MEMBERS :

ALGORITHM EXPOSÉE GROUP II

* AFAH TIMOL AKENDUM(LEADER)
* PEDJI DIMITRI HEASHLEY
* TCHALLE FOKA HELLARY MANUELA
* TCIMETUA MAMNO SIKARI
* PIENERE DIEBSON ARIEL PASCALIN

SUBJECT OF EXERCISE: DYNAMIC PROGRAMMING

WORK CONTENT:

* WHAT IS DYNAMIC PROGRAMMING
* WHERE SHOULD DYNAMIC PROGRAMMING BE USED
* HOW DOES DYNAMIC PROGRAMMING WORK
* TOP-DOWN APPROACH
* BOTTUM-UP
* SIGNS OF DYNAMIC PROGRAMMING SUITABILITY
* DYNAMIC PROGRAMMING ALGORITHMS
* EXAMPLES OF APPLICATIONS OF DYNAMIC PROGRAMMING

DYNAMIC PROGAMMING

What Is Dynamic Programming?

**Dynamic programming is a computer programming technique where an algorithmic problem is first broken down into sub-problems, the results are saved, and then the sub-problems are optimized to find the overall solution — which usually has to do with finding the maximum and minimum range of the algorithmic query.**

Richard Bellman was the one who came up with the idea for dynamic programming in the 1950s. It is a method of mathematical optimization as well as a methodology for computer programming. It applies to issues one can break down into either overlapping subproblems or optimum substructures.

When a more extensive set of equations is broken down into smaller groups of equations, overlapping subproblems are referred to as equations that reuse portions of the smaller equations several times to arrive at a solution.

On the other hand, optimum substructures locate the best solution to an issue, then build the solution that provides the best results overall. This is how they solve problems. When a vast issue is split down into its constituent parts, a computer will apply a mathematical algorithm to determine which elements have the most desirable solution. Then, it takes the solutions to the more minor problems and utilizes them to get the optimal solution to the initial, more involved issue.

This technique solves problems by breaking them into smaller, overlapping subproblems. The results are then stored in a table to be reused so the same problem will not have to be computed again.

For example, when using the dynamic programming technique to figure out all possible results from a set of numbers, the first time the results are calculated, they are saved and put into the equation later instead of being calculated again. So, when dealing with long, complicated equations and processes, it saves time and makes solutions faster by doing less work.

The dynamic programming algorithm tries to find the shortest way to a solution when solving a problem. It does this by going from the top down or the bottom up. The top-down method solves equations by breaking them into smaller ones and reusing the answers when needed. The bottom-up approach solves equations by breaking them up into smaller ones, then tries to solve the equation with the smallest mathematical value, and then works its way up to the equation with the biggest value.

Using dynamic programming to solve problems is more effective than just trying things until they work. But it only helps with problems that one can break up into smaller equations that will be used again at some point.

Where shoulddynamic programming be used?

Dynamic programming is used when one can break a problem into more minor issues that they can break down even further, into even more minor problems. Additionally, these subproblems have overlapped. That is, they require previously calculated values to be recomputed. With dynamic programming, the computed values are stored, thus reducing the need for repeated calculations and saving time and providing faster solutions.

## How Does Dynamic Programming Work?

Dynamic programming works by breaking down complex problems into simpler subproblems. Then, finding optimal solutions to these subproblems. Memorization is a method that saves the outcomes of these processes so that the corresponding answers do not need to be computed when they are later needed. Saving solutions save time on the computation of subproblems that have already been encountered.

Dynamic programming can be achieved using two approaches:

1. Top-down approach

In computer science, problems are resolved by recursively formulating solutions, employing the answers to the problems’ subproblems. If the answers to the subproblems overlap, they may be memorized or kept in a table for later use. The top-down approach follows the strategy of memorization. The memoization process is equivalent to adding the recursion and caching steps. The difference between recursion and caching is that recursion requires calling the function directly, whereas caching requires preserving the intermediate results.

The top-down strategy has many benefits, including the following:

* The top-down approach is easy to understand and implement. In this approach, problems are broken down into smaller parts, which help users identify what needs to be done. With each step, more significant, more complex problems become smaller, less complicated, and, therefore, easier to solve. Some parts may even be reusable for the same problem.
* It allows for subproblems to be solved upon request. The top-down approach will enable problems to be broken down into smaller parts and their solutions stored for reuse. Users can then query solutions for each part.
* It is also easier to debug. Segmenting problems into small parts allows users to follow the solution quickly and determine where an error might have occurred.

Disadvantages of the top-down approach include:

* The top-down approach uses the recursion technique, which occupies more memory in the call stack. This leads to reduced overall performance. Additionally, when the recursion is too deep, a stack overflow occurs.

**2.** Bottom-up approach

In the bottom-up method, once a solution to a problem is written in terms of its subproblems in a way that loops back on itself, users can rewrite the problem by solving the smaller subproblems first and then using their solutions to solve the larger subproblems.

