Dynamic Programming Algorithms

Introduction

In our study of divide-and-conquer algorithms, we noticed that a problem seemed conducive to a divide-and-conquer approach provided

1. it could be divided into one or more subproblems of smaller size that could be recursively solved, and
2. solutions to the subproblems could be combined to form the solution to the original problem within a reasonable (i.e. bounded by a polynomial) number of steps.

Another key property that allows divide-and-conquer algorithms to succeed is that each subproblem occurs at most once in the computation tree induced by the recursion. For example, in quicksort once a portion of the original array has been sorted there is no need to sort it again. And in fact, the quicksort algorithm sorts each portion of the array at most once.

However, the *top down* recursive approach is oftentimes not appropriate for some problems because a top down approach to solving them may cause some sub-problems to re-solved an exorbitant number of times in the computation tree. Consider the following example.

Example 1. The Fibonacci Numbers is a sequence $f_0, f_1, \ldots$ recursively defined as follows:

- **base case.** $f_0 = 0$ and $f_1 = 1$
- **recursive case.** for $n \geq 2$, $f_n = f_{n-1} + f_{n-2}$.

Show that the following recursive algorithm for computing the n th Fibonacci number has exponential complexity with respect to n.

```
int fib(int n)
{
    if(n == 0)
        return 0;
    if(n==1)
        return 1;
    return fib(n-1) + fib(n-2);
}
```

The above recursive algorithm has a simple remedy that characterizes a *dynamic programming* algorithm. On input n, rather than make blind recursive calls, we instead store f_0 and f_1, and use these values to compute f_2. After storing f_2, we then compute f_3 using f_1 and f_2. This process continues until f_n has been computed. We call this approach a **bottom-up** approach, since it begins by computing the solutions for smaller problems, and then using those solutions to compute the solutions for larger problems.

The term **dynamic programming** refers to this bottom-up approach. In general, it solves a problem by first tabulating a list of solutions to subproblems, starting with the most simplest of subproblems, and working its way towards the solution to the original problem. The key step in designing a dynamic-programming algorithm is to develop a **dynamic programming recurrence relation** that relates the solution to a larger problem to that of smaller sub-problems.

Some applications of the dynamic-programming include the following.

0-1 Knapsack Given objects $x_1, \ldots, x_n$, where object x_i has weight w_i and profit p_i (if its placed in the knapsack), determine the subset of objects to place in the knapsack in order to maximize profit, assuming that the sack has capacity M.

Longest Common Subsequence Given an alphabet Σ, and two words X and Y whose letters belong to Σ, find the longest word Z which is a (non-contiguous) subsequence of both X and Y.

Optimal Binary Search Tree Given a set of keys $k_1, \ldots, k_n$ and weights $w_1, \ldots w_n$, where w_i reflects how often k_i is accessed, design a binary search tree so that the weighted cost of accessing a key is minimized.

Matrix Chain Multiplication Given a sequence of matrices that must be multiplied, parenthesize the product so that the total multiplication complexity is minimized.

All-Pairs Minimum Distance Given a directed graph $G = (V, E)$, find the distance between all pairs of vertices in V.

Polygon Triangulation Given a convex polygon $P =< v_0, v_1, \ldots, v_{n-1} >$ and a weight function defined on both the chords and sides of P, find a triangulation of P that minimizes the sum of the weights of which forms the triangulation.

Bitonic Traveling Salesperson given n cities $c_1, \ldots, c_n$, where c_i has grid coordinates (x_i, y_i), and a cost matrix C, where entry C_{ij} denotes the cost of traveling from city i to city j, determine a left-to-right followed by right-to-left Hamilton-cycle tour of all the cities which minimizes the total traveling cost. In other words, the tour starts at the leftmost city, proceeds from left to right visiting a subset of the cities (including the rightmost city), and then concludes from right to left visiting the remaining cities.

Viterbi's algorithm for context-dependent classification Given a set of observations $\vec{x}_1, \ldots, \vec{x}_n$ find the sequence of classes $\omega_1, \ldots, \omega_n$ that are most likely to have produced the observation sequence.

Edit Distance Given two words u and v over some alphabet, determine the least number of edits (letter deletions, additions, and changes) that are needed to transform u into v.

0-1 Knapsack

0-1 Knapsack: given objects $x_1, \ldots, x_n$, where object x_i has weight w_i and profit p_i (if its placed in the knapsack), determine the subset of objects to place in the knapsack in order to maximize profit, assuming that the sack has capacity M.

A recurrence for 0-1 Knapsack can be derived by making the following observations about an optimal solution.

Case 1 The optimal solution includes x_n. Then the rest of the solution is the optimal solution for the knapsack problem in which objects $x_1, \ldots, x_{n-1}$ are given (same weights and profits as before), and for which the sack capacity is now $M - w_n$.

Case 2 The optimal solution does not include x_n. Then the solution is the optimal solution for the knapsack problem in which objects $x_1, \ldots, x_{n-1}$ are given (same weights and profits as before), and for which the sack capacity is M.

We can generalize this observation by considering the sub-problem where objects $x_1, \ldots, x_i$ are to be placed into a knapsack with capacity c. Letting $P(i, c)$ denote the maximum profit for this problem, then the above cases lead to the recurrence

$$P(i,c) = \begin{cases} 0 & \text{if } i = 0 \text{ or } c \leq 0 \\ \max(P(i-1,c), P(i-1,c-w_i) + p_i) & \text{if } w_i \leq c \\ P(i-1,c) & \text{otherwise} \end{cases}$$

Of course, we ultimately desire to compute $P(n, M)$ which may be found by be computing each entry of the matrix $P(i, c)$, for $0 \leq i \leq n$, and $0 \leq c \leq M$. Thus, the algorithm has a running time of $\Theta(nM)$, which is exponential in the two input-size paramters n and $\log M$ (why $\log M$ and not M?).

Example 2. Solve the following 0-1 knapsack problem using a dynamic-programming approach. Assume a knapsack capacity of $M = 10$.

object	**weight**	**profit**
1	3	4
2	5	6
3	5	5
4	1	3
5	4	5

	0	1	2	3	4	5	6	7	8	9	10
0											
1											
2											
3											
4											
5											

Edit Distance Between Two Words

Given two words u and v over some alphabet, The **edit distance**, also known as **Levenshtein distance**, $d(u, v)$ is defined as the the minimum number of edit operations needed to convert one word to another. These edits include adding a character, deleting a character, and changing a character. In this lecture, each edit will be assinged a cost of 1. However, in practice, it may make more sense to assign different costs to different kinds of edits, where the more likely edits (e.g., replacing an 'e' with an 'r', as often occurs in typing errors) are given lower costs

Example 3. Compute the edit distance between the words u=sample and v=dampen.

