# Homework 4

CIS-675 DESIGN AND ANALYSIS OF ALGORITHMS

PROF. IMANI PALMER

6/4/2021

Anthony Redamonti
SYRACUSE UNIVERSITY

## Question 1:

You have a set of N dice, where each die *d* has *m* faces numbered 1, ..., *m*. How many ways are there to arrange the dice so that the total sum is equal to some value M? You don't need to write an algorithm here. Just define how to calculate the answer by breaking the problem into smaller subproblems, and the base case.

Consider the first die. When it is 1, the remaining dice must add up to M-1 for their cumulative sum to equal M.

Die 1 = 1 + Dice 2 to N = M-1

The sub-problems occur when evaluating each die from 1 to N.

Die 1 = 1 + Die 2 = 1 + Dice 3 to N = M-2

The base-case occurs when evaluating the last (N'th) die.

Die 1 = 1 + Die 2 = 1 + ... + Die N = M-(N-1)

On the last die, if a face exists such that the cumulative sum equals M, then this sequence of dice is counted among the possible combinations that yield a sum, M. For a given sequence of dice, if the cumulative sum is greater than M and the base case has not been reached, then that sequence is abandoned, and the next appropriate die is incremented. The search will continue until all dice have been evaluated for all faces.

However, in some cases there may not be a need to evaluate all possible combinations. For example, if the first die is being evaluated on a face value greater than M, then the search can stop.

Recursion can be used to implement the algorithm as well as dynamic programming to improve performance.

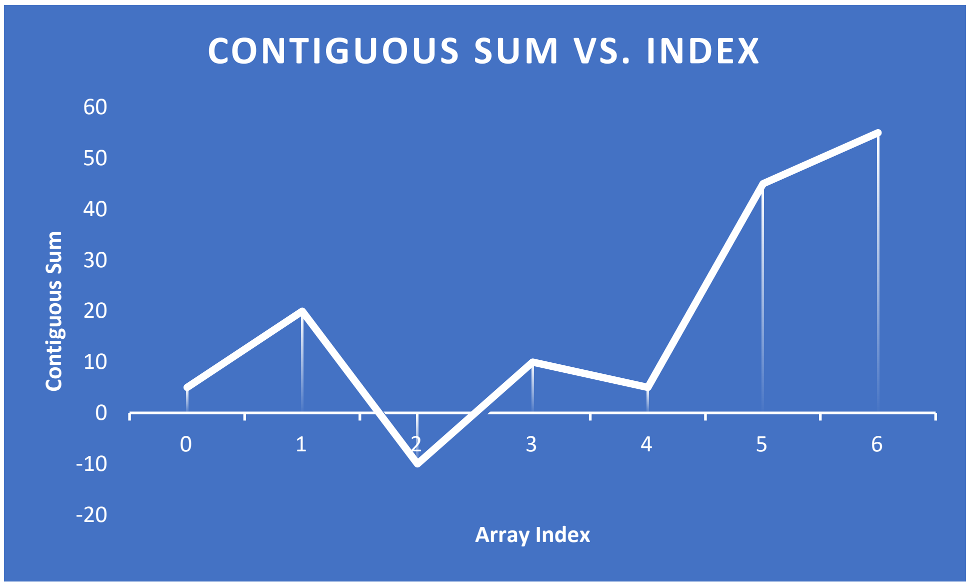
## Question 2:

A contiguous subsequence of a list *S* is a subsequence made up of consecutive elements of *S*. For instance, if *S* is: {5, 15, -30, 10, -5, 40, 10}, then {15, -30, 10} is a contiguous subsequence, but {5, 15, 40} is not. Give a dynamic programming linear-time algorithm for the following task:

