CSE 373 Analysis of Algorithms February 22, 2015

Solutions to HW 1

Steven Skiena

PROBLEM **1**(5pts)

[1–17] **Base case:** When we have $n = 1$ vertex, we have $0 = n - 1$ edges.

Inductive step: Assume for $n = k$, we have $k - 1$ edges. We already know that for any given tree, it must have at least one leaf node. When given a tree with $k + 1$ vertices, we pick an leaf node and delete it, to get a tree with k vertices. According to our assumption, this tree has $k - 1$ edges, thus the original tree has $k - 1 + 1 = k$ edges.

Thus, a tree with n vertices always have $n - 1$ edges. □

[1–19] Assuming I have 30 books, each having around 600 pages, in total I have about $30 \times 600 = 9000$ pages, which is no where close to a million. Suppose the school library has 20 shelves, each having 100 books, then the total pages assuming 600 pages per book is $20 \times 100 \times 600 = 1200000$ pages.

PROBLEM **2**(5pts)

[1–20] We assume that there are about 40 lines per page and about 10 words per line. Multiply by 500 pages and we get about $200,000$ words.

[1–22] The population of US is approximately 300 million and there are approximately $5,0000$ people per city or town. Therefore the answer is $300 \times 1065,0000 = 6,000$.

PROBLEM **3**(10pts)

[2–7] (a) True – $2^{n+1} = 2 \cdot 2^n = O(2^n)$
(b) False – Suppose $2^{2n} = O(2^n)$ is true, then $\exists$ a constant $c, \forall n \geq n_0, 2^{2n} \leq c \cdot 2^n$, thus $c \geq 2^n$, which is impossible.

[2–8] (a) $f(n) = 2logn = \Theta(g(n))$
(b) $f(n) = \Omega(g(n))$
(c) $f(n) = \Omega(g(n))$
(d) $f(n) = \Omega(g(n))$
(e) $f(n) = \Omega(g(n))$
(f) $f(n) = \Theta(g(n))$
(g) $f(n) = \Omega(g(n))$
(h) $f(n) = O(g(n))$

Note that to show $f(n) = \Theta(g(n))$, you need it show $f(n) = O(g(n))$ and $f = \Omega(g(n))$.

PROBLEM **4**(10pts)

[2–19] The functions from lowest to highest order:

$(\frac{1}{3})^n \mid 6 \mid \log\log n \mid \log n, \ln n \mid (\log n)^2 \mid n^{1/3} + \log n \mid \sqrt{n} \mid$

$\frac{n}{\log n} \mid n \mid n \log n \mid n^2, n^2 + \log n \mid n^3 \mid n - n^3 + 7n^5 \mid \frac{3}{2}^n \mid 2^n \mid n!$

Note: (1) For $(\frac{1}{3})^n$ and 6: $\lim_{n->+\infty} \frac{6}{(\frac{1}{3})^n} = +\infty$, thus $6 = \Omega((\frac{1}{3})^n)$
(2) According to Stirling's approximation: $n! = \Theta(\sqrt{2\pi n}(\frac{n}{e})^n) = \Omega(n^n) = \Omega(2^n)$.

PROBLEM **5**(10pts)

[2–21] (a) True.
(b) False.
(c) True.
(d) False.
(e) True.
(f) True.
(g) False.

[2–22] (a) $f(n) = \Omega(g(n))$
(b) $f(n) = O(g(n))$
(c) $f(n) = \Omega(g(n))$

[2–23] (a) Yes. $O(n^2)$ worst case means that on no input will it take more time than that, so of course it can take $O(n)$ on some inputs.
(b) Yes. $O(n^2)$ worst-case means that on no input will it take more time than that. It is possible that all inputs can be done in $O(n)$, which still follows this upper bound.
(c) Yes. Although the worst case is $\Theta(n^2)$, this does not mean all cases are $\Theta(n^2)$.
(d) No. $\Theta(n^2)$ worst case means there exists some input which takes $\Omega(n^2)$ time, and no input takes more than $O(n^2)$ time.
(e) Yes. Since both the even and odd functions are $\Theta(n^2)$.

