****

**حلقة بحث في مادة المعلوماتية**

 **للطالب محمد زياد القباقيبي**

 **الصف الحادي عشر**

dYNAMIC PROGRAMMING

**Contents:**

Introduction………………………………………………………… 2

What is the meaning of (Algorithm)……………………………….3

Recursion..…………………………………………………….…… 4

Divide and Conquer…..…………………………………………… 6

Dynamic Programming.…………………………………………… 9

Top-down vs Bottom-up...………………………………………… 6

Famous DP problems……………………………………………...12

References……………………………………………..…………...15

***Introduction:***

* Properly you heard about "algorithm" in mathematics or in computer science.
* Do you know what an algorithm is and what is the importance of it?
* There are so many algorithms and methods in computer science one of them.

Dynamic Programming (DP) in this research after talking about algorithm, Divide and Conquer, Recursion we will talk about them before to introduce you to DP, hoping you understand the difference, (All the codes in this research will be in C, C++ programming language or pseudo-code).

* We will start explaining about (Algorithm), Recursion, Divide and Conquer (which is close to dynamic programming) then we will talk about Dynamic Programming.

**What is the meaning of (algorithm)?**

[[1]](#footnote-1)What are algorithms? Why is the study of algorithms worthwhile? What is the role of algorithms relative to other technologies used in computers? We will answer these questions.

Informally, an ***algorithm*** is any well-defined computational procedure that takes some value, or set of values, as ***input*** and produces some value, or set of values, as ***output***. An algorithm is thus a sequence of computational steps that transform the input into the output.

We can also view an algorithm as a tool for solving a well-specified ***computational problem***. The statement of the problem specifies in general terms the desiredinput/output relationship. The algorithm describes a specific computational procedurefor achieving that input/output relationship.

For example, we might need to sort a sequence of numbers into non-decreasing order. This problem arises frequently in practice and provides fertile ground for introducing many standard design techniques and analysis tools. Here is how we formally define the **sorting problem**:

**Input:** A sequence of n numbers $\left\{a\_{1},a\_{2},a\_{3},...,a\_{n}\right\} $

**Output:** A permutation (reordering) $\left\{a\_{1}^{'},a\_{2}^{'},a\_{3}^{'},...,a\_{n}^{'}\right\} $ of the input sequence such that$ a\_{1}^{'}\leq a\_{2}^{'}\leq a\_{3}^{'}\leq ...\leq a\_{n}^{'}$.

* For example, given the input sequence$ \left\{31,41,59,26,41,58\right\}$ sorting algorithm returns as output the sequence$ \left\{26,31,41,41,59,58\right\}$. Such an input sequence is called an ***instance*** of the sorting problem. In general, an ***instance of a problem*** consists of the input (satisfying whatever constraints are imposed in the problem statement) needed to compute a solution to the problem.
* Because many programs use it as an intermediate step, sorting is a fundamental operation in computer science. As a result, we have a large number of good sorting algorithms at our disposal. Which algorithm is best for a given application depends on among other factors the number of items to be sorted, the extent to which the items are already somewhat sorted, possible restrictions on the item values, the architecture of the computer, and the kind of storage devices to be used: main memory, disks, or even tapes.

An algorithm is said to be ***correct*** if, for every input instance, it halts with the correct output. We say that a correct algorithm ***solves*** the given computational problem. An incorrect algorithm might not halt at all on some input instances, or it might halt with an incorrect answer. Contrary to what you might expect, incorrect algorithms can sometimes be useful, if we can control their error rate.

**Recursion:**

We all know that you can use or create functions in C and C++, but do you know that you can make a function call itself?

[[2]](#footnote-2)Recursion is a programming technique that allows the programmer to express operations in terms of themselves. In C, this takes the form of a function that calls itself. A useful way to think of recursive functions is to imagine them as a process being performed where one of the instructions is to "repeat the process". This makes it sound very similar to a loop because it repeats the same code, and in some ways, it is similar to looping. On the other hand, recursion makes it easier to express ideas in which the result of the recursive call is necessary to complete the task. Of course, it must be possible for the "process" to sometimes be completed without the recursive call. One simple example is the idea of building a wall that is ten feet high; if I want to build a ten-foot high wall, then I will first build a 9-foot high wall, and then add an extra foot of bricks. Conceptually, this is like saying the "build wall" function takes a height and if that height is greater than one, first calls itself to build a lower wall, and then adds one a foot of bricks.

A simple example of recursion would be:

void recurse()

{

 recurse(); /\* Function calls itself \*/

}

int main()

{

 recurse(); /\* Sets off the recursion \*/

 return 0;

}

This program will not continue forever, however. The computer keeps function calls on a stack and once too many are called without ending, the program will crash. Why not write a program to see how many times the function is called before the program terminates?