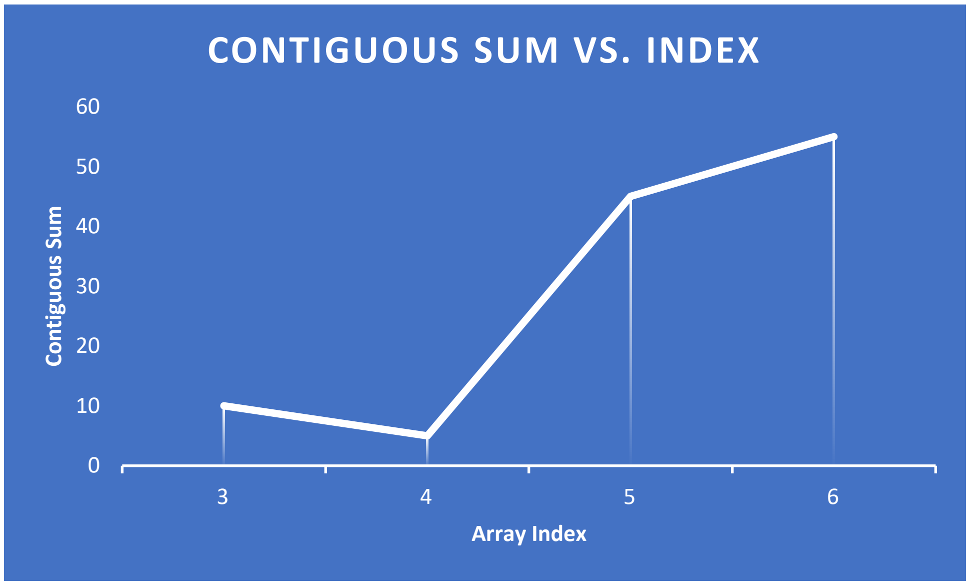
Input: A list of numbers $a^1, a^2, \ldots, a^n$.
Output: The contiguous subsequence of maximum sum (a subsequence of length zero has sum zero)

For the preceding example, the answer would be 10, -5, 40, 10, with a sum of 55. *Hint: For each j ∈(1, 2, …, n), consider contiguous subsequences ending exactly at position j*.

Consider the example sequence: {5, 15, -30, 10, -5, 40, 10}. Below is a plot of the cumulative sum as the array is traversed.



When the contiguous sum is less than zero, begin calculating the contiguous sum after the index where this occurred. In the example above, the sum became negative at index 2, so the sum was reset to zero and recalculated starting at index 3. The contiguous sum starting at index 3 is shown on the next page.

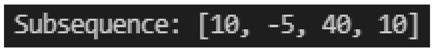


Because the sum does not drop below 0, index 3 must be the start of the contiguous subsequence with the largest sum. Also, index 6 represents the ending of the subsequence because it is at this index where the sum is highest.

The following dynamic linear-time algorithm was implemented in C to find the contiguous subsequence with the largest sum.

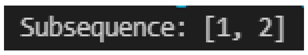
```
#include <stdio.h>
#include <stdlib.h>
#define N 7

//------------------------------------ algorithm --------------------------------
----------------//
// Return the contiguous subsequence with the largest sum.
int* algorithm(int* array){
   int* sub_sequence = (int*)malloc(N*sizeof(int));
   int i;
   int starting_index = 0;
   int next_starting_index = 0; // starting index of next segment
   int ending_index = 0; // index where max sum is highest
   int cumulative_sum = 0;
   int max_sum = 0;

   for(i = 0; i < N; i++){
      cumulative_sum += array[i];
      if(cumulative_sum < 0){
         next_starting_index = i+1; // skip index that makes cumulative sum negat
ive
         cumulative_sum = 0;    // reset the cumulative sum
         continue;
      }
      if(max_sum <= cumulative_sum){ // if the maximum sum is <= cumulative sum,
         starting_index = next_starting_index; // start at the beginning of this
segment
         max_sum = cumulative_sum;   // make the new max sum = cumulative sum.
         ending_index = i; // The ending index of the subsequence.
      }
   }
   int j = 0;
   for(i = starting_index; i <= ending_index; i++){
      sub_sequence[j++] = array[i];
   }
   return(sub_sequence);
}

int main(){
   int i;
   int array[N] = {5, 15, -30, 10, -5, 40, 10};
   int* largest_sum_of_contiguous_sub_sequence = algorithm(array);
   int sub_sequence_size = 2;
   printf("Subsequence: [");
   for(i = 0; i < sub_sequence_size-1; i++){
      printf("%d, ", largest_sum_of_contiguous_sub_sequence[i]);
   }
   printf("%d]\n", largest_sum_of_contiguous_sub_sequence[sub_sequence_size-1]);
}
```