Unlike the top-down approach, the bottom-up approach removes the recursion. Thus, there is neither stack overflow nor overhead from the recursive functions. It also allows for saving memory space. Removing recursion decreases the time complexity of recursion due to recalculating the same values.

The advantages of the bottom-up approach include the following:

* It makes decisions about small reusable subproblems and then decides how they will be put together to create a large problem.

### It removes recursion, thus promoting the efficient use of memory space. Additionally, this also leads to a reduction in timing complexity.

Signs of dynamic programming suitability

Dynamic programming solves complex problems by breaking them up into smaller ones using recursion and storing the answers so they don’t have to be worked out again. It isn’t practical when there aren’t any problems that overlap because it doesn’t make sense to store solutions to the issues that won’t be needed again.

Two main signs that one can solve a problem with dynamic programming: subproblems that overlap and the best possible substructure.

**Overlapping subproblems**

When the answers to the same subproblem are needed more than once to solve the main problem, we say that the subproblems overlap. In overlapping issues, solutions are put into a table so developers can use them repeatedly instead of recalculating them. The recursive program for the Fibonacci numbers has several subproblems that overlap, but a binary search doesn’t have any subproblems that overlap.

A binary search is solved using the divide and conquer technique. Every time, the subproblems have a unique array to find the value. Thus, binary search lacks the overlapping property.

For example, when finding the nth Fibonacci number, the problem F(n) is broken down into finding F(n-1) and F. (n-2). You can break down F(n-1) even further into a subproblem that has to do with F. (n-2).In this scenario, F(n-2) is reused, and thus, the Fibonacci sequence can be said to exhibit overlapping properties.

**Optimal substructure**

The optimal substructure property of a problem says that you can find the best answer to the problem by taking the best solutions to its subproblems and putting them together. Most of the time, recursion explains how these optimal substructures work.

This property is not exclusive to dynamic programming alone, as several problems consist of optimal substructures. However, most of them lack overlapping issues. So, they can’t be called problems with dynamic programming.

You can use it to find the shortest route between two points. For example, if a node p is on the shortest path from a source node t to a destination node w, then the shortest path from t to w is the sum of the shortest paths from t to p and from p to w.

Examples of problems with optimal substructures include the longest increasing subsequence, longest palindromic substring, and longest common subsequence problem. Examples of problems without optimal substructures include the most extended path problem and the addition-chain exponentiation.

* Understanding the Longest Common Subsequence concept in dynamic programming

In dynamic programming, the phrase “largest common subsequence” (LCS) refers to the subsequence that is shared by all of the supplied sequences and is the one that is the longest. It is different from the challenge of finding the longest common substring in that the components of the LCS do not need to occupy consecutive locations within the original sequences to be considered part of that problem.

The LCS is characterized by an optimal substructure and overlapping sub problem properties. This indicates that the issue may be split into many less complex sub-issues and worked on individually until a solution is found. The solutions to higher-level sub problems are often reused in lower-level sub problems, thus, overlapping sub problems.

Therefore, when solving an LCS problem, it is more efficient to use a dynamic algorithm than a recursive algorithm. Dynamic programming stores the results of each function call so that it can be used in future calls, thus minimizing the need for redundant calls.

For instance, consider the sequences (MNOP) and (MONMP). They have five length-2 common subsequences (MN), (MO), (MP), (NP), and (OP); two length-3 common subsequences (MNP) and (MOP); MNP and no longer frequent subsequences (MOP). Consequently, (MNP) and (MOP) are the largest shared subsequences. LCS can be applied in bioinformatics to the process of genome sequencing.

Dynamic Programming Algorithms

When dynamic programming algorithms are executed, they solve a problem by segmenting it into smaller parts until a solution arrives. They perform these tasks by finding the shortest path. Some of the primary dynamic programming algorithms in use are:

1. Greedy algorithms:

An example of dynamic programming algorithms, greedy algorithms are also optimization tools. The method solves a challenge by searching for optimum solutions to the subproblems and combining the findings of these subproblems to get the most optimal answer.

Conversely, when greedy algorithms solve a problem, they look for a locally optimum solution to find a global optimum. They make a guess that looks optimum at the time but does not guarantee a globally optimum solution. This could end up becoming costly down the road.

2. Floyd-Warshall algorithm

The Floyd-Warshall method uses a technique of dynamic programming to locate the shortest pathways. It determines the shortest route across all pairings of vertices in a graph with weights. Both directed and undirected weighted graphs can use it.

This program compares each pair of vertices’ potential routes through the graph. It gradually optimizes an estimate of the shortest route between two vertices to determine the shortest distance between two vertices in a chart. With simple modifications to it, one can reconstruct the paths.

This method for dynamic programming contains two subtypes: Behavior with negative cycles: Users can use the Floyd-Warshall algorithm to find negative cycles. You can do this by inspecting the diagonal path matrix for a negative number that would indicate the graph contains one negative cycle. In a negative cycle, the sum of the edges is a negative value; thus, there cannot be a shortest path between any pair of vertices. Exponentially huge numbers are generated if a negative cycle occurs during algorithm execution.

