Algorithms Analysis & Design
Dynamic Programming

Aditya Harikrish
2020111009

Contents

1 Introduction **3**
1.1 Example . . . 3

2 Fibonacci Sequence **4**
2.1 Without Memoisation: The Naive Solution . . . 4
2.2 With Memoisation . . . 4

3 Frog Jump **6**
3.1 2 Steps . . . 6
3.2 K Steps . . . 7

4 Longest Increasing Subsequence (LIS) **8**
4.1 Version 1 - $O(n^2)$ . . . 8
4.1.1 Time Complexity . . . 9
4.2 Version 2 - $O(n \log n)$ . . . 9
4.2.1 Time complexity . . . 11

5 Minimum Coins **11**
5.1 Without Memoisation . . . 11
5.2 With Memoisation . . . 12

6 0-1 Knapsack **13**
6.1 A Greedy Approach . . . 13
6.1.1 An Offshoot: Fractional Knapsack . . . 14
6.2 0-1 Knapsack: With Repetition . . . 14
6.3 0-1 Knapsack: Without Repetition . . . 15

7 Longest Common Subsequence (LCS) **15**
7.1 First Observation . . . 16
7.2 Second Observation . . . 16
7.3 Algorithm . . . 16
7.4 Time Complexity . . . 16
7.5 C++ Implementation . . . 16

8 Shortest Common Supersequence (SCS) 17
8.1 Algorithm . . . 17
8.2 Time Complexity . . . 17

9 Tower of Hanoi 18
9.1 Examples . . . 18
9.2 Minimum Number of Moves Needed . . . 19
9.3 Algorithm . . . 20

10 Bellman-Ford Algorithm 20
10.1 Pseudocode . . . 21
10.2 Time Complexity . . . 21

11 Floyd-Warshall Algorithm 21
11.1 Pseudocode . . . 22
11.2 C++ Implementation . . . 22
11.3 Time Complexity . . . 23

1 Introduction

Dynamic programming is a technique to solve computational problems by breaking them down into smaller and simpler subproblems. Problems that can be broken down in this manner are said to have the 'optimal substructure property'.

In general, we can split dynamic programming problems into three steps, of which the first two need not be followed in any particular order. They are as follows.

1. The base case. These are some of the already known cases whose answers are known beforehand. Base cases are useful for stopping recursive call stacks and providing some information that allows us to base other solutions off of.
2. The transition function. This is the function which translates a problem into a set smaller subproblems.
3. The final answer. We need to extract the final answer from all the computation that has been performed.

1.1 Example

The Problem: Given a natural number n, find the number of different ways to write it as a sum of 1, 3, and 5.

For example, for $n = 6$, the answer is 7.

$$\begin{aligned} 6 &= 1+1+1+1+1+1 \\ &= 1+1+1+3 \\ &= 1+1+3+1 \\ &= 1+3+1+1 \\ &= 3+1+1+1 \\ &= 1+5 \\ &= 5+1 \end{aligned}$$

Before defining the problem as a set of subproblems, let us define S_n to be the number of ways to write n as a sum of 1, 3, and 5.

Let $n = a_1 + \ldots a_m$ be a solution. Notice that if $a_m = 5$, then $a_1 + \ldots a_{m-1} = n - 5$. Therefore, there are S_{n-5} solutions that end in $a_m = 5$.

The sum can only end in three 1, 3, and 5. Therefore,

$$S_n = S_{n-1} + S_{n-3} + S_{n-5}. \tag{1}$$

The recurrence must end in some base cases. Here,

- $S_0 = 1$
- $S_i = 0$ for $i < 0$

The pseudocode is given below.

```
function S(n):
    if n < 0: return 0
    if n = 0: return 1
    return S(n-1) + S(n-3) + S(n-5)
```

2 Fibonacci Sequence

One of the most popular sequences, the Fibonacci sequence is defined as follows.

1. The first element is 0 and the second element is 1.
2. Each subsequent element is the sum of the previous two elements.

F_i denotes the ith Fibonacci number. From the definition,

$$F_i = \begin{cases} 0 & \text{if } i = 0 \\ 1 & \text{if } i = 1 \\ F_{i-1} + F_{i-2} & \text{if } i > 1 \end{cases} \tag{2}$$

The Problem: Given a positive integer n, find the Fibonacci sequence from F_0 to F_n, i.e. the first $n + 1$ elements.

2.1 Without Memoisation: The Naive Solution

The simplest solution is to directly implement the definition using recursion, shown below.

```
uint64_t Without_Memoisation(std::vector<uint64_t>& FibArray, uint64_t
    n) {
    if (n <= 1)
        FibArray[n] = n;
    else
        FibArray[n] = Without_Memoisation(FibArray, n - 1) +
    Without_Memoisation(FibArray, n - 2);
    return FibArray[n];
}
```

However, this solution has a serious drawback - it runs in exponential time. For all fib(i) where $i \geq 2$, two functions are called - fib($i-1$) and fib($i-2$). And each of these called functions call two more unless they're the first two elements in the sequence. This generates a binary tree of recursive calls, with almost 2^n recursive calls.

Refer to Figure 2.1 for an example.

2.2 With Memoisation

Memoisation is a neat little trick that is widely used to speed up algorithms. Simply put, memoisation is the technique of storing computed results so that the computation need not take place again to determine the same result.

For the Fibonacci sequence, let us create an array called 'FibArray' where

$$\text{FibArray}[i] = F_i.$$

When computing the first $n + 1$ elements, initially, FibArray is empty. Whenever the function computes the ith Fibonacci number, it stores the value in FibArray[i]. Following this, whenever another function needs the value of F_i, the value is returned