The following input was used to test the algorithm: [5, 15, -30, 10, -5, 40, 10]. The output of the code is below.



```
Subsequence: [10, -5, 40, 10]
```

Other inputs were tested: [1, 2, -5, -5, -5, 1, 1]. The output for this input is below.



```
Subsequence: [1, 2]
```

## Question 3:

Given two strings x = $x_1, x_2, \ldots, x_n$ and y = $y_1, y_2, \ldots, y_n$ we wish to find the length of their longest common substring that is, the largest k for which there are indices i and k with $x_i, x_{i+1}, \ldots, x_{i+k-1} = y_j, y_{j+1}, \ldots, y_{j+k+1}$.

Show how to do this in time O(mn) using dynamic programming.

Consider two strings:

$string1 = "applesauce"$. Let M represent the length of string1 and equal 10.

$string2 = "lesaXX"$. Let N represent the length of string2 and equal 6.

To solve this problem using dynamic programming, implement an NxM array.

Let 'i' and 'j' represent the row and column number respectively of the two-dimensional array. They also represent the length of string1 and string2 respectively.

First, fill in the first row with either 0 or 1. If string1[i] equals string2[j] then array[i][j] = 1. If they are not equal, then array[i][j] = 0.

| | **a** | **p** | **p** | **l** | **e** | **s** | **a** | **u** | **c** | **e** |
|---|---|---|---|---|---|---|---|---|---|---|
| **l** | 0 | 0 | 0 | **1** | 0 | 0 | 0 | 0 | 0 | 0 |
| **e** | | | | | | | | | | |
| **s** | | | | | | | | | | |
| **a** | | | | | | | | | | |
| **X** | | | | | | | | | | |
| **X** | | | | | | | | | | |

Next, fill in the first column with 0 or 1 following the same rules as the previous step.

| | **a** | **p** | **p** | **l** | **e** | **s** | **a** | **u** | **c** | **e** |
|---|---|---|---|---|---|---|---|---|---|---|
| **l** | 0 | 0 | 0 | **1** | 0 | 0 | 0 | 0 | 0 | 0 |
| **e** | 0 | | | | | | | | | |
| **s** | 0 | | | | | | | | | |
| **a** | **1** | | | | | | | | | |
| **X** | 0 | | | | | | | | | |
| **X** | 0 | | | | | | | | | |

Fill in the remaining elements of the array by comparing string1[i] with string2[j]. If they are equal, then array[i][j] = array[i-1][j-1] + 1. If they are not equal, then array[i][j] = 0. Keep track of the maximum value in the array and return it.

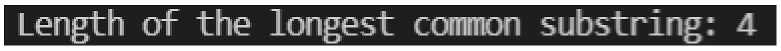
| | **a** | **p** | **p** | **l** | **e** | **s** | **a** | **u** | **c** | **e** |
|---|---|---|---|---|---|---|---|---|---|---|
| **l** | 0 | 0 | 0 | **1** | 0 | 0 | 0 | 0 | 0 | 0 |
| **e** | 0 | 0 | 0 | 0 | **2** | 0 | 0 | 0 | 0 | **1** |
| **s** | 0 | 0 | 0 | 0 | 0 | **3** | 0 | 0 | 0 | 0 |
| **a** | **1** | 0 | 0 | 0 | 0 | 0 | **4** | 0 | 0 | 0 |

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| **X** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **X** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

