- The instructions are the same as in Homework 1, 2, 3 and 4.

There are 5 questions for a total of 88 points.

---

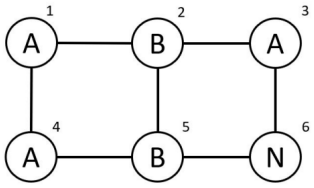
1. (*Abstract Word Search*) Given a directed graph G where each vertex has out-degree at most d, and each vertex v is labelled by a symbol $\ell(v)$, a start vertex v_1, and a sequence of symbols $w_1..w_k$, is there a path $v_1..v_k$ in G starting from v_1, so that $\ell(v_1)\ell(v_2)...\ell(v_k) = w_1...w_k$? *Note that the path might not be a simple path.*

   Here is a recursive back-tracking algorithm that solves *Abstract Word Search*:

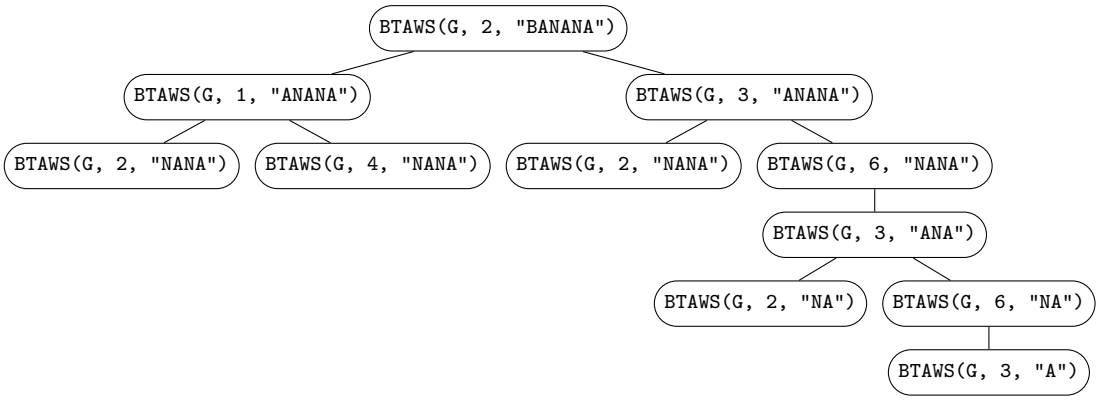
```
BTAWS(G, v, w_1..w_k)
  IF ℓ(v) ≠ w_1 return False
  IF k = 1 return True
  FOR u ∈ N(v) do:
    IF BTAWS(G, u, w_2..w_k)==True THEN return True
  Return False
```

   $N(v)$ in the above pseudocode denotes the neighboring vertices of vertex v (i.e., vertices to which v has an out-going edge). Answer the questions that follow:

   (a) (2 points) Illustrate the tree of recursive calls the above algorithm makes on the input $(G, 2, \text{“}BANANA\text{”})$ (Note: The graph G given below, the starting vertex v is vertex 2, and $w_1..w_6 = BANANA$).



Solution: Assuming that in the adjacency list of the given graph G, all the neighbors, of each vertex, are sorted in increasing order of their vertex numbers, the recursive call tree can be given as:



(b) (2 points) Give an upper bound on the number of total recursive calls the above procedure makes on a general input.

Solution: In the worst case, at every level, each vertex will have exactly d neighbors, and the recursive calls of the first $(d-1)$ neighbors of a vertex will not return *true* in the if condition.

Now we know the depth/height of tree $= k$

And, at each level: Branch Factor or, Fan-out $(f) = d$

$\Rightarrow$ Upper bound on the number of total recursive calls $= O(f^{\text{depth}})$

$\Rightarrow$ Upper bound on the number of total recursive calls $= O(d^k)$

(c) (2 points) Characterize the different subproblems that arise in the BT algorithm above.

