Student: Yu Cheng (Jade)

Math 475

Homework #4

March 17, 2010

Course site Exercises 3

Problem 1: The procedure below takes an array of integers and determines if some elements occur three (or more) times in the array. Which of the following big-O estimates: $O(logn)$, $O(n)$, $O(nlog\ n)$, $O(n^2)$, $O(n^2logn)$, $O(n^3)$, $O(n^3logn)$, $O(n^4)$, and $O(2^n)$ best describes the worst-case running time of the algorithm.

```
public boolean hasThreeEqual(int[] arr) {
    int n = array.length;
    for  (int i = 0; i < n; i++) {
        for  (int j = i + 1; j < n; j++) {
            for  (int k = j + 1; k < n; k++) {
                if  (arr[i] == arr[j] && arr[i] == arr[k]) {
                    return true;
                }
            }
        }
    }
    return false;
}
```

Answer: The big-O runtime complexity of the procedure above is $O(n^3)$. The number of elementary operations in the inner most loop is 2. The total number of elementary operations of the entire procedure can be expressed as the summations form below,

$$\sum_{i=0}^{n-1}\sum_{j=i+1}^{n-1}\sum_{k=j+1}^{n-1} 2 = \sum_{i=0}^{n-1}\sum_{j=i+1}^{n-1} 2(n-1-j)$$

$$= \sum_{i=0}^{n-1}\left[2(n-1)\sum_{j=i+1}^{n-1} 1 - 2\sum_{j=i+1}^{n-1} j\right]$$

$$= \sum_{i=0}^{n-1}(n^2 - 3n - 2ni + 3i + i^2 + 2)$$

$$= (n^2 - 3n + 2)\sum_{i=0}^{n-1} 1 + (3-2n)\sum_{i=0}^{n-1} i + \sum_{i=0}^{n-1} i^2 = \frac{\mathrm{n}^3}{3} - \mathrm{n}^2 + \frac{2\mathrm{n}}{3}.$$

Therefore the runtime complexity of the given procedure is $O\left(\frac{n^3}{3} - n^2 + \frac{2n}{3}\right) = O(n^3)$, as we drop the lower order terms and the constants.

Problem 2: Find a fast algorithm to do problem 1.

Answer: Copied below is a faster procedure that accomplishes the given task, checking if there is a value in an given array that appears three times or more. This procedure belongs to $O(n^2)$. The basic idea is trading space with speed. Instead of a three level nested loop, we have a two level nested loop and a look-up record array. Each record entry contains an integer value and a count number that this value has appeared.

For each value in the given array, we check if it is already in the record. If it is, we increment of count of this record entry, and return true if the count reaches 3. If the value is not already in the record, we add it in the record, and set the count for this record entry as 1.

```
public boolean fastHasThreeEqual(int[] arr) {
    int n = arr.length;
    pair[] record = new pair[n];
    int sizeOfRecord = 0;
    for (int i = 0; i < n; i++) {                      // for each element, arr[i]
        for (int j = 0; j < sizeOfRecord; j++) {
            if (arr[i] == record[j].value) {           // check if it's in the record
                record[j].count++;                     // if it is, increment count
                if (record[j].count == 3) {            // if the count reaches 3
                    return true;                       // return true and terminate
                }
                break;
            }
        }                                              // if arr[i] is not recorded
        records[sizeOfRecord++] = new pair(arr[i], 1); // add it and set count to 1
    }
    return false;                                      // if procedure never returned
}                                                      // in the inner loop, it means
                                                       // there's no such element

private class pair {                                   // a helper object that has
    int value;                                         // a value and keeps a record
    int count;                                         // of how many times this
    public pair(int newV, int newC) {                  // value appeared in the array
        value = newV;
        count = newC;
    }
}
```

The worst case for this procedure is when no value in the given array appears three or more than three times. In this case, our algorithm would evaluate every value and add it into the record, and then return false at the end.

The number of execution for the inner loop body starts from 0 and go all the way to $n-1$. This is by measuring the size of the record as the algorithm proceeds. It is a fixed number of operations that the inner loop body carries. Therefore the runtime complexity is bounded by the number of times that the inner loop body executes. Hence the runtime is bounded by summation below,

$$0+1+2+\cdots+(n-1)=\frac{(n-1)}{2}\times n=\frac{n^2}{2}-\frac{n}{2}.$$

Therefore the runtime complexity of the given procedure is $O\left(\frac{n^2}{2}-\frac{n}{2}\right)=O(n^2)$, as we drop the lower order terms and the constants.