#include <stdio.h>

void recurse ( int count ) /\* Each call gets its own copy of count \*/

{

 printf( "%d\n", count );

 /\* It is not necessary to increment count since each function's

 variables are separate (so each count will be initialized one greater)

 \*/

 recurse ( count + 1 );

}

int main()

{

 recurse ( 1 ); /\* First function call, so it starts at one \*/

 return 0;

}

This function works because it will go through and print the numbers begin to 9, and then as each printnum function terminates it will continue printing the value of begin in each function.
Now we will show you one of the most famous recursion functions:

-Factorial (n!):

[[3]](#footnote-3)/\*  Example Program For Factorial Value Using Function In C++

    little drops @ thiyagaraaj.com

    Coded By:THIYAGARAAJ MP             \*/

#include<iostream>

#include<conio.h>

using namespace std;

//Function

long factorial(int);

int main()

{

     // Variable Declaration

     int counter, n;

     // Get Input Value

     cout<<"Enter the Number :";

     cin>>n;

     // Factorial Function Call

     cout<<n<<" Factorial Value Is "<<factorial(n);

     // Wait For Output Screen

     getch();

     return 0;

 }

// Factorial Function

long factorial(int n)

{

    int counter;

    long fact = 1;

     //for Loop Block

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

     {

         fact = fact \* counter;

     }

  return fact;

}

**Divide and Conquer:**

Before talking about Dynamic-Programming, we will talk about Divide and Conquer algorithms because they are close to DP.

[[4]](#footnote-4)Divide and Conquer is an algorithmic paradigm. A typical Divide and Conquer algorithm solves a problem using following three steps.

**1.** Divide: Break the given problem into sub problems of same type.
**2.** Conquer: Recursively solve these sub problems
**3.**Combine: Appropriately combine the answers

Following are some standard algorithms that are Divide and Conquer algorithms.

**1) Binary Search** is a searching algorithm. In each step, the algorithm compares the input element x with the value of the middle element in array. If the values match, return the index of middle. Otherwise, if x is less than the middle element, then the algorithm recurs for left side of middle element, else recurs for right side of middle element.

**2) Quicksort** is a sorting algorithm. The algorithm picks a pivot element, rearranges the array elements in such a way that all elements smaller than the picked pivot element move to left side of pivot, and all greater elements move to right side. Finally, the algorithm recursively sorts the subarrays on left and right of pivot element.

**3) Merge Sort** is also a sorting algorithm. The algorithm divides the array in two halves; recursively sorts them and finally merges the two-sorted halves.

**4) Closest Pair of Points** the problem is to find the closest pair of points in a set of points in $x,y $plane. The problem can be solved in $O\left(n^{2}\right)$ time by calculating distances of every pair of points and comparing the distances to find the minimum. The Divide and Conquer algorithm solves the problem in $O\left(n.log\_{2}n\right)$time.

Binary search algorithm relies on a divide and conquer strategy to find a value within an already-sorted collection. Binary search locates the position of an item in a sorted array. Binary search compare an input search key to the middle element of the array and the comparison determines whether the element equals the input, less than the input or greater. The return value is the element position in the array.

### Binary Search Algorithm complexity

* Worst case performance $O(log n)$
* Best case performance $O(1)$
* Average case performance $O(log n)$
* Worst case space complexity $O(1)$

C++ Code:

#include <cstdlib>

#include <iostream>

**using** **namespace** std**;**

int binary\_search**(**int array**[],** int first**,** int last**,** int value**);**

int main**()** **{**

 int list**[**10**];**

 **for** **(**int k **=** 0**;** k**<**11**;** k**++)**

 list**[**k**]** **=** 2 **\*** k **+** 1**;**

 cout **<<** "binary search results: " **<<** binary\_search**(**list**,** 1**,** 21**,** 11**)** **<<** endl**;**

 **return** 0**;**

**}**//end of main

int binary\_search**(**int array**[],** int first**,** int last**,** int search\_key**)**

**{**

 int index**;**

 **if** **(**first **>** last**)**

 index **=** **-**1**;**

 **else**

 **{**

 int mid **=** **(**first **+** last**)** **/** 2**;**

 **if** **(**search\_key **==** array**[**mid**])**

 index **=** mid**;**

 **else**

 **if** **(**search\_key **<** array**[**mid**])**

 index **=** binary\_search**(**array**,** first**,** mid **-** 1**,** search\_key**);**

 **else**

 index **=** binary\_search**(**array**,** mid **+** 1**,** last**,** search\_key**);**

 **}** // end if

 **return** index**;**

**}**// end binarySearch

**Dynamic Programming:**

[[5]](#footnote-5)(usually referred to as **DP** ) is a very powerful technique to solve a particular class of problems. It demands very elegant formulation of the approach and simple thinking and the coding part is very easy. The idea is very simple, If you have solved a problem with the given input, then save the result for future reference, so as to avoid solving the same problem again... shortly *'Remember your Past'* :) .  If the given problem can be broken up in to smaller sub-problems and these smaller sub problems are in turn divided in to still-smaller ones, and in this process, if you observe some over-lapping sub problems, then it is a big hint for DP. Also, the optimal solutions to the sub problems contribute to the optimal solution of the given problem ( referred to as the Optimal Substructure Property ).

