Algorithm Design
Dynamic Programming

Hikmat Farhat
Given a graph $G = (V, E)$ an independent set is a subset of vertices $S \subseteq V$ such that for all $x, y \in S$, $(x, y) \not\in E$.

Usually we need to an independent set such that $|S|$ is maximum. This general problem is hard to solve in the general case (see NP chapter later).

In this lecture we consider a special case of the independent set: when all vertices are on the same line.

The solution for this particular case is trivial: select every other node.

We make things interesting by adding a weight to each node and asking: what is the independent set which has the largest total weight?
An example of the (linear) independent set problem with weights is shown below. Clearly the solution in this case is \( \{x_2, x_4\} \).
Dynamic Programming Solution

- Given a set of nodes \( \{x_1, \ldots, x_n\} \) with associated weights \( \{w_1, \ldots, w_n\} \) we need to find the independent set \( S \) such that \( \sum_{x \in S} \) is maximum.

- Assume that the solution \( O \) and the optimal value (which we don’t know how to compute yet) is \( opt(n) \). The dependence on \( n \) comes from the fact that we will express this value in terms of smaller subproblems.

- Consider the last node \( x_n \). Either \( x_n \) contributed to \( opt(n) \) (i.e. \( x_n \in O \)) or not.

- In the first case \( x_n \in O \) implies that \( x_{n-1} \notin O \). If we can compute the optimal value for \( \{x_1, \ldots, x_{n-2}\} \) then we add to it \( w_n \) to obtain the optimal value.

- In the second case \( x_n \notin O \) then the optimal value is the same as the one obtained for \( \{x_1, \ldots, x_{n-1}\} \).
From the analysis above we conclude that

\[ opt(n) = \max \begin{cases} w_n + opt(n - 2) \\ opt(n - 1) \end{cases} \]

Applying the above recursive formula to the previous example while noting that \( opt[0] = 0, opt[1] = 2 \) (only \( x_1 \) is included) we get

\[ opt[5] = \max \begin{cases} 4 + opt[3] \\ opt[4] \end{cases} \]
\[\begin{align*}
\text{opt}[4] &= \max \left\{ 7 + \text{opt}[2], \text{opt}[3] \right\} \\
\text{opt}[3] &= \max \left\{ 3 + \text{opt}[1], \text{opt}[2] \right\} \\
\text{opt}[2] &= \max \left\{ 8 + \text{opt}[0], \text{opt}[1] \right\}
\end{align*}\]

Since \(\text{opt}[0] = 0\) and \(\text{opt}[1] = 2\) then \(\text{opt}[2] = 8\), \(\text{opt}[3] = 8\), \(\text{opt}[4] = 15\), \(\text{opt}[5] = 15\)
Finding the solution

- Our method allowed us to compute the optimal value. What if we want the list of vertices for the optimal solution?
- In all dynamic programming problems the way to find the optimal solution from the optimal value is almost the same: walk backwards.
- In the example above: we start with $x_5$ is it in the solution?
- Since $opt[5] = 15$ and $opt[4] = 15$ then $x_5$ was not selected in the solution.
- Since $opt[4] = 15$ and $opt[3] = 8$ then $x_4$ was selected in the solution, which means $x_3$ cannot be in the solution.
- Since $opt[2] = 18$ and $opt[1] = 2$ then $x_2$ was selected in the solution which means $x_1$ cannot be in the solution.
- From the above the optimal solution includes $\{x_2, x_4\}$.
A quick look at the recursive solution we can see that the values of \( opt[2] \) and \( opt[3] \) were needed twice (for the computation of \( opt[3] \), \( opt[4] \), \( opt[5] \)).

If the size of the problem was bigger then these values (among others) would have been needed even more.

In fact this top down approach will lead to exponential complexity because many of the values have to be repeatedly computed.