Example 4. Show that the edit distance over the set of words of some alphabet represents a **metric space**; meaning that the distance is only zero when the two words are equal, is symmetric on the two input words, and satisfies the triangle inequality

$$d(u, v) \leq d(u, w) + d(w, v),$$

for any three words u, v, w.

Theorem 1. Let u and v be words, with, $|u| = m$, and $|v| = n$. For $0 \leq i \leq m$ and $0 \leq j \leq n$, define $d(i, j)$ as the edit distance between $u[1 : i]$ and $v[1 : j]$, where, e.g., $u[1 : i]$ is the prefix of word u having length i. Then

$$d(i,j) = \begin{cases} j & \text{if } i = 0 \\ i & \text{if } j = 0 \\ \min(d(i-1,j)+1, d(i,j-1)+1, d(i-1,j-1)+(u_i \neq v_j)) & \end{cases}$$

Proof of Theorem 1. Consider an optimal sequence of edits that transforms $u[1 : i]$ to $v[1 : j]$. We may assume that these edits are performed on u from right to left. Consider the first operation that is performed on u.

Case 1. The first edit in u's transformation is to add a letter to the end of u. In this case the added letter will match v_j. Hence, after adding this final letter we must now transform $u[1 : i]$ to $v[1 : j-1]$. This can be optimally done in $d(i, j-1)$ steps. Hence, adding 1 for the initial letter addition yields $d(i, j-1) + 1$ steps.

Case 2. The first edit in u's transformation is to remove the last letter of u. If we are optimally transforming u to v, and the first step is to remove a letter at the end of u, then clearly this letter must be u_i. Hence, after the removal, we must subsequently transform $u[1 : i-1]$ to $v[1 : j]$. This can be optimally done in $d(i-1, j)$ steps. Hence, adding 1 for the initial letter deletion yields $d(i-1, j)+1$ steps.

Case 3. The first edit in u's transformation is to change u_i to v_j. If this is the first edit, then we must subsequently transform $u[1 : i-1]$ to $v[1 : j-1]$. The number of steps to perform this is $d(i-1, j-1)$. Hence, adding 1 for the initial letter change yields $d(i-1, j-1) + 1$ steps.

Case 4. The first edit that transforms $u[1 : i]$ to $v[1 : j]$ occurs somewhere before u_i, and hence $u_i = v_j$. In this case we are actually transforming $u[1 : i-1]$ to $v[1 : j-1]$, which can be optimally performed in $d(i-1, j-1)$ steps.

Example 5. Let $u =$ claim and $v =$ climb. In optimally transforming u to v, what case applies for the first edit? Same question for $u =$ gingerale and $v =$ ginger, $u =$ storm and $v =$ warm, $u = recieve$ and $v = receive$, and $u = hurts$ and $v = hertz$.

Notice that $d(u, v)$ can be computed by the method of dynamic programming, since the edit distance between two words is reduced to finding the edit distance between two smaller words that are prefixes of the two original words. Here the problem is to compute $d(m, n)$, and the subproblems are to compute $d(i, j)$, for each $0 \leq i \leq m$ and $0 \leq i \leq n$. The subproblems are stored in a matrix for future access. This is called **memoization**. For example, $d(3, 2)$ can be computed by comparing $d(3, 1)$, $d(2, 2)$, and $d(2, 1)$, which have all been previously stored in the matrix. This yields a total of $\Theta(mn)$ steps to compute the edit distance.

Example 6. Show the dynamic-programming matrix that is formed when computing the edit distance of $u =$ fast and $v =$ cats.

Optimal Binary Search Tree

Suppose a binary tree $\mathcal{T}$ holds keys $k_1, \ldots k_n$ (henceforth, and without loss of generality, assume that $k_i = i$). Let w_i denote a weight that is assigned to k_i. A large (respectively, small) weight means that k_i is accessed more (respectively, less) often. Now, if d_i denotes the depth of k_i (e.g. the root has depth 1), then the **weighted access cost** of $\mathcal{T}$ is defined as

$$\mathrm{wac}(\mathcal{T}) = \sum_{i=1}^{n} w_i d_i.$$

Example 7. Suppose keys 1-5 have respective weights 50,40,20,30,40, and are inserted into an intially empty binary search tree $\mathcal{T}$ in the order 1,5,2,4,3. Determine $\mathrm{wac}(\mathcal{T})$.

Thus, the problem to be solved is to find a binary search tree that holds keys $1, \ldots, n$, and has minimal weighted access cost. The main insight that leads to a dynamic-programming recurrence for solving this problem is to observe that a binary search tree itself is a recursive structure. Moreover, consider the optimal tree T_{opt} and suppose it has root $k \in \{1, \ldots, n\}$. Let T_L and T_R denote the respective left and right subtrees of the root. Then T_L and T_R are themselves **optimal substructures**, meaning that T_L is the solution to finding a binary-search tree that has minimum weighted access cost, and which holds keys $1, \ldots, k-1$, and T_R is the solution to finding a binary-search tree that has minimum weighted access cost, and which holds keys $k+1, \ldots, n$. Suppose otherwise. For example, suppose T' is a binary search tree that holds keys $1, \ldots, k-1$, and for which $\mathrm{wac}(T') < \mathrm{wac}(T_L)$. Then we could replace T_L with T' in T_{opt} and obtain a binary search tree over keys $1, \ldots, n$ that has a smaller weighted access cost than T_{opt}, which is a contradiction.