There are two ways of doing this.

**1.) Top-Down :**Start solving the given problem by breaking it down. If you see that the problem has been solved already, then just return the saved answer. If it has not been solved, solve it and save the answer. This is usually easy to think of and very intuitive. This is referred to as ***Memorization***.

**2.) Bottom-Up**: Analyze the problem and see the order in which the sub-problems are solved and start solving from the trivial sub problem, up towards the given problem. In this process, it is guaranteed that the sub problems are solved before solving the problem. This is referred to as ***Dynamic Programming***.

Note that divide and conquer is slightly a different technique. In that, we divide the problem in to non-overlapping subproblems and solve them independently, like in **mergesort** and **quick sort.**

[[6]](#footnote-6)The two main properties of a problem that suggest that the given problem can be solved using Dynamic programming.

1) Overlapping Sub problems
2) Optimal Substructure

**1) Overlapping Sub problems:**
Like Divide and Conquer, Dynamic Programming combines solutions to sub-problems. Dynamic Programming is mainly used when solutions of same sub problems are needed again and again. In dynamic programming, computed solutions to sub problems are stored in a table so that these don’t have to recomputed. So Dynamic Programming is not useful when there are no common (overlapping) subproblems because there is no point storing the solutions if they are not needed again. For example, Binary Search doesn’t have common sub problems. If we take example of following recursive program for Fibonacci Numbers, there are many sub problems which are solved again and again.

|  |
| --- |
| /\* simple recursive program for Fibonacci numbers \*/int fib(int n){   if ( n <= 1 )      return n;   return fib(n-1) + fib(n-2);} |

Recursion tree for execution of fib (5)

 fib(5)

 / \

 fib(4) fib(3)

 / \ / \

 fib(3) fib(2) fib(2) fib(1)

 / \ / \ / \

 fib(2) fib(1) fib(1) fib(0) fib(1) fib(0)

 / \

fib(1) fib(0)

We can see that the function $f(3) $is being called 2 times. If we would have stored the value of $f\left(3\right),$then instead of computing it again, we would have reused the old stored value.

/\* C program for tabulated version(top-down) \*/

#include<stdio.h>

int fib(int n)

{

  int f[n+1];

  int i;

  f[0] = 0;   f[1] = 1;

  for (i = 2; i <= n; i++)

      f[i] = f[i-1] + f[i-2];

  return f[n];

}

int main ()

{

  int n = 9;

  printf("Fibonacci number is %d ", fib(n));

  return 0;

}

[[7]](#footnote-7)

//bottom-up

1public int fibDP**(**int x**)** **{**

 int fib**[]** **=** **new** int**[**x **+** 1**];**

 fib**[**0**]** **=** 0**;**

 fib**[**1**]** **=** 1**;**

 **for** **(**int i **=** 2**;** i **<** x **+** 1**;** i**++)** **{**

 fib**[**i**]** **=** fib**[**i **-** 1**]** **+** fib**[**i **-** 2**];**

 **}**

 **return** fib**[**x**];**

 **}**

**Top-down vs Bottom-up**

You will ask yourself what is the different between Top-Down and Bottom-up and if you can use any of them at any time…

[[8]](#footnote-8)Here is a compare between them:

**Here are** $9 $**of the most famous Dynamic programming problems:**

* **Longest Increasing Subsequence**: The longest Increasing Subsequence (LIS) problem is to find the length of the longest subsequence of a given sequence such that all elements of the subsequence are sorted in increasing order. For example, length of LIS for $\{10, 22, 9, 33, 21, 50, 41, 60, 80\}$ is $6$ and LIS is$\{10, 22, 33, 50, 60, 80\}$.[[9]](#footnote-9)
* **Longest Common Subsequence**: [[10]](#footnote-10) Given two sequences, find the length of longest subsequence present in both of them. A subsequence is a sequence that appears in the same relative order, but not necessarily contiguous. For example, $“abc”, “abg”, “bdf”, “aeg”, ‘”acefg”, . etc. $are subsequences of $“abcdefg”. $so a string of length n has $2^{n}$ different possible subsequences.
* **Edit Distance**: [[11]](#footnote-11)Given two strings str1 and str2 and below operations that can performed on str1. Find minimum number of edits (operations) required to convert ‘str1′ into ‘str2′.
1. Insert
2. Remove
3. Replace

All of the above operations are of equal cost.

