Missouri University of Science & Technology — Department of Computer Science

**Spring 2023** — **CS 2500: Algorithms**

# Homework 3: Advanced Design Techniques

**Instructor:** *Sid Nadendla* — **Due:** *April 10, 2023*

## Problem 1: Task Selection — *25 points*

Consider a total of $n$ jobs. Let the $i^{th}$ job be designated with a start time $s_i$, a finish time $f_i$ and a net value $v_i$. Two jobs are said to be *compatible* if they do not overlap in time. Your goal is to find the subset of mutually compatible jobs that have the maximum total value. Present the following four stages of your design approach to this problem:

1. Model the above problem as a multi-stage decision problem, identify the state and decision variables, define the state transitions and derive the Bellman equation. **(3+3+4 points)**
2. Using the Bellman equation, write a pseudocode to compute the optimal solution using dynamic programming approach. **(5 points)**
3. Write down the pseudocode for the greedy solution to this problem. **(5 points)**
4. Implement in Python, both the dynamic programming and greedy solutions to this problem and compare the value of the solutions returned for random inputs when there are a total of $n = 10$ jobs. **(5 points)**

**SOLUTION:**

Consider a set $S_N$ with $N$ tasks, where the $i^{th}$ task has a start time $s_i$, finish time $f_i$ and value $v_i$ if scheduled. For the sake of simplicity, assume that these tasks are labeled such that their finish times are sorted in an increasing order. In other words, we have $f_1 \leq f_2 \leq \cdots \leq f_n$.

**Note:** Depending on how one models the multi-stage decision problem, there could be more than one dynamic programming algorithm to the same problem. This is just one approach.

1. One way to model this problem as a multi-stage decision problem is to **define $i^{th}$ stage as a problem to selecting the $i^{th}$ task**.

   Note that the index $i$ is sufficient to identify all the compatible tasks at any given stage in the stated multi-stage decision problem model. Therefore, the **state** of the problem at the $i^{th}$ stage can be denoted by $i$ (a scalar value), and the **decision** made in this stage is $x_i = 0$ (task $i$ is not selected), or $x_i = 1$ (task $i$ is selected).

   Let $V[i]$ denote the maximum value that can be obtained by optimally selecting amongst the first $i$ tasks. Note that such a stage can originate from two possibilities: (i) $x_i = 0$, or (ii) $x_i = 1$. Note that if the optimal sequence of task decisions that led to state $i$ included the choice $x_i = 0$, then we have $V[i] = V[i-1]$ since the same sequence of choices would also have resulted in the maximum value at state $i-1$. On the other hand, if $V[i]$ is obtained with $x_i = 1$, then $V[i] = V[j_{comp}(i)] + v_i$, where $V[j_{comp}(i)]$ is the optimal value obtained by solving the task selection problem on the first $j_{comp}(i)$ tasks that are all compatible with the $i^{th}$ task. Specifically, $j_{comp}(i)$ can be found by identifying the largest task-index $j \in \{1, \cdots, i\}$ such that $f_j \leq s_i$. In other words, the **Bellman recursion** is given as

$$V[i] = \max \begin{cases} V[i-1], & \text{if } x_i = 0 \\ V[j_{comp}(i)] + v_i, & \text{if } x_i = 1. \end{cases} \tag{1}$$

A more formal way to write the same equation is as follows:

$$V[i] = \max_{x_i \in \{0,1\}} \left[ (1 - x_i) \cdot V[i-1] + x_i \cdot \Big( V[j_{comp}(i)] + v_i \Big) \right]. \tag{2}$$

2. The pseudocode for the above algorithm can be written as follows:

```
TASKSELECT-DP($s, f, v$)
$N = s.length$
if $f.length = N$ and $v.length = N$
    $V[1] = v[1]$
    for $i = 2$ to $N$
        Initialize $V[i] = V[i-1]$
        $j_{comp}[i] = \max\{j \in \{1, \cdots, i\} \mid f[j] \leq s[i]\}$
        if $V[i] < V[j_{comp}[i]] + v[i]$
            $V[i] = V[j_{comp}[i]] + v[i]$
    return $V[n]$
else
    Print 'Dimension Mismatch Error'
```

3. A simple greedy algorithm is as follows:

   - Sort all the tasks in an increasing order of normalized value, i.e. (value per unit time taken to complete the task).
   - In an iterative fashion according to the sorted order of tasks, pick the task if it is compatible with all the previously chosen tasks. If it is incompatible, ignore it.
   - Upon exhausting all the $n$ tasks, return the total value of selected tasks.