Solution: All of the recursive calls that `BTAWS` makes are of the form `BTAWS`$(G, v_i, w_j..w_k)$, where $i \in [1...n]$, n is the number of vertices, $1 \leq j \leq k$ and, $i, j, k \in \mathbb{N}$

(d) (2 points) Define an array or matrix based on the sub-problem characterization of part (c).

Solution: A 2-D boolean array/matrix called $dp[1...n, 1...k]$ can be defined as follows:

$dp[i, j]$ = whether there exists a possible path from $v_i...v_k$ in G, starting from v_i, so that $\ell(v_i)\ \ell(v_{i+1})...\ell(v_k) = w_j...w_k$

(e) (4 points) Compose the base cases of your array and express your array entries recursively. (give a brief justification of why your recursion works.)

Solution: The base case of the array can be adopted from the given Backtracking algorithm, i.e., for the last word w_k and some vertex v_i:

$$\text{return } \begin{cases} true & \text{, if } \ell(v_i) = w_k \\ false & \text{, if } \ell(v_i) \neq w_k \end{cases}$$

Equivalently, we can represent matrix entries as:

For last word, i.e., $j = k$ and, any vertex v_i (for all $i \in [1...n]$):

$$dp[i][k] = \begin{cases} true & \text{, if } \ell(v_i) = w_k \\ false & \text{, if } \ell(v_i) \neq w_k \end{cases}$$

Also, in the sample recursion tree we drew in 1 a), we can see that the value of i sometimes increases, and sometimes decreases but, the value of j always gets incremented. And the solution of any subproblem in a column depends on the solution to subproblems, somewhere in the next column. Hence, we must fill the *dp* table from right to left in terms of columns, but within a column, we can pick any arbitrary order.

Now clearly, we can observe a one-to-one mapping between the subproblems and the entries of the table as:

$$dp[i][j] = \begin{cases} true & \text{, if some neighbour of } i \text{ returns } true \text{ on string } w_{j+1}...w_k \\ false & \text{, otherwise} \end{cases}$$

or,

$$dp[i][j] = \begin{cases} true & \text{, if } \exists\ m \in [1...n],\ dp[m][j+1] = true \wedge (v_i, v_m) \in E \\ false & \text{, otherwise} \end{cases}$$

Hence, the entire correctness of our DP algorithm lies on the correctness of the Backtracking algorithm, which is correct due to the Principle of Mathematical Induction (PMI).

(f) (2 points) Convert the above backtracking algorithm into a dynamic programming algorithm.

Solution:

Input: A graph $G = (V, E)$, where each vertex has out-degree at most d, and each vertex v is labelled by a symbol $\ell(v)$, an integer s denoting start vertex v_1, and a sequence of symbols/string $w_1..w_k$.

Output: A boolean value denoting, if there exist a path $v_1..v_k$ in G starting from v_1, such that $\ell(v_1)\ \ell(v_2)...\ell(v_k) = w_1...w_k$.

Algorithm:

Procedure `DPAWS`$(G, s, w_1...w_k)$:

- Initialize a 2-D boolean array called dp of size $n \times k$ to all $false$.
- for $i = 1, 2, 3, ..., n$:
  - if $(\ell(v_i) == w_k)$:
    - Set $dp[i][k] = true$
  - else:
    - Set $dp[i][k] = false$
- for $j = k-1, k-2, ..., 1$:
  - for $i = 1, 2, 3, ..., n$:
    - if $(\ell(v_i) == w_j)$:
      - for all m in $(v_i, v_m) \in E$:
        - Set $dp[i][j] = dp[i][j]\ ||\ dp[m][j+1]$
    - else:
      - Set $dp[i][j] = false$
- return $dp[s][1]$

Note: In the above algorithm:

→ *= means assignment.*

→ *== means equality comparison operator.*

→ *|| means logical OR operator that returns false iff both its operands are false, else it returns true.*

→ *Indentation, along with hyphen (-), denotes the scope.*

(g) (2 points) Analyze the running time of your dp algorithm

Solution:

Running Time Analysis:

1. Initialization of the 2-D array takes $O(n.k)$ time.
2. Each entry of the table gets filled in $O(d)$ time, where d denotes the number of neighbors of vertex v_i at i^{th} row of the table. And, we have to fill $n.k$ such entries.

