Algorithm Designing Techniques

Algorithm designing techniques are crucial for creating efficient and effective algorithms. Here are some common techniques:

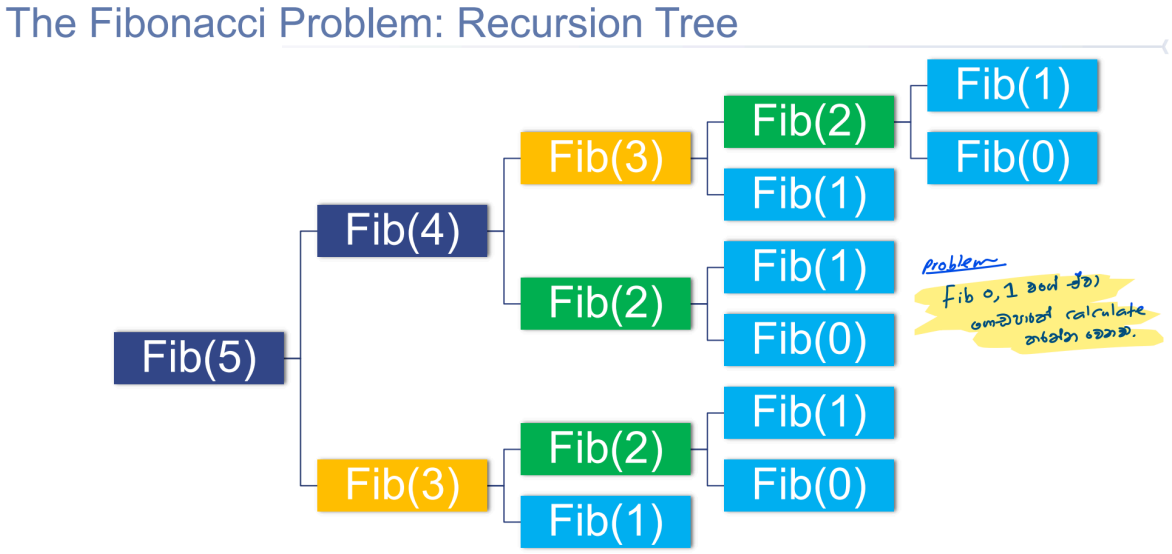
1. **Divide and Conquer**:
    - Break a problem into smaller subproblems, solve each subproblem recursively, and then combine their solutions to solve the original problem.
    - Examples: Merge Sort, Quick Sort, Binary Search.
2. **Dynamic Programming**:
    - Solve complex problems by breaking them down into simpler subproblems and storing the results of subproblems to avoid redundant computations.
    - Examples: Fibonacci sequence, Knapsack problem, Longest Common Subsequence.
3. **Greedy Algorithms**:
    - Make a series of choices, each of which looks best at the moment, without worrying about the global optimum.
    - Examples: Prim's and Kruskal's algorithms for Minimum Spanning Tree, Huffman coding.
4. **Recursion**:
    - Solve a problem by solving smaller instances of the same problem.
    - Examples: Factorial calculation, Tower of Hanoi.

> - `Divide and conquer is a strategy for algorithm design that often involves recursion, but they are not exactly the same concept. Divide and conquer specifically involves breaking down problems into multiple smaller subproblems. Recursion might involve only reducing the problem size in a single dimension (e.g., decrementing a number).`
> - `While recursion is a fundamental technique that can be applied to many problems, dynamic programming builds on recursion by storing and reusing results to avoid redundancy, making it a powerful tool for solving complex problems efficiently.`

1.The Fibonacci Problem

$$F(n) = F(n-2) + F(n-1) \ \ where \ \ F(0) = 0, F(1) = 1$$

1.The Recursion Tree



Algorithm

```
Fib(n)
{
    if (n == 0)
        return 0;
    if (n == 1)
        return 1;
    Return Fib(n-1)+Fib(n-2)
}
```

$$T(n) = T(n-1) + T(n-2) + O(1)$$

$$= T(n-2) + T(n-3) + O(1) + T(n-3) + T(n-4) + O(1) + O(1)$$

$$= T(n-2) + 2T(n-3) + T(n-4) + 3O(1)$$

$$T(n) = O(2^n)$$

2.Dynamic Programming

`Typically applied to optimization problems.`

Dynamic programming (DP) does not always use recursion, although it often can. Dynamic programming is a method for solving complex problems by breaking them down into simpler subproblems and solving each subproblem just once, storing the solutions in a table (memorization) or reusing them iteratively (tabulation). There are two main approaches to implement dynamic programming: top-down (which often uses recursion with memorization) and bottom-up (which typically uses an iterative approach).

Top-Down Approach (Memorization)

In the top-down approach, dynamic programming is implemented using recursion with memorization. This means solving subproblems recursively and storing the results to avoid redundant computations.