The above insight suggest that we define $\mathrm{wac}(i, j)$ as the minimum weighted access cost that can be attained by a binary search tree that holds keys $i, \ldots, j$. Then the above insight also suggests the following recurrence:

$$\mathrm{wac}(1, n) = w_k + \mathrm{wac}(1, k-1) + \mathrm{wac}(k+1, n).$$

In words, the weighted access cost of T_{opt} is the cost $w_k(1)$ of accessing the root, plus the total cost of accessing keys in T_L, plus the total cost of accessing keys in T_R. But the above recurrences in not quite correct. For example the quantity $\text{wac}(1, k-1)$ makes the assumption that T_L is not a subtree of some other tree. In other words, it assumes that the root r of T_L has a depth of 1, the children of r have a depth of 2, etc.. However, since T_L is the left subtree of the root of T_{opt}, r instead has a depth of *2*, while its children have a depth of *3*, etc.. In general, if node i of T_L contributes $w_i d_i$ to $\text{wac}(1, k-1)$ then it must contribute $w_i(d_i + 1)$ when T_L is a subtree of T_{opt}. This means that an additional w_i must be added to $\text{wac}(1, k-1)$. Hence, the above recurrence must be re-written as

$$\text{wac}(1, n) = w_k + \text{wac}(1, k-1) + \sum_{r=1}^{k-1} w_r + \text{wac}(k+1, n) + \sum_{r=k+1}^{n} w_r =$$

$$\text{wac}(1, k-1) + \text{wac}(k+1, n) + \sum_{r=1}^{n} w_r.$$

Finally since, we do not know offhand which key will serve as the root to T_{opt}, we must minimize the above quantity over the different possible values of k. This leads to the following theorem.

Theorem 1. Let $\text{wac}(i, j)$ denote the minimum attainable weighted access cost for any binary search tree that holds keys $i, i+1, \ldots, j$. Then

$$\text{wac}(i, j) = \begin{cases} 0 & \text{if } j < i \\ w_i & \text{if } i = j \\ \min\limits_{k=i}^{j}(\text{wac}(i, k-1) + \text{wac}(k+1, j)) + \sum_{r=i}^{j} w_r & \text{otherwise.} \end{cases}$$

Theorem 1 provides the dynamic-programming recurrence needed to solve the Optimal Binary Search Tree problem. We must compute each entry $\text{wac}(i, j)$ of the wac matrix for all values $1 \le i \le j \le n$. It is an exercise to show that this matrix can be computed in $\Theta(n^3)$ steps.

Example 8. Use dynamic programming to determine the binary search tree of minimum weighted-access cost, and whose keys and weights are provided in Example 7.

Matrix-Chain Multiplication

A product P of matrices is said to be **fully parenthesized** iff

- **basis step:** $P = (A)$, for some matrix A
- **inductive step:** $P = (P_1 \cdot P_2)$, where P_1 and P_2 are fully parenthesized matrix products.

Example 9. How many ways can a product of four matrices be fully parenthesized?

Given a fully parenthesized product of matrices P, the **multiplication complexity**, $mc(P)$, is defined inductively as follows.

- **basis step:** if $P = (A)$, for some matrix A, then $mc(P) = 0$.
- **inductive step:** if $P = (P_1 \cdot P_2)$, where P_1 and P_2 are fully parenthesized matrix products which respectively evaluate to $p \times q$ and $q \times r$ matrices, then

$$mc(P) = mc(P_1) + mc(P_2) + pqr.$$

Example 10. Given matrices A,B,and C with respective dimensions 10×100, 100×5, and 5×50, find $mc(A(BC))$ and $mc((AB)C)$. Conclude that the parenthesization of a product of matrices affects the multiplication complexity.

Matrix-Chain Multiplication Problem: Given a chain of matrices $A_1, \dots, A_n$, find a full parenthesization P of the sequence for which $mc(P)$ is minimum.

Similar to the Optimal Binary Search Tree problem, we make the following key obervations about an optimal parenthesization P.

1. if $P = (A)$ for some matrix A, then $mp(P) = 0$
2. if $P = (P_1 P_2)$ where P_1 is a full parenthesization for $A_1, \dots, A_k$ and P_2 is a full parenthesization for $A_k, \dots, A_n$, then P_1 is the optimal parenthesization for the problem of minimizing the multiplication complexity for $A_1, \dots, A_k$, and P_2 is the optimal parenthesization for the problem of minimizing the multiplication complexity for $A_k, \dots, A_n$. In other words, if P is an optimal strucutre, then P_1 and P_2 are optimal substructures.

Thus, we see that the matrix-chain multiplication problem may also be solved by a bottom-up dynamic-programming algorithm, where the subproblems involve finding the optimal parenthesization of matrix sequences of the form $A_i, \dots, A_j$. Let $p_0, p_1, \dots, p_n$ denote the dimension sequence associated with the matrix sequence $A_1, \dots, A_n$ (for example, A_1 is a $p_0 \times p_1$ matrix). Let $\text{mc}(i, j)$ denote the minimum multiplication complexity for the sequence $A_i, \dots, A_j$. Let P be the optimal full parenthesization for $A_i, \dots, A_j$. If $P = (A_i)$, then $i = j$ and $\text{mc}(i, j) = 0$. Otherwise, $P = (P_1 P_2)$ where e.g. P_1 is the optimal full parenthesization of $A_i, \dots, A_k$, for some $i \le k < j$. Then

$$\text{mc}(i, j) = \text{mc}(i, k) + \text{mc}(k + 1, j) + p_{i-1} p_k p_j. \tag{1}$$

We summarize as follows:

1. if $i = j$, then $\text{mc}(i, j) = 0$
2. if $i < j$, then $\text{mc}(i, j) = \min_{i \leq k < j}\{\text{mc}(i, k) + \text{mc}(k + 1, j) + p_{i-1}p_kp_j\}$

We may store the values $\text{mc}(i, j)$ in a matrix M, and store the values k which minimize the complexity in a matrix S. For example, $s(i, j) = k$ means that k is the index which minimizes the sum in Equation 1.

Example 11. Given the five matrices below, find a full parenthesization of $A_1, \ldots, A_5$ with minimum multiplication complexity.

matrix	**dimension**
A_1	2×4
A_2	4×2
A_3	2×1
A_4	1×5
A_5	5×2

Matrix M

	$j = 1$	2	3	4	5
$i = 1$					
$i = 2$					
$i = 3$					
$i = 4$					
$i = 5$					

Matrix S

	$j = 1$	2	3	4	5
$i = 1$					
$i = 2$					
$i = 3$					
$i = 4$					
$i = 5$					

Floyd-Warshall Algorithm

Consider the problem of finding the minimum-cost path between two vertices of a weighted graph $G = (V, E, c)$, where $c : E \to R^+$ assigns a travel cost to each edge $e \in E$. Then the cost of the path $P = e_1, e_2, \ldots, e_m$ is defined as

$$c(P) = \sum_{i=1}^{m} c(e_i).$$

Moreover, the **distance** $d(u, v)$ **between** u **and** v is defined as the cost of the path from u to v which minimizes the above sum. In other words, $d(u, v)$ is the cost of the minimum-cost path from u to v.

