Homework 5

Tags	Homework
Date	

Problem 1. You are given a one dimensional array that may contain both positive and negative integers, find the sum of contiguous subarray of numbers which has the largest sum using the divide-and-conquer paradigm. Analyze the time complexity of your algorithm

For example, if the given array is {-2, -5, **6, -2, -3, 1, 5**, -6}, then the maximum subarray sum is 7 (see highlighted elements 6, -2, -3, 1, 5).

ALGORITHM

Function takes original set of numbers, and then two arrays for the recursive calls that are made

```
inputs:
let arr be the input array
let start be the beginning index
let end be the end index

function:
if arr is empty
  no sum

if arr has one element
  element is the contiguous sum

mid = middle index of arr
rightMax = arbitrarily large negative number
tempMax = temporary maximum, set to 0

for each element from mid+1 to end
  add the element to tempMax
  if tempMax > leftMax
    set leftMax = tempMax

leftMax = arbitrarily large negative number
tempMax = temporary maximum, set to 0
```

```
for each element from mid to start (incrementing backwards)
  add the element to tempMax
  if tempMax > leftMax
    set leftMax = tempMax

  set recursiveMax = choose maximum from
    recursive search from passing arr, start, mid for left side
    recursive search from passing arr, mid+1, end for right side

  set currMax = leftMax + rightMax

  return maxiumum between recursiveMax and currMax
```

PROOF

- Base condition:
  - Consider the case where there is only one element in the given array, that element is a contiguous subarray of the entire array, therefore it must be the largest sum.
- Induction step:
  - Assume that the algorithm correctly executes the largest contiguous subarray sum for k elements, we must prove it must also correctly execute for k-1 elements.
  - To prove this, we must show that separating the array of k-1 elements in halfs yields the correct contiguous array.
  - There are three cases where the contiguous max can lie when separating an array in half
    - entirely on the left side of the middle index
    - entirely on the right side of the middle index
    - between the left and right side
  - For each (k-1)/2 array, we take the maximum overlapping the middle element and compare that with the largest maximum from the left OR right side
    - we get the maximum overlapping the middle element because we increment/decrement from the middle to the right and left side, respectively,

to determine the maximum contiguous sum from the middle to start and middle to end and then add those two values

- for the k-1 array we find the location of the maximum in the three places by taking the added value above and recurse over the left and right sides until we inevitably left with 2 elements whether k-1 is odd
- Dividing the array continuously means whether k-1 is odd or even yields a point where we are left with 2 elements where we can determine the maximum by checking whether the left element is greater/less than the right or if the combined sum is greater as proven by the base condition.
- Choosing the maximum between these 3 values yields the largest contiguous array between the 2 elements, which transcends similarly to 4 elements up until k-1 elements.
- Henceforth, the algorithm executes correctly for k-1 elements which by induction proves that the algorithm is true for k elements as well.
- By induction, the algorithm finds the largest sum of the contiguous array.

RUNTIME

- This divide and conquer approach is similar to that of a merge sort where we divide each of the arrays we find in half to find the maximum, effectively decreasing the number of calls and searches through the array by degrees of 2^n where n is the number of elements in the arrays that are passed
- Henceforth, the worst case runtime **O(nlog(n))** where n is the number of elements in the the initial array given

Problem 2. Exercise 3 on page 314

a)

Let a graph with nodes $v_1, v_2, ..., v_6$ be an ordered graph with the following set of edges: $(v_1, v_2), (v_2, v_3), (v_2, v_4), (v_3, v_5), (v_4, v_5), (v_5, v_6)$

The given algorithm would execute and return the value 3 for the following sequence $(v_1, v_2), (v_2, v_3), (v_3, v_5)$

When the solution really is 4 for the following sequence
$(v_1, v_2), (v_2, v_4), (v_4, v_5), (v_5, v_6)$

The issue lies in the fact that the node that w is set to HAS to be the smallest value j from the rest of the nodes w connect to.

b)

ALGORITHM

```
input: G, ordered graph
function:
let n = number of nodes in the ordered graph G
let arr = array[0...n]
let arr[0], arr[1] = 0, and rest indexes to be -1
for all i from 2 to n
  for all nodes j that has an edge (j,i)
    if arr[j] >= 0
      set arr[i] to the max between arr[i] to and arr[j]+1
return arr[n]
```

