EECS 336

Litong Wang

Homework 2

Due on April 18, 2016

1. Suppose that you are choosing between 3 algorithms A, B, and C for your algorithmic problem:

   (a) Algorithm A solves the problem with an input of size n by dividing the problem into 5 sub problems of half the original input size in linear time, recursively solving the subproblems, and then combining the solutions in linear time.

   (b) Algorithm B solves the problem with an input of size n by creating two subproblems of input size n 1 in constant time, recursively solving the two subproblems, and then combining the solutions in constant time.

   (c) Algorithm C solves the problem with an input of size n by dividing the problem in to 9 subproblems of input size n/3 in cubic time, recursively solving the subproblems, and then combining the solutions in quadratic time.

   What are the time complexities of these algorithms, and which one would you prefer? Prove the time complexities.

   the Master Theorem:

$$
\begin{aligned}
&Case1 : x < \alpha \; T(n) = \Theta(n^{\alpha}(\log n)^{\beta}) \\
&Case2 : x = \alpha \; T(n) = \Theta(n^{\alpha}(\log n)^{\beta+1}) \\
&Case3 : x > \alpha \; T(n) = \Theta(n^{x})
\end{aligned}
$$

   Analysis of algorithm A:

$$
\begin{aligned}
&T(n) = 5T(\frac{n}{2}) + 2n \\
&a = 5,\ b = \frac{1}{2},\ \alpha = 1,\ \beta = 0 \\
&5 \times (\frac{1}{2})^x = 1 \\
&x = \lg 5 > \alpha \\
&\therefore Case3 \\
&\therefore T(n) = \Theta(n^{\lg 5})
\end{aligned}
$$

Analysis of algorithm B:

$$
\begin{aligned}
&T(n) = 2T(n-1) + c \\
&T(n) = 2(2T(n-1) + c) + c \\
&T(n) = 2(2(2T(n-1) + c) + c) + c \\
&T(n) = 2^k T(n-k) + c2^{k-1} + c2^{k-2} + \cdots + c \\
&T(n) = 2^k T(n-k) + c(2^k - 1) \\
&set\ k = n - 1 \\
&T(n) = 2^{n-1} T(1) + c(2^{n-1} - 1) \\
&\therefore T(n) = \Theta(2^n)
\end{aligned}
$$

Analysis of algorithm C:

$$
\begin{aligned}
&T(n) = 9T(\frac{n}{3}) + n^3 + n^2 \\
&T(n) = 9T(\frac{n}{3}) + \Theta(n^3) \\
&a = 9,\ b = \frac{1}{3},\ \alpha = 3,\ \beta = 0 \\
&9 \times (\frac{1}{3})^x = 1 \\
&x = 2 < \alpha \\
&\therefore Case1 \\
&\therefore T(n) = \Theta(n^3)
\end{aligned}
$$

Becase $2^n = \Omega(n^3)$ $n^3 = \Omega(n^{\lg 5})$, we should choose algorithm A.

2. Exercise 4.3 - 9 in the textbook. Change the equation to the following. Prove your solution.

   Solve the recurrence $T(n) = 3 \cdot T(n^{\frac{1}{3}}) + 24 \cdot T(n^{\frac{1}{6}}) + (\log n)^2 \cdot (\log \log n)^{1.5} + 101,000,078$ by making a change of variables. Your solution should be asymptotically tight. Do not worry about whether values are integral.

   $m = \log n$
   $T(2^m) = 3 \cdot T(2^{\frac{m}{3}}) + 24 \cdot T(2^{\frac{m}{6}}) + m^2 \cdot (\log m)^{1.5}$
   $S(m) = T(2^m)$
   $S(m) = 3 \cdot T(\frac{m}{3}) + 24 \cdot T(\frac{m}{6}) + m^2 \cdot (\log m)^{1.5}$
   $a_1 = 3, a_2 = 24, b_1 = \frac{1}{3}, b_2 = \frac{1}{6}, \alpha = 2, \beta = 1.5$
   $a_1(b_1)^x + a_2(b_2)^x = 1$
   $x = 2 = \alpha, Case\ 2:$
   $S(m) = \Theta(m^2 \cdot (\log m)^{1.5+1})$
   $T(n) = T(2^m) = S(m) = \Theta(m^2 \cdot (\log m)^{2.5}) = \Theta(\log^2 n \cdot (\log \log n)^{2.5})$

3. Solve the following recurrences. Your solutions should be asymptotically tight. Do not worry about whether values are integral. Prove your solutions.