To avoid this exponential blowup we either have to use memoization (saving the computed values for later use) or solve the problem in a bottom up manner by using iteration.
The iterative solution for the independent set (linear) problem can be written as

\[
\text{IS}(G)
\]

\[
\begin{align*}
opt[0] &= 0 \\
opt[1] &= w_1 \\
\text{for } 2 = 1 \text{ to } n \text{ do} \\
| \quad opt[i] &= \max(opt[i - 1], w[i] + opt[i - 2]) \\
\text{return } opt[n]
\end{align*}
\]
The weighted interval scheduling is a generalization of the interval scheduling we studied using greedy approach.

In the greedy approach the intervals were considered to be equivalent.

In this generalization each interval $i$ has a weight $v_i$ in addition to starting $e(i)$ and ending $f(i)$ time.

A simple example shows that the greedy approach no longer works when the intervals have weights.
Let \( I \) be a set of intervals where each interval \( i \) starts at \( e(i) \), ends at \( f(i) \) and has values \( v(i) \).

Our goal is to find a subset of non-overlapping intervals \( S \subseteq I \) such that \( V = \sum_{i \in S} v(i) \) is maximum.

First we sort the intervals by ending time as we did in the case of greedy approach.

Given interval \( i \) let \( p(i) \) be the largest index \( k < i \) such that intervals \( k \) and \( i \) do not overlap.

As an example let \( I = \{(1, 2, 2), (1, 5, 7), (4, 7, 3), (6, 8)\} \) where \((a, b, c)\) denotes an interval that starts at \( a \), ends at \( b \) and has value \( c \).

The greedy approach will select the smallest ending time first so the greedy solution gives

\[
S = \{(1, 2, 2), (4, 7, 3)\} \quad \text{for a value of 5}
\]
Whereas the optimal solution is

\[ S = \{(1, 5, 7), (6, 8, 4)\} \] for a value of 11

The function \( p \) in this example has the following values

\[
\begin{align*}
p(1) &= 0 \\
p(2) &= 0 \\
p(3) &= 1 \\
p(4) &= 2
\end{align*}
\]

For the general problem where \( I = \{i_1, \ldots, i_n\} \) let \( opt(n) \) be the optimal solution and consider the last interval \( i_n \).
There are two cases

1. \( i_n \in \text{opt}(n) \). In this case all the intervals \( i_{p(n)+1}, \ldots, i_{n-1} \)
   overlap with \( i_n \) and therefore cannot be in \( \text{opt}(n) \)

2. \( i_n \notin \text{opt}(n) \). In this case the solution with \( n \) intervals is the
   same as the solution for \( n - 1 \) intervals since we know \( i_n \notin \text{opt} \)

3. Since we don’t know which case we have, we take the larger one. This means that we compute both possibilities and the
   larger of the two values is chosen as the optimal solution

\[
\text{opt}(n) = \max \begin{cases} 
  v_n + \text{opt}(p(n)) & \text{if } i_n \in \text{opt}(n) \\
  \text{opt}(n - 1) & \text{if } i_n \notin \text{opt}(n)
\end{cases}
\]

The above is a recursive equation that will allow us to compute the optimal value.

If we use the equation as it is it might lead to exponential blow up.

Instead we do the computation bottom up
OPT(\(I\))

\(opt[0] = 0\)

\textbf{for} \(i = 1\) \textbf{to} \(n\) \textbf{do}

\(\quad opt[i] = \max(opt[i-1], v[i] + opt[p[i]])\)

\textbf{return} \(opt[n]\)

- Clearly the complexity of the above algorithm is \(\Theta(n)\).
- Also we need to sort the intervals by ending time which is \(\Theta(n \log n)\).
- Once the intervals are sorted computing the \(p[i]\)'s can be done in \(\Theta(\log n)\) for each \(i\) (similar to binary search) for a total of \(\Theta(n \log n)\)
Max subarray sum

- Recall: given an array of numbers $A_0, A_1, \ldots, A_{n-1}$ let

$$S_{ij} = \sum_{k=i}^{j} A_k$$

- Goal: find $\max_{ij} S_{ij}$
- Brute force (see first lecture): $\Theta(n^3)$
- Divide and conquer (see first lecture): $\Theta(n \log n)$
- Can we do better? yes.
Let $M_j = \max_i S_{ij}$. $M_j$ is the maximum sum ending at $j$ for all possible values of $0 \leq i \leq j$. (see next slide for details)

The max subsequence sum can be written as $\max_j M_j$.