[2–24] (a) No.
(b) Yes. Note that $\log 3^n = n \log 3$ and $\log 2^n = n \log 2$. $\log 2$ and $\log 3$ are constants.
(c) Yes.
(d) Yes.

PROBLEM **6**(15pts)

[3–2]

```
struct node
{
```

```
    int data;
    struct node* next;
};

struct node* reverse(struct node* head)
{
    struct node* prev = NULL, *current = head, *next = NULL;
    while (current)
    {
        next = current -> next;
        current -> next = prev;
        prev = current;
        current = next;
    }
    return prev;
}
```

PROBLEM **7**(10pts)

[3–4] Note that your query space is $\{1, \ldots, n\}$. So you can just use a bit array where $A[i] = 1$ if i is in the array, otherwise $A[i] = 0$. To insert i you just need to make $A[i] = 1$ if it is not already so. Similarly for deletion you make $A[i] = 0$. To Search i, just check the value of $A[i]$: $A[i] = 1$, then return true, else return false. Hence they are all $O(1)$.

PROBLEM **8**(15pts)

[3–10] Using any kind of balanced binary search tree to store the bins, because it supports insertion, deletion and search in $O(\log n)$ time. The keys of the bins are their free spaces.

(a) We try to put the objects one by one. We search in the tree for the bins which has the smallest amount of extra room and the extra room is sufficient to hold the object.

More specifically, for each object we start from the root node. If root node can hold current object, then update the smallest amount of extra room we current have, and go down to the left node. Else, go down to the right node. Repeat this process until we reach a null node. If we find a suitable bin, then put current object in the best-fit bin, and update the BST. Else, we use a new bin to hold this object, and insert it into the tree.

Note that do an in-order traversal in the tree is incorrect: this may gives you a time complexity of $O(n)$.

(b) We try to put the objects one by one. We search in the tree for the bins which has the largest amount of extra room. If such bin exists and is sufficient to hold the object, we put objects in it and update our tree; otherwise, we use a new bin and insert it into the tree.

Note: This is an NP-Complete problem. Neither best-fit nor worst-fit can always give you the optimal solution.

PROBLEM **9**(20pts)

[3-11] (a) In this question we can take any amount of preprocessing time, can only use $O(n^2)$ space, and answer the range minimum queries in $O(1)$ time. Just use a $n \times n$ matrix where position (i, j) stores the minimum of $x_i, \ldots, x_j$.

(b) In this question we can take any amount of preprocessing time, but we can only use $O(n)$ space, and the range minimum queries should take $O(\log n)$ time.

Here, we would like to build a binary tree (actually this is called segment tree) such that:

(1) It has a leaf node for each number x_i
(2) Each node represents the minimum of all the nodes below it. In other words, it represents the minimum value in a certain interval.

Let $min(i, j) = min(x_i, \ldots, x_j)$. If a node in the tree denotes $min(i, j)$ then its left child is $min(i, \lfloor i + j/2 \rfloor)$ and the right child is $min(\lfloor i + j/2 \rfloor + 1, j)$. Here we build such a tree for an array with 8 numbers $[1, 7, 2, 3, 10, 4, 5, 9]$. See the figure below:

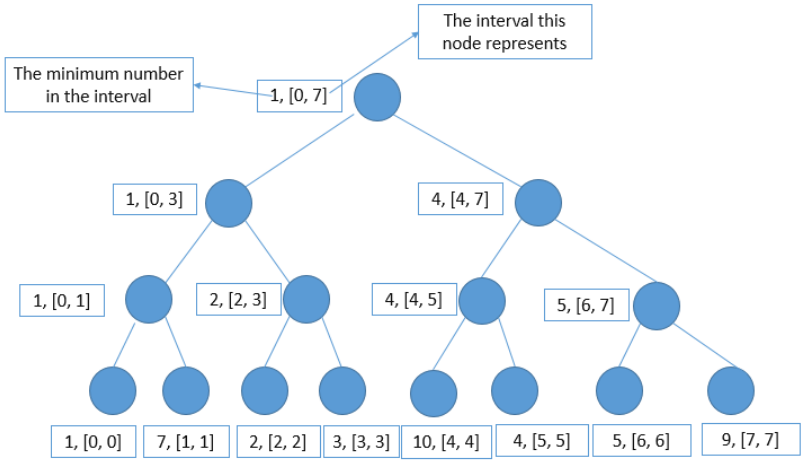

FIGURE 1. Segment tree

Each node in the tree is associated with the interval it represents, and the minimum number in that interval. For example, the root node represents the interval $[0, 7]$, and the minimum number in this interval is 1.

Now that we have stored our values, we answer the queries in the following way:

(1) Start from the root node, and recursively find the O(log n) nodes which are fully covered by the query interval. Note that when we find a node which is fully covered, we simply return the minimum value in its interval, and don't look at its left/right child.
(2) The answer to the query is the minimum of all these nodes.

Below are some figures to show you how to process a query.

