Homework 5

Jiahao Liu February 10, 2024

1. Solve the following recurrences by giving tight Θ-notation bounds in terms of n for sufficiently large n. Assume that T(n) represents the running time of an algorithm, i.e., T(n) is a positive and non-decreasing function of n. For each part below, briefly describe the steps along with the final answer.

a. $T(n) = 9T(n/3) + n^2 logn$
a = 9, b = 3, $n^{log_b a} = n^2$. f(n) = $n^2 logn$. ~~Fall into case 3, now we check whether $\exists c < 1, s.t.\ af(n/b) \leq cf(n)$. $9 * (n/3)^2 log(n/3) = n^2(logn - log3)$, we can't find c < 1 to meet the inequality. So we can't use the master theorem to solve this problem.~~
Fall into case 2 generalized case, $f(n) = \Theta(n^2 log^k n), k = 1 \geq 0$, we can get $T(n) = \Theta(n^2 log^2 n)$.
Or recall the recursion tree, we have the following:

$$\begin{aligned}
T(n) &= f(n) + a * f(n/b) + a^2 * f(n/b^2) + \cdots + a^k * f(n/b^k) \quad where\ k = logn - 1 \\
&= n^2 logn + 9 * (n/3)^2 log(n/3) + 9^2 * (n/3^2)^2 log(n/3^2) + \cdots + 9^k * (n/3^k)^2 log(n/3^k) \\
&= n^2 logn + n^2 log(n/3) + n^2 log(n/3^2) + \cdots + n^2 log(n/3^k) \\
&= n^2 (log \frac{n^{k+1}}{3^{0+1+2+\cdots+k}}) \\
&= n^2 (log \frac{n^{logn}}{3^{(logn(logn-1))/2}}) \\
&= n^2 (log(n^{logn}) - log3^{(logn(logn-1))/2}) \\
&= n^2 (log^2 n - \frac{logn(logn-1)}{2} log3) \\
&= n^2 ((1 - \frac{log3}{2}) log^2 n + \frac{log3}{2} logn) \Rightarrow T(n) = \Theta(n^2 log^2 n)
\end{aligned}$$

b. $T(n) = 4.01T(n/2) + n^2 logn$
a = 4.01, b = 2, $n^{log_b a} \approx n^{2.0036}$. f(n) = $n^2 logn$. Because:

$$\lim_{n\to\infty} \frac{n^{2.0036}}{n^2 logn} = \lim_{n\to\infty} \frac{n^{0.0036}}{logn} = \lim_{n\to\infty} \frac{0.0036 n^{0.0036-1}}{n^{-1}} = \lim_{n\to\infty} 0.0036 n^{0.0036} = \infty$$

Or because $\forall\ 0 < \epsilon < n^{log_2 4.01} - 2, f(n) = O(n^{log_2 4.01 - \epsilon})$. Fall into case 1, so $T(n) = \Theta(n^{log_2 4.01})$.
c. $T(n) = \sqrt{6000} T(n/2) + n^{\sqrt{6000}}$
a = $\sqrt{6000}$, b = 2, $n^{log_b a} = n^{log_2 \sqrt{6000}}$. f(n) = $n^{\sqrt{6000}}$. Fall into case 3, now we check regualrity condition. $\sqrt{6000} * f(n/2) \leq cn^{\sqrt{6000}} \Leftrightarrow \sqrt{6000}(\frac{n}{2})^{\sqrt{6000}} \leq cn^{\sqrt{6000}} \Leftrightarrow c \geq \frac{\sqrt{6000}}{2^{\sqrt{6000}}} > \frac{77.46}{2^{77.46}}$, we can find c < 1 to meet the inequality. So $T(n) = \Theta(n^{\sqrt{6000}})$.

d. $T(n) = 10T(n/2) + 2^n$
a = 10, b = 2, $n^{log_b a} = n^{log_2 10}$. f(n) = 2^n. Fall into case 3, now we check regualrity condition. $10 * f(n/2) \leq cn^{log_2 10} \Leftrightarrow 10 * 2^{n/2} \leq c2^n \Leftrightarrow c \geq \frac{10}{2^{n/2}}$, we can find c < 1 to meet the inequality. So $T(n) = \Theta(2^n)$.

