
for j=n-1 to 2:
  minval = min(s[j], T[j - 1, j + 1] + s[j + 1]) #Case 1 & 2
  for k=j+2 to n-1:
    minval = min(minval, T[j - 1, j + 1] + T[j, k] + c[j + 1][k]) #Case 3
    minval = min(minval, T[j - 1, k] + T[j, j + 1] + c[j + 1][k]) #Case 4

  c[j - 1][j] = minval
  for i=1 to j-2:
```

```
    c[i][j] = c[j - 1][j] + s[i] - s[j]

# Marco starts from 1, and Polo starts from j
final_answer = c[1][2]
for j=3 to n:
    final_answer = min(final_answer, c[1][j])
return final_answer
```

---

4. Time Complexity: Start from $i = n - 2 \to 1$ and loop through $j = n \to i + 1$. For $i < j - 1$ ($O(n^2)$ such pairs) each update takes $O(1)$ and for $i = j - 1$ ($O(n)$ such pairs) each update takes $O(n)$. The pre-processing of array s takes $O(n)$ time. So, in total time complexity of the algorithm is $O(n + n^2 \times 1 + n \times n) = O(n^2)$.

[Rubic (20 points):

- [3 points]: Correct base case, and pointing out marco should travel cities i to $j - 1$ if $i \neq j - 1$.
- [12 points]: for covering all 4 mentioned cases, 3 points each. 2 cases: single city to travel by each traveler. 2 cases: adding cities to an already existing set of cities.
- [5 points]: Computing time complexity correctly.
- No reduction of point if they consider ending cities instead of starting city. In that case, they should solve opt[i, j] using previous sub-problems not the next ones!
- If they did not use array s to speed up the optimization (and they loop over to compute the total sum), and they end-up having $O(n^3)$, and they compute the correct time complexity, then deduct -2 points.

]

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: First of all, we can use greedy to prove that Erica will learn at most k lectures everyday. This is because Erica is only required to learn at least k lectures every two days, so if she's already learned k lectures in some day, the requirements on yesterday-today and today-tomorrow will both be satisfied, and there's no need to study more. We use dynamic programming to solve this problem.

1. Defining sub-problem: let $opt_{i,j}$ be the minimum exhaustion Erica could get after she studies lectures at days $1 \dots i$ and learned exactly j lectures at day i. Apparently $i \leq n$ and $j \leq k$.

2. Recurrence Relation: For Erica to study j lectures at day i, she should study *at least* $k - j$ lectures at day $i - 1$. So, the recursive formula would be:

$$opt_{1,j} = \max(0, (j - a_1)), \forall 0 \leq j \leq k$$
$$opt_{i,j} = \min_{k-j \leq t \leq k} (opt_{i-1,t} + \max(0, (j - a_i))), \forall 2 \leq i \leq n, 0 \leq j \leq k$$

3. Pseudo Code

```
for j=0 to k:
  opt[1][j] = max(0, j - a[1])
for i=2 to n:
  for j=0 to k:
    opt[i][j] = infinity
    for t=k-j to k:
      opt[i][j] = min(opt[i][j], opt[i - 1][t] + max(0, j - a[i]))
final_answer = opt[n][0]
for j=1 to k:
    final_answer = min(final_answer, opt[n][j])
return final_answer
```

4. Time Complexity The complexity for this algorithm is $O(n^3)$, since the total number of states is $O(n^2)$ (we have n days and $k = O(n)$) and each recursive formula costs $O(n)$ time to compute.

Additionally, there's a faster solution which only cost $O(n)$ of time. It is not required for this answer, but for the sake of learning, we mention it here. The high-level idea is using the *lazy strategy*, which is, leave as much as possible lectures that will cause exhaustion to tomorrow. See if you can design this algorithm too.

[Rubic (15 points):

- [2 points]: Pointing out that Erica will not learn more than k lectures everyday.

- [4 points]: Designing the subproblems and recursive formula that can minimize the exhaustion.
- [7 points]: Taking care of the constraint of learning not less than k lectures.
- [2 points]: Computing time complexity correctly.
- Designing an non dynamic programming algorithm will get at most 14 points.
- It's okay to design a dp algorithm faster than the solution using 'lazy strategy'.

]

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:

1. Defining sub-problem: Let opt_i be maximum points you can get starting from position i. Define array S as $S_i = max\{opt_j | i \leq j\}$. We are going to use S as a helper for updating opt in the recurrence relation.

2. Recurrence Relation: Array S could capture the best move (maximum points) after being in cell i. As a base case, $opt_n = S_n = a_n$. The recursive formula can be:

$$opt_i = a_i + S_{i+a_i}$$
$$S_i = \max(opt_i, S_{i+1})$$

The final output is S_1.

3. Pseudo Code

