CSE 373 Analysis of Algorithms Date: September 26, 2016

Solution to HW1

Total Points: 100

Steven Skiena

Problem 1

1-17 *[5]*

Base case: When we have $n = 1$ vertex, then we have $0 = n - 1$ edges. *[1]*
Inductive step: Assume for $n = k$, we have $k - 1$ edges. To show that for $n = k+1$, we have k edges. To prove this, note that no matter how T_k (k-vertex tree) looks like, two things must hold in T_{k+1}, due to the property of trees:

(1) The $k + 1^{th}$ vertex has to be incident on any one of the vertices of T_k. If not, the tree will be disconnected! *[2]*

(2) The $k + 1^{th}$ vertex cannot be incident on more than one vertex of T_k. Suppose it is incident on two vertices $u, v \in T_k$. What goes wrong? Remember that since T_k was a tree (and hence connected), there was a path from u to v in T_k. Now our $k + 1^{th}$ vertex, call it x forms another path (u, x, v) from u to v. Thus we get a cycle! This cannot happen if T_{k+1} is a tree. Thus, for $n = k + 1$, we have exactly $k - 1 + 1 = k$ edges. *[2]*

Remark: 5 points are distributed for each step and each case.

1-19 *[4]*

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. *[2]*

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. *[2]*

Remark: 3 points are distributed for each sub-problem. This is just a sample answer for your reference. We will give credits for any reasonable assumption and estimation.

Problem 2

1-20 *[3]*

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. *[3]*

Remark: This is just a sample answer for your reference. We will give credits for any reasonable assumption and estimation.

1-22 *[3]*

The population of US is approximately 300 million and there are approximately $50,000$ people per city or town. Therefore the answer is $300,000,000/50,000 = 6,000$. *[3]*

Remark: This is just a sample answer for your reference. We will give credits for any reasonable assumption and estimation.

Problem 3

2-7 *[2]*

(a). True. $2^{n+1} = 2 \cdot 2^n = O(2^n)$ *[1]*
(b). False. $2^{2n} = 4^n$ *[1]*

2-8 *[16]*

(a). Θ
(b). Ω
(c). Ω
(d). Ω
(e). Ω
(f). Θ
(g). Ω
(h). O
Remark: 2 points for each correct answer.

Problem 4

2-19 *[9]*

The functions from lowest to highest order:

$\frac{1}{3}^n \mid 6 \mid \log\log n \mid \log n, \ln n \mid (\log n)^2 \mid n^{\frac{1}{3}} + \log n \mid \sqrt{n} \mid \frac{n}{\log n} \mid n \mid n \log n \mid n^2, n^2 + \log n \mid n^3 \mid n - n^3 + 7n^5 \mid \frac{3}{2}^n \mid 2^n \mid n!$

Note: (1). $\frac{1}{3}^n$ will always be less than 1. (2). $n! \approx n^n$ (See Stirling's approximation).
Remark: 1 point for 2 correct orders.

Problem 5

2-21 *[7]*

(a). True
(b). False
(c). True

(d). False
(e). True
(f). True
(g). False
Remark: 1 point for each correct answer.

2-22 *[3]*

(a). Ω *[1]*
(b). O *[2]*
(c). Ω *[2]*
Remark: 1 point for each correct answer.

2-23 *[5]*

(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 time, and no input takes more than $O(n^2)$ time.
(e) Yes. Since both the even and odd functions are $\Theta(n^2)$.
Remark: 1 point for each correct answer.

2-24 *[4]*

(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.
Remark: 1 point for each correct answer.

Problem 6

3-2 *[6]*

```
typedef struct list{
  item_type item;
  struct list *next;
}list;
```

```
void reverse_list(list **l){
  list *p = NULL;
  list *q = *l;
  list *r;
  while( q != NULL ){
    r = q -> next;
    q -> next = p;
    p = q;
    q = r;
  }
  *l = q;
}
```

Remark: 6 points for completely correct. 4 points for correctly reversing the list with minor errors.

Problem 7

3-4 *[8]*

Note that your query space is $\{1, ..., n\}$. So you can just use a bit array or integer array to indicate $A[i] = 1$ 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 $A[i] = 0$. Search i is just i if $A[i] = 1$, else not present. Hence they are all $O(1)$. Pseudocode is presented below:

```
int search(int A[n], i){
  if (i < 1 or i > n)
    return error;
  else {
      if (A[i] == 1)
        return YES;
      else
        return NO;
  }
}
void insert(int A[n], i){
  if (i < 1 or i > n)
    return error;
  else{
     A[i] = 1;
  }
}
void delete(int A[n], i){
  if (i < 1 or i > n)
    return error;
  else{
     A[i] = 0;
```

```
  }
}
```

Remark: propose the array data structure - 2 points; search in $O(1)$ - 2 points; insertion in $O(1)$ - 2 points; deletion in $O(1)$ - 2 points.