   Formally, the above algorithm can be written as a pseudocode as follows:

```
TaskSelect-Greedy($s, f, v$)
 1  $N = s.length$
 2  if $f.length = N$ and $v.length = N$
 3      Sort all tasks in the increasing order of $\frac{v[i]}{f[i] - s[i]}$.
 4      $U[1] = v[1]$
 5      for $i = 2$ to $N$
 6          if $i$ is compatible with $\{1, \cdots, i-1\}$
 7              $U[i] = U[i-1] + v[i]$
 8          else
 9              $U[i] = U[i-1]$
10      return $U[n]$
11  else
12      Print 'Dimension Mismatch Error'
```

## Problem 2: String Edit Problem *25 points*

The *string edit* problem is to find the cheapest way to modify two strings so that they are the same. The permitted operations are *deletions*, *insertions* and *substitutions*.

Let the two strings be denoted as $a_1 a_2 \cdots a_m$ and $b_1 b_2 \cdots b_n$, where each $a_i$ and each $b_j$ are characters in the set $S$. If $s_i$ and $s_j$ are any two characters in $S$, let

- the cost of deleting $s_i$ = $D_i > 0$
- the cost of inserting $s_i$ = $I_i > 0$
- the cost of substituting $s_i$ with $s_j$ = $C_{ij} \geq 0$.

Assume $C_{ij} = C_{ji}$ for all $i, j$ and $C_{ij} = 0$ if and only if $i = j$.

Then, present the following four stages of your design approach to this problem:

1. Model the above problem as a multi-stage decision problem, identify the state and decision variables, define the state transitions and derive the Bellman equation. **(3+3+4 points)**
2. Using the Bellman equation, write a pseudocode to compute the optimal solution using dynamic programming approach. **(5 points)**
3. Write down the pseudocode for the greedy solution to this problem. **(5 points)**
4. Implement in Python, both the dynamic programming and greedy solutions to this problem and compare the value of the solutions returned for random pairs of strings. **(5 points)**

**SOLUTION:**

1. Assume that the string-edit problem is solved in multiple-stages where, in each stage, the last symbol in the first sequence is modified using one of the three permitted operations, namely *deletion*, *insertion*, or *substitution*. At an arbitrary stage, we would typically have the following subproblem to be solved:

$$\begin{bmatrix} a_1\, a_2\, \cdots\, a_i \\ b_1\, b_2\, \cdots\, b_j \end{bmatrix}$$

Therefore, the **state** of the above arbitrary stage can be represented as $(i, j)$. The **decision** variable at any stage can be modeled as

$$x[i,j] = \begin{cases} -1, & \text{if } a_i \text{ is deleted} \\ 0, & \text{if } b_j \text{ is inserted as } a_{i+1}. \\ 1, & \text{if } a_i \text{ is substituted by } b_j. \end{cases}$$

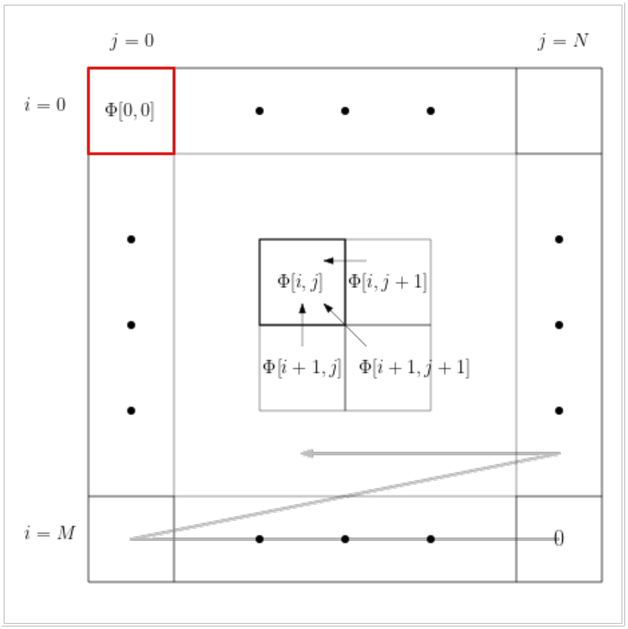
Note that the $(i, j)^{th}$ stage is manifested due to one of the three following decisions made in the prior stages:

$$\begin{tikzcd}
\begin{bmatrix} a_1\ a_2\ \cdots\ a_i\ a_{i+1} \\ b_1\ b_2\ \cdots\ b_j \end{bmatrix} \arrow[dr, "\text{Delete } a_{i+1}"] & \\
\begin{bmatrix} a_1\ a_2\ \cdots\ a_i \\ b_1\ b_2\ \cdots\ b_j\ b_{j+1} \end{bmatrix} \arrow[r, "\text{Insert } b_{j+1}"] & \begin{bmatrix} a_1\ a_2\ \cdots\ a_i \\ b_1\ b_2\ \cdots\ b_j \end{bmatrix} \\
\begin{bmatrix} a_1\ a_2\ \cdots\ a_i\ a_{i+1} \\ b_1\ b_2\ \cdots\ b_j\ b_{j+1} \end{bmatrix} \arrow[ur, "\text{Substitute } a_{i+1} \text{ with } b_{j+1}"'] &
\end{tikzcd}$$

The "*Delete* $a_{i+1}$" decision in the $(i + 1, j)^{th}$ stage leaves us with a residual problem of solving the problem in $(i, j)^{th}$ stage. Similarly, the "*Insert* $b_{j+1}$" decision in the $(i, j + 1)^{th}$ stage matches $a_{i+1}$ with $b_{j+1}$, thus leaving us with a residual problem of solving the problem in $(i, j)^{th}$ stage. Finally, the "*Substitute* $a_{i+1}$ with $b_{j+1}$" decision in the $(i + 1, j + 1)^{th}$ stage matches $a_{i+1}$ with $b_{j+1}$, thus leaving us with a residual problem of solving the problem in $(i, j)^{th}$ stage. Note that one of these decisions will be result in the minimum cost. Therefore, if $\Phi[i, j]$ denotes the minimum cost to reach the $(i, j)^{th}$ stage, then the corresponding **Bellman recursion** will be given by

$$\Phi[i,j] = \min \begin{cases} \Phi[i+1,j] + D[a[i+1]], & \text{if } x[i+1,j] = -1, \\ \Phi[i,j+1] + I[b[j+1]], & \text{if } x[i+1,j] = -1, \\ \Phi[i+1,j+1] + C[a[i+1], b[j+1]], & \text{if } x[i+1,j] = -1. \end{cases}$$

2. Since the Bellman recursion shows that $\Phi[i,j]$ depends on $\Phi[i+1,j]$, $\Phi[i,j+1]$ and $\Phi[i+1,j+1]$, we design the recursion as shown in the figure below:



Note that the target is to reach $\Phi[0,0]$, since, upon matching the two strings $\boldsymbol{a}$ and $\boldsymbol{b}$, we are left with a subproblem that contains zero characters in $\boldsymbol{a}$ and $\boldsymbol{b}$ that need to be matched.

Therefore, the pseudocode for the dynamic programming algorithm to solve the String Edit problem is as follows:

```
STRINGEDIT-DP$(a, b, D, I, C)$
1   $\Phi = $ ZEROS$(M+1, N+1)$ // Initialize $\Phi$ as a $(M+1) \times (N+1)$ all-zero matrix.
2   for $i = M$ to 0
3       for $j = N$ to 0
4           if $i = M$ and $j \neq N$
5               $\Phi[M,j] = \Phi[N,j+1] + I[b(j+1)]$
6           else if $i \neq M$ and $j = N$
7               $\Phi[i,N] = \Phi[i+1,N] + D[a(i+1)]$
8           else
9               $\Phi[i,j] = \min \begin{cases} \Phi[i+1,j] + D[a(i+1)], \\ \Phi[i,j+1] + I[b(j+1)], \\ \Phi[i+1,j+1] + C[a[i+1], b[j+1]] \end{cases}$
10  return $\Phi[0,0]$
```

3. A simple greedy algorithm is as follows:

```
STRINGEDIT-GREEDY$(a, b, D, I, C)$
1  $K = \max\{a.length, b.length\}$
2  for $k = 1$ to $K$
3      if $k \leq a.length$ and $k \leq b.length$
4          $\Psi[k] = \Phi[k-1] + \min \begin{cases} D[a(k)], \\ I[b(k)], \\ C[a[k], b[k]] \end{cases}$
5      else if $k > b.length$
6          $\Psi[k] = \Phi[k-1] + D[a[k]]$
7      else
8          $\Psi[k] = \Phi[k-1] + I[b[k]]$
9  return $\Psi[K]$
```

## Problem 3: Palindromic Subsequences *25 points*

A subsequence is *palindromic* if it is the same whether read from left to right, or right to left. For instance, the sequence $ACGTGTCAAAATCG$ has many palindromic subsequences, including $ACGCA$ and $AAAA$. On the other hand, $ACT$ is a subsequence that is not palindromic.