   In class, we have discussed one version of the Master Theorem in detail and mentioned the Akra-Bazzi Master Theorem. The textbook gives another version of the Master Theorem. Apply all three theorems to each of the following recurrence equations.

   (a) $T(n) = 5 \cdot T(\frac{n}{2}) + 176 \cdot T(\frac{n}{4}) + n^4 (\log n)^{-3}$

   Method 1: in-class
   $a_1 = 5, a_2 = 176, b_1 = \frac{1}{2}, b_2 = \frac{1}{4}, \alpha = 4, \beta = -3$

In this case, we cannot apply the method discussed in the class. Because that method requires $\beta > 0$.

Method 2: the Akra-Bazzi Master Theorem

$$
\begin{aligned}
&P = 4 \\
&T(n) = \Theta(n^4(1 + \int_2^n \frac{u^4(\log u)^{-3}}{u^5} du)) \\
&T(n) = \Theta(n^4(1 + \int_2^n \frac{(\log u)^{-3}}{u} du)) \\
&T(n) = \Theta(n^4(1 + \int_2^n (\log u)^{-3} d\log u)) \\
&T(n) = \Theta(n^4(1 + \int_{\log 2}^{\log n} v^{-3} dv)) \\
&T(n) = \Theta(n^4(1 - \frac{1}{2}\log^{-2} n + \frac{1}{2} \times \frac{1}{\log 2^2})) \\
&T(n) = \Theta(n^4)
\end{aligned}
$$

Method 3: textbook

The Method 3 is not suitable for this problem, because Method 3 requires that $T(n) = aT(\frac{n}{b}) + f(n)$. However, in this question we have multiple terms.

(b) $T(n) = 5 \cdot T(\frac{n}{3}) + 4 \cdot T(\frac{n}{6}) + 26 \cdot T(\frac{n}{9}) + n^2$

Method 1: in-class

$$
\begin{aligned}
&a_1 = 5, a_2 = 4, a_3 = 24, b_1 = \frac{1}{3}, b_2 = \frac{1}{6}, b_3 = \frac{1}{9}, \alpha = 2, \beta = 0 \\
&a_1 \cdot (b_1)^x + a_2 \cdot (b_2)^x + a_3 \cdot (b_3)^x = 1 \\
&x < 2 = \alpha \Rightarrow case\ 1: \\
&\therefore T(n) = \Theta(n^2)
\end{aligned}
$$

Method 2: the Akra-Bazzi Master Theorem

$$
\begin{aligned}
&T(n) = \Theta(n^P(1 + \int_1^n \frac{u^2}{u^{P+1}} du)) \\
&T(n) = \Theta(n^P(1 + \int_1^n u^{1-P} du)) \\
&T(n) = \Theta(n^P(1 + \frac{n^{2-P} - 1}{2 - P})) \\
&T(n) = \Theta(n^2)
\end{aligned}
$$

Method 3: textbook

The Method 3 is not suitable for this problem, because Method 3 requires that $T(n) = aT(\frac{n}{b}) + f(n)$. However, in this question we have multiple terms.

(c) $T(n) = 25 \cdot T(\frac{n}{5}) + n^2 \log n$

Method 1: in-class

$$
\begin{aligned}
&a = 25, b = \frac{1}{5}, \alpha = 2, \beta = 1 \\
&a \cdot (b)^x = 1 \\
&x = 2 = \alpha \Rightarrow case\ 2: \\
&\therefore T(n) = \Theta(n^2 (\log n)^2)
\end{aligned}
$$

Method 2: the Akra-Bazzi Master Theorem

$$
\begin{aligned}
&P = 2 \\
&T(n) = \Theta(n^2 (1 + \int_1^n \frac{u^2 \log u}{u^3} du)) \\
&T(n) = \Theta(n^2 (1 + \int_1^n \frac{\log u}{u} du)) \\
&T(n) = \Theta(n^2 (1 + \int_1^n \log u d \log u)) \\
&T(n) = \Theta(n^2 (1 + \int_0^{\log n} v dv)) \\
&T(n) = \Theta(n^2 (1 + (\frac{1}{2} \log^2 n - 0))) \\
&T(n) = \Theta(n^2 (\log n)^2)
\end{aligned}
$$

Method 3: textbook

$\because f(n) = n^2 \log n, a = 25, b = 5$

$\because f(n) \neq \mathcal{O}(n^{\log_b a - \varepsilon})$ & $f(n) \neq \Theta(n^{\log_b a})$ & $f(n) \neq \Omega(n^{\log_b a + \varepsilon})$ and $af(\frac{n}{b}) < cf(n)$

$\therefore$ we cannot use master method here.