Problem 8

3-10 *[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. If such bin exists, we put objects in it and update our tree; if not, we use a new bin and insert it to the tree. Pseudocode is presented below: *[5]*

```
int bestfit(node, w) {
  if ( w == 0 )  return 1;
  if ( node == NULL )  return 0;
  if (node->space == w)
    node->space = 0;
    Adjust the tree to be the balanced binary search tree;
  else if (node->space < w) {
    if ( !bestfit(node's right child, w) ) {
      Create a new node and set its space to 1 − w;
      Insert the new node to the tree and adjust it to be balanced.
      return 1;
     }
  }
  else {
   if ( !bestfit(node's left child, w) ) // All its left descendents can't hold w.
     {
      node->space = node->space - w;
      Adjust the tree to be the balanced binary search tree;
      return 1;
    }
  }
}
int main() {
  Node *root = new Node;
  root->space = 1;
  root->left = NULL;
  root->right = NULL;
  for i = 1 to n {
```

```
    bestfit(root, w[i]);
  }
  return the number of nodes of the tree;
}
```

(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 to the tree. Pseudocode is presented below: *[5]*

```
int worstfit(node, w) {
  if ( w == 0 )  return 1;
  if ( node == NULL )  return 0;
  if ( !worstfit(node's right child, w) ) {
    Create a new node and set its space to 1 − w;
    Insert the new node to the tree and adjust it to be balanced.
    return 1;
  }
}
int main() {
  Node *root = new Node;
  root->space = 1;
  root->left = NULL;
  root->right = NULL;
  for i = 1 to n {
    bestfit(root, w[i]);
  }
  return the number of nodes of the tree;
}
```

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

Problem 9

3-11 *[15]*

(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 $x_i, ..., x_j$. *[5]*

(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.

Let $min(i, j) = min(x_i, ..., x_j)$.

Build a binary tree in the following manner: the root of the tree is $min(1, n)$ the left child of the root is $min(1, \lfloor n/2 \rfloor)$ and the right child of the root is $min(\lfloor n/2 \rfloor + 1, n)$. We do this recursively, that is, 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 $min(\lfloor (i+j)/2 \rfloor + 1, j)$. The leaves of the tree are $x_1, ..., x_n$.

See figure below.

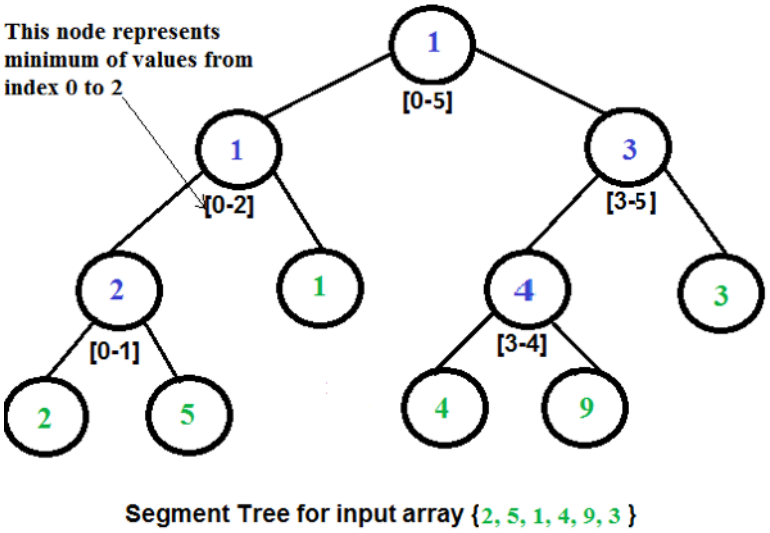

Figure 1: Segment tree

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

```
//  qs - query start index, qe - query end index
int RMQ(node, qs, qe) {
  if range of node is within qs and qe
    return value in node
  else if range of node is completely outside qs and qe
    return error
  else
    return min( RMQ(node's left child, qs, qe),
                RMQ(node's right child, qs, qe) )
}
```

Since the algorithm only stores n elements, the tree uses $O(n)$ space.

Since the tree is the balanced binary tree, and the height of the tree is $O(\log n)$, the search time is $O(\log n)$. *[10]*