Devise an algorithm that takes a sequence $x[1:n]$ of length $n$, and returns the length of the longest palindromic subsequence. **(5+5+10+5 points)**

*Note:* Its run time should be $O(n^2)$.

**SOLUTION:** Given a sequence $\boldsymbol{a}$ of length $N$, consider a subsequence starting from the left index $i$ and ending at right index $j$. In other words, the **state** of the subproblem of finding the longest palindromic subsequence of $\boldsymbol{a}[i:j]$ is given by $(i, j)$. The **decision** at each stage is whether or not to append a new character on both sides to form a palindromic subsequence.

Let the length of the longest palindromic subsequence of $\boldsymbol{a}[i:j]$ be denoted as $L[i, j]$. Note that, if $i = j$, then there is only one character in the subsequence. Therefore, there is only one palindromic subsequence, which consists of the character itself, , i.e. $L[i, j] = 1$.

On the other hand, if $i < j$, then the **Bellman recursion** contains the following three cases:

(i) If $\boldsymbol{a}[i] = \boldsymbol{a}[j]$, then

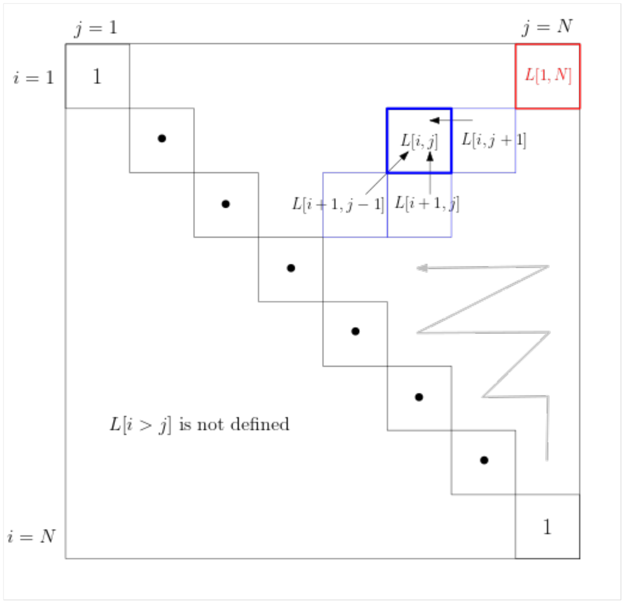
$$L[i,j] = \begin{cases} 2, & \text{if } j = i+1, \quad \text{(i.e. two identical characters in } \boldsymbol{a}[i:j]) \\ 2 + L[i+1, j-1], & \text{otherwise.} \end{cases}$$

Note that the subsequence $\boldsymbol{a}[i+1:j-1]$ is the sequence we obtain upon removing the first and last characters in $\boldsymbol{a}[i:j]$. Therefore, we have the above recursion, whenever $j > i+1$.

(ii) If $\boldsymbol{a}[i] \neq \boldsymbol{a}[j]$, then

$$L[i,j] = \max\Big\{L[i+1,j],\ L[i,j+1]\Big\}.$$

Based on the above Bellman recursion, the following recursion can be visualized in the computation of $L$ matrix.



Therefore, we have the following **pseudocode**[1].

```
PALINDROME-DP($\boldsymbol{a}$)
1   $N = \boldsymbol{a}.length$
2   $L =$ ZEROS$(N, N+1)$ // Initialize $L$ as a $N \times (N+1)$ all-zero matrix.
3   for $i = N-1$ to 1
4       for $j = N$ to $i+1$
5           if $\boldsymbol{a}[i] = \boldsymbol{a}[j]$
6               if $j = i+1$
7                   $L[i,j] = 2$
8               else
9                   $L[i,j] = 2 + L[i+1,j-1]$
10          else
11              $L[i,j] = \max\{L[i+1,j],\ L[i,j+1]\}$
12  return $L[1,N]$
```

[1]Note that an extra column of $j = N+1$ is included in the matrix as a dummy term so that the recursion goes smoothly, although it is not used in the recursion.