Example: Fibonacci Sequence with Memorization

```
def fib(n, memo={}):
    if n in memo:
        return memo[n]
    if n <= 1:
        return n
    memo[n] = fib(n-1, memo) + fib(n-2, memo)
    return memo[n]

print(fib(10))  # Output: 55
```

In this example, `fib` is a recursive function that uses a dictionary `memo` to store previously computed values of the Fibonacci sequence.

Bottom-Up Approach (Tabulation)

In the bottom-up approach, dynamic programming is implemented iteratively. This method builds the solution from the base cases up to the desired value, filling in a table (usually an array) along the way.

Example: Fibonacci Sequence with Tabulation

```
def fib(n):
    if n <= 1:
        return n
    table = [0] * (n + 1)
    table[1] = 1
    for i in range(2, n + 1):
        table[i] = table[i-1] + table[i-2]
    return table[n]

print(fib(10))  # Output: 55
```

In this example, `fib` is an iterative function that fills in the `table` array with Fibonacci values from the bottom up.

Key Differences

- **Top-Down (Memorization)**:
  - Uses recursion.
  - Stores results of subproblems in a cache (dictionary or array).
  - Typically easier to implement if the recursive structure of the problem is clear.
- **Bottom-Up (Tabulation)**:
  - Uses iteration.
  - Builds a table iteratively from the base case up to the desired value.
  - Often more efficient in terms of space and stack usage since it avoids the overhead of recursive function calls.

2.What is Dynamic Programming?

Dynamic programming (DP) is a method for solving complex problems by breaking them down into simpler subproblems. There are three key components of dynamic programming that are crucial for its implementation:

Three Components of Dynamic Programming.

1. Optimal Substructure (i.e. recursive)

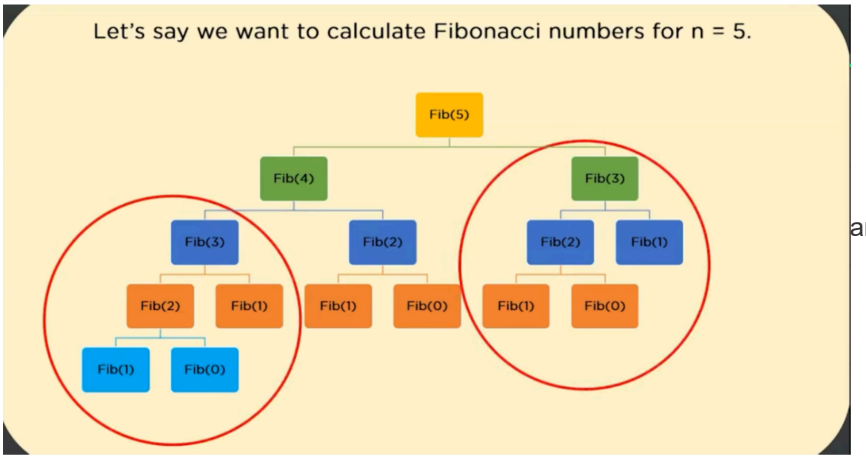
Optimal substructure means that the optimal solution to a problem can be constructed from the optimal solutions of its subproblems. In other words, if you can solve smaller instances of a problem optimally, you can combine these solutions to solve the larger problem optimally.

Example: In the shortest path problem, the shortest path from node A to node C through node B is the sum of the shortest path from A to B and the shortest path from B to C.

2. Overlapping Subproblems (repeating subproblems)

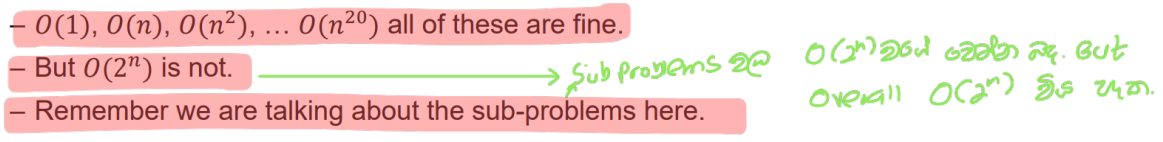
Overlapping subproblems mean that the problem can be broken down into subproblems that are reused multiple times. Rather than solving the same subproblem repeatedly, DP solves each subproblem once and stores the result for future use.

Example: In the Fibonacci sequence, the computation of `F(n-1)` and `F(n-2)` are reused multiple times when computing `F(n)`.



an example for overlapping subproblems.