(d) $T(n) = 5 \cdot T(\frac{n}{5}) + n^2 \log n$

Method 1: in-class

$$a = 5, b = \frac{1}{5}, \alpha = 2, \beta = 1$$
$$a \cdot (b)^x = 1$$
$$x = 1 < \alpha \Rightarrow case\ 1:$$
$$\therefore T(n) = \Theta(n^2 \log n)$$

Method 2: the Akra-Bazzi Master Theorem

$$P = 1$$
$$T(n) = \Theta(n(1 + \int_1^n \frac{u^2 \log u}{u^2} du))$$
$$T(n) = \Theta(n(1 + \int_1^n \log u du))$$
$$T(n) = \Theta(n(1 + (n \log n - n) - 1))$$
$$T(n) = \Theta(n^2 \log n)$$

Method 3: textbook

$$\because f(n) = n^2 \log n, a = 5, b = 5$$
$$\therefore f(n) = \Omega(n^{\log_b a + \varepsilon})$$
$$\therefore T(n) = \Theta(f(n)) = \Theta(n^2 \log n)$$

4. Generalize Exercise 9.3-8 in the textbook by changing the input from two arrays X and Y to five arrays A, B, C, D, E each containing n numbers already in sorted order. Prove the time complexity and correctness of your algorithm.
   Solutions:
   Algorithm:

Data: A , B , C , D , E
Result: *the median* M *of five arrays*
$k = 0$;
while *true* **do**
    find the medians of five arrays: $m1, m2, m3, m4, m5 = medians\ of\ A, B, C, D, E$;
    $median_{largest} = max(m1, m2, m3, m4, m5)$;
    $median_{smallest} = min(m1, m2, m3, m4, m5)$;
    $x = min(index(median_{smallest}) - 0, n - index(median_{largest}))$;
    if $a = 0$ && $number\ of\ arrays\ \geq 3$ **then**
        remove the array which only has one element ;
    else
        remove x elements which are smaller than $median_{smallest}$ in the array where the smallest median is found ;
        remove x elements which are larger than $median_{largset}$ in the array where the largest median is found ;
    end
end
return right;

Correctness:

Proof.

1) Given five of arrays A, B, C, D, E with same length n. The algorithm can find the median of five arrays.

2) Base case: $n = 1$, we can easily find the median of the five numbers.

3) Induction hypothesis:

In step k: remove the same number of elements which are smaller than $median_{smallest}$ and ones larger than $median_{largest}$ in the array where these two medians are found. The median of the five new arrays is still the same as the median of the original five arrays.

In the next step, we remove the same number of elements which are smaller than $median_{smallest}$ and ones larger than $median_{largest}$ in the array where these two medians are found. Withour loss of generosity, array A is the array contains the smallest median and array E is the array contains the largest median. All the elements in A before the smallest median are smaller than all the medians, which means all these elements are smaller than all the elements after five medians. These elements are at least smaller than half number of elements in all five arrays which makes median of five arrays cannot

be in them. It's similar to the largest median and all the elements after it. Therefore, the median of the five new arrays is still the same as the median of the original five arrays.

4) Because all the inductive step works, the median of five new arrays is still the same as the one of the original five arrays. Because during each iteration, we reduce the size of arrays, so the input size will always decrease. The algorithm will eventually become the base case, so this algorithm will always terminate.

Time complexity:

In each step, we reduce the size of array. In the worst case, we need $\log(n)$ times operations to remove an array of size n, there are only five arrays. So we can know that the time complexity of this algorithm is $\mathcal{O}(\log n)$.

5. Consider a strictly decreasing function $f : N \to Z$(that is, a function defined on the natural numbers and taking integer values, such that $f(i) > f(i+1)$. Assuming we can evaluate f at any i in constant time,we want to find $n = min\{i \in N | f(i) \leq 0\}$ (that is, we want to find the point where f becomes nonpositive).