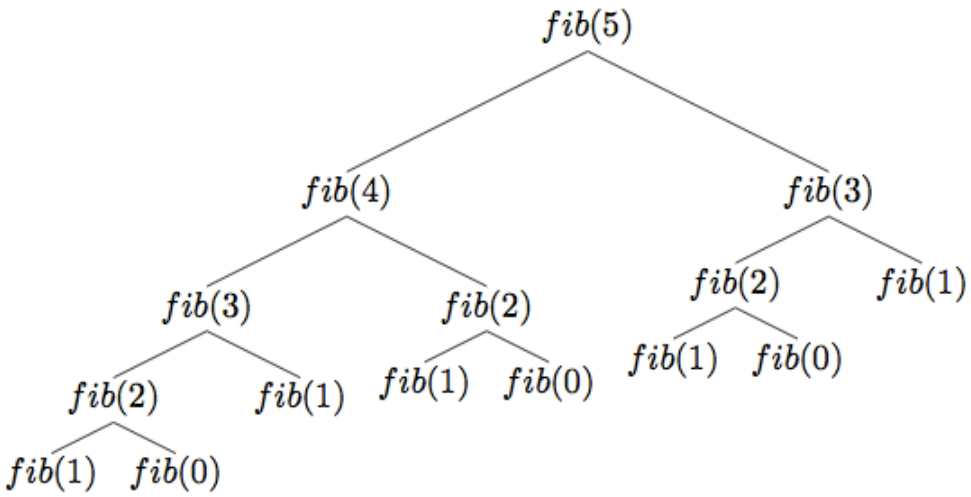


Figure 1: The call tree for fib(5) (without memoisation)

from FibArray[i] in $O(1)$, which is a huge improvement the earlier exponential time complexity.

Here, each function call from fib(0) to fib(n) computes the value of its corresponding Fibonacci number only once. Thus, all but one instance of each of these function calls occur in $O(1)$. The time complexity is $O(n)$. Refer to Figure 2.2 for a comparison in the performace of the two solutions to the Fibonacci Sequence problem.

```
std::vector<uint64_t> FibArray(n + 1, 4);
uint64_t With_Memoisation(std::vector<uint64_t>& FibArray, uint64_t n) {
    // The default value to fill the array is 4, because 4 is not a Fibonacci number
    // Checking if the value is present in the memo
    if (FibArray[n] != 4) return FibArray[n];

    // Value not present in memo at this point
    if (n <= 1)
        FibArray[n] = n;
    else
        FibArray[n] = With_Memoisation(FibArray, n - 1) + With_Memoisation(FibArray, n - 2);
    return FibArray[n];
}
```

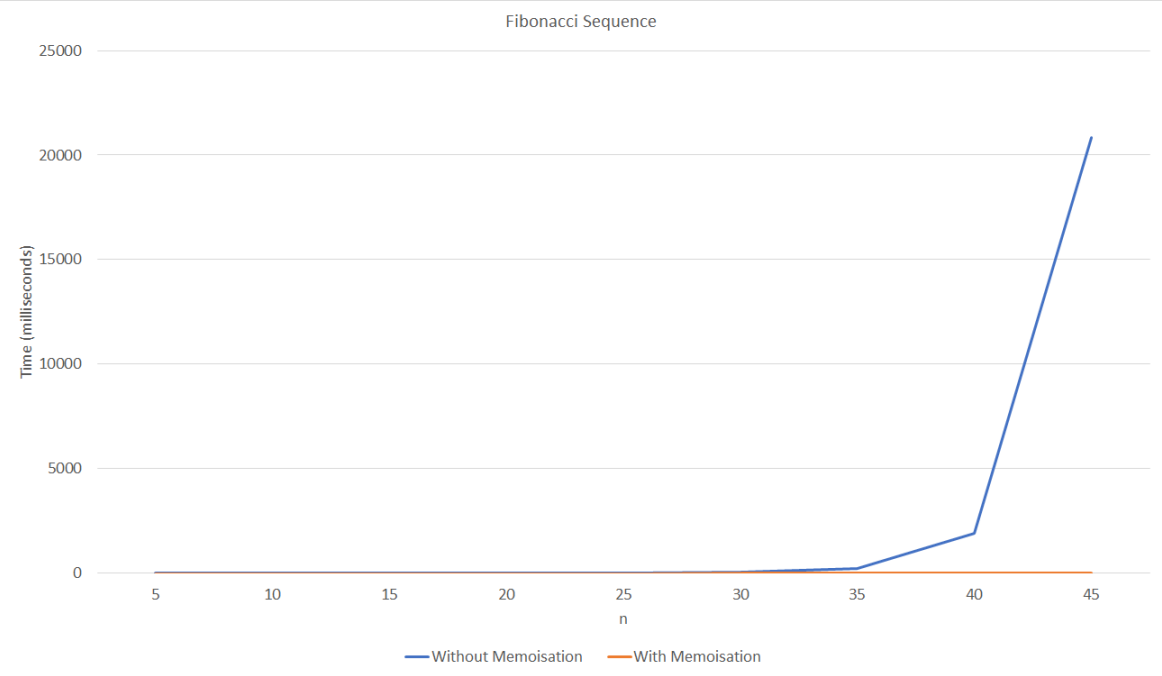


Figure 2: Time taken to calculate the first $n+1$ elements of the Fibonacci sequence with and without memoisation

3 Frog Jump

The Problem: There are n stones in a line (numbered $0, 1, \ldots, n-1$) and a frog is on Stone 0. Each stone has a height. The ith stone has height h_i. The cost of jumping from Stone i to Stone j is $|h_j - h_i|$. Find the minimum cost for the frog to go from Stone 0 to Stone $n-1$.

3.1 2 Steps

In this version of the problem, if the frog is on Stone i, it can only jump to Stone $i+1$ or Stone $i+2$ in a single jump.

Let us create an array called 'dp' such that

$$\text{dp}[i] = \text{minimum cost for the frog to go from Stone 0 to Stone } i-1. \tag{3}$$

Transition function. We notice that the ith stone can be reached from either the $(i-1)$th stone or the $(i-2)$th stone (for $i \geq 2$). That means that dp$[i]$ is the minimum of these two cases.

$$\text{dp}[i] = \min\{\text{dp}[i-1] + |h_i - h_{i-1}|, \text{dp}[i-2] + |h_i - h_{i-2}|\} \text{ when } i \geq 2 \tag{4}$$

Base cases. There are at most two stones which do not have two preceding stones - Stone 0 and Stone 1. Thus, we shall manually define dp[0] and dp[1] beforehand.

$$\text{dp}[0] = 0 \text{ as the frog starts at Stone 0, and} \tag{5}$$

$$\text{dp}[1] = |h_1 - h_0|. \tag{6}$$

Final answer. From eq. (3), our final answer is dp$[n-1]$.

```
#include <iostream>
#include <vector>
int main(void) {
    int n;
    std::cin >> n;
    std::vector<int> h(n), dp(std::max(n, 2));
    for (auto &it : h) std::cin >> it;

    dp[0] = 0;
    dp[1] = abs(h[1] - h[0]);

    for (int i = 2; i < n; ++i) {
        dp[i] = std::min(dp[i - 1] + abs(h[i] - h[i - 1]),
                         dp[i - 2] + abs(h[i] - h[i - 2]));
    }

    std::cout << dp[n - 1] << "\n";

    return 0;
}
```

3.2 K Steps

In this version of the problem, if the frog is on Stone i, it has k different stones which it can jump to in one go: Stones $i+1$, $i+2$, . . . ,$i+k$. The solution is just an extension and a generalisation of section 3.1. We use the same definition for the dp array as shown in eq. (3).