Problem 3: Let V be a vector of size n representing a partition as described in class. Describe an algorithm that modifies V so that each tree (block) has depth at most 1 and runs in time $O(n)$.

Answer: The following code demonstrates a procedure that accomplishes this task in time $O(n)$.

```
public void reArrangeToDepthOne(A, V) {
    int n = A.length;
    for (int i = 0; i < n; i++) {
        int a = A[i];                       // for each element of A, a
        int b = V[a];                       // obtain its parent (or negative #), b
        if (b < 0)                          // if b is a negative number
            continue;                       // move on to the next value
        root(a, V);                         // otherwise call the root subroutine
    }
}

private int root(a, V) {                    // root subroutine rearrange the parents
    int b = V[a];                           // to be the base root for all values
    if (b < 0)                              // encountered on the way up the branch
        return a;
    int r = root(b, V);
    V[a] = r;
    return r;
}
```

Proof of correctness: This is obvious. The *root* subroutine returns the base root of the specified value and rearranges every value encountered on the way up to the root, including the specified value itself, to be a child of the root directly. After the execution of the main routine, other than the root values, every other value has gone through the *root* procedure. They were reassigned to be the root's children. The result tree has, therefore, a depth of only one.

Proof of big-O, $O(n)$: The procedure appears to be slower than $O(n)$ because of the nested recursion method call. But the number of recursion steps is bounded by the outer loop, n. Therefore, the runtime ***cannot*** be calculated simply with $O(n)\times O(largestDepth)=O(n\times largestDepth)$.

In this procedure, when a value is evaluated and becomes a child of the base root, its children will not need to redo this procedure in order to find the base root. Instead they will be assigned to the root through their parent. In other words, the procedure of going up a particular branch to look for the root happens only once. There's no repeat, every portion of a branch in a tree is covered once and only once in the repeated *root* subroutine calls. The summation of the branches of all trees linearly corresponds to the size of the forest, which is n. Therefore the entire procedure belongs to $O(n)$.

Problem 4: Let V be a vector of size n representing a partition and suppose the trees have depth at most 1. Find an algorithm that runs in time $O(n)$ that changes V so that the root of each tree is the smallest member of that block (and the depth stays at most 1).

Answer: The following code demonstrates a procedure that accomplishes this task in time $O(n)$.

```
public void reArrayToSmallestRoot(A, V) {
    int n = A.length
    for (int i = 0; i < n; i++) {
        int a = A[i];                       // for each element of A, a
        int b = V[a];                       // obtain its parent (or negative #), b
        if (b < 0)                          // if b is a negative number
            continue;                       // move on to the next value
        if (b > a) {
            int tmp = V[a];                 // otherwise compare a with b
            V[a] = V[b];                    // if parent b is larger than child a
            V[b] = tmp;                     // swap the values
        }
    }
}
```

Proof of correctness: This is obvious. The loop invariant in this procedure is root value is no greater than the evaluated children values. The invariant is true before the execution because none children are evaluated yet. The invariant is true after each round of loop body execution because we only swap the parent and child when the parent has greater value. After swapping the parent becomes no greater than this child. Procedure for sure terminates because we loop a fixed number of times and there's no nested loop or recursion of any kind.

Proof of big-O, $O(n)$: This is also obvious. Procedure contains a loop with executes its loop body n times. There's no nested loop or recursion of any kind in the loop body. Therefore, the time complexity of this procedure is $O(n)$.

Problem 5: The lattice of all subsets of an n-element set has order dimension n; that is, it is the intersection of n linear extensions and no fewer. We outlined in class that it is the intersection of n linear extensions, so you don't need to do that part. But prove that it cannot be written as the intersection of fewer than n linear extensions.

Answer: We have a lattice for a subset relation of an n-element set. We can provide n' linear extensions for this partial order set, where $n' < n$. We will then prove that these n' linear extensions' intersection cannot equal to the subset relation. It contains cover relation between two elements that are not comparable in the original subset relation.