The key idea is that the values $M_j$ can be stored in an array and computed in $\Theta(n)$ time.

What is the relationship between the different values of $M_j$?

It is clear that if $M_{j-1} > 0$ then $M_j = M_{j-1} + A[j]$.

Otherwise $M_{j-1} \leq 0$ then $M_j = A[j]$

The above two cases can be combined to obtain

$$M_j = \max(A[j], M_{j-1} + A[j])$$
\[
M = \max_j \max_i \sum_{k=i}^{j} A[k] = \max_j \left( \max_i \sum_{k=i}^{j} A[k] \right) \\
= \max_j M_j
\]

Now

\[
M_j = \max_i \sum_{k=i}^{j} A[k] = \max(j \sum_{k=1}^{j} A[k], \sum_{k=2}^{j} A[k], \ldots, A[j]) \\
= \max(j \sum_{k=1}^{j-1} A[k] + A[j], \sum_{k=2}^{j-1} A[k] + A[j], \ldots, A[j])
\]
\[
\begin{align*}
= \max \left( \sum_{i=1}^{j-1} A[k], \sum_{i=2}^{j-1} A[k], \ldots, 0 \right) + A[j] \\
= \max(M_{j-1} + A[j], A[j])
\end{align*}
\]
Example

Consider the array

\[ [2, -4, 6, 3, -7, 4, 5, -5, -6, 4, 6, -4, 3] \]

One can guess by inspection that the subarray \([6, 3, -7, 4, 5]\) will give the largest sum.

using the proposed algorithm we get

\[ [2, -2, 6, 9, 2, 6, 11, 6, 0, 4, 10, 6, 9] \]

By scanning the result we obtain the maximum 11 as it should be.

Note that this method not only computes the result for \(n = 13\) elements but also solves all subproblems.
MaxSubarraySum(A)

\[ M[0] \leftarrow 0 \]
\[ \text{max} \leftarrow 0 \]
for \( i = 1 \) to \( n \) do
\[ M[i] \leftarrow \text{max}(A[i], A[i] + M[i - 1]) \]
/* Now compute the maximum of all sums */
for \( i = 1 \) to \( n \) do
    if \( M[i] > \text{max} \) then
        \[ \text{max} \leftarrow M[i] \]
return \( \text{max} \)
Consider two strings $X$ and $Y$. Our goal is to find the longest sequence that is contained in both $X$ and $Y$.

For example: $BDBDC$ and $ADBC$ then the longest common sequence is $DBC$.

Note that the characters in the sequence do not have to be necessarily consecutive.

This problem is different from the common substring problem where the characters in the common sequence are required to be consecutive.

In this example the longest common substring is $DB$. 

Consider two strings \( X = x_1x_2 \ldots x_{n-1}x_n \) and \( Y = y_1y_2 \ldots y_{m-1}y_m \). We denote the prefix of length \( k \) of \( X \) by \( X_k = x_1 \ldots x_k \) so that \( X = X_n \) and \( Y = Y_m \).

Let \( LCS(X, Y) \) be the length of the longest common subsequence between \( X \) and \( Y \). Then we can write:
\[
LCS(X, Y) = LCS(X_n, Y_m).
\]

If \( x_n = y_m \) then the common subsequence must include \( x_n \). In this case:
\[
LCS(X_n, Y_m) = LCS(X_{n-1}, Y_{m-1}) + 1.
\]

If \( x_n \neq y_m \) then:
\[
LCS(X_n, Y_m) = \max(LCS(X_n, Y_{m-1}), LCS(X_{n-1}, Y_m)).
\]
First we note that if one substring has 0 elements the length of the common subsequence is 0. Therefore $LCS[i][0] = 0$ and $LCS[0][j] = 0$ for all $i, j$.