Transition function. The ith stone can be reached directly from any one of Stones $i-1$, $i-2$, . . . , $i-k$ (for $i \geq k$). That means that dp$[i]$ is the minimum of these k cases.

$$\text{dp}[i] = \min_{j=1,2,\ldots,n} \{\text{dp}[i-j] + |h_i - h_{i-j}|\} \text{ when } i \geq k \tag{7}$$

Base cases. There are at most k stones which do not have k preceding stones - Stones 0 through $k-1$. Thus, we shall manually define dp[0], dp[1],...,dp$[k-1]$ beforehand.

$$\text{dp}[0] = 0 \text{ as the frog starts at Stone 0, and} \tag{8}$$

$$\text{dp}[i] = \min_{j=0,1,\ldots,i-1} \{dp[j] + |h_i - h_j|\} \text{ when } i < k. \tag{9}$$

Final answer. From eq. (3), our final answer is dp$[n-1]$.

```
int MinCostDP(const std::vector<int>& h, int k) {
    int n = h.size();
    std::vector<int> dp(n);
    dp[0] = 0;

    for (int i = 1; i < n; ++i) {
        int curr_min = dp[i - 1] + abs(h[i] - h[i - 1]);

```

```
9           for (int j = i - 2; j >= i - k && j >= 0; --j) {
10              curr_min = std::min(curr_min, dp[j] + abs(h[i] - h[j]));
11          }
12
13          dp[i] = curr_min;
14      }
15
16      return dp[n - 1];
17  }
```

4 Longest Increasing Subsequence (LIS)

A subsequence L of a sequence S is a sequence that can be obtained by deleting some or none of the elements in S, without changing the order in which the elements appear.

> **The Problem:** Given a sequence of n numbers, $a_0, a_1, \ldots, a_{n-1}$, find the longest subsequence of this sequence that is strictly increasing. If there are multiple such subsequences, find any of them.

The brute-force approach would be to check every possible subsequence, but that would take exponential time as there are 2^n possible subsequences in a sequence of n elements. Thus, we shall not discuss this approach.

4.1 Version 1 - $O(n^2)$

Let us define an array called 'dp' as follows.

$$\text{dp}[i] = \text{length of the LIS ending with } a_i. \tag{10}$$

Let us also define an array called 'prev' such that prev$[i]$ is the index of the element (in the sequence $a_0, \ldots, a_{n-1}$) immediately preceding a_i in the longest increasing subsequence that a_i is a part of.

Base cases. We observe that dp$[i]$ must be at least one as the subsequence that contains only a_i is also increasing. So, we initialise dp$[i] = 1$ and prev$[i] = -1$ (since we have no information about any LIS in the beginning) for all $i = 0, 1, \ldots, n-1$.

Transition Function. Observe that for some $j < i$, if

1. $a_i > a_j$ (i.e., it is possible for a_i to succeed a_j in an increasing subsequence), and
2. dp$[i]$ ¡ dp$[j] + 1$,

then we set

- dp$[i] =$ dp$[j] + 1$, and
- prev$[i] = j$

We also keep track of the last index of the longest increasing subsequence known so far (in the variable 'last_index_of_LIS') along with its length (in the variable 'max_length').

Final answer. The length of the LIS is 'max_length'. To find the LIS itself, start from i = last_index_of_LIS and go to prev[i] and update i to prev[i] until prev[i] is -1. Note that this gives the LIS in reverse order, so you might prefer to reverse this list.

```cpp
int LIS::lengthOfLIS_v1(const std::vector<int>& a) {
    if (a.size() < 1) return 0;
    std::vector<int> dp(a.size(), 1);    /* DP[i] = length of the LIS ending with a[i] */
    std::vector<int> prev(a.size(), -1); /* Only needed to find the LIS itself in addition to its length */
    int max_length = 1;
    int last_index_of_LIS = -1;
    for (int i = 1; i < a.size(); ++i) {
        for (int j = 0; j < i; ++j) {
            if (a[i] > a[j] && dp[j] + 1 > dp[i]) {
                dp[i] = dp[j] + 1;
                prev[i] = j;
            }
        }
        if (dp[i] > max_length) {
            max_length = dp[i];
            last_index_of_LIS = i;
        }
    }

    /* Print the LIS */
    std::vector<int> LIS_in_reverse;
    for (auto i = last_index_of_LIS; i >= 0;) {
        LIS_in_reverse.push_back(a[i]);
        i = prev[i];
    }
    std::cout << "LIS: ";
    for (auto it = LIS_in_reverse.rbegin(); it != LIS_in_reverse.rend(); ++it) {
        std::cout << *it << " ";
    }
    std::cout << "\n";
    return max_length;  // Length of the LIS
}
```

4.1.1 Time Complexity

We iterate through each element in the array a. For each element, we go through each of the previous elements, in $O(1)$ for each of the previous elements. Thus, the overall time complexity is $O(n^2)$.

4.2 Version 2 - $O(n \log n)$

Let use create an array called 's' defined as follows.

$$s[i] = \text{ index of the smallest integer that ends an increasing sequence of length } i. \tag{11}$$

We define an array called '*parent*' such that $parent[i]$ is the index of the previous element in the LIS containing a_i. Now, iterate through the sequence. Let the index of the current iteration be i.

1. If a_i is greater than the last element in s, then this means that we have found a larger increasing sequence than the one currently known and thus, we append a_i to s.

2. Otherwise, we find the smallest element in whose index is in s which is greater than or equal to a_i and change it to a_i. Notice that this element can be found using binary search in $O(n \log n)$ because s is always sorted.

The length of the LIS is simply the length of s. To find the LIS itself, we update the *parent* array as follows:

1. If a_i is greater than the last element in s, then $parent[i] =$ the last element in s since the parent of the new element is the previously last element.