* **Coin Change**: [[12]](#footnote-12) Given a value N, if we want to make change for N cents, and we have infinite supply of each of $S = \{S1, S2,…. Sm\} $valued coins, how many ways can we make the change? The order of coins does not matter.
* [[13]](#footnote-13) $0/1$ **Knapsack Problem**: Given weights and values of n items, put these items in a knapsack of capacity W to get the maximum total value in the knapsack.
* **Cutting a Rod**:[[14]](#footnote-14) Given a rod of length n inches and an array of prices that contains prices of all pieces of size smaller than$ n $, determine the maximum value obtainable by cutting up the rod and selling the pieces.
* **Maximum Sum Increasing Subsequence**:[[15]](#footnote-15) Given an array of n positive integers. Write a program to find the sum of maximum sum subsequence of the given array such that the integers in the subsequence are sorted in increasing order. For example, if input is $\{1, 101, 2, 3, 100, 4, 5\}$, then output should be 106 $(1 + 2 + 3 + 100)$, if the input array is {3, 4, 5, 10}, then output should be $22 (3 + 4 + 5 + 10)$ and if the input array is $ \{10, 5, 4, 3\},$ then output should be 10.
* **Longest Bitonic Subsequence**: [[16]](#footnote-16) Given an array arr[0 … n-1] containing n positive integers, a subsequence of arr[] is called Bitonic if it is first increasing, then decreasing. Write a function that takes an array as argument and returns the length of the longest bitonic subsequence.
* **Floyd Warshall Algorithm**:[[17]](#footnote-17) or solving the All Pairs Shortest Path problem. The problem is to find shortest distances between every pair of vertices in a given edge weighted directed Graph.

**In The End:**

**What makes DP technique different?**

1-Faster than Recursion and Divide-and-Conquer Techniques.

2-Usually uses more memory.

3-Always gives us the best answer in smart way☺.

References:

* Books:

Competitive Programming 3 (Steven Halim, Felix Halim)

Introduction to Algorithms 3rd Edition (Thomas H.Cormen, Charles E.Leiserson, Ronald L.Rivest, Clifford Stein)

* Websites (I took the all information in 18/Dec/2015)

<http://www.geeksforgeeks.org/>

<http://algorithms.tutorialhorizon.com>

<https://www.codechef.com/>

<http://www.cpp.thiyagaraaj.com>

<http://www.cprogramming.com/tutorial/c/lesson16.html>

1. Introduction to Algorithms 3rd Edition (Thomas H.Cormen, Charles E.Leiserson, Ronald L.Rivest, Clifford Stein) page 5. [↑](#footnote-ref-1)
2. <http://www.cprogramming.com/tutorial/c/lesson16.html> 17/Dec/2015 [↑](#footnote-ref-2)
3. http://www.cpp.thiyagaraaj.com/c-programs/c-common-example-program/factorial-using-function-example-program-in-c [↑](#footnote-ref-3)
4. http://www.geeksforgeeks.org/divide-and-conquer-set-1-find-closest-pair-of-points/ [↑](#footnote-ref-4)
5. https://www.codechef.com/wiki/tutorial-dynamic-programming [↑](#footnote-ref-5)
6. http://www.geeksforgeeks.org/dynamic-programming-set-1/ [↑](#footnote-ref-6)
7. http://algorithms.tutorialhorizon.com/introduction-to-dynamic-programming-fibonacci-series/ [↑](#footnote-ref-7)
8. Competitive Programming 3 (Steven Halim, Felix Halim) page 102 [↑](#footnote-ref-8)
9. http://www.geeksforgeeks.org/dynamic-programming-set-3-longest-increasing-subsequence/ [↑](#footnote-ref-9)
10. http://www.geeksforgeeks.org/dynamic-programming-set-4-longest-common-subsequence/ [↑](#footnote-ref-10)
11. http://www.geeksforgeeks.org/dynamic-programming-set-5-edit-distance/ [↑](#footnote-ref-11)
12. http://www.geeksforgeeks.org/dynamic-programming-set-7-coin-change/ [↑](#footnote-ref-12)
13. http://www.geeksforgeeks.org/dynamic-programming-set-10-0-1-knapsack-problem/ [↑](#footnote-ref-13)
14. http://www.geeksforgeeks.org/dynamic-programming-set-13-cutting-a-rod/ [↑](#footnote-ref-14)
15. http://www.geeksforgeeks.org/dynamic-programming-set-14-maximum-sum-increasing-subsequence/ [↑](#footnote-ref-15)
16. http://www.geeksforgeeks.org/dynamic-programming-set-15-longest-bitonic-subsequence/ [↑](#footnote-ref-16)
17. http://www.geeksforgeeks.org/dynamic-programming-set-16-floyd-warshall-algorithm/ [↑](#footnote-ref-17)