```
opt[n] = a[n]
s[n] = opt[n]
for i=n-1 to 1:
    opt[i] = a[i]
    if i + a[i] <= n:
        opt[i] += s[i + a[i]]
    s[i] = max(opt[i], s[i + 1])
return s[1]
```

4. Time Complexity: The complexity of this algorithm is $O(n)$, since there are $O(n)$ subproblems and each recursive formula costs $O(1)$ time to compute.

[Rubic (10 points):

- [2 points]: correct subproblem.
- [3 points]: correct recursive formula that can maximize the prize.
- [3 points]: The algorithm only cost $O(n)$ time.
- [2 points]: Computing time complexity correctly.

]

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 $O(n^2)$ time (where n is the length of strings).

Solution:

To make our proof straightforward, we first change the statement to another equivalent: For every string a of length n and string b, b will be J-similar to a only if length of b is equal to n.

We use induction on n to prove the statement.

The base case ($n = 1$) is trivial.

Assume we now have proved the statement for all $n < k$. Now we focus on $n = k$. Regarding to the definition, there are two cases that b can be J-similar to a. Firstly, if a is equal to b, then apparently they have the same length. On the other hand, if the second case happens, by the induction either one of the following will be correct:

$$len(a_1) = len(b_1), len(a_2) = len(b_2)$$
$$or\ len(a_1) = len(b_2), len(a_2) = len(b_1)$$

In both case we have $len(a_1)+len(a_2) = len(b_1)+len(b_2)$, which means the length of b is also n. □

The $O(n^2)$ algorithm is very easy to design: we just need to follow the two cases. Specifically, we design a function J_similar(a) as follows:

```
def J_similar(a, b)
{
    if a == b:
        return True # they are identical
    if len(a) % 2 == 1:
        return False # Unable to cut strings of odd length

    a1, a2 = a[:len(a)/2], a[len(a)/2:] # Cut string to half
    b1, b2 = b[:len(b)/2], b[len(b)/2:] # Cut string to half

    if J_similar(a1, b1) and J_similar(a2, b2):
        return True
    elif J_similar(a1, b2) and J_similar(a2, b1):
        return True
    else:
        return False
}
```

this algorithm will results in $T(n) = 4 \cdot T(n/2) + O(n)$, which gives us $T(n) = O(n^2)$. Luckily, this is already enough to get full credit. ;-)

However, we can go further trying to accelerate it. First of all, we can easily prove that if $J(a,b), J(b,c)$, then $J(a,c)$. This is called the Transitive Property. Using this property we can reduce one recursive call in every call of J_similar:

```
def J_similar(a, b)
{
    if a == b:
        return True # they are identical
    if len(a) % 2 == 1:
        return False # Unable to cut strings of odd length

    a1, a2 = a[:len(a)/2], a[len(a)/2:] # Cut string to half
    b1, b2 = b[:len(b)/2], b[len(b)/2:] # Cut string to half

    a1_b1 = J_similar(a1, b1)
    a1_b2 = J_similar(a1, b2)
    if a1_b1==False and not a1_b2==False:
        return False
    elif a1_b1==True and a1_b2==False:
        return J_similar(a2,b2)
    elif a1_b1==False and a1_b2==True:
        return J_similar(a2,b1)
    else:
        return J_similar(a2,b1) # if J_similar(a2,b1)=False, then
```

```
            J_similar(a2,b2)=False too.
}
```

The key point here is that if $J\_similar(a1, b1) = J\_similar(a1, b2) = True$, then $J\_similar(b1, b2) = True$. So, we will have $J\_similar(a2, b1) = J\_similar(a2, b2)$, which can be checked in only one time of recursion.

In this version, we have $T(n) = 3 \cdot T(n/2) + O(n)$, which gives us $T(n) = O(n^{\log_2 3}) \approx O(n^{1.58})$. It's still not $O(n \log n)$!(which is required in the original version)

Further more, let's explain how to reach $O(n \log n)$:

What we do here is: we rearrange both of the two strings by some certain movements, then we prove they will project to the same string if and only if they are J-similar to each other.

How to design this movement? To begin with, it's obvious that for any string a of even length, once we split it into two halves (a_1, a_2), $a' = a_2 a_1$ should be also J-similar to a. Based on this, we can design a divide and conquer approach to rearrange a, b into their lexicographically minimum equivalents. Specifically, we design a function J-sort(a) as follows:

```
def J-sort(a)
{
    if len(a) % 2 == 1:
        return a # Unable to cut strings of odd length
    a1, a2 = a[:len(a)/2], a[len(a)/2:] # Cut string to half
    a1_m = J-sort(a1)
    a2_m = J-sort(a2)
    if a1_m < a2_m: # lexicographical order
        return a1_m + a2_m # concatenate
    else:
        return a2_m + a1_m
}
```

It's not hard to conclude that J-sort(a) has the lowest lexicographical order among all the strings that is J-similar to a (Just use induction like above). Because of this (and the same thing to b), we can further prove that a is J-similar to b if and only if J-sort(a) is **equal** to J-sort(b).

Let $T(n)$ be the complexity of calculating J-sort(a) with respect to length n. Based on the previous analysis, we have $T(n) = 2 \cdot T(n/2) + O(n)$. The Master Theorem tells us $T(n) = O(nlogn)$. Aside from that, checking the equivalence between two strings costs only $O(n)$ time, which is negligible.

[Rubic (15 points):

- [5 points]: Proving the statement correctly
- [7 points]: Designing the divide-and-conquer algorithm.
- [3 points]: Computing time complexity correctly.

]

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 super-array. 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 ($n \ll a \ll b$, $\ll$ 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:

Rather than directly calculate the sum between element a and element b, it's easier for us to calculate the prefix sum for a and b separately, then subtract them to get the answer.

Let S_i denotes the prefix sum of $\hat{p}$ from index 1 to i, which means $S_i = \sum_{1 \leq j \leq i} \hat{p}_j$. Then:

$$\sum_{i=a}^{b} \hat{p}_i = \sum_{i=1}^{b} \hat{p}_i - \sum_{i=1}^{a-1} \hat{p}_i = S_b - S_{a-1}$$

Calculating S_b for arbitrary b can be finished in two steps: First, if $b = 2^k \cdot n$ where $k > 1$, then $\hat{p}_{1 \ldots b}$ will be one of the arrays Chris gets during his movements. Thus, we can directly calculate S_b from $S_{b/2}$:

$$S_b = S_{b/2} + (b/2 - S_{b/2}) = b/2$$

Second, for arbitrary b, let c be the biggest number among set $\{2^k \cdot n | 1 \leq k\}$ that satisfies $c \leq b$. We can divide the array $\hat{p}_{1 \ldots b}$ into two parts: $\hat{p}_{1 \ldots c}$ and $\hat{p}_{c+1 \ldots b}$. For $\hat{p}_{1 \ldots c}$, we can use the first step to calculate S_c. The tricky thing is the second part. Since $\hat{p}_{c+1 \ldots 2c}$ is the bit-inverted version of $\hat{p}_{1 \ldots c}$, we have $\hat{p}_i = 1 - \hat{p}_{i-c}, \forall c < i \leq b$. Thus:

$$\sum_{i=c+1}^{b} \hat{p}_i = (b - c) - S_{b-c}$$

Combining the two steps we can get:

$$S_b = \begin{cases} b/2, & \text{if } b = 2^k \cdot n \\ (c/2) + (b - c) - S_{b-c}, & \text{otherwise} \end{cases}$$

Let $T(b)$ be the time complexity to calculate S_b. For every possible b, calculate c will cost $O(\log b)$ time, and since $b - c \leq b/2$ (because $b \leq 2c$), we have $T(b) = T(b-c) + O(\log b) \leq T(b/2) + O(\log b)$. By the Master Theorem we can get $T(b) = O(\log^2 b) = o(b)$.

[Rubic (20 points):

- [4 points]: Solving the problem for base case where $b = n \times 2^k$
- [5 points]: Designing the divide-and-conquer recursion that calculate the right answer.
- [6 points]: The algorithm runs quicker than O(b).
- [5 points]: Computing time complexity correctly.

]