We have n', where $n' < n$, number of linear extensions. Let's extract the sub linear extension from each linear extension that contains only the single element subsets. They are illustrated as below,

$$\left.\begin{array}{rccc} linear\ extentions\ \#1: & \emptyset & \cdots & \{a_1, a_2, \cdots, a_n\} \\ linear\ extentions\ \#2: & \emptyset & \cdots & \{a_1, a_2, \cdots, a_n\} \\ \vdots & \vdots & \vdots & \vdots \\ linear\ extentions\ \#n': & \emptyset & \cdots & \{a_1, a_2, \cdots, a_n\} \end{array}\right\} n'$$

$$\Rightarrow \left.\begin{array}{rccc} sub\ linear\ extentions\ \#1: & \{a_i\} & \cdots & \{a_j\} \\ sub\ linear\ extentions\ \#2: & \{a_k\} & \cdots & \{a_l\} \\ \vdots & \vdots & \vdots & \vdots \\ sub\ linear\ extentions\ \#n': & \{a_m\} & \cdots & \{a_n\} \end{array}\right\} n'$$

Since $n' < n$, there is at least one single element set that does not take the last position of the n' orderings for the single element sets. Let's call this element a_{middle}. Therefore we have the following relation,

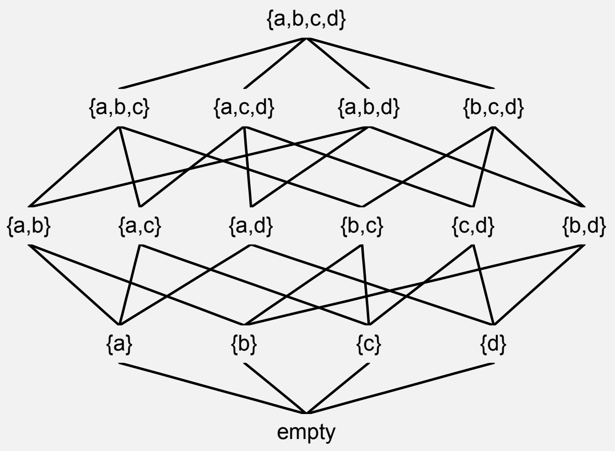
$$\left.\begin{array}{rcccc} sub\ linear\ extension\ \#1 & \cdots & \{a_{middle}\} & \cdots & \{a_i\} \\ sub\ linear\ extension\ \#2 & \cdots & \{a_{middle}\} & \cdots & \{a_j\} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ sub\ linear\ extension\ \#n' & \cdots & \{a_{middle}\} & \cdots & \{a_n\} \end{array}\right\} n'$$

According to sub linear extension #1, $\{a_{middle}\}$ appears before any subset that contains a_i. According to sub linear extension #2, $\{a_{middle}\}$ appears before any subset that contains a_j, and so on. Combining all n' linear extensions, we conclude that in the intersection of these n' linear extensions, the subset $\{a_{middle}\}$ will appear before any subset that contains $a_i, a_j, \cdots, a_n$. The smallest one of such subset is $\{a_i, a_j, \cdots, a_n\}$. Therefore, with n' linear extensions, we always have $\{a_{middle}\} \prec \{a_i, a_j, \cdots, a_n\}$, where a_{middle} is an element that does not appear as the last single element subset in the n' linear extensions, and $a_i, a_j, \cdots, a_n$ are the elements that do appear as the last single element subset in the n' linear extensions.

However, according to the subset partial order relation, $\{a_{middle}\}$ and $\{a_i, a_j, \cdots, a_n\}$ are incomparable subsets. There is no covering relation between these two subsets in the original relation. Therefore, with n' linear extensions, where $n' < n$, the intersection of these linear extensions cannot recover the original subset poset. Extra covering relations exist in the intersection, and more linear extensions are need to eliminate these extra covering relations.

Hence, we've proven that the minimal number of linear extensions required to intersect into the original subset lattice is larger than n'. Therefore, the dimension of the subset partial order relation is at least n.

For example, we have a four element subset partial order relation, and it is illustrated as below,



We provide three linear extensions for this subset lattice, and in these linear extensions, element b does not appear as the last single element subset, while the rest of the elements, a, c, d, all do. Then, we have the following relation,