   We can easily solve the problem in $\mathcal{O}(n)$ time by evaluating $f(1), f(2), f(3), \cdots, f(n)$. Give an $\mathcal{O}(\log n)$-time algorithm. Prove the time complexity and correctness of your algorithm.

   Solutions:

   Algorithm:

Data: $N, f(i)$
Result: $n = min\{i \in N | f(i) \leq 0\}$
$k = 0$;
Step-1:
while *true* **do**
    $value = f(2^k)$;
    if $value > 0$ **then**
        $k++$;
    else
        break;
    end
end
Step-2:
$left = 2^{k-1}, right = 2^k, mid = left + (right - left)/2$;
while $left < right$ **do**
    $value = f(mid)$;
    if $value > 0$ **then**
        $left = mid + 1$;
    else
        $right = mid$;
    end
end
return right;

Correctness:
Claim. Given a sequence of numbers $left, \cdots, right$ and a decreasing function f. Step-2 correctly returns the minimum n such that $left \leq n \leq right$ and $f(n) \leq 0$.
Proof.
1) Claim. Given a sequence of numbers $left, \cdots, right$ and a decreasing function f, $right - left \leq m$ and $f(right) \leq 0$. Step-2 correctly returns the minimum n such that $left \leq n \leq right$ and $f(n) \leq 0$.
2) Base case: $m = 0$, then $f(left) = f(right) \leq 0$, return $n = right$, the algorithm is correct.
3) Induction hypothesis:
In step k: suppose that P(n) is true for $right - left \leq m$; that is, given a sequence of numbers $left, \cdots, right$ we can find the smallest n such that $left \leq n \leq right$ and

$f(n) \leq 0$.

In next step, we have two cases, if $f((left + right)/2) > 0$, then $left' = (left + right)/2 + 1$; else $right' = (left + right)/2$. In either case, we would find n between $left', \cdots, right'$ such that $right' - left' \leq \frac{m}{2}$. Because $\frac{m}{2} \leq m$, the recursive call can be assumed to work correctly by the induction hypothesis.

4) Because all the inductive step works, we can conclude that Step-2 is correct.

Because during each iteration, we reduce the search range by half, so the input size will always decrease. The algorithm will eventually become the base case, so this algorithm will always terminate.

Time complexity:

The runtime of Step-1 is $k = \log n$.

The time complexity of Step-2:

The base case, when m = 0, it only takes constant time, and so

$T(n) = \Theta(1)$

The recursive case occurs when $m > 0$. Each of the subproblems solved is a subarray of $\frac{n}{2}$ elements and so we spend $T(\frac{n}{2})$ time solving each of them. We have two subproblem, the left subarray and the right subarray, so the contribution of this part is $2T(\frac{n}{2})$. It takes $\Theta(1)$ time to divide the problem.

$$T(n) = \begin{cases} \Theta(1) & \text{if } m = 0 \\ T(n) + \Theta(1) & \text{if } m > 0 \end{cases} \tag{1}$$

$$\begin{aligned} &T(n) = T(\frac{n}{2}) + c \\ &a = 1, b = \frac{1}{2}, \alpha = 0, \beta = 0 \\ &a \cdot (b)^x = 1 \\ &x = 0 = \alpha \\ &This\ is\ case\ 2: \\ &\therefore T(n) = \Theta(\log n) \end{aligned}$$

$\therefore$ The time complexity of this algorithm is $\mathcal{O}(\log n)$.

6. The our-secret-name problem (OSNP) is defined as follows. Given an array $A[1 \cdots n]$ of integers, find values of i and j with $1 \leq i \leq j \leq n$ such that is maximized.

$$(\sum_{k=i}^{j} A(k))^2$$

Give an $\mathcal{O}(n \log n)$ -time divide-and-conquer algorithm to solve OSNP. Prove the time complexity and correctness of your algorithm.

Solutions:

In order to achieve the maximized value of $(\sum_{k=i}^{j} A(k))^2$, we only need to find values of i and j such that $|\sum_{k=i}^{j} A(k)|$ is maximized.