Thus,

$$\Rightarrow \text{T.C.} = O\left(n.k.d\right)$$

2. (18 points) (Laying rugs) You have a long corridor in your palace. Unfortunately, there are n unsightly marks on the corridor floor, at $x_1, ..., x_n$ yards from the door. You will use some costly rugs whose width fills the corridor and are each one yard long to cover the marks. For simplicity, we will allow rugs to overlap and part of the rug to extend past the end of the corridor. We identify rugs with intervals of length 1, and it hides the mark at x_i if x_i is in the interval. Unfortunately, we have a fixed number of rugs R, and the goal is to use them to hide as many marks as possible. Design an algorithm to minimize the number of unhidden marks, given $x_1, ..., x_n$, and R as inputs. Your algorithm only needs to output the minimum number of unhidden marks possible.

Solution: In the below solution, I am safely assuming that indices start from 0, without any loss of generality.

Description of sub-problems:
The subproblems can be represented by a 2-D array called dp, of size $(R+1) \times (n+2)$, such that:

$\therefore dp[i,j]$ = minimum number of unhidden marks left after using atmost i rugs and considering $(n-j+1)$ marks on the corridor floor, at $x_j ... x_n$, yards from the door.

Base Case(s):
If we have 0 rugs left then, we obviously can't cover any mark, so the number of unhidden marks is equal to the size of array x, into consideration.

$\therefore dp[0][j] = (n-j+1)$, $1 \leq j \leq n+1, \forall j \in \mathbb{N}$

Similarly, whenever, we consider any $x[j...n]$, $j > n$, we need not require any rug as there is no such valid mark on the corridor floor, so we will set $dp[i][n+1] =, 0 \leq i \leq R, \forall i \in \mathbb{N}$. Note that this will be taken care of automatically when we initialize all array entries of dp to 0.

Recursion with justification:
The other entries can be recursively represented as:

$$dp[i][j] = \begin{cases} 0 & \text{, if } i \geq (n-j+1) \\ \min\left(1 + dp[i][j+1], dp[i-1][k]\right); x[k] <= x[j]+1, \forall j \leq k \leq n, & \text{, otherwise} \end{cases}$$

The correctness of the above recursive relation is justified by the equivalent Backtracking algorithm for the above problem which states that the minimum number of unhidden marks is equal to 0 if the number of rugs available is equal to or, more than the number of marks into consideration. Else, it is equal to the minimum number of unhidden marks if we don't cover the current mark and use the same number of rugs to cover the remaining marks or, if we use the current rug to cover the current mark at x_i and thus some subsequent marks within distance $(x_i + 1)$, and use remaining rugs to cover remaining marks. And this is exactly what our DP algorithm simulates. Hence, the recursion is justified.

Order in which sub-problems are solved:
Since, we can observe that a solution to a subproblem $dp[i][j]$ depends on the solution of the sub-problem, either in the subsequent column and the same row, or in the subsequent column and some row above it.

Hence, the correct order of solving the subproblems is across the columns - higher column number to lower column number (right-to-left) but, within a column, any arbitrary order can be followed. *(I have considered from bottom-to-top, within a column).*

Form of output:
The output rests at position $dp[R][1]$, which is an integer, denoting the minimum number of unhidden marks possible using atmost R rugs and considering marks on the corridor floor, at $x_1, ..., x_n$ yards from the door.