Example 12. Let $G = (V, E, c)$ be a directed graph, where

$$V = \{1, 2, 3, 4, 5\}$$

and the edges-costs are given by

$$E = \{(1, 2, 1), (1, 3, 3), (2, 3, 1), (2, 4, 1), (2, 5, 4), (3, 4, 1), (4, 5, 1), (5, 1, 1)\}.$$

Draw the graph and determine the distance of each vertex from vertex 1.

We now give the Floyd-Warshall dynamic-programming algorithm for finding the distance between all pairs of graph vertices. It is based on the following simple principle. Suppose $P = u = v_0, v_1, \ldots, v_m = v$ is a minimum-cost path between vertices u and v of graph G. Then for any $0 < k < n$, the two paths

$$P_1 = u = v_0, v_1, \ldots, v_k$$

and

$$P_2 = v_k, v_{k+1}, \ldots, v_m$$

must both be minimum-cost paths. For otherwise one of them could be replaced by a minimum-cost path that would then yield a shorter distance from u to v. In other words, a minimum-cost path has optimal substructures, in that every subpath of a minimum-cost path is also a minimum-cost path. It is interesting to note that the same is *not* true for *maximum-cost* paths.

The benefit of optimal substructures in minimum-cost paths is the following. Let m be the number of edges in a minimum-cost path from vertex u to vertex v in graph G. Then there are two cases.

- **Case 1:** $m = 1$. Then the $d(u, v)$ is the cost of the edge from u to v in graph G (Note: if u is not adjacent to v, then we may assume an edge between them with a cost of ∞).
- **Case 2:** $m > 1$. Then there exists vertex w such that $d(u, v) = d(u, w) + d(w, v)$, where the lengths of the minimum-cost paths from u to w and from w to v do not exceed $m - 1$.

Now assume $\{1, 2, \ldots, n\}$ are the vertices of G, and let c_{ij} denotes the cost of the edge connecting vertex i to vertex j. The above observations suggest that we build up a minimum-cost path from u to v by piecing together two paths, each of smaller length. But rather than allowing for the "joining" vertex w to be chosen from any of the n vertices, we instead build in stages, where in stage k, $k = 1, 2, \ldots, n$, we allow for the joining vertex to be vertex k. With this in mind, we let d_{ij}^k denote the distance from vertex i to vertex j taken over all those paths whose internal vertices (i.e. vertices occuring between i and j) are a subset of $\{1, 2, \ldots, k\}$.

For $k = 0$, we set $d_{ij}^0 = c_{ij}$, if $i \neq j$, and zero otherwise. In this case d_{ij}^0 represents the distance over all paths from i to j which do not possess any internal vertices. Of course, such paths must have lengths equal to either zero (a single vertex) or one (two vertices connected by edge e_{ij} and having cost c_{ij}).

For $k > 0$ and $i \neq j$, consider d_{ij}^k and the minimum-cost path P from i to j that is restricted to using internal vertices $\{1, \ldots, k\}$. If P does not visit vertex k, then P is also a minimum-cost path from i to j that is restricted to using internal vertices $\{1, \ldots, k-1\}$. Hence, $d_{ij}^k = d_{ij}^{k-1}$. Otherwise, if P visits k, then, by the principle of optimal subpaths of an optimal path, $P = P_1P_2$, where P_1 is a minimum-cost path from i to k, and P_2 is a minimum-cost path from k to j, where both paths are restricted to using internal vertices $\{1, \ldots, k-1\}$. Hence, $d_{ij}^k = d_{ik}^{k-1} + d_{kj}^{k-1}$. Hence, d_{ij}^k can be computed by taking the minimum of the quantities d_{ij}^{k-1} and $d_{ik}^{k-1} + d_{kj}^{k-1}$.

The above paragraphs are now summarized by the following dynamic-programming recurrence.

$$d_{ij}^k = \begin{cases} 0 & \text{if } k = 0 \text{ and } i = j \\ c_{ij} & \text{if } k = 0 \text{ and } i \neq j \\ \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}) & \text{if } k \geq 1 \end{cases}$$

Notice that d_{ij}^n gives the distance from vertex i to vertex j over all paths whose internal vertices form a subset of $\{1, 2, \ldots, n\}$, which is precisely the distance from i to j over all possible paths, and is precisely what we desired to compute. The following example provides some intuition as to why the minimum-cost path between i and j will always be constructed. In other words, the restrictions placed on the joining vertex at each stage is not detrimental to obtaining the optimal path.

Example 13. Suppose $P = 8, 6, 1, 4, 2, 5$ is a minimum-cost path. Show the order in which the Floyd-Warshall Algorithm (as implied by the dynamic-programming recurrence relation) pieces together P from smaller minimum-cost paths. For each path, indicate the k value for which the cost of the path is first recorded in d_{ij}^k.

Example 14. Run the Floyd-Warshall Algorithm on the Graph from Example 12. Show each of the matrices d_{ij}^k.

Exercises

1. Let f_n denote the n th Fibonacci number. If f_n has closed-form

$$f_n = c_1(\frac{1+\sqrt{5}}{2})^n + c_2(\frac{1-\sqrt{5}}{2})^n,$$

determine the values of c_1 and c_2.

2. Prove that f_n has exponential growth. Hint: what can be say about the asymptotic growth of a sum of functions?

3. Solve the following 0-1 knapsack problem using dynamic-programming.

object	**weight**	**profit**
1	1	2
2	2	4
3	3	5
4	4	5
5	2	2
6	1	3

with knapsack capacity 8.

4. For 0-1 knapsack, devise recursive algorithm that takes as inputs matrix $P(i, c)$, the array w of weights, the array p of profits, the number of objects i, and the knapsack capacity c, and prints a list of the objects that are used to obtain a profit $P(i, c)$.

5. Determine the edit distance between u = lovely and v = waver.

6. Provide the dynamic-programming matrix that is needed to efficiently compute the edit-distance between u=block and v=slacks. Circle one of the optimal paths and provide the sequence of edits that this path represents in transforming u to v.

7. Provide the dynamic-programming matrix that is needed to efficiently compute the edit-distance between u=paris and v=alice. Circle one of the optimal paths and provide the sequence of edits that this path represents in transforming u to v.

8. Suppose keys 1-5 have respective weights 30,60,10,80,50, and are inserted into an intially empty binary search tree $\mathcal{T}$ in the order 5,2,1,3,4. Determine wac($\mathcal{T}$).

9. Use dynamic programming to determine the binary search tree of minimum weighted-access cost, and whose keys and weights are provided in the previous problem.