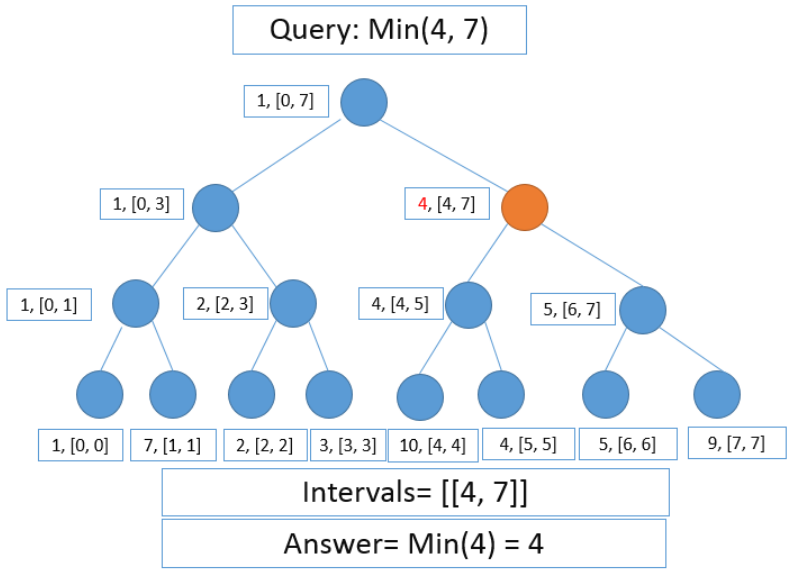

FIGURE 2. Query 1

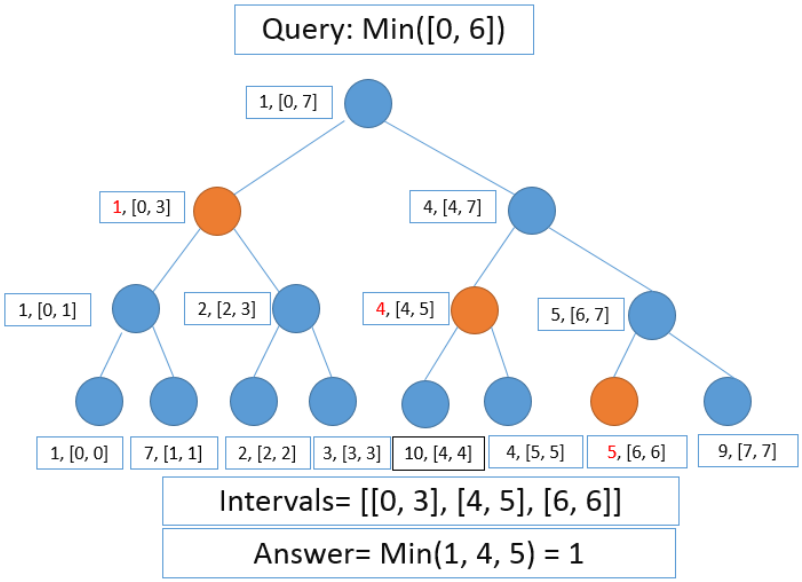

FIGURE 3. Query 2

The psuedocode for the query is as follows:

```
//[left, right] is the query interval
//root is the root node in the segment tree
//this function return the min value in interval [left, right]
Query(root, left, right)
{
```

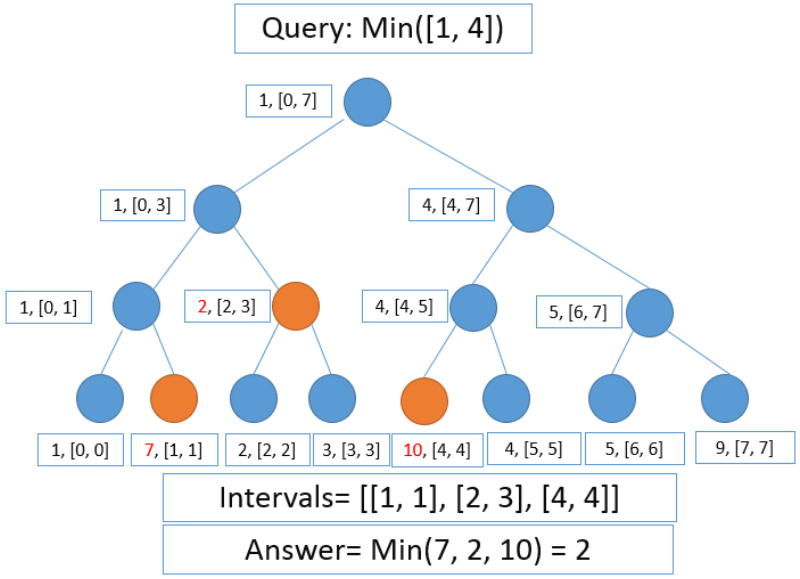

FIGURE 4. Query 3

```
    //case 1: current interval fully covered by the query interval
    if (root.left >= left AND root.right <= right)
        return root.minValue
    //case 2: current interval don't intersect with the query interval, return error
    else if (root.left > right OR root.right < left)
        return ERROR;
    //case 3: current interval partially intersect with the query interval
    else
    {
        //get the middle of current interval.
        //the left child represents the interval [root.left, mid]
        //the right child represents the interval [mid + 1, root.right]
        mid = (root.left + root.right) / 2;
        leftMinValue = rightMinValue = INT_MAX
        if (mid >= left)
            leftMinValue = Query(root.leftchild, l, r)
        if (mid < right)
            rightMinValue = Query(root.rightchild, l, r)
        return min(leftMinValue, rightMinValue)
    }
}
```

For a given query $[left, right]$, we simply execute $Query(rootnode, left, right)$ and it returns the min value in interval $[left, right]$.

Firstly we prove that the above tree uses $O(n)$ space. The tree have $n, n/2, n/4, \cdots, 1$ nodes on each level respectively, thus the total number of nodes $= \sum_{i=0} \frac{1}{2^i} n \leq 2n = O(n)$.

Secondly we prove that query time is $O(\log n)$. Apparently, this tree has at most $O(logn)$ levels. For each level, we visit at most two nodes. Thus, the query time should be $O(\log n)$.

Note: The range minimum query (RMQ) problem has many interesting solutions with different preprocessing/space and time trade-offs. Another interesting solution is building a sparse table, which gives us space complexity of $O(nlogn)$ and time complexity of O(1). You can learn more about RMQ in this website: RMQ and LCA