PROOF

- Base condition:
  - consider one node in graph G, that means the longest path is 0, and the algorithm returns a[1] = 0
- Induction:
  - Assume the algorithm to return the correct longest path for k nodes in G that meet the conditions of the problem
  - We must prove that is true for k+1 nodes in G
  - If k+1 nodes, then that means the arr[k] gives us longest path before adding the extra node.
  - If that new node has any edges (j,k+1) such that the path arr[j] exists at the length of 0 at the least, we set the max length if adding any edge (j,k+1) increases the current maximum that exists at arr[k+1]. Regardless of it does or not, arr[k+1] will store the largest path because it will either be incremented or stay at the current maximum.
  - By induction, the algorithm returns the longest path.

RUNTIME

Because we use a nested iteration to iterate through all the nodes and its connected edges, each node at the worst will have to be iterated **O(n^2)** times.

Problem 3. Exercise 5 on page 316.

ALGORITHM

let quantity[a,b] be the total quality of the segmentation from y[a] to y[b]

```
input: string y
function:
let n = length of string y
let arr = array of optimal segmentation qualities
          where arr[i] is the optimal segmentation of characters from y[0] to y[i]
arr[0] = 0, arr[1] = quanlity[y[1]]
for all i from 2 to n
  for all j from 0 to j-1
    if i >= j
      set arr[i] = minimum between arr[i] and (arr[j-1] + quality(j,n))
return arr[n]
```

PROOF

- Base condition:
  - let string y be a singular character, that means that the quantity of y[1] yields the optimal segmentation quality. therefore the base condition is true.
- Induction step:
  - Assume the algorithm gives us the the correct max total quality for k elements. we must prove the algorithm correctly returns the max total quality for k+1 elements, that is, arr[k+1] should be the returned element.
  - For an extra element, i would iterate and for its iteration, each take each possible continuous substring of the characters and determine whether the current maximum total value at the ith element when greater than j in the dynamic array is smaller than a compared value.

- i must be greater than or equal to j as therefore the sequence of characters containing the first j-1 characters must be optimal total quality. That is because arr[j] gives us the value of the optimal segmentation of that relevant sequence of characters, which means that arr[i] optimal total quality would be either its current value or the arr[j] + quality(j,k+1)
- Therefore, the algorithm finds the value for each last sequence of characters up until k+1 in y, and then determines the total optimal segmentation by comparing these values and yield the output in arr[k+1].
- Henceforth, the algorithm finds the segmentation of the maximum total quality.

RUNTIME

The algorithm uses nested loops to traverse the character array, y, to find each possible iteration of a continuous word in y. Therefore, there are at most n*n continuous plausible words that could be iterated over, yielding a worst case time complexity of **O(n^2)** if we assume the quality call is a single call.

Problem 4. Exercise 8 on page 319

a)

We can take the example directly from the book:

i	1	2	3	4
x_i	1	10	10	1
$f(i)$	1	2	4	8

The algorithm finds the smallest value where $f(j) \geq 1$, that is $j = 1$. So we activate the EMP on the 1st second, giving us min(1,1) which is 1 robot destroyed. After that, we recurse on x_1, x_2, x_3. Because there is no value here where $f(j) \geq 10$, we set j = 3 and activate the EMP on the 3rd second after 2 seconds of reload, resulting in min(10,2) destroying 2 robots. Therefore, in total the algorithm has destroyed 3 robots.

This is not the expected output, as the textbook explains that the best solution would be to activate the EMP in the 3rd and 4th seconds, yielding the destruction of 5 robots.

b)

ALGORITHM

```
input: data on robot arrivals x_1, x_2,...,x_n
function:
let n = number of robots
let arr = the max number of robots that can destroyed on x_1,x_2,...,x_n
set arr[0] = 0
for all i from 1 to n
  for all j fron 2 to i
      set arr[i] = maximum between (arr[i], (arr[i-j] + minimum between (x_i, f(j)))
return arr(n)
```