10. Let $K(i, j)$ denote the key of the root for the optimal bst that stores keys $i, i+1, \ldots, j$. Devise recursive Java algorithm that takes as inputs K and n (the dimension of K, and hence the number of keys in the tree) that returns an optimal bst. Use the class

```
public class BST
{
    int key;  //key value stored in the root of this BST
    BST left; //the left subtree of the root of this BST
    BST right; //the right subtree of the root fo this BST

    BST(int key,BST left, BST right); //constructor
}
```

11. Determine the running time of the Optimal Binary Search Tree dynamic programming algorithm. Hint: first determine a formula for the number of steps needed to compute $\text{wac}(i, j)$. Then sum over i and j.

12. How many ways are there to fully parenthesize a product of five matrices.

13. What is the multiplication complexity of $A(BC)$ if A has size 2×10, B has size 10×20, and C has size 20×7? Same question for $(AB)C$.

14. Use a dynamic programming algorithm to determine the minimum multiplication complexity of $A_1A_2A_3A_4A_5$, where the dimension sequence is $5, 7, 3, 9, 4, 2$.

15. Let $G = (V, E, c)$ be a directed graph, where
$$V = \{1, 2, 3, 4, 5, 6\}$$
and the directed edges-costs are given by
$$E = \{(1, 5, 1), (2, 1, 1), (2, 4, 2), (3, 2, 2), (3, 6, 8), (4, 1, 4), (4, 5, 3), (5, 2, 7), (6, 2, 5)\}.$$
Provide a geometical representation of graph G.

16. Find all-pairs distances for graph G in the previous problem by performing the Floyd-Warshall Algorithm. Show the matrix sequence $d^0, \ldots, d^6$.

17. Write a recursive algorithm which, on inputs $d^{(0)}$, $d^{(n)}$, i, and j, prints the sequence of vertices for a minimum-cost path from i, to j, $1 \le i, j \le n$, where $d^{(0)}$ is the $n \times n$ cost-adjacency matrix, while $d^{(n)}$ is the final matrix obtained from the Floyd-Warshall algorithm.

18. In the Floyd-Warshall algorithm, give a geometrical argument as to why, when computing matrix d^k, $1 \le k \le n$, neither row k nor column k change from $d^{(k-1)}$ to d^k.

19. Consider a simple path for a directed network that represents a *maximum-cost* path from vertex i to j. Do such paths have the optimal substructure property? Explain.

20. The following is another method for finding distances in a graph $G = (V, E, c)$ between all pairs of vertices. Again, assume the vertex set is $V = \{1, \ldots, n\}$, and let D equal the connection-cost matrix. In other words, D_{ij} is equal to the cost of the edge from vertex i to vertex j, where $D_{ij} = \infty$ if there is no edge from i to j. Thus, D_{ij} is the distance from i to j when restricted to a path of length 1 or less. Now let matrix D^k be the matrix for which D^k_{ij} is the distance from i to j when restricted to paths of length k or less. Then, since $|V| = n$, and the optimal path from i to j is a simple path, it follows that the distance from i to j is equal to D^{n-1}_{ij}, since $n - 1$ is the maximum length of a simple path that can be formed in G. Prove that, for all positive r and s,
$$D^{r+s} = D^r D^s,$$
where the right side represents matrix multiplication, but with multiplication replaced by addition, and addition replaced by the min operator. In particular, we have
$$D^{r+s}_{ij} = \min(D^r_{i1} + D^s_{1j}, \ldots, D^r_{in} + D^s_{nj}).$$

21. Let D^{n-1} be defined as in the previous exercise. Describe how to compute D^{n-1} from D in $\Theta(n^3 \log n)$ steps.

22. Let $G = (V, E, c)$ be a directed graph, where

$$V = \{1, 2, 3, 4, 5, 6\}$$

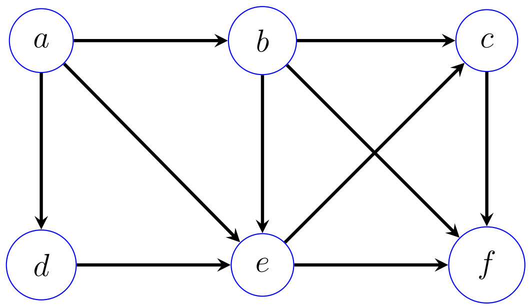
and the directed edges-costs are given by

$$E = \{(1, 5, 1), (2, 1, 1), (2, 4, 2), (3, 2, 2), (3, 6, 8), (4, 1, 4), (4, 5, 3), (5, 2, 7), (6, 2, 5)\}.$$

Provide a geometical representation of graph G. Then use the previous two exercises to find the distances between all pairs of vertices of G by performing a sequence of matrix multiplications. Show the sequence of matrix multiplications.

23. For a directed acyclic graph $G = (V, E)$, let $l(u)$ denote the length of the longest path that begins at vertex u. Provide a dynamic-programming recurrence relation for $l(u)$. Hint: express $l(u)$ in terms of the children of u; i.e., the vertices v for which $(u, v) \in E$.

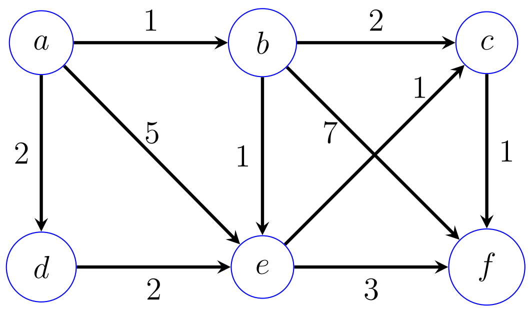
24. Use the dynamic-programming recurrence from the previous exercise to compute the $l(v)$, for each for each vertex v in the following graph.



25. Given an array of n integers, the problem is to determine the longest increasing subsequence of integers in that array. Note: the sequence does not necessarily have to be contiguous. Show how the previous problem can be used to solve this problem, by constructing the appropriate graph. Hint: the vertices of the graph should consist of the n integers.

26. For a weighted directed acyclic graph $G = (V, E, c)$ and source vertex $s \in V$. let $d(v)$ denote the distance from s to v. Provide a dynamic-programming recurrence relation for $d(v)$. Hint: express $d(v)$ in terms of the parents of v; i.e., the vertices u for which $(u, v) \in E$. Note: this is a simpler alternative to Dijkstra's algorithm in case the graph is acyclic.