Pseudocode:

Procedure `DPLR`$(x[1...n], R)$:

- Sort the given array x.
- Initialize a 2-D integer array called dp of size $(R+1) \times (n+2)$ to all 0's.
- for $j = 1, 2, 3, ..., n+1$:
  - Set $dp[0][j] = n - j + 1$
- for $j = n, n-1, n-2, ..., 1$:
  - for $i = R, R-1, R-2, ..., 1$:
    - if $(i \geq n - j + 1)$:
      - Set $dp[i][j] = 0$
    - else:
      - Initalize an integer variable exc to $1 + dp[i][j+1]$.
      - Declare an integer variable k.
      - k = `BinSearch(x, j, n, x[j]+1.0)`
      - Initalize an integer variable inc to $dp[i-1][k]$.
      - Set $dp[i][j] = \min(exc, inc)$
- return $dp[R][1]$

`BinSearch`$(arr, low, high, key)$:

- while $(low \leq high)$:
  - Initialize a variable $mid = low + \lfloor (high - low)/2 \rfloor$
  - if $(arr[mid] > key)$:
    - if $(mid - 1 \geq low)$:
      - if $(arr[mid-1] \leq key)$:
        - return mid
      - else:
        - Set $high = mid - 1$
    - else:
      - return mid
  - else:
    - Set $low = mid + 1$
- return -1

Note: In the above pseudocode:

→ *= means assignment.*

→ *&& means logical AND operator which returns true iff both the operands are true else, it returns false.*

→ *min is a function that returns the minimum of two values in $O(1)$ time, using if-else statements.*

→ *Indentation, along with hyphen (-), denotes the scope.*

Runtime analysis:

1. Sorting the array x of size n takes $O(n.logn)$ time.
2. Initializing a 2-D array of size $(R+1) \times (n+2)$ to all 0's takes $O(n.R)$ time.
3. We have to fill $O(n.R)$ table entries where each table entry invokes a call to the Binary Search function which takes in worst case $O(logn)$ time.

Thus,

$$\Rightarrow \text{T.C.} = O\left(n. \log n + n.R + n.R. \log n\right)$$
$$\Rightarrow \text{T.C.} = O\left(n.R. \log n\right)$$

3. (18 points) (Palindromic subsequence) A subsequence is *palindromic* if it is the same whether read left to right or right to left. For instance, the sequence

$$A, C, G, T, G, T, C, A, A, A, A, T, C, G$$

has many palindromic subsequences, including A, C, G, C, A and A, A, A, A (on the other hand, the subsequence A, C, T is *not* palindromic). Devise an algorithm that takes a sequence $x[1 \ldots n]$ and returns the (length of the) longest palindromic subsequence. Its running time should be $O(n^2)$.

Solution: In the below solution, I am safely assuming that indices start from 0, without any loss of generality.

Description of sub-problems:
The subproblems can be represented by a 2-D array called dp, of size $(n+1) \times (n+1)$, such that:

$\therefore$ $dp[i, j]$ = length of longest common subsequence between $x[i...n]$ and $y[j...n]$, where y is the reverse sequence of x.

Base Case(s):
When either of the sequence is empty, i.e., either x is empty or, y is empty, the corresponding array entries are also 0.
Hence,

$dp[i][0] = 0$ and, $dp[0][j] = 0$, $\forall i \in [1...n+1]$, $j \in [1...n+1]$

Recursion with justification:
The other entries can be recursively represented as:

$$dp[i][j] = \begin{cases} 1 + dp[i-1][j-1] & \text{, if } s1[i-1] == s2[j-1] \\ \max(dp[i-1][j], dp[i][j-1]) & \text{, otherwise} \end{cases}$$

Finding the longest palindromic subsequence in a sequence is equal to finding the longest common subsequence between a sequence, and the sequence formed by reversing the elements of the original sequence. Hence, the above question reduces to finding the longest common subsequence between a sequence and its reverse. Thus, the recursion is justified by the recursive relation of the LCS problem.