e. $T(n) = 2T(\sqrt{n}) + log_2 n$
we can't exactly define b in the original problem, let's try linearize the recurrence. Let $m = \log_2 n$, implying that $n = 2^m$. The recurrence relation then becomes: $T(2^m) = 2T(2^{m/2}) + m$. let $S(m) = T(2^m)$. Substituting into our new recurrence gives: $S(m) = 2S(m/2) + m$.
Because now m is not integer, we can not directly use master theorem to solve it. However, we can still see that there are at most log(m) layers for this recursion tree, on each layer there are totally m operations, so $S(m) = \Theta(mlogm)$. we substitute back to express our solution in terms of n: $S(m) = T(2^m) = T(n) = \Theta(\log n \cdot \log(\log n))$
Use recursion tree, we can see that at problem size n, the amount of work we need to put is $log_2 n$; On the 2 subproblems in the next level, the problem size becomes $\sqrt{n}$, and the amount of work for each subproblem is $log_2 \sqrt{n} = (log_2 n)/2$, so the total amount of work we need on the next level is $2 * (log_2 n)/2 = log_2 n$; Hence, we can see that the total amount of work for each level stays the same $log_2 n$.
If the size of each subproblem is being halved from what it was before, and the problem size was n at level 0, we would need $log_2 n$ levels till it would become trivial.
But our problem size is not being halved, it is being square rooted, so we need $log_2 log_2 n$ levels. At each level, the workload is $log_2 n$, so the total time complexity is $\Theta(logn * log(logn))$.

2. Hidden surface removal is a problem in computer graphics that scarcely needs an introduction: when Woody is standing in front of Buzz, you should be able to see Woody but not Buzz; when Buzz is standing in front of Woody, ... well, you get the idea.
The magic of hidden surface removal is that you can often compute things faster than your intuition suggests. Here's a clean geometric example to illustrate a basic speed-up that can be achieved. You are given n nonvertical lines in the plane, labeled $L_1, L_2, \cdots, L_n$ with i^{th} line specified by the equation $y = a_i x + b_i$. We will make the assumption that no three of the lines all meet at a single point. We say line L_i is uppermost at a given x-coordinate x_0 if its y-coordinate at x_0 is greater than the y-coordinates of all the other lines at $x_0, \forall j \neq i, a_i x_0 + b_i > a_j x_0 + b_j$. We say line L_i is visible if there is some x-coordinate at which it is uppermost, intuitively some portion of it can be seen if you look down from $y = \infty$.
Give an algorithm that takes n lines as input and in O(nlogn) time returns all of the ones that are visible.

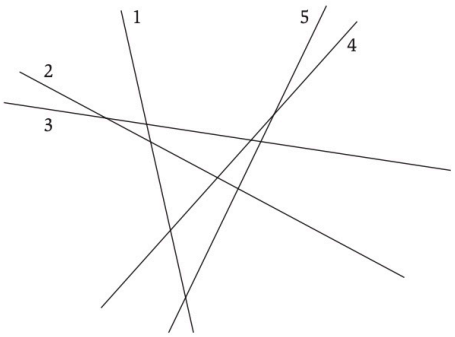


Figure 5.10 An instance of hidden surface removal with five lines (labeled *1–5* in the figure). All the lines except for *2* are visible.

Algorithm 1: FindVisibleLines(a set of lines)

Input: A array of n lines $L_1, L_2, \cdots, L_n$
Sort the lines $\{L_i : y = a_i x + b_i\}$ by their **slopes** a_i in non-decreasing order; O(nlogn)
return helper($\{L_i\}$);

Algorithm 2: helper(a set of sorted lines)

Input: A array of n sorted lines $L_1, L_2, \cdots, L_n$
if *n == 1* **then**
  return $\{L_1\}$*, empty set of intersection points;*
end
else if *n == 2* **then**
  return $\{L_1, L_2\}$*, the intersection point of the two lines;*
end
else
  S_1, P_1 = helper($L_1, L_2, \cdots, L_{\lfloor \frac{n}{2} \rfloor}$); Visible lines and their intersections from left half
  S_2, P_2 = helper($L_{\lfloor \frac{n}{2} \rfloor + 1}, \cdots, L_n$); Visible lines and their intersections from right half
  foreach *point* $P_{1i} = (x_{P_1 i}, y_{P_1 i})$ *in* P_1 *O(n)* **do**
    Take the line with **smallest slope** in $S_2 : y = a_{S_2} x + b_{S_2}$; O(1)
    if $a_{S_2} x_{P_1 i} + b_{S_2} > y_{P_1 i}$ **then**
      Find the two lines in S_1 that intersect at P_{1i}; Delete one with **bigger value of slope** from S_1
    end
  end
  foreach *point* $P_{2i} = (x_{P_2 i}, y_{P_2 i})$ *in* P_2 *O(n)* **do**
    Take the line with **biggest slope** in $S_1 : y = a_{S_1} x + b_{S_1}$; O(1)
    if $a_{S_1} x_{P_2 i} + b_{S_1} > y_{P_2 i}$ **then**
      Find the two lines in S_2 that intersect at P_{2i}; Delete one with **smaller value of slope** from S_2
    end
  end
  Merge the two sets of visible lines $S = S_1 \cup S_2$;
  return S*, all the intersections of lines in* S
end

Preparation: We sort the lines by their slopes, which takes $\Theta(nlogn)$ time.
Divide: Divide the lines into two halves, which takes $\Theta(1)$ time.
Conquer: We recursively find the visible lines and their intersection points from the left half and right half.

