# CSC-421 Applied Algorithms and Structures
# Winter 2021-22

**Instructor:** Iyad Kanj
**Office:** CDM 832
**Phone:** (312) 362-5558
**Email:** `ikanj@cs.depaul.edu`
**Office Hours (Office/Zoom)**: Tuesday and Wednesday 4:00-5:30
**Course Website**: https://d2l.depaul.edu/

## Solution Key to Assignment 3

1. (a) The recursive definition is ($1 \leq j \leq i$):

$$C[i,j] = \begin{cases} 1 & \text{if } j = i \text{ or } j = 1 \\ C[i-1,j] + C[i-1,j-1] & \text{otherwise} \end{cases}$$

   (b) We design a subroutine Rec-C($i$, $j$) that computes the value of $C[i,j]$.

```
Rec-C(i, j)

if (j = i) or (j=1) then
  return(1);
return(Rec-C(i-1, j) + Rec-C(i-1, j-1));
```

   If you draw a recursion-tree for the call Rec-C(6, 4), you see that the algorithm performs overlapping computations.

(c) The values $C[i, j]$ can be computed in a bottom-up fashion using dynamic programming. We use a table C and fill it out starting at the first raw using the recursive definition given above.

```
Combinations(C)

for i = 1 to n do
  for j = 1 to i do
    if (j=1) or (j=i) then
        C[i, j] = 1;
    else C[i, j] = C[i-1, j] + C[i-1, j-1];
```

The elements that we computed in the table form a triangle; this triangle is called Pascal's Triangle (named after the mathematician Blaise Pascal, 1623-1662). The running time of the algorithm is $O(n^2)$. The running time of the recursive algorithm is exponential. The dynamic programming solution is better.

2. (a) If $n = 8$ then the proposed algorithm results in 3 bills of values 6, 1, 1, which is not optimal, as the optimal solution in this case consists of two bills of value 4.

   (b) The observation is that to change an amount $n$, one has to use one of the bills of values $d_1, \ldots, d_k$. Therefore, it costs (the same amount of) 1 bill to go from amount $n$ to each of the amounts $n - d_1, \ldots, n - d_k$. Hence, the optimal solution for $n$ can be expressed as the minimum of the optimal solutions for $n - d_1, \ldots, n - d_k$, plus 1 (accounting for the single bill to go from $n$ to each of $n - d_1, \ldots, n - d_k$). Here is the recursive algorithm:

```
Change-Rec(n)

1. if n =0 then return 0;
2. if n < 0 then return n+1
/* A large value that does not affect the min function */
2. return (1 + min{Change-Rec(n-d_1), ..., Change-Rec(n-d_k));
```

   (c) We can use the same recursive formulation above in a dynamic programing bottom-up approach to compute the minimum number of bills for changing the amount $n$. We use a one-dimensional table/array $T$ to store the solutions to the subproblems. The algorithm is given below.

```
Change-Dynamic(n)
1. Create a table T[0..n] and initialize T[0]=0;
2. for i=1 to n do
     T[i] = 1+ min{T[i-d_1], ..., T[n-d_k]};
     /* if i-d_j is negative we do not consider it
          in the computation of T[i] */
3. return T[n];
```

The running time is $O(k \cdot n)$. If we want to find an optimal change, we can add a field to each entry $i$ to $T$ pointing back to index $j$ based on which $T[i]$ was computed (i.e., corresponding to the minimum value);

3. Define the subproblem $L(i)$, for $i = 1, \ldots, n$, to be the sum of the elements of a largest-sum contiguous subarray of $A$ ending at index $i$. Observe that $L(i) = L(i-1) + A(i)$ if $A(i-1) >= 0$, and $L(i) = A(i)$ is $A(i-1) < 0$. Using a table $L[0..n]$, where $L(0) = 0$, this recursive definition (optimal substructure property) can be used to compute $L(i)$, for $i = 1, \ldots, n$, in time $O(n)$. Afterwards, we go over $L$ and output the maximum $L(i)$. (Note we assume that the desired contiguous subarray is not allowed to empty.) Here is the algorithm:

```
Largest Sum Subarray(A)
1. Create a table L[0..n] and initialize L(0)=0;
2. for i=1 to n do
     if A(i-1) >= 0 then L(i) = L(i-1) +A(i);
     else L(i) = A(i);
3. return the maximum L(i), where i ranges over 0,..., n;
```

4. (a) We observe that if we only allow insertions and deletions (i.e., we do not allow substitutions) in the Edit Distance problem, then the alignment corresponding to the edit distance between $A$ and $B$ corresponds to a longest common subsequence of $A$ and $B$: the sequence of characters in $A$ and $B$ aligned together in such an alignment is a longest common subsequence of $A$ and $B$. More formally, if we denote by $edit - distance_{InDel}(A, B)$ the edit distance between $A$ and $B$ w.r.t. to only insertions and deletions,

and by $lcs(A, B)$ the length of a longest common subsequence of $A$ and $B$, then we can observe that $edit - distance_{InDel}(A, B) = length(A) + length(B) - 2lcs(A, B)$. Therefore, $lcs(A, B)$ can be computed by computing $edit - distance_{InDel}(A, B)$. To compute $edit - distance_{InDel}(A, B)$, we tweak the Edit Distance algorithm so that it only considers insertions and deletions. This can be done by only changing the computation of $Edit(i, j)$ in the algorithm as follows. If $A[i] = B[j]$ then we set $Edit(i, j) = 1 + Edit(i - 1, j - 1)$; else, we set $Edit(i, j) = 1 + \min\{Edit(i - 1, j), Edit(i, j - 1)\}$. The running time of the algorithm remains $O(nm)$.

(b) Observe that a longest palindromic subsequence of a sequence $X$ is a longest common subsequence between $X$ and its reverse seqeuencd $X^R$ (i.e., obtained by reversing $X$). Therefore, we can solve the problem by invoking the algorithm in part (a) on $X$ and $X^R$, which runs in time $O(n^2)$.