\[
\text{LCS}(X, Y)
\]

\[
\text{for } i = 1 \text{ to } n \text{ do}
\]

\[
\text{for } j = 1 \text{ to } m \text{ do}
\]

\[
\text{if } X[i] = X[j] \text{ then}
\]

\[
LCS[i][j] \leftarrow 1 + LCS[i - 1][j - 1]
\]

\[
\text{else}
\]

\[
LCS[i][j] \leftarrow \max(LCS[i - 1][j], LCS[i][j - 1])
\]

\[
\text{return } LCS[n][m]
\]
Example

Consider the two strings $X = ABCD$ and $Y = AABDC$
Given a knapsack of capacity $W$ and $n$ items $x_1, \ldots, x_n$ with weights $w_1, \ldots, w_n$ and values $v_1, \ldots, v_n$ we need to find a subset $S \subseteq \{x_1, \ldots, x_n\}$ such that

$$\sum_{x_i \in S} w_i \leq W$$

$$\sum_{x_i \in S} v_i \text{ is maximum}$$
Let $S_n$ be the optimal solution when the problem contains $n$ items and consider item $x_n$. There are two possibilities:

Either $x_n \in S_n$. Suppose that we know $S_{n-1}$ the optimal solution for the subproblem including the first $n-1$ items. Can one add to that solution $x_n$? No, because when the weight of $x_n$ is added, the total weight might be greater than $W$. Instead we look for the optimal solution for the first $n-1$ items with a knapsack of capacity $W - w_n$ to leave room for item $x_n$. Therefore $S_n(W) = v_n + S_{n-1}(W - w_n)$

Or $x_n \notin S_n$. Since $x_n \notin S_n$ then we don’t need to leave room for it when we compute $S_{n-1}$. In this case $S_n(W) = S_n(W)$
Because we don’t know apriori if \( x_n \in S_n \) we compute both values and take the largest.

\[
S(n, W) = \max \left\{ v_n + S(n - 1, W - w_n), S(n - 1, W) \right\}
\]

Before writing a bottom up solution we need to take care of the boundary conditions.

Anytime the capacity of the knapsack \( W = 0 \) the solution is just 0.

Also we need to take into account the case when \( w_n > W \). In this case \( S(n, W) = S(n - 1, W) \)
Knapsack\((w, v, W)\)

/* if \(W = 0\) then solution for any number of items is 0 */

for \(i = 1\) to \(n\) do
    S\([i][0]\)=0
/* for any \(W\) zero items gives 0 */

for \(i = 1\) to \(W\) do
    S\([0][i]\)=0
/* Now the solution */

for \(j = 1\) to \(W\) do
    for \(i = 1\) to \(n\) do
        S\([i][j]\)=\(\max(S[i-1][j], v[i]+S[i-1][j-w[i]])\)
Consider the following instance of the knapsack problem where the knapsack size is $W = 13$.

<table>
<thead>
<tr>
<th>$v_1$</th>
<th>$w_1$</th>
<th>$\frac{v_1}{w_1}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3</td>
<td>0.66</td>
</tr>
<tr>
<td>4</td>
<td>6</td>
<td>0.66</td>
</tr>
<tr>
<td>5</td>
<td>7</td>
<td>0.71</td>
</tr>
<tr>
<td>6</td>
<td>8</td>
<td>7.5</td>
</tr>
</tbody>
</table>

Note that a greedy approach would choose items $v_4$ and $v_1$ for a total value of 8 whereas the optimal is 9 by choosing items 3 and 2.
<table>
<thead>
<tr>
<th>$n \backslash w$</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>9</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>
We are given an array $A$ of $n$ elements and an integer value $S$ and we ask if there is a subset of $A$ whose sum is $S$.

We start with the last element. There are two choices:

1. Either the last element is in the sum in which case we have to find a subset of of the first $n-1$ elements whose sum is $S - A[n]$.
2. Or the last element is not in the sum in which case we have to find a subset of the first $n-1$ elements whose sum is $S$.