Combine: According to the explanation in helper function, we can merge the two sets of visible lines and their intersection points in O(n) time.
The recursion equation is $T(n) = 2T(n/2) + n$, with a = b = 2, we have $n^{log_a b} = n, f(n) = \Theta(n)$, fall into case 2, so the recursion process takes $\Theta(nlogn)$.
The total time complexity of this algorithm is $\Theta(nlogn)$.

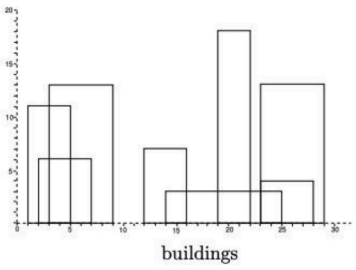
3. Assume that you have a blackbox that can multiply two integers. Describe an algorithm that when given an n-bit positive integer a and an integer x, computes x^a with at most O(n) calls to the blackbox.

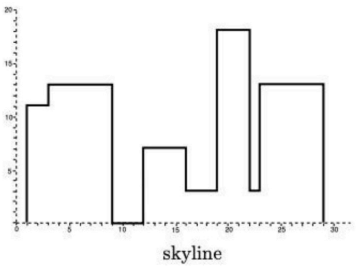
We can get the binary representation of a with n-bit, and use the quick power algorithm to solve this problem. Because the while loop iterate at most n times, each iteration call blackbox at most twice, the overall call O(n) times of blackbox.

Algorithm 3: QuickPower(int a, int x)

```
Input: A n-bit positive integer a, an intege x
if x == 0 then
    return 0;
end
Initialize result = 1;
while a != 0 do
    if a & 1 == 1, i.e. the last bit of a is 1 then
        result = blackbox(result, x);
    end
    x = blackbox(x, x); // square x;
    a = a » 1; // shift right 1 bit to get the next bit of a;
end
return result;
```

4. A city's skyline is the outer contour of the silhouette formed by all the buildings in that city when viewed from a distance. A building B_i is represented as a triplet (L_i, H_i, R_i) where L_i and R_i denote the left and right x coordinates of the building, and H_i denotes the height of the building. Describe an O(nlogn) algorithm for finding the skyline of n buildings. For example, the skyline of the buildings {(3, 13, 9), (1, 11, 5), (12, 7, 16), (14, 3, 25), (19, 18, 22), (2, 6, 7), (23, 13, 29), (23, 4, 28)} is {(1, 11),(3, 13), (9, 0), (12, 7), (16, 3), (19, 18), (22, 3), (23, 13), (29, 0)}. (Note that the x coordinates in a skyline are sorted)





We can solve this by divide and conquer algorithm. We first sort all building by their L_i in non-decreasing order, then divide the them into two halves, and use a helper function to recursively find the skyline of the left half and right half, then merge the left and right skylines to get the final skyline. (Please refer to the detail explanation in the following helper function of how to combine two skylines)
The Preparation step of sorting buildings on their L_i takes $\Theta(nlogn)$.
The recursion equation is $T(n) = 2T(n/2) + O(n)$, with a = b = 2, we have $n^{log_a b} = n, f(n) = \Theta(n)$, fall into case 2, so the recursion process takes $\Theta(nlogn)$.
The total time complexity of this algorithm is $\Theta(nlogn)$.

Algorithm 4: FindSkyline(a set of buildings)

```
Input: A array of n buildings {(L_i, H_i, R_i)}
Sort the buildings by their L_i in non-decreasing order; O(nlogn)
return helper({(L_i, H_i, R_i)}, 0, buildings.size() − 1);
```

Algorithm 5: helper(a set of sorted buildings, left, right)