27. Use the dynamic-programming recurrence from the previous exercise to compute the $d(v)$, for each for each vertex v in the following graph. Assume that a is the source.



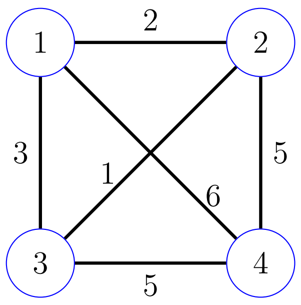
28. A production plant has two assembly lines. Each line has 3 stations. Matrix

$$A = \begin{pmatrix} 3 & 5 & 6 \\ 4 & 2 & 7 \end{pmatrix}$$

is a 2×3 matrix for which a_{ij} gives the processing times for an item that is being assembled at station j of line i. When an item finishes stage $j < 3$ of line i, it has the option of either remaining on line i and proceeding to stage $j+1$, or moving to the other line, and then proceeding to stage $j+1$. If it chooses to stay on the same line, then it requires 0.5 units of time to move to the next station. Otherwise it requires 2 units of time to move to the next station located on the other line. Explain how Exercise 26 may be used to find the least total time (the sum of all processing and transfer times) that is required to process a single item.

29. Given n cities, let C be an $n \times n$ matrix whose C_{ij} entry represents the cost in traveling from city i to city j. Determine a dynamic-programming recurrence relation that will allow one to determine a minimum-cost tour that begins at city 1 and visits every other city at most once. Hint: let $\text{mc}(i, A)$ denote the minimum-cost tour that begins at city i and visits every city in A which is a subset of the cities, and not include i. Determine a recurrence for $\text{mc}(i, A)$.

30. Apply the recurrence from the previous exercise to the following graph. Again, assume the tour begins at 1.



Solutions to Exercises

1. $\pm\sqrt{5}/5$

2. Prove that $f_n = \Theta((\frac{1+\sqrt{5}}{2})^n)$

3.

x_i, c	0	1	2	3	4	5	6	7	8
x_1	0	2	2	2	2	2	2	2	2
x_2	0	2	4	6	6	6	6	6	6
x_3	0	2	4	6	7	9	11	11	11
x_4	0	2	4	6	7	9	11	11	12
x_5	0	2	4	6	7	9	11	11	13
x_6	0	3	5	7	9	10	12	14	14

4.
```
void print_objects(Matrix P, int[] w, int[] p, int i, int c)
{
      //base case
      if(i == 0 || c == 0)
            return;

      if(P[i-1,c] != P[i,c])
      {
            print_objects(P, w, p,i-1,c-w[i]);
            print i;
      }

      print_objects(P, w, p,i-1,c);
}
```

5. 4

6. In matrix, "l" stands for left, "u" for up, and "d" for diagonal.

	λ	s	l	a	c	k	s
λ	0	1,l	2,l	3,l	4,l	5,l	6,l
b	1,u	1,l	2,d	3,d	4,d	5,d	6,d
l	2,u	2,du	1,d	2,l	3,l	4,l	5,l
0	3,u	3,du	2,u	2,d	3,ld	4,ld	5,ld
c	4,u	4,du	3,u	3,du	2,d	3,l	4,l
k	5,u	5,du	4,u	4,du	3,u	2,d	3,l

Optimal transformation: $b \to s$, $o \to a$, $+s$

7. In matrix, "l" stands for left, "u" for up, and "d" for diagonal.

	λ	a	l	i	c	e
λ	0	1,l	2,l	3,l	4,l	5,l
p	1,u	1,d	2,ld	3,ld	4,ld	5,ld
a	2,u	1,d	2,ld	3,ld	4,ld	5,ld
r	3,u	2,u	2,d	3,ld	4,ld	5,ld
i	4,u	3,u	3,du	2,d	3,l	4,l
s	5,u	4,u	4,du	3,u	3,d	4,l

Optimal transformation: $-p$, $r \to l$, $s \to c$, $+e$

8. $\text{wac}(\mathcal{T}) = 50 + (2)(60) + 3(30 + 10) + (4)(80) = 610$

9. Entries of the form $\text{wac}(i, j)/k$ give the optimal weighted access cost , followed by the root k of the corresponding optimal tree.

i/j	1	2	3	4	5
1	30	120/2	140/2	310/2	420/4
2	0	60	80/2	230/4	330/4
3	0	0	10	100/4	200/4
4	0	0	0	80	180/4
5	0	0	0	0	50

The optimal tree root has key 4, while the optimal left sub-tree has root key 2.

10.
```
BST optimal_bst(Matrix K, int i, int j)
{
     if(i==j)
         return new BST(i,null,null)

     int k = K[i,j];
     return new BST(k,optimal_bst(K,i,k-1),optimal_bst(K,k+1,j);
}
```

11. $\Theta(n^3)$. Ciomputing entry (i, j) requires $\Theta(j - i + 1)$ steps. Hence, the running time is proportional to $\sum_{i=1}^{n} \sum_{j=i}^{n} (j - i + 1)$. Moreover, using basic summation formulas (see the exercises from the Big-O lecture), one can show that this sum is $\Theta(n^3)$.

12. 14

13. $A(BC)$: 1540, $(AB)C$: 680

14. Each entry is of the form $m(i, j), k$

i/j	1	2	3	4	5
1	0	105	240/2	273/2	238/1
2	0	0	189	192/2	168/2
3	0	0	0	108	126/3
4	0	0	0	0	72
5	0	0	0	0	0

15. (a, b, c) means the directed edge incident with a and b, and having cost c.

16.

$$d^0 = \begin{pmatrix} 0 & \infty & \infty & \infty & 1 & \infty \\ 1 & 0 & \infty & 2 & \infty & \infty \\ \infty & 2 & 0 & \infty & \infty & 8 \\ 4 & \infty & \infty & 0 & 3 & \infty \\ \infty & 7 & \infty & \infty & 0 & \infty \\ \infty & 5 & \infty & \infty & \infty & 0 \end{pmatrix}$$

$$d^2 = \begin{pmatrix} 0 & \infty & \infty & \infty & 1 & \infty \\ 1 & 0 & \infty & 2 & 2 & \infty \\ 3 & 2 & 0 & 4 & 4 & 8 \\ 4 & \infty & \infty & 0 & 3 & \infty \\ 8 & 7 & \infty & 9 & 0 & \infty \\ 6 & 5 & \infty & 7 & 7 & 0 \end{pmatrix}$$

$d^3 = d^2$ (why?) $d^4 = d^3$.

$$d^5 = \begin{pmatrix} 0 & 8 & \infty & 10 & 1 & \infty \\ 1 & 0 & \infty & 2 & 2 & \infty \\ 3 & 2 & 0 & 4 & 4 & 8 \\ 4 & 10 & \infty & 0 & 3 & \infty \\ 8 & 7 & \infty & 9 & 0 & \infty \\ 6 & 5 & \infty & 7 & 7 & 0 \end{pmatrix}$$

$d^5 = d^6$ (why?)