2. Otherwise, using the the index of the smallest element found using binary search earlier, (say '*index*'), we set $parent[i] = s[index - 1]$.

```
int LIS::lengthOfLIS_v2(const std::vector<int>& a) {
    if (a.size() < 1) return 0;
    std::vector<int> s; /* s[i] = index of the smallest integer that
   ends an increasing sequence of length i */
    s.push_back(0);
    std::vector<int> parent(a.size(), -1);
    for (int i = 1; i < a.size(); ++i) {
        if (a[i] > a[s.back()]) {
            parent[i] = s.back();
            s.push_back(i);
        } else {
            /* Binary search in s for the smallest integer >= a[i] */
            int l = 0, r = s.size() - 1;
            int mid = l + (r - l) / 2;
            int index = r; /* Index of the target in s */
            while (l <= r) {
                mid = l + (r - l) / 2;
                if (a[s[mid]] >= a[i]) {
                    index = std::min(index, mid);
                    r = mid - 1;
                } else {
                    l = mid + 1;
                }
            }
            if (index - 1 >= 0)
                parent[i] = s[index - 1];
            else
                parent[i] = -1;
            s[index] = i;
        }
    }

    /* Print the LIS */
    std::vector<int> LIS_in_reverse;
```

```
    for (int i = s.back(); i >= 0;) {
        LIS_in_reverse.push_back(a[i]);
        i = parent[i];
    }
    std::cout << "LIS: ";
    for (auto it = LIS_in_reverse.rbegin(); it != LIS_in_reverse.rend
    (); ++it) {
        std::cout << *it << " ";
    }
    std::cout << "\n";

    return s.size();  // Length of the LIS
}
```

4.2.1 Time complexity

We iterate through each of the n elements in the sequence, and perform one binary search that takes $O(\log n)$ time. Thus, the overall time complexity is $O(n \log n)$.

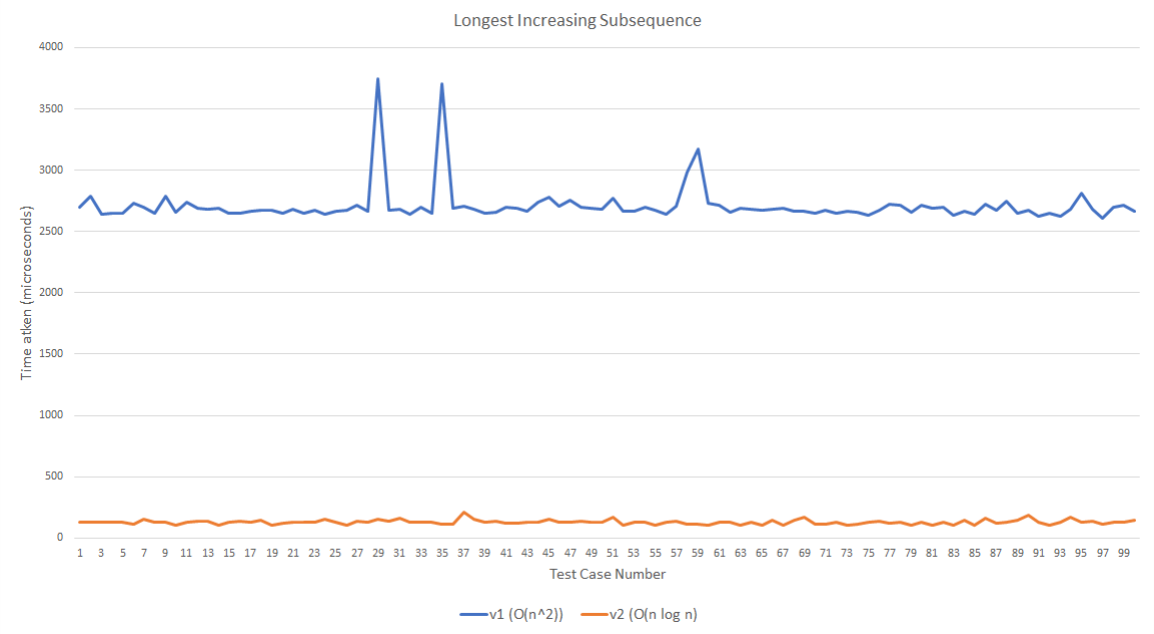


Figure 3: LIS: time taken for $n = 1000$ over 100 test cases

5 Minimum Coins

5.1 Without Memoisation

```
int MinCoins::WithoutMemoisation(int n, const std::vector<int>& denom)
    {
    if (n == 0) return 0;
    int min = INT_MAX;
    for (auto& denomination : denom) {
        if (n - denomination >= 0) {
```