Input: A array of n sorted buildings $\{(L_i, H_i, R_i)\}$, left and right index
if *left == right* **then**
  return $\{(L_{left}, H_{left}), (R_{left}, 0)\}$;
end
Initialize $mid = \lfloor \frac{left+right}{2} \rfloor$;
S_l = helper($\{(L_i, H_i, R_i)\}, left, mid$); Skyline of left half S_l: list of points
S_r = helper($\{(L_i, H_i, R_i)\}, mid + 1, right$); Skyline of right half S_r: list of points
Initialize pointer for points in left and right skyline: $i = 0, j = 0$, the heigth for left and right skyline $h_l = 0, h_r = 0$, and the point coordinates to be added to merged skyline (x, y), the final skyline S to $\varnothing$;
while $i < S_l.size()\&\&j < S_r.size()$ **do**
  // Each time we choose one point from left or right skyline to update the (x, y) for merged skyline;
  // When the right skyline is empty or the next point in left skyline has a smaller x-coordinate;
  if $j == S_r.size()| \,|(i < S_l.size()\&\&S_l[i].x < S_r[j].x)$ **then**
    $x = S_1[i].x$; $h_l = S_1[i].y$; $i++$;
  end
  // When the left skyline is empty or the next point in right skyline has a smaller x-coordinate;
  else
    $x = S_2[j].x$; $h_r = S_2[j].y$; $j++$;
  end
  $y = \max(h_l, h_r)$; now we have got the coordinate (x, y) might be added to merged skyline;
  // When the new point has the same x-coordinate with the last point in merged skyline S, update its y-coordinate (height);
  if $S.size() > 0\&\&x == S.back().x$ **then**
    $S.back().y = y$;
  end
  // When the new point has the same y-coordinate with the last point in merged skyline S, ignore this point because of the skyline definition.;
  else if $S.size() > 0\&\&y == S.back().y$ **then**
    continue;
  end
  // Apart from the above two cases, add the new point to the merged skyline S;
  else
    $S.push\_back((x, y))$;
  end
end
return S;

5. Emily has received a set of marbles as her birthday gift. She is trying to create a staircase shape with her marbles. A staircase shape contains k marbles in the kth row. Given n as the number of marbles help her to figure out the number of rows of the largest staircase she can make.(Time complexity < O(n))

Use binary search to find the largest k such that $1 + 2 + 3 + \cdots + k \leq n$ with time complexity $O(logn)$.

Algorithm 6: FindLargestStaircase(int n)

Input: A integer n
Initialize $left = 0, right = n$;
while *left + 1 != right* **do**
  Initialize $mid = \lfloor \frac{left+right}{2} \rfloor$;
  if $\frac{mid*(mid+1)}{2} \leq n$ **then**
    $left = mid$;
  end
  else
    $right = mid$;
  end
end
// Now left is the right bound of k such that $1 + 2 + 3 + \cdots + k \leq n$;
// right is the left bound of k such that $1 + 2 + 3 + \cdots + k > n$;
return *left*;

6. Suppose you're consulting for a bank that's concerned about fraud detection, and they come to you with the following problem. They have a collection of n bank cards that they've confiscated, suspecting them of being used in fraud. Each bank card is a small plastic object, containing a magnetic stripe with some encrypted data, and it corresponds to a unique account in the bank. Each account can have many bank cards corresponding to it, and we'll say that two bank cards are equivalent if they correspond to the same account. It's very difficult to read the account number off a bank card directly, but the bank has a high-tech equivalence tester that takes two bank cards and, after performing some computations, determines whether they are equivalent.
Their question is the following: among the collection of n cards, is there a set of more than n/2 of them that are all equivalent to one another? Assume that the only feasible operations you can do with the cards are to pick two of them and plug them in to the equivalence tester. Show how to decide the answer to their question with only O(nlogn) invocations of the equivalence tester.

First, we can observe that if there is 1 set of more than n/2 of cards that are all equivalent and we divide n cards into two n/2 halves, at least one half has more than n/4 equivalent cards. Given this fact, we can use divide and conquer algorithm to solve this problem.
If there are fewer than 2 cards, directly return the card.
Otherwise, we divide the cards into left and right halves L, R, note that $|L|, |R| \leq \lceil n/2 \rceil$, and recursively find the equivalent card in L, R. Each half should return at most one card that represents the equivalent card class, we have $0 \leq |S_L|, |S_R| \leq 1$.
In the combine step, if the card in S_L, S_R is same, we return this card; otherwise, we use the equivalence tester to compare each card in $S_L \cup S_R$ with all the cards in $L \cup R$ to check if there are more than $(|L| + |R|)/2$ equivalent cards. If we find such a set, just return one representation card, otherwise, return $\varnothing$.
Because the combine step on each layer of the recursion tree needs 2*n equivalence tester invocations, we have the recursion equation $T(n) = 2T(n/2) + 2n$. a = b = 2, $n^{log_b a} = n, f(n) = 2n = \Theta(n^{log_b a})$, Fall into case 2, so $T(n) = \Theta(nlogn)$

Algorithm 7: FindEquivalentCard(a set of cards)

```
Input: A array of n cards
if n == 1 then
    return the card;
end
else if n == 2 then
    return the card if they are equivalent, otherwise ∅;
end
else
    L, R = divide the cards into two halves;
    S_L = FindEquivalentCard(L); S_R = FindEquivalentCard(R);
    if S_L == S_R then
        return S_L;
    end
    else
        foreach card c in S_L ∪ S_R do
            if the number of cards in L ∪ R that are equivalent to c is more than (|L| + |R|)/2 then
                return c;
            end
        end
        return ∅;
    end
end
```