Order in which sub-problems are solved:
Since, we can observe that a solution to a subproblem $dp[i][j]$ depends on the solution of the sub-problem, either in the previous column and the same row, or in the previous row and the same column, or, in the previous column and previous row (diagonally).

Hence, the correct order of solving the subproblems is across the columns - lower column number to higher column number (left-to-right) but, within a column, top-to-bottom must be followed..

Form of output:
The output rests at position $dp[n+1][n+1]$, which is an integer, denoting the length of the longest palindromic subsequence, in the given sequence $x[1...n]$.

Pseudocode:

Procedure `DPLPS`($x[1...n]$):

- Initialize a new sequence $y[1...n]$ equal to $x[1...n]$.

- Reverse the sequence $y[1...n]$.
- Declare a 2-D integer array called dp of size $(n+1) \times (n+1)$.
- for $i = 1, 2, 3, ..., n+1$:
  - Set $dp[i][1] = 0$
- for $j = 1, 2, 3, ..., n+1$:
  - Set $dp[1][j] = 0$
- for $j = 2, 3, ..., n+1$:
  - for $i = 2, 3, ..., n+1$:
    - if $(x[i-1] == y[j-1])$:
      - Set $dp[i][j] = 1 + dp[i-1][j-1]$
    - else:
      - Set $dp[i][j] = \max(dp[i-1][j], dp[i][j-1])$
- return $dp[n+1][n+1]$

Note: *In the above pseudocode:*

→ $=$ *means assignment.*

→ $==$ *means equality comparison operator.*

→ *max is a function that returns the maximum of two values in* $O(1)$ *time, using if-else statements.*

→ *Indentation, along with hyphen (-), denotes the scope.*

Runtime analysis:

1. Initalizing a new subsequence $y[1...n]$ to reverse of $x[1...n]$ requires a single traversal of x and thus, takes $O(n)$ time.
2. Each entry of the table gets filled in $O(1)$ time as we are filling the rows from top-to-bottom and columns left-to-right in such a way that the required entries of the array gets computed correctly. And, we have to fill $O(n^2)$ such entries.

Thus,

$$\Rightarrow \text{T.C.} = O\left(n + n^2\right)$$
$$\Rightarrow \text{T.C.} = O\left(n^2\right)$$

4. (18 points) (Shipping boxes) You are a manager at a shipping company. On a particular day, you must ship n boxes with weights $w[1...n]$ that are positive integers between 1 and 100. You have three trucks, each with a weight limit of D, a positive integer. Design an algorithm to determine if it is possible to pack all the n boxes into the three trucks such that for every truck, the total weight of boxes in the truck is at most D (i.e., the weight limit is not exceeded). Your algorithm should output "yes" if it is possible to pack and "no" otherwise.

Solution: In the below solution, I am safely assuming that indices start from 0, without any loss of generality.

Description of sub-problems:
The sub-problems can be represented by a 4-D array called dp, of size $n \times D \times D \times D$, such that:

$\therefore dp[i, a, b, c]$ = whether it is possible to pack n boxes with weights $w[i...n]$ can fit into either $truck_1$ with weight limit a or, $truck_2$ with weight limit b or, $truck_3$ with weight limit c.

Base Case(s):

- First base case:
  if $((a \leq 0)||(b \leq 0)||(c \leq 0))$, we will return $dp[i][a][b][c] = 0$
- Second base case:
  if $(i == (n + 1))$, we will return $dp[i][a][b][c] = 1$

Recursion with justification:
The other entries can be recursively represented as:

dp[i][a][b][c] = `DPSB`(w, n, i+1, a - w[i], b, c) || `DPSB`(w, n, i+1, a, b - w[i], c) || `DPSB`(w, n, i+1, a , b , c - w[i])

The above recursion is justified as it simulates the backtracking algorithm which states that for every weight, try to fit it in one of the three trucks. If it is possible to fit it in $truck_1$ or, $truck_2$ or, $truck_3$, satisfying the weight limit left of that truck, then return true (1) else return false (0), which is exactly what our recursive relation states.

