Bryan Arnold

CSE 3500

4/3/17

Homework 8

Maximum Profit

For this problem, we want to find the optimal number day in which to buy a stock to maximize profit between day i and j. This is an improvement of an earlier algorithm, so it needs a dynamic programming approach to reduce the runtime to O(n).

First, we want a collection of all the stocks predictions for n days, so 0 through n – 1 days. Then. We want to go through the collection in reverse and denote the maximum profit initially as day n-1. Next go to n – 2 day profit, check whether it is greater than the initially given maximum profit of day n – 1. If it is, assign maximum profit as the day n – 2, otherwise, add max-stocks[j], where j=current index, to your current profit. Finally, you simply display the maximum profit you will receive.

Algorithm:

Input: the collection of stock predictions in an Array denoted as A

Initialize the last stock day as the current maximum profit: max = A[n-1]

for j from n-1 to 0

if current stock being looked at is greater than the current max: max = stocks[j];

Sum up the profit gained: profit = profit + max-stocks[j];

End for j

Display the profit to finish the program.

Maximum Consecutive Subarray

Algorithm:

Input: the array of integers to find the maximum consecutive subarray; A

First, initialize two integers to 0 to look at two different maximums at a time. One is the overall found maximum, the other is the maximum of each loop iteration:

```
totalMax = maxCurrent = 0
```

Next, loop through the array of integers:

```
For loop i from 0 to length of A
```

Set the current max of this iteration to itself plus the current element in the array in the iteration:

```
maxCurrent = maxCurrent + A[i]
```

Now, if the total maximum is less than the current maximum is found, set the total maximum to the current iteration maximum. Also, take another integer value to store the location for the start of the array once the maximum sum is found:

```
If totalMax < maxCurrent, totalMax = maxCurrent

iterationTracker = i
```

Now, check if the current sum is below 0, and reset it to 0 and set another integer to denote the end of the iteration:

```
If maxCurrent  < 0, maxCurrent = 0

iterationEnd = i

end for i
```

Finally, use the placeholders to display the integers that add to the maximum consecutive sum:

```
For j from iterationTracker to iterationEnd

Display element in A[j]

End for j
```

The recursive solution isn’t the correct way to do this algorithm, even if we know the answer from 0 to i-1 and we know the ith element, because we don't know whether the answer for 0 to i - 1 ends at which index. The final ending sequence should be a consecutive path throught he array of integers, but we can't just directly add ith element to o to i-1. This could cause the found answer to not be consecutive throughout the array. The answer could have ended at any jth element less than equal to i-1 and directly adding ith element will give not give us the maximum consecutive sum and the elements that make up the maximum sum. Since both of the loops in this problem are separate and loop through the elements once, the runtime is linear: O(n).

Interleaving Strings

Algorithm:

First, we want to take the input of the two sequences from the two ships, x and y, as well as the signal being listened to, s:

Input: Strings x, y, and s.

Next, separate the characters composing the strings into an array, so one character per element in the array:

toArray → s, x, y. They turn into s[], x[], and y[].

Next, iterate through the length of s[], and initializing two new variables to track the location and size of the current character:

For i from 0 to the length of s[]

Int a = i; //track location

Int n = 0; //length of matching so far

Now, we iterate through the length of string x and if characters match, increase the first for loop's iterator as well as the length of matching so far:

For j from 0 t o length of x[]

If s[i] is the same character as x[j]: i++ and n++

End if

Also, if n equals the length of x, we now iterate over the length of s[] again to check for y, also initializing new tracking and length variables:

If n is the length of s[]:

For k from 0 to the length of s[]

int a1 = k; //track location

int m = 0; //track length matching

Now, repeat the process done for x prior to this but for y:

For p from 0 to length of y[]

```
If s[k] is the same as y[p]: k++ and m++
End if
```

Now, if the length tracker of matching is the same as the size of y, the two strings are interleaving:

```
If m = the length of y[]
Display that the strings are interleaving
p = a1
end if
end for
end for
end if
i = a; //keep looping for checks
end for
```

Since you must go through multiple for loops, the runtime for this will be in polynomial time (O(n^3)).

Exercise 24

For this algorithm, first, we need to take in input for all the precincts and voters within each:

Input: the collection of precincts with voters in each. Denoted as P[]

Now, we must split up all the precincts in half into districts denoted as X and Y. We then initialize four variables to hold the number of voters that vote for the same party in district X and Y, as well as half the number of precincts in each district and voters in each precinct:

```
Divide total precincts in P[], half in new X[] and Y[].
int a = total voters party A has in a precinct
int b = total precincts party A has in a district
int t = total precincts in district/2 //precincts in a district
int t1 = voters in current precinct/2
```

Now, we must iterate through both districts. First, looking at their voters. If party A wins a precinct, add one on to the total precincts that party A has won:

```
For i from 0 to length of X[]

t1 = voters in X[i]  //to check majority of voters in precinct

if voter is voting for party A, add it to a

If(a > t1) a majority exists, so add 1 to b.

End if

End for
```

Now, check whether b is greater than t: half the total precincts in the current district. If it is, make a note of with a boolean:

If b > t: create new boolean k = true, otherwise make k = false.

Reset all variables used, so b= 0, a = 0 and reinitialize to proper values for next respective district.

Now, repeat the same process for district Y[]:

```
For i from 0 to length of Y[]

t1 = voters in Y[i]  //to check majority of voters in precinct

if voter is voting for party A, add it to a

If(a > t1) a majority exists, so add 1 to b.

End if

End for
```

Finally, check whether b is greater than t: half the total precincts in the current district. If it is, make a note of it with a boolean and compare to the other boolean. If they are both true, gerrymandering is possible and should be displayed. If one isn’t, gerrymandering isn’t a possibility:

If b > t: create new boolean j = true, otherwise make j false

If k == j, then display gerrymandering is possible, otherwise say it isn’t.

This will find out if gerrymandering is possible for a district with a given party. To find out fully, one would have to input the different party and different split of districts. Although those aren’t included in the algorithm I gave, they simply require input differences of the party to be checked as well as different splitting of the precincts into districts. This program has to loop through the number of precincts in each district, n, as well as the voters in each precinct in each district, m. So, the loops make the running m*n, for a runtime of O(m*n).