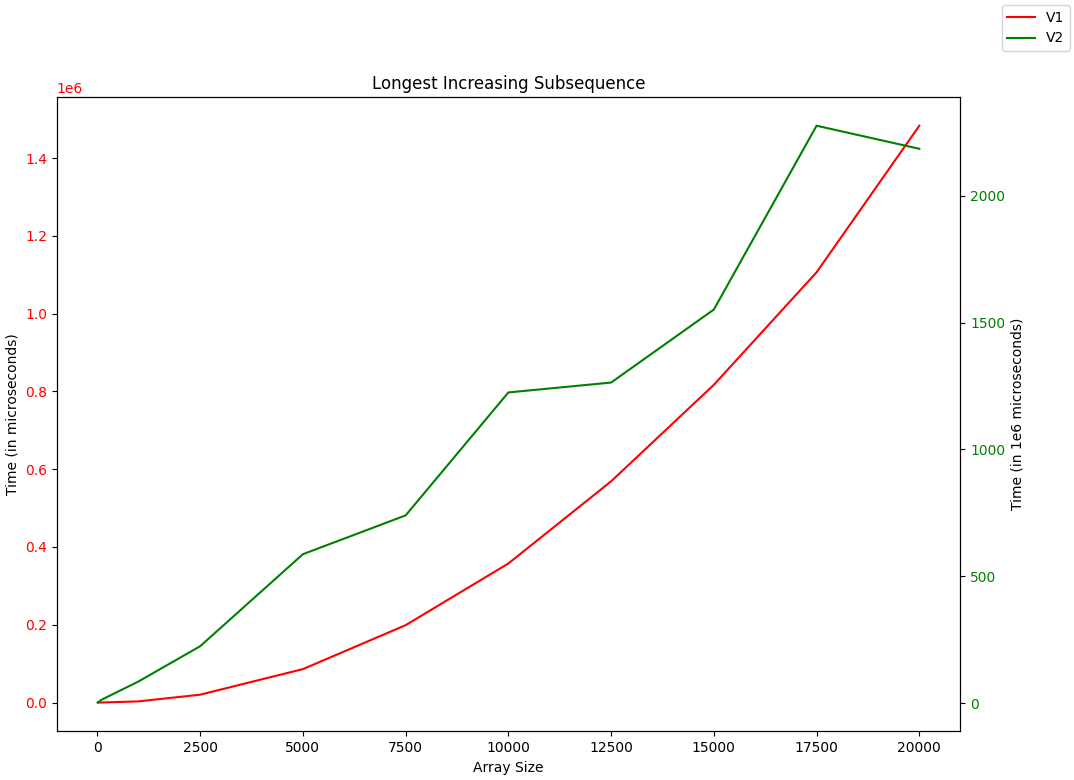


Figure 4: LIS: time taken, for different values of n

```
            min = std::min(min, 1 + WithoutMemoisation(n -
    denomination, denom));
        }
    }
    return min;
}
```

5.2 With Memoisation

```
int MinCoins::WithMemoisationHelper(int n, const std::vector<int>&
    denom, std::vector<int>& CoinMemo) {
    if (CoinMemo[n] != -1) return CoinMemo[n];
    int min = INT_MAX;
    for (auto& denomination : denom) {
        if (n - denomination >= 0) {
            min = std::min(min, 1 + WithMemoisationHelper(n -
    denomination, denom, CoinMemo));
        }
    }
    CoinMemo[n] = min;
    return min;
}

int MinCoins::WithMemoisation(int n, const std::vector<int>& denom) {
    std::vector<int> CoinMemo(n + 1, -1);
    CoinMemo[0] = 0;
```

```
16      return WithMemoisationHelper(n, denom, CoinMemo);
17  }
```

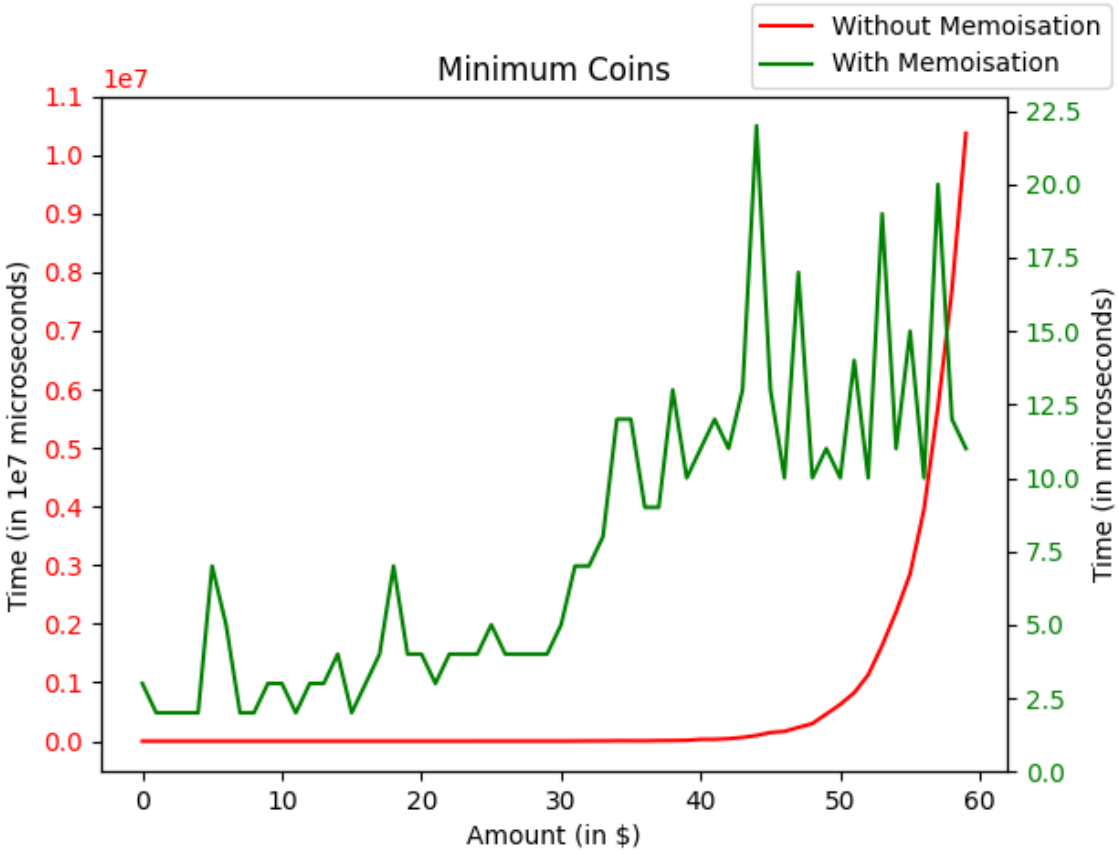


Figure 5: Time taken to calculate the minimum number of coins needed with and without memoisation for the denominations 1, 5, 9 and 23

6 0-1 Knapsack

The Problem: Given a knapsack with a maximum weight capacity W and a set of N items, each with its own weight w and value v, find the maximum value of the items that can be fit into the knapsack. Each item must either be chosen or not chosen, hence this problem is called 0-1 knapsack problem.

6.1 A Greedy Approach

One of the first few approaches that one might try to solve the 0-1 Knapsack problem is the greedy approach—to sort the items in decreasing order of $\frac{value}{weight}$ and choose items with the highest $\frac{value}{weight}$ ratios. However, this approach is incorrect. Consider the following counterexample.

$$W = 10,\ N = 3$$

S. No.	0	1	2
w	5	5	8
v	4	4	7
$\frac{v}{w}$	0.8	0.8	0.875

Using the greedy method, we would pick only item 2, as it has the highest $\frac{v}{w}$ ratio and might seem like the most efficient use of the available space at first glance. However, picking items 1 and 2 instead would give us a total value of 8, which is higher. Hence, the greedy approach is faulty.

6.1.1 An Offshoot: Fractional Knapsack

In this version of the problem, we are allowed to take fractional quantities of each item. That is, if one of the items was a kilogram of rice, then we'd be allowed to take anywhere from no rice to a few grams to the entire kilogram.