The above reasoning will lead to

\[
\text{subset}(A, n, S) \{
\text{if } (n==0) \text{ return false}
\text{if } (S==0) \text{ return true}
\text{return subset}(A, n-1, S) || \text{subset}(A, n-1, S-A[n])
\}
\]
Consider the array $A = 2, 7, 3, 1$ and $S = 6$.

\[
\begin{align*}
\text{subset}(A, 4, 6) &= \text{subset}(A, 3, 6) \lor \text{subset}(A, 3, 5) = \text{false} \lor \text{true} \\
\text{subset}(A, 3, 6) &= \text{subset}(A, 2, 6) \lor \text{subset}(A, 2, 3) = \text{false} \lor \text{false} = \text{false} \\
\text{subset}(A, 3, 5) &= \text{subset}(A, 2, 5) \lor \text{subset}(A, 2, 2) = \text{false} \lor \text{true} = \text{true} \\
\text{subset}(A, 2, 6) &= \text{false} \\
\text{subset}(A, 2, 3) &= \text{false} \\
\text{subset}(A, 2, 5) &= \text{false} \\
\text{subset}(A, 2, 2) &= \text{true}
\end{align*}
\]
We use the recurrence relation that we have obtained to solve the subsetsum problem in a bottom up manner. First note that if the required sum is 0 then the answer is always true. Also if the number of elements is 0 then with the exception of sum=0 all the answers should be false.
SS(A, S)

/* if S = 0 then solution for any number of items is True */
for i = 0 to n do
  R[i][0] = True
/* for any S other than zero */
/* 0 items is False */
for j = 1 to S do
  R[0][j] = False
/* Now the solution */
for i = 1 to n do
  for j = 1 to S do
    R[i][j] = R[i - 1][j] or R[i - 1][j - A[i]]
Suppose that you have a set of $n$ integers $A = \{A_1, \ldots, A_n\}$.

Let $S_1$ and $S_2$ be a partition of $A$ i.e. $S_1 \cup S_2 = A$ and $S_1 \cap S_2 = \emptyset$.

Let $sum_1 = \sum_{a \in S_1} a$ and $sum_2 = \sum_{a \in S_2}$ be the sum of elements in each partition.

Our goal is to find $S_1$ and $S_2$ such that $|\text{sum}_1 - \text{sum}_2|$ is minimum.
Let $\sigma = \sum_{a \in A} a$, i.e. the sum of all elements in $A$.

Given $A$ we know how to find if there is a subset of $A$ whose sum is $k$.

Let $p(i, j)$ be true if the first $i$ elements of $A$ have a subset whose sum is $j$.

This is the subset sum problem we have studied previously.

Computer $p(i, j)$ for all $1 \leq i \leq n$ and $0 \leq j \leq \sigma/2$.

then consider the set

$$\{ |j - \sigma/2| \mid p(n, j) = 1 \}$$

Our answer is

$$\min_j \{ |j - \sigma/2| \mid p(n, j) = 1 \}$$
Consider the set $\{1, 6, 11\}$ whose sum is 18 so $\sigma/2 = 9$. We compute the subset sum problem in a bottom up fashion we get
For example

\[ \text{ACTGG} - \text{ATT} \]
\[ \text{ACGGGTATG} \]

We need to align two sequences to minimize the ”cost”.

Cost the sum of costs where each dash incurs a cost \( \alpha_{\text{gap}} \) and cost of mismatch \( \alpha \).

Note that we are assuming that all mismatches are the same cost even though it is easy to modify the cost to depend on the type of mismatch.
Given strings $X = x_1 \ldots x_n$ and $Y = y_1 \ldots y_m$ we need to find the "best" alignment: the one that leads to the smallest cost.

Let $X_i$ be the prefix of $X$ of size $i$, i.e. $X_i = x_1 \ldots x_i$.

Let $opt(X_n, Y_m)$ be the optimal solution for prefixes $X_n = X$ and $Y_m = Y$. We can write

$$opt(X_n, Y_m) = \min \begin{cases} 
\alpha + opt(X_{n-1}, Y_{m-1}) \\
\alpha_{gap} + opt(X_{n-1}, Y_m) \\
\alpha_{gap} + opt(X_n, Y_{m-1})
\end{cases}$$
Before we implement the algorithm we need to determine the value of the base cases: $opt(X_i, 0)$ and $opt(0, Y_i)$.