The following algorithm was implemented in C:

```c
#include <stdio.h>
#include <stdlib.h>
#define N 10
#define M 6

//------------------------------------ algorithm --------------------------------
----------------//
// Return the length of the longest common substring.
int algorithm(char* string1, char* string2){
   int i, j;
   int array[N][M];

   int max_value = 0;

   // fill in 0 or 1 for first row
   for(i = 0; i < 1; i++){
      for(j = 0; j < M; j++){
         if(string1[i] == string2[j]){
            array[i][j] = 1;
            max_value = 1;
         }
         else{
            array[i][j] = 0;
         }
      }
   }

   // fill in 0 or 1 for first column
   for(i = 1; i < N; i++){
      for(j = 0; j < 1; j++){
         if(string1[i] == string2[j]){
            array[i][j] = 1;
            max_value = 1;
         }
         else{
            array[i][j] = 0;
         }
      }
   }
   // fill in rest of array
   for(i = 1; i < N; i++){
      for(j = 1; j < M; j++){
         if(string1[i] == string2[j]){
            array[i][j] = array[i-1][j-1] + 1;
            if(array[i][j] > max_value){
               max_value = array[i][j];
            }
         }
         else{
            array[i][j] = 0;
         }
      }
   }

   return(max_value);
}

int main(){
   char string1[N] = {'a', 'p', 'p', 'l', 'e', 's', 'a', 'u', 'c', 'e'};
   char string2[M] = {'l', 'e', 's', 'a', 'z', 'z'};
   int length = algorithm(string1, string2);
   printf("Length of the longest common substring: %d\n", length);
}
```

The output of the algorithm in below.



```
Length of the longest common substring: 4
```

## Question 4:

Amortized analysis of stacks. There is a stack with three operations: push, pop, and multipop(k). The multipop(k) operation is implemented as a series of k pops. push and pop each take 1 unit of time, and multipop(k) takes k unit of time. Using amortized analysis, we determined that the average cost of an operation, over n operations, was 2 units of time. Suppose that in addition to multipop(k), we want to implement a new operation multipush(k), which is a series of k pushes. Can we modify our amortized analysis to this case? Does amortized analysis help in this case? Why or why not? (Hint: think about what it means for amortized analysis to help. When would we want to use amortized analysis over standard analysis?)

Does amortized analysis help in this case? No, it does not.

Suppose there are only three operations able to be performed on a stack: push, pop, and multipop(k). Performing amortized analysis to compute the average cost of 'k' number of operations would result in O($k^2$) average cost but not tight. The expensive multipop operation can only pop 'k' number of elements off the stock if there are 'k' number of elements on the stack when it is called. Thus, multipop is dependent on push to add elements to the stack.

The amortized analysis would no longer apply if multipush(k) were implemented because it is an independent function in the function set. Introducing this function would allow the user to push k elements onto the stack and pop k elements off the stack using two consecutive function calls.

## Question 5:

Consider the extensible array data structure that we learned in class. Suppose there is no extra cost for allocating memory. Suppose that we want to add another operation to this data structure: remove, which deletes the last element added. In order to make sure that the array doesn't take up too much space, we say that if the array is at least half empty, we will reallocate memory that is only half the size of the current array, and copy all the elements over. This is basically the opposite of the insertion operation from before.

a) Does amortized analysis make sense here? In other words, does it allow us to get a tighter bound than the standard analysis?

Yes, it does help. Because remove() is dependent on append(), it is beneficial to use **amortized analysis** to weigh the cost of a single remove() operation averaged over many remove() operations and achieve a tighter bound than the standard analysis. Adding elements to the extensible array takes O(1) cost. Remove() will take O(1) cost except when half of the array is not in use. In this case, remove() will transfer the 'n' remaining elements into a new array, taking O(n) cost.

b) Instead of reallocating memory if the array becomes half empty, we reallocate/copy if the array becomes at least 3/4 empty (only 1/4 of the array cells are being used). Analyze the running time of a sequence of n operations using amortized analysis. Hint: this is a straightforward modification of the original extensible array analysis. If we shrink an array, how many elements get added before we next double it? If we are deleting elements, how many do we delete before shrinking it?

Suppose append() is used 10 times resulting in the following array:

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | | | | | | |

For the array to resize, remove() would have to be called until its contents were equal to a quarter of its size. Thus, after six remove() operations, the array would resize, and all elements copied to the new array shown below.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | | | | |

Banker's Method:

Let one unit of cost be charged for each append/delete operation. Let another unit of cost be charged for copying elements to a new array. Therefore, each appended item should generate 2 units of cost to pay for these operations. But what if an appended item needs to be copied to a new array a second time, i.e. if the array doubles twice? There would not be enough generated revenue to cover the cost of copying the older elements. Therefore, each appended item should generate 3 units of cost. A constant unit of cost reduces to O(1) cost for a sequence of n operations.