17.
```
//Note: include_start vertex is needed so that no vertex is
//printed more than once
void print_optimal_path(Matrix d0, Matrix dn, int i, int j,
    bool include_start_vertex)
{
    if(d0[i,j]==dn[i,j]])//optimal path is a direct connection
    {
        if(include_start_vertex)
            print i

        print j;
    }

    //optimal path must pass through intermediate vertex k. Find k
    int k;

    for(k = 1; k <= n; r++)
    {
        if(dn[i,k]+dn[k,j] == dn[i,j])//found k
        {
            print_optimal_path(d0,dn,i,k,true);
            print_optimal_path(d0,dn,k,j,false);
```

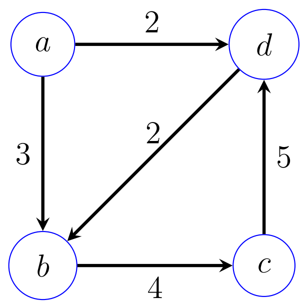


Figure 1: Graph for Problem 20

```
            }
        }
    }
```

18. Row k represents distances from k, while column k represents distances to k. Thus, none of these distances can improve when allowing k to be an intermediate vertex, since the optimal paths either begin or end at k.

19. See the graph below. $P = a, b, c, d$ is a maximum-cost simple path from a to d, but $P^{‘} = a, b$ is *not* a maximum-cost path from a to b. Hence, maximum-cost simple paths do not possess optimal substructures.

20. Let r and s be two positive integers, and consider entry D_{ij}^{r+s}. First suppose $D_{ij}^{r+s} = \infty$. Then there is no path from i to j of length not exceeding $r + s$. Then for each $k = 1, \ldots, n$, we must have either $D_{ik}^r = \infty$, or $D_{kj}^s = \infty$. Otherwise, there would be a path from i to k of length not exceeding r, and a path from k to j of length not exceeding s. Thus, concatenating these two paths yields a path from i to j of length not exceeding $r + s$, which is a contradiction. Hence,

$$\min(D_{i1}^r + D_{1j}^s, \ldots, D_{in}^r + D_{nj}^s) = \infty,$$

which means that the (i, j) entry of $D^r D^s$ is equal to D_{ij}^{r+s}.

Now suppose $D_{ij}^{r+s} < \infty$ is finite. First suppose that D_{ij}^{r+s} is realized by a path of length not exceeding r. In this case $D_{ij}^{r+s} = D_{ij}^r$. Then we must have

$$\min(D_{i1}^r + D_{1j}^s, \ldots, D_{in}^r + D_{nj}^s) = D_{ij}^r + D_{jj}^s = D_{ij}^r + 0 = D_{ij}^r.$$

Otherwise, suppose there is a k for which $D_{ik}^r + D_{kj}^s < D_{ij}^r < \infty$. Then this would imply i) a path from i to k of length not exceeding r with cost D_{ik}^r, and ii) a path from k to j of length not exceeding s with cost D_{kj}^s. Concatenating these two paths makes a path from i to j with cost $D_{ik}^r + D_{kj}^s < D_{ij}^r$, and of length not exceeding $r + s$. But this implies $D_{ij}^{r+s} \leq D_{ik}^r + D_{kj}^s < D_{ij}^r$, which is a contradiction. Hence, the (i, j) entry of $D^r D^s$ is equal to $D_{ij}^r = D_{ij}^{r+s}$.

Finally, suppose $D_{ij}^{r+s} < \infty$ is finite and that D_{ij}^{r+s} is realized by a path P with $|P| = l > r$. Then P can be written as $P_1 P_2$ (i.e. path concatenation), where $|P_1| = r$ and $|P_2| = l - r$. Let vertex k be the ending (respectively, starting) vertex of path P_1 (repsectively, P_2). Then, by the property of optimal subpaths of a minimum-cost path,

$$D_{ij}^{r+s} = \text{cost}(P_1) + \text{cost}(P_2) = D_{ik}^r + D_{kj}^s.$$

In other words, P_1 must be the minimum-cost path from i to k of length not exceeding r, and P_2 must be the minimum-cost path from k to j of length not exceeding s. Moreover, by an argument similar to the one in the previous paragraph, it follows that

$$\min(D^r_{i1} + D^s_{1j}, \ldots, D^r_{in} + D^s_{nj}) = D^r_{ik} + D^s_{kj}.$$

Hence, the (i, j) entry of $D^r D^s$ is equal to $D^r_{ik} + D^s_{kj} = D^{r+s}_{ij}$.

Therefore, since r, s, i, and j were arbitrary, $D^{r+s} = D^r D^s$.

21. A single matrix multiplication requires $\Theta(n^3)$ steps. Also, if $p \geq n-1$, then we must have $D^p = D^{n-1}$, since a minimum-cost path must be simple, and the longest simple path can have a length of at most $n-1$. Finally, let k be the first integer for which $2^k \geq n-1$. Then computing

$$D^2 = D \cdot D,\ D^4 = D^2 \cdot D^2, \ldots, D^{n-1} = D^{2^k} = D^{2^{k-1}} \cdot D^{2^{k-1}}$$

requires k multiplications. But since $2^k \leq 2(n-1)$, it follows that

$$k \leq \log(2(n-1)) = \log 2 + \log(n-1) = \Theta(\log n).$$

Therefore, D^{n-1} can be computed with $\Theta(\log n)$ multiplications, each requireing $\Theta(n^3)$ steps, for a total of $\Theta(n^3 \log n)$ steps.