The greedy approach is a valid solution for the fractional knapsack problem. As the issue of space efficiency does not crop up anymore—since fractional items are allowed, the most efficient way to pack the knapsack will always be to fill it with items in descending order of v/w.

6.2 0-1 Knapsack: With Repetition

In this version of the problem, we are allowed to pick any number of each item. Let us create an array called 'm' defined as

$$m_w = \text{highest value possible with a knapsack of capacity } w. \tag{12}$$

For each item i and a knapsack of capacity w, we have two possibilities: either the optimal solution corresponding to m_w contains item i or it doesn't. If the optimal solution corresponding to m_w contains item i, then if we remove item i from this set of items, we are left with the optimal solution to a knapsack with capacity $w - w_i$. This implies that m_w is the maximum of at most N possibilities—one for including / excluding each of the N items in the optimal solution. Thus, the time complexity is $O(NW)$.

```
int WithRepetition(int N, int W, const std::vector<int>& w, const std
    ::vector<int>& v) {
    std::vector<int> m(N, 0);
    m[0] = 0;
    for (int capacity = 1; capacity <= W; ++capacity) {
        for (int i = 0; i < w.size(); ++i) {
            if (w[i] <= capacity) {
                m[capacity] = std::max(m[capacity], m[capacity - w[i]
    + v[i]]);
            }
        }
    }
}
```

6.3 0-1 Knapsack: Without Repetition

In this version of the problem, we are only allowed to pick one of each item.

Let us define $m(i, w)$ to be the maximum value that can be attained with the first i items and a knapsack with capacity w. We need to find $m(N, W)$.

- $m(0, w) = 0 \quad \forall w$ where $0 \leq w \leq W$.
- $m(i, 0) = 0 \quad \forall i$ where $0 \leq i < N$ (assuming zero-based indexing).
- $m(i, w) = m(i-1, w) \quad$ if $w_i > w$ (that is, it's not possible to fit in the ith item)
- If $w_i \leq w$, there there are 2 possibilities. Either an optimal solution contains the ith item or it doesn't. If it doesn't, then $m(i, w) = m(i-1, w)$. If it does, $m(i, w) = m(i-1, w-w_i) + v_i$. To find out which of these possibilities is optimal, we simply take their maximum.

$$m(i, w) = \max\{m(i-1, w), m(i-1, w-w_i) + v_i\}$$

The C++ program of the above algorithm is as follows.

```
int KnapsackWithoutRepetition(int N, int W, const std::vector<int>& w,
    const std::vector<int>& v) {
  std::vector<std::vector<int>> m(N + 1, std::vector<int>(W + 1));
  for (int i = 0; i <= N; ++i) {
    for (int j = 0; j <= W; ++j) {
      if (i == 0 || j == 0)
        m[i][j] = 0;
      else if (j >= w[i-1])
        m[i][j] = std::max(m[i-1][j-w[i-1]] + v[i-1], m[i-1][j]);
      else
        m[i][j] = m[i-1][j];
    }
  }
  return m[N][W];
}
```

7 Longest Common Subsequence (LCS)

The Problem: Given two sequences, find the longest sequence that is a subsequence of both of these sequences.

We divide the LCS problem into subproblems that are simpler. That is, LCS has the optimal substructure property.

Definition 7.1 (Prefix)**.** The prefix s_n of a string s is defined as the first n characters of s.

7.1 First Observation

For any character A,

$$LCS(X+A,\ Y+A) = LCS(X,\ Y)+A,$$

where + denotes concatenation. For example, *LCS('window', 'widow') = LCS('windo', 'wido') + 'w'.*

7.2 Second Observation

If A and B are different characters, then $LCS(X+A,\ Y+B)$ is maximal-length string of the set $\{LCS(X+A,\ Y),\ LCS(X,\ Y+B)\}$. If both have the same length, then any one of the two may be chosen.

7.3 Algorithm

From the two observations above, we come to the conclusion below.

$$LCS(X_i,\ Y_i) = \begin{cases} \phi & \text{if } i = 0 \text{ or } j = 0 \\ LCS(X_{i\text{-}1},\ Y_{j\text{-}1})+x_i & \text{if } i, j > 0 \text{ and } x_\text{i} = y_\text{j} \\ \text{maximal length of } \{LCS(X_{i\text{-}1},\ Y_j), LCS(X_i,\ Y_{j\text{-}1})\} & \text{Otherwise} \end{cases} \tag{13}$$

Initially, i and j are the lengths of the strings X and Y respectively.

A similar algorithm could also be written using suffixes.

7.4 Time Complexity

For only two sequences, the time complexity is at worst quadratic in the length of the longer sequence (when no pair of characters of the two seqeunces matches) and at best linear (when the two sequences are identical). Mathematically,

$$T(n) = O(n^2),\ \text{and}$$
$$T(n) = \Omega(n)$$

where n is the length of the longer sequence of the two given ones.

In general, in order to find the LCS of k sequences, the problem becomes NP-hard and the time complexity becomes $O(n^k)$, where n is the length of the longest sequence of the k given sequences.

7.5 C++ Implementation

```
#include <iostream>
#include <string>

std::string LCS(const std::string& a, const std::string& b, int i, int
    j) {
    if (i == -1 || j == -1)
        return "";
    else if (i == j) {
```

```
8           return LCS(a, b, i - 1, j - 1) + a[i];
9       } else {
10          std::string x = LCS(a, b, i - 1, j);
11          std::string y = LCS(a, b, i, j - 1);
12          return x.size() > y.size() ? x : y;
13      }
14  }
15
16  int main(void) {
17      std::string a, b;
18      std::cin >> a >> b;
19
20      std::cout << "LCS: " << LCS(a, b, a.size(), b.size());
21
22      return 0;
23  }
```

8 Shortest Common Supersequence (SCS)

Definition 8.1. X is said to be a supersequence of Y if Y is a subsequence of X.

> **The Problem:** Given two seqeuces, find the shortest sequence that is a supersequence of both of these sequences.

For example, for the strings 'window' and 'widow', the SCS is 'window'. For 'mouse' and 'blouse', there are multiple possible shortest common supersequences, of which we can choose any. They are 'mblouse', 'bmlouse' and 'blmouse'.

