Alexander Szostek
October 10th, 2014
CISC220-010

# Lab 5 - Drozdek Exercises

---

## Exercise 2.11.7

**Question:**

The algorithm presented in this chapter for finding the length of the longest subarray with the numbers in increasing order is inefficient, because there is no need to continue to search for another array if the length already found is greater than the length of the subarray to be analyzed. Thus, if the entire array is already in order, we can discontinue the search right away, converting the worst case into the best. The change needed is in the outer loop, which now has one more test:

```
for (i = 0, length = 1; i < n-1 && length < n - i; i++)
```

What is the worst case now? Is the efficiency of the worst case still $O(n^2)$?

**Answer:**

The worst case for this function is if the length of the subarray is the length of the array or, in other words, the array is in increasing order. The efficiency of the worst case now is O(n). The reason why is because the function only has to loop through the array once to determine the length of the longest subarray with increasing numbers. Before the change, the function would continue even if the length of the subarray was greater than the number of remaining elements in the next possible subarray.

---

## Exercise 2.11.8

Find the complexity of the function used to find the kth smallest integer in an unordered array of integers:

```
int selectkth(int a[], int k, int n) {
int i, j, mini, tmp;

for (i = 0; i < k; i++) {
        mini = i;
        for (j = i+1; j < n; j++)
                if (a[j]<a[mini])
                        mini = j;
                tmp = a[i];
                a[i] = a[mini];
                a[mini] = tmp;
                }
return a[k-1];
```

}

**Answer:**

In the worst case, we give selectkth an unorganized array and we ask for the nth smallest element, where n is the length of the array. In this case, the outer loop would loop n times and the inner loop would need to iterate the entire array to check for smaller elements. Thus, the complexity of selectkth is $O(n^2)$ in the worst case.

---

## Exercise 2.11.10

**Question:**

Find the computational complexity for the following four loops:

```
a. for (cnt1 = 0, i = 1; i <= n; i++)
        for (j = 1; j <= n; j++)
            cnt1++;
b. for (cnt2 = 0, i = 1; i <= n; i++)
        for (j = 1; j <= i; j++)
            cnt2++;
c. for (cnt3 = 0, i = 1; i <= n; i *= 2)
        for (j = 1; j <= n; j++)
            cnt3++;
d. for (cnt4 = 0, i = 1; i <= n; i *= 2)
        for (j = 1; j <= i; j++)
            cnt4++;
```

**Answers:**

**Key**

Occurs 1 time

Occurs n times

Occurs 2n times

Occurs (n/2) times

Occurs $\log_2(n)$ times

Occurs $n * \log_2(n)$ times

Occurs $n^2$ times

a.
```
for (cnt1 = 0, i = 1; i <= n; i++)
    for (j = 1; j <= n; j++)
        cnt1++;
```

$f(n) = 1 + 1 + n + n + n^2 + n^2$

$f(n) = 2 + 2n + 2n^2$

$f(n)$ is $O(n^2)$.

b.
```
for (cnt2 = 0, i = 1; i <= n; i++)
        for (j = 1; j <= i; j++)
                cnt2++;
```

$f(n) = 1 + 1 + n + n + (n/2) + (n/2)$
$f(n) = 2 + 2n + n = 2 + 3n$
$f(n)$ is $O(n)$.

c.
```
for (cnt3 = 0, i = 1; i <= n; i *= 2)
        for (j = 1; j <= n; j++)
                cnt3++;
```

$f(n) = 1 + 1 + \log_2(n) + \log_2(n) + n*\log_2(n) + n*\log_2(n)$
$f(n) = 2 + 2*\log_2(n) + 2n*\log_2(n)$
$f(n)$ is $O(n*\log_2(n))$.

d.
```
for (cnt4 = 0, i = 1; i <= n; i *= 2)
        for (j = 1; j <= i; j++)
                cnt4++;
```

$f(n) = 1 + 1 + \log_2(n)\ (\log_2(n) + 2n + 2n)$
$f(n) = 2 + \ (\log_2(n)^2 + 4n * \log_2(n))$
$f(n)$ is $O(n * \log_2(n))$.

* As n increases, j++ and cnt4++ approach running 2n times as n approaches infinity.

---

**Question:**

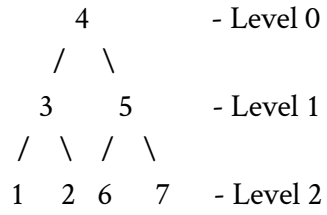
Depending on the "shape" of the tree, *insert()* for a binary search tree might be as good as *O(log n)* (where *n* is the number of nodes in the tree) or it might be as bad as *O(n)*. Explain in detail what characteristics lead to both of these outcomes.

**Answer:**

To start, the absolute best case of inserting a node into a binary search tree is inserting the node into an empty tree, and the complexity in this situation is simply O(1). In this case, the complexity of inserting this node does not depend on anything other than the speed of doing the insertion itself, which is constant.

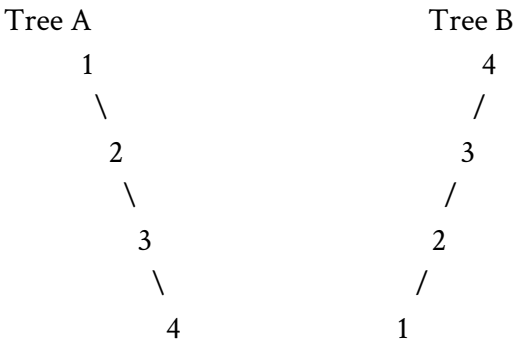
Alexander Szostek
October 10th, 2014
CISC220-010

In the average case for inserting a binary search tree is as good as $O(\log_2 n)$. Suppose we had a tree:




As we move down the tree level by level, we are effectively eliminating a half of the possible places that the node that we're inserting could be. If we're at Level 0, there are 8 possible places that we could insert a node. At Level 1, there are only 4 possible places. At Level 2, only 2 possible places. (Assuming that nodes that are inserted into a tree will always become a leaf). In this case, the complexity of inserting a node into this tree is $O(\log_2 n)$.

In the absolute worst case scenario, an unbalanced binary search tree could resemble either of the two examples below:




As we move down the tree, we are only eliminating one possible location that the node would be inserted at each level. At Level 0, there are 5 possible places to insert a node. At Level 1, there are 4 possible places, etc. In the example above, inserting a node could be as simple as $O(1)$, such as inserting at the right of 4 in Tree A, and as complex as $O(n)$, inserting left of 1 in Tree B.