University of Waterloo
CS341 - Winter 2016
Assignment 1

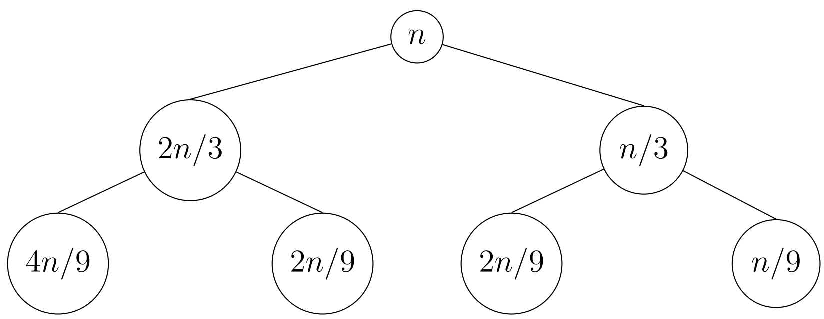
Due Date: Monday January 25 at 11:59pm

Problem 1 Counting Inversion

Ans: The code is in the a1p1.cpp.

Problem 2 Solving Recurrence

1. $T(n) = T(2n/3) + T(n/3) + n^2$
   Ans: According to the tree,



....

   each operations are relatively n^2, $\frac{5}{9}n^2$, $\frac{25}{81}n^2$ ...
   So, by the sum of them(geometric sum), we get $Sum = \frac{9}{4}n^2 \in O(n^2)$,
   So, $T(n) = O(n^2)$.

2. $T(n) = \sqrt{n} * T(\sqrt{n}) + n$
   Ans: we can list equations:

$$T(n) = n^{\frac{1}{2}} * T(n^{\frac{1}{2}}) + n$$

$$T(n) = n^{\frac{1}{2}} * (n^{\frac{1}{4}} * T(n^{\frac{1}{4}}) + n^{\frac{1}{2}}) + n$$

$$T(n) = n^{\frac{3}{4}} * T(n^{\frac{1}{4}}) + n + n$$

$$T(n) = n^{\frac{7}{8}} * T(n^{\frac{1}{8}}) + n + n + n$$

....

   for n, $n^{\frac{1}{2}}$, $n^{\frac{1}{4}}$, ..., $n^{\frac{1}{2^i}}$, because $2^i = \log n$, then $i = \log\log n$
   And by the above equations, we get each step takes n operations
   So, $T(n) = O(n \log\log n)$.

Problem 3 Incomparable Pairs

Ans:

According to the question, there is a 2D-plane with n points; And a pair of points (x_i, y_i) and (x_j, y_j) is incomparable if $x_i \geqslant x_j$ but $y_i \leqslant y_j$, or $x_i \leqslant x_j$ but $y_i \geqslant y_j$.

Firstly, we can sort the x_i, which cost $O(n \log n)$. Then we can look y_i as a list of integers. Because x_i is in the increasing order right now, then we can just count the number of inversion of y_i by the method in the Problem 1, which cost $O(n \log n)$. Finally the number we get is the number of incomparable pair.

Pseudo-code:

Incomparable-Count(A)

A: a list of points (x_i, y_i), $i = 1, 2, .., n$

1. Sort(A,x_i) (sort A by x_i)
2. Split A into two arrays B and C with length s and t, by y_i
3. $i \leftarrow 1$;$j \leftarrow 1$;$k \leftarrow 0$; $D \leftarrow \emptyset$
4. while $i < s$ and $j < t$,
5.   if $(B[i] < C[j])$ then
6.   append $B[i]$ to D
7.   i++
8.   else
9.   append $C[j]$ to D
10.   j++
11. $k \leftarrow k + m - i + 1$
12. Append $B[i, s]$ and $C[j, t]$ to D
13. Return k

k is the final result, and the time complexity, $T(n) = O(n \log n) + O(n \log n) = O(n \log n)$.

Problem 4 Finding Maximum Space

(a)Ans:

As we know, width of all buildings is assumed to be 1. For every building w, we calculate the area with w as the smallest building in the rectangle.

Firstly, we need to know the indexes of the first building which is smaller than w on left of w and on right of w. Let us call these indexes as lindex and rindex.

We read all buildings from left to right, create an array (a stack) for buildings. Every building is pushed to stack once. When we get a lower building, a building is popped from stack. When a building is popped, its height would be used to calculate the area. At the same time, it is seen as smallest bar. i.e. the current index is the rindex and index of previous item in stack is the lindex.

Pseudo-code:

Max-Space(H)

H: an array of heights of buildings, for every building $H[i]$, where $i = 0, 1, .., n - 1$

1. Create an empty stack.
2. Start from first building, and do following for every building $H[i]$ where i increases from 0 to n-1.

   - 1) if ($stack\ is\ empty$ or $H[i] > the\ height\ of\ building\ at\ top\ of\ stack$ $(H[top])$) then
     - push i to stack
   - 2) if $(H[i] < H[top])$ then
     - remove the top of stack
     - calculate area of rectangle with $H[top]$ as smallest building (For $H[top]$, the lindex is previous item in stack (i.e. it is previous to top) and rindex is the current index (i.e. i))

3. if $stack\ is\ not\ empty$ then
   - one by one remove all buildings from stack and do step 2.2 for every popped building.

4. return a number which is the maximum space

Because we only push and pop once (most n numbers), the time complexity is linear, $O(n)$.

(b)Ans:

Obviously, from the bottom to up, we can run the function *Max-Space(H)* given by (a) for each row. There are n rows and n columns, for each row's H, $H[i]$ is the number of cells for this column until meeting the occupied space (if the first cell is occupied space, $H[i] = 0$), for $i = 0, 1, .., n-1$. And let r represents the index of row, for $r = 1, 2, .., n$.

We can get a maximum area from each row, finally we pick the maximal one among them. And this is the maximum rectangular unoccupied space.

Pseudo-code:

Max-unoccupied

```
create r ← 1, R ← 0, S ← 0
while (r < n) do
   create an n-size array H as above description
   R=Max-Space(H)
   if (R > S) then
      S ← R
   r++
return S, which is the maximum rectangular unoccupied space
```

Because Max-Space(H) costs $O(n)$ and we run it n times, The time complexity is $T(n) = T(n-1) + cn = O(n^2)$.