3. Difference instances of your recurrence should be bounded by Polynomial



Comparison

Feature/Criteria	Dynamic Programming	Brute Force	Divide and Conquer
Approach	Breaks problem into overlapping subproblems; solves each subproblem once and stores the results (memoization or tabulation).	Systematically tries all possible solutions to find the correct one.	Breaks problem into non-overlapping subproblems, solves each independently, and combines solutions.
Optimal Substructure	Yes	No	Yes
Overlapping Subproblems	Yes	No	No
Efficiency	Typically more efficient, avoids redundant calculations	Typically less efficient due to exhaustive search	More efficient than brute force, but efficiency depends on the nature of the problem
Time Complexity	Usually polynomial (e.g., O(n^2), O(n^3))	Often exponential (e.g., O(2^n), O(n!))	Varies, often O(n log n) or O(n^c) for some constant c
Space Complexity	Uses additional space for memoization or DP table	Minimal space for simple cases; can be large for complex cases	Depends on recursion depth, often O(log n) for balanced problems
Use Cases	Problems with overlapping subproblems and optimal substructure, e.g., Fibonacci, Knapsack, shortest paths	Simple problems or when no better algorithms are known, e.g., small combinatorial problems	Problems that can be divided into smaller independent subproblems, e.g., Merge Sort, Quick Sort, Binary Search
Examples	Fibonacci sequence, Knapsack problem, Matrix chain multiplication	String matching (naive), Traveling Salesman Problem (TSP), Subset sum problem	Merge Sort, Quick Sort, Binary Search, Strassen's matrix multiplication
Implementation Complexity	Moderate to High	Low	Moderate
Redundant Calculations	Avoids by storing results	High, due to repeated work	Minimal, subproblems solved independently

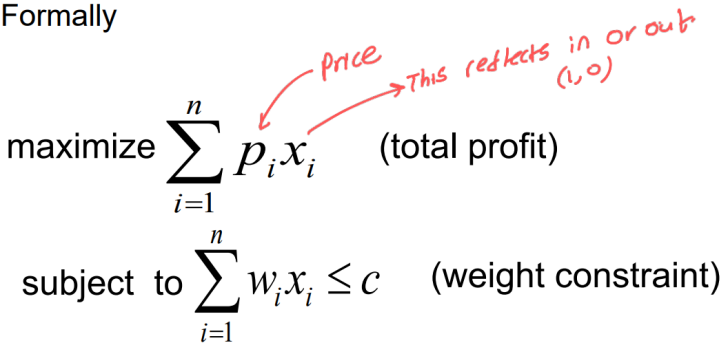
Feature/Criteria	Dynamic Programming	Brute Force	Divide and Conquer
Parallelizability	Difficult to parallelize due to dependencies	Easy to parallelize, each solution independent	Can be parallelized, subproblems independent
Base Case Identification	Required	Not required	Required

- **Dynamic Programming** is efficient for problems with overlapping subproblems and optimal substructure. It avoids redundant calculations but may use significant extra space.
- **Brute Force** is a simple, exhaustive search approach that guarantees finding a solution but is often inefficient and impractical for large problems due to exponential time complexity.
- **Divide and Conquer** is suitable for problems that can be split into independent subproblems. It can be more efficient than brute force and dynamic programming for certain types of problems, especially those that can be divided into balanced parts.

3.The Knapsack Problem

The Knapsack Problem

- Let $x_i = 1$ denote item i is in the knapsack and $x_i = 0$ denote it is not in the knapsack
- Problem Stated Formally



Recursive Solution

Base Case: If no items are left or the capacity of the knapsack is 0, the maximum value is 0.

Recursive Case:

- If the weight of the current item is greater than the remaining capacity, skip the item.
- Otherwise, consider the maximum value obtained by either including the item or excluding it.

Recursive Solution

- Consider the First Item $i = 1$
- If it is Selected;

$$\text{maximize} \quad \sum_{i=2}^{n} p_i x_i \quad \text{subject to} \quad \sum_{i=2}^{n} w_i x_i \le c - w_1$$

- If it is Not Selected;

$$\text{maximize} \quad \sum_{i=2}^{n} p_i x_i \quad \text{subject to} \quad \sum_{i=2}^{n} w_i x_i \le c$$

- Compute Both Cases, Select the Better Option

- $P(i,k)$ Maximum Possible Profit Using Items $i, i+1,\ldots,n$ and Capacity k
- Recursive Expression for $P(i,k)$
  - Case 1: $i = n$
  - Case 2: $i < n$

$$P(i,k) = \begin{cases} 0 & i = n \;\&\; w_i > k \\ P_n & i = n \;\&\; w_i \le k \\ P(i+1,k) & i < n \;\&\; w_i > k \\ \max\{P(i+1,k),\ p_i + P(i+1,k-w_i)\} & i < n \;\&\; w_i \le k \end{cases}$$

item weight > weight capacity

weight limit gets updated.

not taking it

taking it.

- We Can Write a Program for the Recursive Solution Based on the 4 Cases
  - Recursive program will take $O(2^n)$ time (subproblems 2^n තෙක්ම).
- Inefficient Because $P(i,k)$ for the **Same i and k** Will be **Computed Many Times**
- Let's Consider an Example
  - $n = 5, c = 10,$
  - $w = [2, \quad 2, \quad 6, \quad 4, \quad 5]$
  - $p = [12, 25, 24, 15, 14]$

Recursive Solution