Order in which sub-problems are solved:
To solve dp[i][a][b][c] we need to recursively first solve $dp[i + 1....n][a'][b'][c']$, where $a' < a, b' < b$ and $c' < c$.

Form of output:
The output rests at position $dp[1][D][D][D]$, which is an integer, ultimately leading to our algorithm to output "yes" if it is possible to pack all the n boxes into the three trucks such that for every truck, the total weight of boxes in the truck is at most D, else it outputs "no".

Pseudocode:

Procedure `main`$(w[1...n], D)$:

- Initialize a 4-D integer array called dp of size $(n+1) \times (D+1) \times (D+1) \times (D+1)$ to all -1's.
- if (`DPSB`$(w[1...n], 1, a, b, c) == true$):
  - Output "yes"

- else:
  - Output "no"

Procedure DPSB($w[1...n], i, a, b, c$):

- if ($dp[i][a][b][c] != -1$):
  - return $dp[i][a][b][c]$
- if ($a < 0 || b < 0 || c < 0$):
  - return $dp[i][a][b][c] = 0$
- if ($i == n + 1$):
  - return $dp[i][a][b][c] = 1$
- return dp[i][a][b][c] = DPSB($w, n, i+1, a-w[i], b, c$) || DPSB($w, n, i+1, a, b-w[i], c$) || DPSB($w, n, i+1, a, b, c-w[i]$)

Note: In the above pseudocode:

→ *= means assignment.*

→ *== means equality comparison operator.*

→ *!= means equality comparison operator.*

→ *|| means logical OR operator that returns false iff both its operands are false, else it returns true.*

→ *Indentation, along with hyphen (-), denotes the scope.*

Runtime analysis:

1. Initializing a 4-D array of size $(n+1) \times (D+1) \times (D+1) \times (D+1)$ to all -1's takes $O(n.D^3)$ time.
2. Each entry of the table gets filled in $O(1)$ time as we are filling the entries of the table in such a way that the required entries of the array get computed correctly. And, we have to fill $O(n.D^3)$ such entries.

Thus,

$$\Rightarrow \text{T.C.} = O\left(n.D^3\right)$$

5. (18 points) (Testing centers) A town has n residents labeled $1, ..., n$. All n residents have houses along a single road. The town authorities suspect a virus outbreak and would like to set up k testing centers along this road. They want to set up these k testing centers in locations that minimize the sum of squared distance that all the residents need to travel from their house to their nearest testing center. Moreover, the centers will be opened in residents' houses to reduce the operating cost. None of the residents has any objection if a center is opened in their house. You have been asked to design an algorithm for finding the optimal locations of the k testing centers.

   Since all residents live along a single road, the house location of a resident can be identified by the distance along the road from a single reference point (which can be thought of as the town's starting point). As input, you are given integer n, integer k, and the house location of the residents in an integer array $A[1...n]$ where $A[i]$ denotes the house location of resident i. Moreover, $A[1] \leq A[2] \leq A[3] \leq ... \leq A[n]$. Your algorithm should aim to find an integer array $C[1...k]$ of house numbers such that the following quantity gets minimized:

$$\sum_{i=1}^{n} D(i), \text{ where } D(i) = \min_{j \in \{1,...,k\}} (A[i] - A[C[j]])^2$$

   Note that $D(i)$ denotes the squared distance resident i has to travel to get to the nearest testing center out of centers at houses $C[1], ..., C[k]$.

   (*For example, consider $k = 2$ and $A = [1, 2, 3, 7, 8, 9]$. A solution for this case is $C = [2, 5]$. Note that for testing centers at locations 2 and 8, the total squared distance travelled by residents will be $(1 + 0 + 1 + 1 + 0 + 1) = 4$.*)

   Design a DP algorithm for this problem that outputs the minimum achievable value of the squared distance.