Note that when one string is empty we need $i$ gaps to match the other so

$$opt(X_i, 0) = opt(0, Y_i) = i \cdot \alpha_{gap}$$
Example

Consider the two strings $X = "CG"$ and $Y = "CA"$ with the cost of a gap $\alpha_g=3$ and cost of mismatch $\alpha = 7$. The solution obtained from the table below is $CG\rightarrow$ and $C \leftarrow A$.

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$X \backslash Y$</td>
<td>&quot;&quot;</td>
<td>C</td>
<td>CA</td>
</tr>
<tr>
<td>&quot;&quot;</td>
<td>0</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>C</td>
<td>3</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>CG</td>
<td>6</td>
<td>3</td>
<td>6</td>
</tr>
</tbody>
</table>
The Bellman-Ford algorithm uses dynamic programming to compute the shortest path in graph \( G = \langle V, E \rangle \).

The quantity to be optimized is the shortest distance using at most \( i \) edges.

Let \( d[u, i] \) be the shortest path distance from some source \( s \) to destination \( u \) using at most \( i \) edges.

Let \( c_{v,u} \) be the cost of edge \( (v, u) \in E \) or \( \infty \) otherwise.

Then \( d[u, i + 1] \) can be written as

\[
d[u, i + 1] = \min(d[u, i], \min_{v \in V}(d[v, i] + c_{v,u}))
\]
In the graph below we would like to use the Bellman-Ford algorithm to compute the shortest path from node a to all other nodes.
<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>b</td>
<td>∞</td>
<td>-1</td>
<td>-2</td>
<td>-2</td>
<td>-2</td>
<td>-6</td>
</tr>
<tr>
<td>c</td>
<td>∞</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>d</td>
<td>∞</td>
<td>∞</td>
<td>5</td>
<td>4</td>
<td>-3</td>
<td>-3</td>
</tr>
<tr>
<td>e</td>
<td>∞</td>
<td>∞</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>f</td>
<td>∞</td>
<td>∞</td>
<td>7</td>
<td>-2</td>
<td>-2</td>
<td>-2</td>
</tr>
</tbody>
</table>
Implementation

- Encoding the edge weights
  0-1: -1
  0-2: 3
  1-3: 6
  2-1: -5
  2-4: 2
  2-5: 4
  3-1: -3
  4-5: -7
  5-3: -1

- We choose a large value compared to the edge weights in the graph, say 100.
<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>b</td>
<td>100</td>
<td>-1</td>
<td>-2</td>
<td>-2</td>
<td>-2</td>
<td>-6</td>
</tr>
<tr>
<td>c</td>
<td>100</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>d</td>
<td>100</td>
<td>99</td>
<td>5</td>
<td>4</td>
<td>-3</td>
<td>-3</td>
</tr>
<tr>
<td>e</td>
<td>100</td>
<td>100</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>f</td>
<td>100</td>
<td>93</td>
<td>7</td>
<td>-2</td>
<td>-2</td>
<td>-2</td>
</tr>
</tbody>
</table>
```python
import numpy as np
num=6
opt=np.full((num,num),100)
opt[0,0]=0
edge=np.full((num,num),100)

f=open("graph1.txt","r")
input=f.read()
lines=input.splitlines()
for line in lines:
    x=line.split("":"")
    cost=x[1]
    y=x[0].split("-")
    s=y[0]
    d=y[1]
    edge[int(s),int(d)]=int(cost)
```
```python
pred = np.full(num, -1)
for l in range(1, num):  # iterate over length
    for n in range(0, num):  # iterate over nodes
        opt[n, l] = opt[n, l - 1]
        for m in range(0, num):  # iterate over neighbors
            s = opt[m, l - 1] + edge[m, n]
            if s < opt[n, l]:
                opt[n, l] = s
                pred[n] = m
```