Algorithm: Maximum Subarray

Data: $A[1, \cdots, n]$

Result: i and j which makes $|\sum_{k=i}^{j} A(k)|$ maximized

$left = 1, right = n, mid = left + (right - left)/2,$

$left_{sum} = right_{sum} = 0,$

$left_{max} = left_{min} = right_{max} = right_{min} = 0,$

$i = mid, j = mid + 1,;$

while $i \geq left$ **do**

  $left_{sum} += A(i);$

  if $left_{sum} > left_{max}$ **then**

    $left_{max} = left_{sum};$

  end

  if $left_{sum} < left_{min}$ **then**

    $left_{min} = left_{sum};$

  end

  $i - -;$

end

while $j \leq right$ **do**

  $right_{sum} += A(j);$

  if $right_{sum} > right_{max}$ **then**

    $right_{max} = right_{sum};$

  end

  if $right_{sum} < right_{min}$ **then**

    $right_{min} = right_{sum};$

  end

  $j + +;$

end

$value_{across} = max(|right_{max} + left_{max}|, |right_{min} + left_{min}|);$

$value_{left} = Maximum\ Subarray(A[1, mid - 1]);$

$value_{right} = Maximum\ Subarray(A[mid + 1, n]);$

Keep track of the lower index i and upper index j of each value ;

return $max(value_{across}, value_{left}, value_{right})$ *and* $i,\ j$;

Correctness:

1) Let A be an array. The algorithm $MaximumSubarray(A)$ returns the maximum contiguous subsequence absolute value of sum in A and its index.

2) Base case: the length of A is 1, we can just return 1.

3) Induction hypothesis:
Let A be a array of length $n > 1$, and assume inductively that for any sequence A' of length $n' < n$, $MaximumSubarray(A')$ correctly computes the maximum contiguous subsequence absolute value of sum and return its lower and upper index.
Now consider the array A and let L and R denote the left and right subsequences resulted from spliting A in half according to the algorithm. Furthermore, let $a = [A(i), \cdots, A(j)]$ be a subarray of A that has the largest absolute value of sum among all subarrays of A. Say $|\sum_{k=i}^{j} A(k)| = value$, so $max(value_{left}, value_{right}, value_{across}) \leq value$ since these are all subarrays of A.
There are three possibilies corresponding to how subarray a lies with respect to the spliting element.
a) If a lies entirely in left of the spliting element, then it follows from our inductive hypothesis that $MaximumSubarray(L) = value$, which means the answer we return $max(value_{left}, value_{right}, value_{across}) \geq value_{left} = value$. This, together with $max(value_{left}, value_{right}, value_{across}) \leq value$, implies that the value we find is the largest, and the index we return are correct.
b) If a lies entirely in right of the spliting element, then it follows from our inductive hypothesis that $MaximumSubarray(R) = value$, which means the answer we return $max(value_{left}, value_{right}, value_{across}) \geq value_{right} = value$. This, together with $max(value_{left}, value_{right}, value_{across}) \leq value$, implies that the value we find is the largest, and the index we return are correct.
c) If a lies across the spliting element. In this case, $value_{across} = value$, which means the answer we return $max(value_{left}, value_{right}, value_{across}) \geq value_{across} = value$. This, together with $max(value_{left}, value_{right}, value_{across}) \leq value$, implies that the value we find is the largest, and the index we return are correct
4) Because all the inductive step works, we can conclude that $MaximumSubarray\ Algorithm$ is correct.
Time complexity:
The base case, when n = 1, it only takes constant time, and so
$T(n) = \Theta(1)$
The recursive case occurs when $n > 1$. Each of the subproblems solved is a subarray of $\frac{n}{2}$ elements and so we spend $T(\frac{n}{2})$ time solving each of them. We have two subproblem, the left subarray and the right subarray, so the contribution of this part is $2T(\frac{n}{2})$. It takes $\Theta(n)$ time to find the max absolute value across left subarray and right subarray.

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2T(n) + \Theta(n) & \text{if } n > 1 \end{cases} \tag{2}$$

$$\begin{aligned} &T(n) = 2T(\frac{n}{2}) + n \\ &a = 2, b = \frac{1}{2}, \alpha = 1, \beta = 0 \\ &a \cdot (b)^x = 1 \\ &x = 1 = \alpha \\ &This\ is\ case\ 2: \\ &\therefore T(n) = \Theta(n \log n) \end{aligned}$$

$\therefore$ The time complexity of this algorithm is $\mathcal{O}(n \log n)$.