Solution: In the below solution, I am safely assuming that indices start from 0, without any loss of generality.

Description of sub-problems:

cost[i][j] = minimum possible total distance travelled from resident at i^{th} index to resident at j^{th} index by putting the etsting center at median position among A[i:j]
The sub-problem is defined as follows: DPTC($A[1...n], k, 1, dp$) = minimum possible distance value by placing k testing centers from nth resident to kth resident, i.e., last location.

Base Case(s):

if ($k == 0 \&\& i == n$), we return 0 as we have used all the testing centers to cover all residents.
if ($k == 0 || i == n$), we return ∞ as either we have covered all with less than k testing centers, or we have used all the testing centers but didn't cover all the residents.

Recursion with justification:

DPTC(A, k, j, dp) = $min_{i \leq j \leq n}$(DPTC(A, k-1, j+1, dp)+$cost[i][j]$)

We try locating a testing center to k group of consecutive residents [i.....j]. We obtain a valid solution if we can distribute K testing centers to all n residents such that we obtain k groups. Among all the solutions we choose which is the minimum one.
In our recursive code we compute the minimum among distances if every place are testing centre between i and j and then recursively find the minimum distance if we need to place $(K-1)$ centres from $(j+1)^{th}$ location. Also $cost[i][j]$ return the minimum possible distances for placing testing

centres between i^{th} to j^{th} index location and we always place the centre at median because then the resident at left are equally distance to the resident at right. Hence,we place the centre at medium to get minimum squared distance possible.

Order in which sub-problems are solved:
The order in which the sub-problems are solved is from higher column number to lower column number and rows from bottom to top.

Form of output:
The output rests at position $dp[k][1]$,which is an integer denoting the minimum achievable value of the squared distance .

Pseudocode:

Procedure `main`$(A[1...n], k)$:

- Declare a 2-D integer array called *cost* of size $n \times n$.
- for i = 1, 2, 3,...., n:
  - for j = 1, 2, 3,...., n:
    - Initialize an integer variable called *dist* to $A[i + \lfloor (j-i)/2 \rfloor]$
    - for $k = i, i+1, i+2, ..., j$:
    - if $(dist > A[x])$:
      - Set $cost[i][j] = (cost[i][j] + dist - A[x])^2$
    - else:
      - Set $cost[i][j] = (cost[i][j] + dist - A[x])^2$
- Initialize a 2-D integer array called *dp* of size $k \times n$ to all -1's.
- return `DPTC`$(A[1...n], k, 1, dp)$

Procedure `DPTC`$(A[1...n], k, 1, dp)$:

- if $(k == 0 \&\& i == n)$:
  - return 0
- if $(k == 0 || i == n)$:
  - return ∞
- if $(dp[k][i] != -1)$:
  - return $dp[k][i]$
- Initialize a variable called res to ∞.
- for $j = i, i+1, i+2, ..., n$:
  - result = min(result, cost[i][j], `DPTC`$(A, k-1, j+1, dp)$)
- return $dp[k][i] = result$

Note: In the above pseudocode:

→ *= means assignment.*

→ *== means equality comparison operator.*

→ *!= means equality comparison operator.*

→ *|| means logical OR operator that returns false iff both its operands are false, else it returns true.*

→ *&& means logical AND operator which returns true iff both the operands are true else, it returns false.*

→ *Indentation, along with hyphen (-), denotes the scope.*

Runtime analysis:

1. Initializing two 2-D arrays of size $n \times n$ and $n \times k$ takes $O(n(n+k))$ time.
2. The time taken to compute *cost* matrix is $O(n^3)$.
3. The time taken to compute *dp* matrix is $O(n^2.k)$.

Thus,

$$\Rightarrow \text{T.C.} = O\left(n^2 + n.k + n^3 + n^2.k\right)$$
$$\Rightarrow \text{T.C.} = O\left(n^2(n+k)\right)$$