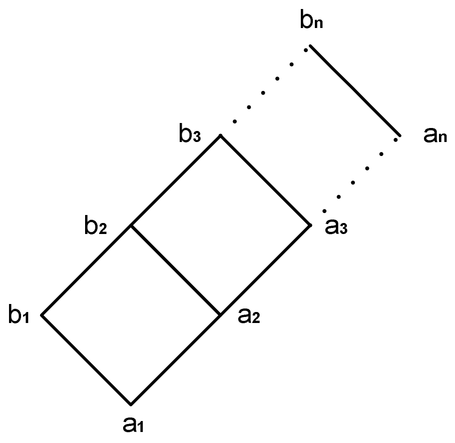
$$\begin{array}{lccccc} linear\ extension\ \#1: & \cdots & \{b\} & \cdots & \{a\} & \cdots \\ linear\ extension\ \#2: & \cdots & \{b\} & \cdots & \{c\} & \cdots \\ linear\ extension\ \#3: & \cdots & \{b\} & \cdots & \{d\} & \cdots \end{array}$$

$$\begin{array}{l} \\ \Rightarrow \\ \\ \end{array} \begin{array}{lccccccc} linear\ extension\ \#1: & \cdots & \{b\} & \cdots & \{a\} & \cdots & \{a,c,d\} & \cdots \\ linear\ extension\ \#2: & \cdots & \{b\} & \cdots & \{c\} & \cdots & \{a,c,d\} & \cdots \\ linear\ extension\ \#3: & \cdots & \{b\} & \cdots & \{d\} & \cdots & \{a,c,d\} & \cdots \end{array} \begin{array}{l} \\ \Rightarrow \{b\} \\ \\ \end{array}$$

$$\prec \{a,c,d\} \in intersection\ of\ linear\ extension\ \#1, \#2, and\ \#3$$

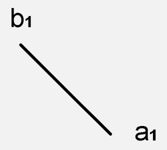
Subsets $\{b\}$ and $\{a,c,d\}$ are, however, not comparable in the original subset relation. More linear extensions, where $\{a,c,d\} \prec \{b\}$ are needed to eliminate this extra cover relation.

Problem 5: Show the lattice in the picture has $\frac{1}{n+1}\binom{2n}{n}$ number of linear extensions.

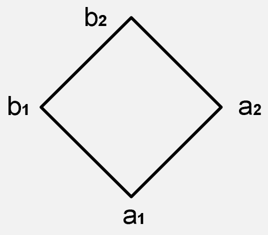


Answer: In order to map this problem onto Theorem 8.1.1, we need to prove this problem is the same as looking for the number of sequences of a $2n$ element set that contains n +1' and n -1's, where the partial sums are always positive, $a_1 + a_2 + \cdots + a_k \geq 0, (k = 1, 2, \cdots, 2n)$.

We look at the base cases first. If $n = 1$, there is one linear extension, $\{a_1, b_1\} \leftrightarrow \{+1, -1\}$



If $n = 2$, there are two linear extensions, $\{a_1, b_1, a_2, b_2\} \leftrightarrow \{+1, -1, +1, -1\}$ and $\{a_1, a_2, b_1, b_2\} \leftrightarrow \{+1, +1, -1, -1\}$. The bases case are good.



By looking at the base case example pattern, we now use +1 to represent $a_i, (i = 1, 2, \cdots, n)$, and -1 to represent $b_j, (j = 1, 2, \cdots, n)$. The final linear extensions will be sequences of +1's and -1's.

Prove the one-to-one mapping: Since $a_1 \prec a_2 \prec \cdots \prec a_n$ and $b_1 \prec b_2 \prec \cdots \prec b_n$, once the final sequence of +1's and -1's is chosen, it corresponds to a unique sequence of a's and b's. It works the other way too. A defined sequences of a's and b's, following the ordering relationship, has a unique corresponding sequence of +1's and -1's, where a's are replaced by +1's and b's are replaced by -1's.

Prove the correctness of the mapping: The constraint of $a_1 \prec a_2 \prec \cdots \prec a_n$ and $b_1 \prec b_2 \prec \cdots \prec b_n$ are considered in the previous section. Another constraint of this partial order relation is

$a_k \prec b_k$. With a +1's and -1's sequence where $a_1 + a_2 + \cdots + a_k \geq 0, (k = 1, 2, \cdots, 2n)$, at any stage, the number of a's equals or outnumbers of number of b's. Since a's and b's are laid out so that a_j is after a_i, if $j > i$, and b_j is after b_i, if $j > i$, we know b_k is always laid out later than a_k. Therefore the constraint $a_k \prec b_k$ is preserved in the +1's and -1's representation. Therefore, we've shown that all of the covering relations on the given poset are preserved in Theorem 8.11's model.

Conclude from the two proves, the given problem is the same problem as the one in Theorem 8.1.1. Hence the number of linear extensions of the given pattern of partial order set is $\frac{1}{n+1}\binom{2n}{n}$, according the theorem.