8.1 Algorithm

Let X and Y be the two sequences whose SCS we wish to find.

1. Find the longest common subsequence of X and Y. (Refer to section 7.)
2. Insert the characters in X and Y that are not in their LCS into the LCS, while maintaining their relative order to the LCS characters.

For example, let $X =$ 'overlap' and $Y =$ 'rapper'. Their LCS is 'rap'. Now, we insert the non-LCS characters in X (namely 'ovel') and Y (namely 'per') into 'rap'. The SCS becomes 'overlapper'. There is only one SCS in this case.

Now consider $X =$ 'subsequence' and $Y =$ 'supersequence'. Their LCS is 'susequence'. The non-LCS characters in X are 'b' and in Y are 'per'. All possible shortest common supersequences are 'subpersequence', 'supbersequence', 'supebrsequence', and 'superbsequence'.

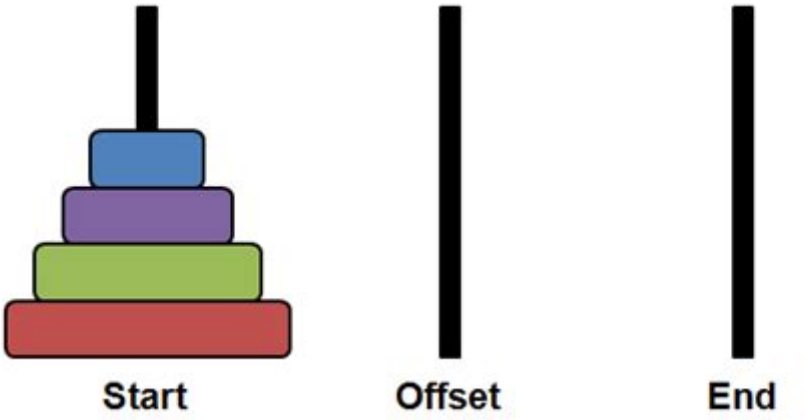
8.2 Time Complexity

The time complexity is identical to that of the LCS problem. For k input sequences of maximum length n, the time complexity is $O(n^k)$.

9 Tower of Hanoi

This is one of my all-time favourite problems and a classic puzzle that was coined by French mathematician Édouard Lucas in 1883. The problem, while easy to understand, might have one scratching their heads for days or weeks.

The Problem: There are 3 rods (let's call then rods 1, 2 and 3) and n discs with holes in their centres that would fit a rod. No two discs are of the same size. All of the discs are initially on rod 1, such that the largest disc is at the bottom and every other disc is on top of a larger disc. Each move in the game consists of removing the topmost disc in a rod and placing it on top of another rod's stack, subject to the rule that a rod may only be placed on top of a larger rod.
What is the minimum number of moves needed to move all the discs from rod 1 to rod 3 and what are they?



I recommend trying out a few examples for yourself before looking at the solution. Mathsiffun.com has a nice webpage for to play the puzzle here.

9.1 Examples

For $n = 1$, trivially, we just move the disc from rod 1 to rod 3.

For $n = 2$, move the topmost disc to rod 2, then move the disc in rod 1 to rod 3 and finally put the other disc on rod 2 in rod 3.

Observe that all we need to describe a move is the source and destination rod that was used to move the disc in that move. Thus, for the sake of brevity, we shall use this to denote the moves for $n = 3$:

1. rod 1 $\rightarrow$ rod 3
2. rod 1 $\rightarrow$ rod 2
3. rod 3 $\rightarrow$ rod 2
4. rod 1 $\rightarrow$ rod 3
5. rod 2 $\rightarrow$ rod 1
6. rod 2 $\rightarrow$ rod 3
7. rod 1 $\rightarrow$ rod 3

9.2 Minimum Number of Moves Needed

By taking a few examples and looking for a pattern, we might come to the conclusion that the minimum number of moves needed for the puzzle with n discs is $2^n - 1$. In fact, this is true. Let's prove this using mathematical induction.

Proof. We need to prove two claims:

1. It is always possible to solve the puzzle in $2^n - 1$ moves.
2. It is impossible to solve it in less than $2^n - 1$ moves.

Let us first prove the first claim. The induction hypothesis is that it is always possible to solve the puzzle in $2^n - 1$ moves.

Base case. For $n = 1$, we need only one move. This is trivially shown to be true.

Induction step. From the induction hypothesis, we assume that we can move n discs from rod 1 to rod 3 in $2^n - 1$ moves. We need to show that assuming this to be true, the hypothesis holds also true for $n + 1$ discs. This is elegantly shown. To move $n + 1$ discs from rod 1 to rod 3, we

- first move the topmost n discs to rod 2 in $2^n - 1$ moves,
- then move the bottom-most disc from rod 1 to rod 3 in 1 move, and,
- then move the n discs from rod 2 to rod 3 in $2^n - 1$ moves.

The total number of moves used is $(2^n - 1) + 1 + (2^n - 1) = 2 * 2^n - 1 = 2^{n+1} - 1$. Thus, the induction hypothesis holds true and it is always possible to solve a Tower of Hanoi puzzle with n discs in $2^n - 1$ moves.

Now, we shall prove that $2^n - 1$ is indeed the minimum possible number of moves need to solve it (this is the induction hypothesis).

Base case. For $n = 1$, trivially, the number of moves needed is 1, which is equal to $2^1 - 1$.

Induction step. Now consider the induction hypothesis for $n + 1$ discs. Observe that in order to move a disc from rod 1 to rod 3, all other discs smaller than this disc *must* be on rod 2. So, in order to move $n + 1$ discs from rod 1 to rod 3, it is imperative to move the top n discs to rod 2 first, then move the largest disc to rod 3, and then move the n discs from rod 2 to rod 3. From the induction hypothesis, this takes a minimum of $2^n - 1$ moves, 1 move, and $2^n - 1$ moves. Summing them up, we get $2^{n+1} - 1$ moves. Thus, from the principle of mathematical induction, the minimum number of moves needed to solve the Tower of Hanoi puzzle with n discs is $2^n - 1$ moves.