PROOF

- Base condition
  - Assume there to be only one robot arriving, that means in i=1, x_i = 1 and f(i) ≥ 1, therefore the base condition is true since the EMP can destroy at least one robot in one second.
- Induction
  - Assume that the algorithm is true for k seconds and k robots. We must prove the algorithm retains true for k+1 seconds and k+1 robots.
  - If there is an additional robot and second to destroy as many robots as possible, then i = k+1 and for all j from 2 to k+1, we should observe whether the current maximum at arr[k+1] robots that can be destroyed is greater or less than arr[k+1-j] + the minimum between the number of robots arriving and the number of robots that can destroyed from the EMP after i-j seconds of reload
  - if the latter is greater than the current maximum, when the loop is done iterating we will have the max number of robots destroyed at arr[k+1]
  - By induction, the algorithm must find the most maximizing number of robots destroyed for n robots arriving in n seconds.

RUNTIME

We have to iterate through each robot at a maximum of n*n times because of the nested loop, therefore the runtime is at a worst of **O(n^2)**

Problem 5. Given n dice each with m faces, numbered from 1 to m, find the number of ways to get sum X. X is the summation of values on each face when all the dice are thrown. You need to use dynamic programming to solve this problem.

ALGORITHM

```
input: n die, m faces each, x is the sum we want
function:
let arr[1...n][1...m] all be initialized to 0
let min be the minimum between x and m
for all i from min
  arr[1][i] = 1

for all i from 2 to n
  for all j from 1 to x
    for all k from 1 to the minimum between m or j
      add arr[i-1][j-k] to arr[i][j]

return arr[n][m]
```

PROOF

- For the number of ways to get sum X with only die must mean sum x is between 1-6, which means taking the minimum value between the number of faces on the die and the sum x must yield a maximum number to find sum x as only 1 way
- Then for every other number of die and number of faces, we apply the same concept and take the minimum between the total number of faces and the face j that we have already found the total number of ways to get sum x with
- then adding the number of ways to get sum x with i-1 die and the j-k gives us the number of ways to get sum X because it gives us the calculation for the sum for i-1 die and the number of faces we have for the new die and how many sums we can get from those values
- henceforth the algorithm correctly gives us the number of ways to get sum X

RUNTIME

- Because we have to iterate through each n die, each m faces, and for sum x that could exist with each m face, the runtime of this algorithm is **O(n*m*x)**

Problem 6. You are given a set of n types of rectangular 3-D boxes, where the i-th box has height h(i), width w(i) and depth d(i) (all real numbers). You want to create a stack of boxes which is as tall as possible, but you can only stack a box on top of another box if the dimensions of the 2-D base of the lower box are each strictly larger than those of the 2-D base of the higher box. Of course, you can rotate a box so that any side functions as its base. It is also allowable to use multiple instances of the same type of box.

ALGORITHM

```
input: n types of rectangular 3d boxes
function:
rot = array with all possible rotations the given boxes
sort the array in a decreasing order
arr = array with all the maximum stack heights

initialize all arr values with the stack heights of the boxes from rot
for i from 1 to n
  for j from 0 to i
    if the rot(w(i) and d(i)) < rot(w(j) and d(j))
      choose maximum between arr[i] and arr[j] + rot(h(i))

find maximum value from arr[i] and return it
```

PROOF

- First we must consider 3 rotations of each of the 3d boxes and store that in the rotation array. it is enough to consider those 3 rotations because the possibilities that we must consider for this algorithm are the dimensions that make the top and bottom of the boxes perpendicular to each other
- then we sort the rotation array in a decreasing order from the highest boxes first so that we can iterate through the array to find the largest efficient heights from the stack of boxes

- then we iterate through the list and determine whether the rotation width and depth are perpendicular to each other in order for the boxes to be stacked on top of each other.
- if the width and depth of the iteration of the box is less than the width and depth of all other iterations of the boxes that could fit it, then we know that the rotated box is perpendicular and fits on top of the box underneath it. therefore we find whether the height is greater than all other heights of the possible boxes, and we thus find the tallest stack of boxes

RUNTIME

- Creating the array with all rotations of n boxes takes n computations, then sorting takes nlog(n) computations, and then iterating through all the arr values takes n^2 computations because we iterate each possible combination of boxes that could be determined to have the highest stack height.
- Therefore, the worst time runtime is **O(n^2)**