Time complexity: The Floyd-Warshall algorithm has three loops, each with constant complexity. As a result, the Floyd-Warshall complexity has a time complexity of O(n3). Wherein n represents the number of network nodes.

3. Bellman Ford algorithm

The Bellman-Ford Algorithm determines the shortest route from a particular source vertex to every other weighted digraph vertices. The Bellman-Ford algorithm can handle graphs where some of the edge weights are negative numbers and produce a correct answer, unlike Dijkstra’s algorithm, which does not confirm whether it makes the correct answer. However, it is much slower than Dijkstra’s algorithm.

The Bellman-Ford algorithm works by relaxation; that is, it gives approximate distances that better ones continuously replace until a solution is reached. The approximate distances are usually overestimated compared to the distance between the vertices. The replacement values reflect the minimum old value and the length of a newly found path.

This algorithm terminates upon finding a negative cycle and thus can be applied to cycle-canceling techniques in network flow analysis.

Examples of Applications of Dynamic Programming

Here are a few examples of how one may use dynamic programming:

1. Example: Matrix Chain Multiplication

Why Dynamic Programming?

The Matrix Chain Multiplication problem involves finding the most efficient way to multiply a given sequence of matrices. Dynamic programming is suitable for this problem because the optimal parenthesization for the multiplication of matrices can be built by considering the subproblems of multiplying smaller matrix chains.

C Program Code:

#include <stdio.h>

#include <limits.h>

// Function to find the minimum number of scalar multiplications

// needed to multiply a chain of matrices

int matrixChainOrder(int p[], int n) {

int m[n][n];

for (int i = 1; i < n; i++)

m[i][i] = 0;

for (int L = 2; L < n; L++) {

for (int i = 1; i <= n - L + 1; i++) {

int j = i + L - 1;

m[i][j] = INT\_MAX;

for (int k = i; k <= j - 1; k++) {

int cost = m[i][k] + m[k+1][j] + p[i-1] \* p[k] \* p[j];

if (cost < m[i][j])

m[i][j] = cost;

}

}

}

return m[1][n-1];

}

int main() {

int arr[] = {30, 35, 15, 5, 10, 20, 25};

int n = sizeof(arr) / sizeof(arr[0]);

printf("Minimum number of scalar multiplications is %d\n", matrixChainOrder(arr, n));

return 0;

}

In this example, dynamic programming is used to find the optimal parenthesization for matrix chain multiplication, minimizing the total number of scalar multiplications needed. The "matrix chain order " function computes the minimum cost by considering subproblems of multiplying smaller matrix chains.

2.Example: Computing Factorials using Dynamic Programming

Why Dynamic Programming?

Dynamic programming is well-suited for problems where the solution to a larger instance of the problem can be efficiently computed using solutions to smaller instances. Factorial computation involves repetitive calculations of smaller factorials, making it an ideal candidate for dynamic programming.

C Program Code:

#include <stdio.h>

// Function to compute factorial using dynamic programming

unsigned long long factorial(int n) {

if (n == 0 || n == 1) {

return 1;

}

unsigned long long dp[n+1];

dp[0] = dp[1] = 1;

for (int i = 2; i <= n; i++) {

dp[i] = i \* dp[i-1];

}

return dp[n];

}

int main() {

int n = 5;

printf("Factorial of %d is %llu\n", n, factorial(n));

return 0;

}

In this example, dynamic programming optimizes the computation of factorials by storing intermediate results in an array (dp). The function factorial efficiently calculates factorials up to the given input n.

3. Example: Longest Increasing Subsequence (LIS)

Why Dynamic Programming?

The Longest Increasing Subsequence (LIS) problem involves finding the length of the longest subsequence of a given sequence such that all elements of the subsequence are sorted in strictly increasing order. Dynamic programming is suitable for this problem because the length of the LIS for a given index depends on the lengths of LIS for smaller indices.

C Program Code:

#include <stdio.h>

// Function to find the length of Longest Increasing Subsequence

int lis(int arr[], int n) {

int lis[n];

int max\_length = 1;

// Initialize LIS values for all indices

for (int i = 0; i < n; i++)

lis[i] = 1;

// Compute LIS values in a bottom-up manner

for (int i = 1; i < n; i++) {

for (int j = 0; j < i; j++) {

if (arr[i] > arr[j] && lis[i] < lis[j] + 1)

lis[i] = lis[j] + 1;

}

if (lis[i] > max\_length)

max\_length = lis[i];

}

return max\_length;

}

int main() {

int arr[] = {10, 22, 9, 33, 21, 50, 41, 60};

int n = sizeof(arr) / sizeof(arr[0]);

printf("Length of LIS is %d\n", lis(arr, n));

return 0;

}

In this example, the program efficiently computes the length of the Longest Increasing Subsequence using dynamic programming by maintaining an array (lis) to store the lengths of LIS for each index.