□

9.3 Algorithm

The recursive algorithm, as already mentioned above, is simply to move $n-1$ discs from rod 1 to rod 2, then move 1 disc from rod 1 to rod 3, and finally move $n-1$ discs from rod 2 to rod 2. The recursion stops when there's only 1 disc in a rod, in which case we simply move it to its destination without any recursion. We store the moves we made in an array called '`Moves`'. Each move can be uniquely characterised by the source rod and the destination rod (along with the index in `Moves`, but that is not something we need to store explicitly), so that is what we shall store for each move.

```
// Move n discs from source to dest, assuming that it is possible
// to do so without moving the discs on the 2 rods other than the
    source rod.
// 'Moves' is a list of pairs - Move[i] stores the source and the
// destination rod for the ith move
void MoveDisc(int n, int source, int dest, std::vector<std::pair<int,
    int>>& Moves) {
    if (n == 1) {
        Moves.push_back({source, dest});
        return;
    }
    int otherRod = 6 - source - dest;  // whichever of {1,2,3} is not
    source or dest
    MoveDisc(n - 1, source, otherRod, Moves);
    Moves.push_back({source, dest});
    MoveDisc(n - 1, otherRod, dest, Moves);
}
```

10 Bellman-Ford Algorithm

The Problem: Given a weighted graph $G = (V, E)$ and a vertex s, find the shortest distance from s to all the vertices in V.

The Bellman-Ford algorithm is used to find the single-source shortest path in both directed and undirected graphs and even in unweighted graphs. It is often used in real life in many protocols in the internet such as the routing information protocol and the distance-vector routing protocol, usually for transmitting information on a network.

Notice that if the graph contains negative edge cycles, then that implies that for some vertices, the shortest distance does not exist, i.e. it is $-\infty$.

We use the idea of relaxation in the Bellman-Ford algorithm. Relaxation refers to gradually and continuously shortening the distance between vertices. Let $E = \{e_1, e_2, \ldots, e_{|E|}\}$ and $V = \{v_1, v_2, \ldots, v_{|V|}\}$. Let w_{e_i} refer to the weight of e_i. Then the algorithm is as follows:

1. Initialiase the distance from s to every vertex as infinity.
2. For every vertex, create a 'previous' vertex which is the vertex that appears before it in the shortest path to s. Initially, this is unknown.
3. Set the distance from s to itself as 0.
4. Then, $|V| - 1$ times, iterate over every edge in E and relax it.

5. Relaxation of an edge $e = (u, v)$ is the process of checking if the previously known shortest distance from s to v is longer than the path from s to u to v, and if it is, setting the new shortest distance to v as the distance from s to $u + w_{(u,v)}$.

10.1 Pseudocode

```
for v in V:
    dist[v] = infinity
    prev[v] = None
dist[s] = 0
for i in [1 ... |V|-1]:
    for (u, v) in E:
        relax(u, v)

function relax(u, v):
    if dist[v] > dist[u] + w(u, v):
        dist[v] = dist[u] + w(u, v)
        prev[v] = u
```

10.2 Time Complexity

Dijkstra's algorithm is faster than the Bellman-Ford algorithm at $O(V + E)$, but the latter can handle negative edge weights which the former cannot.

Each relaxation operation takes $O(1)$ and we perform $(|V| - 1) \cdot |E|$ iterations. Therefore, the worst-case time complexity is $O(|V| \cdot |E|)$.

11 Floyd-Warshall Algorithm

The Problem: Find the shortest path between all pairs of vertices in a graph with weighted edges assuming that the graph contains no negative cycle (i.e., cycles whose sum of edge weights is negative).

Let the given graph be $G = (V, E)$ where V is the set of vertices and E is the set of weighted edges. Then, the Floyd-Warshall algorithm is as follows.

1. Initialise the shortest path between every pair of vertices to infinity.
2. Define an array called '`dist`' as follows.

$$\texttt{dist[i][j]} = \text{shortest path from vertex } \texttt{i} \text{ to vertex } \texttt{j}. \tag{14}$$

3. Find the shortest path between every pair of vertices that used 0 intermediate vertices (i.e., there were 0 vertices between the source and destination vertices). Then, do the same for 1 intermediate vertex, 2 intermediate vertices, and so on till $|V|$ intermediate vertices.
4. Throughout step 2, keep track of the minimum distance between every pair of vertices over every possible number of intermediate vertices in `dist`.

11.1 Pseudocode

```
Create a |V| x |V| matrix, dist, that will describe the distances
    between vertices
for each cell (i, j) in dist:
    if i == j:
        dist[i][j] = 0
    if (i, j) is an edge in E:
        dist[i][j] = weight(i, j)        ## Given as input
    else:
        dist[i][j] = infinity
for k from 1 to |V|:
    for i from 1 to |V|:
        for j from 1 to |V|:
            if dist[i][j] > dist[i][k] + dist[k][j]:
                dist[i][j] = dist[i][k] + dist[k][j]
```

11.2 C++ Implementation

```
#include <iostream>

void FloydWarshall(int **dist, int v) {
    for (int k = 0; k < v; ++k) {
        for (int i = 0; i < v; ++i)
            for (int j = 0; j < v; ++j)
                if (dist[i][k] + dist[k][j] < dist[i][j]) {
                    dist[i][j] = dist[i][k] + dist[k][j];
                }
    }
}

int main(void) {
    int v, e;
    std::cout << "Enter number of vertices: ";
    std::cin >> v;

    std::cout << "Enter number of edges: ";
    std::cin >> e;

    int **dist = new int *[v];
    for (int i = 0; i < v; ++i) {
        dist[i] = new int[v];
        for (int j = 0; j < v; ++j) {
            if (i == j)
                dist[i][j] = 0;
            else
                dist[i][j] = INT_MAX;
        }
    }

    std::cout << "Enter the edges in this format: 'tail head weight'"
                << "\n";
    for (int a = 0; a < e; ++a) {
        int i, j, w;
        std::cin >> i >> j >> w;
        dist[i][j] = w;
```

```
38      }
39
40      FloydWarshall(dist, v);
41
42      std::cout << "The minimum distance matrix is:\n";
43      for (int i = 0; i < v; ++i) {
44          for (int j = 0; j < v; ++j) {
45              std::cout << dist[i][j] << " ";
46          }
47          std::cout << "\n";
48      }
49
50      for (int i = 0; i < v; ++i) {
51          delete[] dist[i];
52      }
53      delete[] dist;
54      return 0;
55  }
```

11.3 Time Complexity

From lines 9 to 12 of the pseudocde (see section 11.1), it is clear that the algorithm contains three nested loops, each with $|V|$ iterations. Thus, the time complexity is $O(|V|^3)$.