## Problem 4: Polygon Triangulation *25 points*

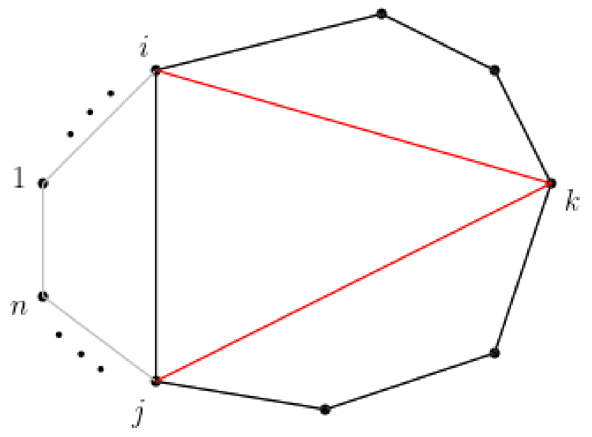
Consider a convex polygon $\mathcal{P}$ on $n$ vertices in a plane. A *triangulation* of $\mathcal{P}$ is a collection of $n-3$ diagonals of $\mathcal{P}$ such that no two diagonals intersect (except possibly at their end-points). Notice that a triangulation splits the polygon's interior into $n-2$ disjoint triangles. Let the cost of triangulation be the sum of the lengths of the diagonals in it.

Then, give the pseudocode for the optimal algorithm for finding the triangulation that minimizes the cost. **(5+5+10+5 points)**

**SOLUTION:**

Let the coordinates of $n$ vertices of the convex polygon[2] be denoted as $\boldsymbol{x}$. Assume that the vertices are ordered such that two vertices with consecutive indices are adjacent to each other.

Consider a subproblem with a convex polytope formed by vertices $\boldsymbol{x}(i:j)$. Then, the **state** of the subproblem is $(i, j)$. In each stage, the **decision** is to choose a vertex $k$ between $i$ and $j$ so that diagonals $[i, k]$ and $[k, j]$ are included into $\mathcal{P}$, as shown in the figure below.



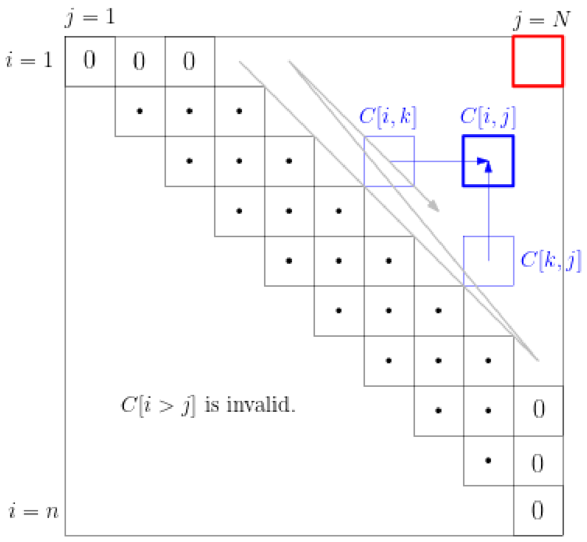
Let the minimum cost of triangulation of this polytope be denoted by $C[i, j]$. Then, for all $j <= i+3$, we have $C[i, j] = 0$ since we have either a single vertex, two vertices, or three vertices. For all $j > i+3$, we have the following **Bellman recursion**:

$$C[i, j] = \min_{i<k<j} C[i, k] + C[k, j] + d(i, k) + d(k, j),$$

where $d(a, b)$ is the Euclidean distance between $\boldsymbol{x}[a]$ and $\boldsymbol{x}[b]$.

Given the Bellman recursion stated above, we can define the recursion as depicted in the figure below:

[2]A region contained by $n$ vertices is a convex polygon if the cross product of any two consecutive edges formed by three consecutive vertices will have the same sign.



Consequently, we have the following **pseudocode**:

```
TRIANGULATION-DP($\boldsymbol{x}$)
1  $n = \boldsymbol{x}.length$
2  $C = \text{ZEROS}(n, n)$ // Initialize $C$ as a $n \times n$ all-zero matrix.
3  for $\ell = 4$ to $n$
4      for $i = 1$ to $n - \ell + 1$
5          $j = i + \ell - 1$
6          $C[i,j] = \min_{i<k<j} \left\{ C[i,k] + C[k,j] + \text{DIST}(i,k) + \text{DIST}(k,j) \right\}$
7  return $C[1,n]$
```