22.

$$D = \begin{pmatrix} 0 & \infty & \infty & \infty & 1 & \infty \\ 1 & 0 & \infty & 2 & \infty & \infty \\ \infty & 2 & 0 & \infty & \infty & 8 \\ 4 & \infty & \infty & 0 & 3 & \infty \\ \infty & 7 & \infty & \infty & 0 & \infty \\ \infty & 5 & \infty & \infty & \infty & 0 \end{pmatrix}$$

$$D^2 = D \cdot D = \begin{pmatrix} 0 & 8 & \infty & \infty & 1 & \infty \\ 1 & 0 & \infty & 2 & 2 & \infty \\ 3 & 2 & 0 & 4 & \infty & 8 \\ 4 & 10 & \infty & 0 & 3 & \infty \\ 8 & 7 & \infty & 9 & 0 & \infty \\ 6 & 5 & \infty & 7 & \infty & 0 \end{pmatrix}$$

For example $D^2_{42} = \min(4 + \infty, \infty + 0, \infty + 2, 0 + \infty, 3 + 7, \infty + 5) = 10$

$$D^4 = D^2 \cdot D^2 = \begin{pmatrix} 0 & 8 & \infty & 10 & 1 & \infty \\ 1 & 0 & \infty & 2 & 2 & \infty \\ 3 & 2 & 0 & 4 & 4 & 8 \\ 4 & 10 & \infty & 0 & 3 & \infty \\ 8 & 7 & \infty & 9 & 0 & \infty \\ 6 & 5 & \infty & 7 & 7 & 0 \end{pmatrix}$$

For example $D^4_{35} = \min(3+1, 2+2, 0+\infty, 0+\infty, 3+3, 8+\infty) = 4$. Finally, $D^5 = D^4 D = D^4$ (why?).

23.
$$l(u) = \begin{cases} 0 & \text{if } u \text{ has no children} \\ \max\limits_{(u,v)\in E} (1 + l(v)) & \text{otherwise} \end{cases}$$

24. Start with vertices that have no children, then proceed to the next vertex u for which $l(v)$ has already been computed, for each child v of u.

$$l(f) = 0.$$

$$l(c) = 1 + l(f) = 1.$$
$$l(e) = 1 + \max(l(c), l(f)) = 2.$$
$$l(b) = 1 + \max(l(c), l(e), l(f)) = 3.$$
$$l(d) = 1 + l(e) = 3.$$
$$l(a) = 1 + \max(l(b), l(d), l(e)) = 4.$$

25. Define the directed graph $G = (V, E)$ so that V is the set of integers and $(m, n) \in E$ iff $m \leq n$ and m appears before n in the array. Then the longest increasing subsequence corresponds with the longest path of G.

26.
$$d(v) = \begin{cases} 0 & \text{if } v = s \\ \min\limits_{(u,v)\in E} (d(u) + c(u, v)) & \text{otherwise} \end{cases}$$

27. Start with the source, then proceed to the next vertex v for which $d(u)$ has already been computed, for each parent u of v.

$$d(a) = 0.$$
$$d(b) = d(a) + 1 = 1.$$
$$d(d) = d(a) + 2 = 2.$$
$$d(e) = \min(d(a) + 5, d(b) + 1, d(d) + 2) = 2.$$
$$d(c) = \min(d(b) + 2, d(e) + 1) = 3.$$
$$d(f) = \min(d(b) + 7, d(c) + 1, d(e) + 3) = 4.$$

28. Define a weighted directed acyclic graph as follows for each station S_{ij} define nodes e_{ij} and f_{ij}, where e_{ij} denotes the entry point to S_{ij}, and f_{ij} denotes its finish point. Add the edge (e_{ij}, f_{ij}) and give it weight a_{ij}, the time needed to process an item at S_{ij}. Next, for each node f_{ij}, $j < 3$, add the edges $(f_{ij}, e_{i(j+1)}, 0.5)$, $(f_{ij}, e_{i'(j+1)}, 2)$, where $i' = 2$ if $i = 1$, and $i' = 1$ if $i = 2$. Finally, add source vertex s and connect it to both e_{11} and e_{21} using 0-weight edges. The graph for the production-plant assembly lines is shown below. The least processing time can now be found by computing the minimum of $d(f_{13})$ and $d(f_{23})$.

$$\begin{tikzcd}
s \arrow[r, "0"] \arrow[dr, "0"'] & e_{11} \arrow[r, "3"] & f_{11} \arrow[r, "0.5"] \arrow[dr, "2"'] & e_{12} \arrow[r, "5"] & f_{12} \arrow[r, "0.5"] \arrow[dr, "2"'] & e_{13} \arrow[r, "6"] & f_{13} \\
 & e_{21} \arrow[r, "4"] & f_{21} \arrow[r, "0.5"] \arrow[ur, "2"] & e_{22} \arrow[r, "2"] & f_{22} \arrow[r, "0.5"] \arrow[ur, "2"] & e_{23} \arrow[r, "7"] & f_{23}
\end{tikzcd}$$

29. Let $\mathrm{mc}(i, A)$ denote the minimum-cost tour that begins at city i and visits every city in A, where A is a subset of cities that does not include i. Then

$$\mathrm{mc}(i, A) = \begin{cases} 0 & \text{if } A = \emptyset \\ C_{ij} & \text{if } A = \{j\} \\ \min\limits_{j \in A}(C_{ij} + \mathrm{mc}(j, A - \{j\}) & \text{otherwise} \end{cases}$$

30. Start with $\mathrm{mc}(1, \{2, 3, 4\})$ and proceed to compute other mc values as needed.

$$\mathrm{mc}(1, \{2, 3, 4\}) = \min(2 + \mathrm{mc}(2, \{3, 4\}), 3 + \mathrm{mc}(3, \{2, 4\}), 6 + \mathrm{mc}(4, \{2, 3\})).$$

$$\mathrm{mc}(2, \{3, 4\}) = \min(1 + \mathrm{mc}(3, \{4\}), 5 + \mathrm{mc}(4, \{3\})) = \min(1 + 5, 5 + 5) = 6.$$

$$\mathrm{mc}(3, \{2, 4\}) = \min(1 + \mathrm{mc}(2, \{4\}), 5 + \mathrm{mc}(4, \{2\})) = \min(1 + 5, 5 + 5) = 6.$$

$$\mathrm{mc}(4, \{2, 3\}) = \min(5 + \mathrm{mc}(2, \{3\}), 5 + \mathrm{mc}(3, \{2\})) = \min(5 + 1, 5 + 1) = 6.$$

Therefore,

$$\mathrm{mc}(1, \{2, 3, 4\}) = \min(2 + \mathrm{mc}(2, \{3, 4\}), 3 + \mathrm{mc}(3, \{2, 4\}), 6 + \mathrm{mc}(4, \{2, 3\})) =$$

$$\min(2 + 6, 3 + 6, 6 + 6) = 8.$$