# Dynamic Programming

## Dynamic Programming (DP)

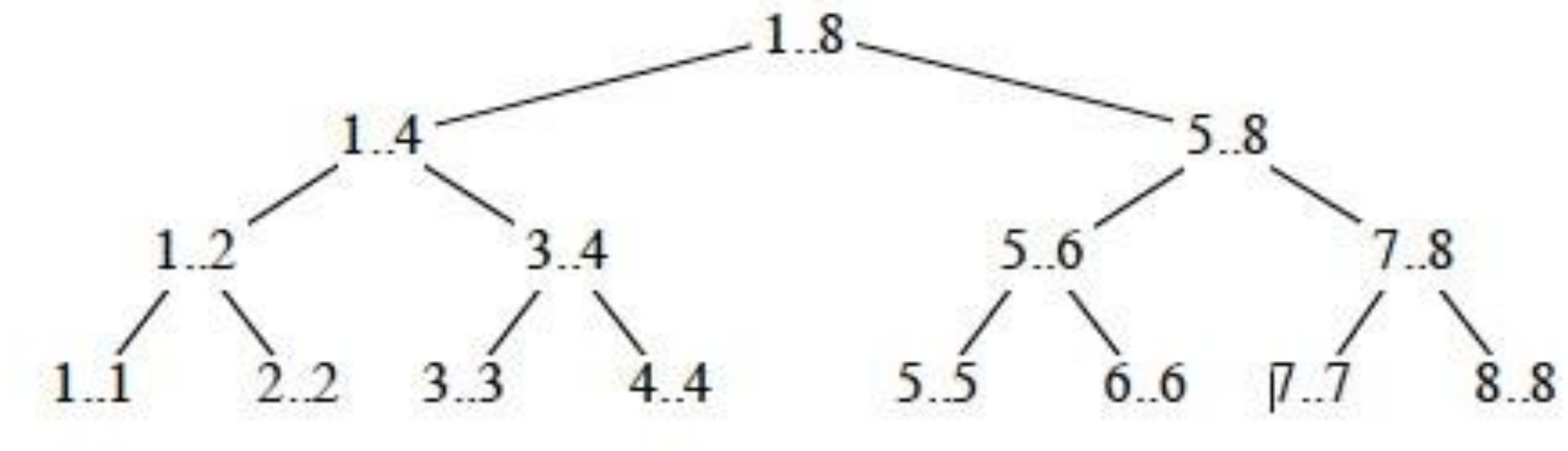
- Identify a small number of sub problems
- It can quickly and correctly solve "large sub problems" given solution to "smaller sub problems"
- After solving all sub problems, it can quickly compute the final solution [usually, its just the answer to the biggest sub problem]

# Comparison of Divide and Conquer with DP

- Dynamic Programming (DP) bears similarities to Divide and Conquer (D&C)
  - Both partition a problem into smaller sub problems and build solution of larger problems from solutions of smaller problems.

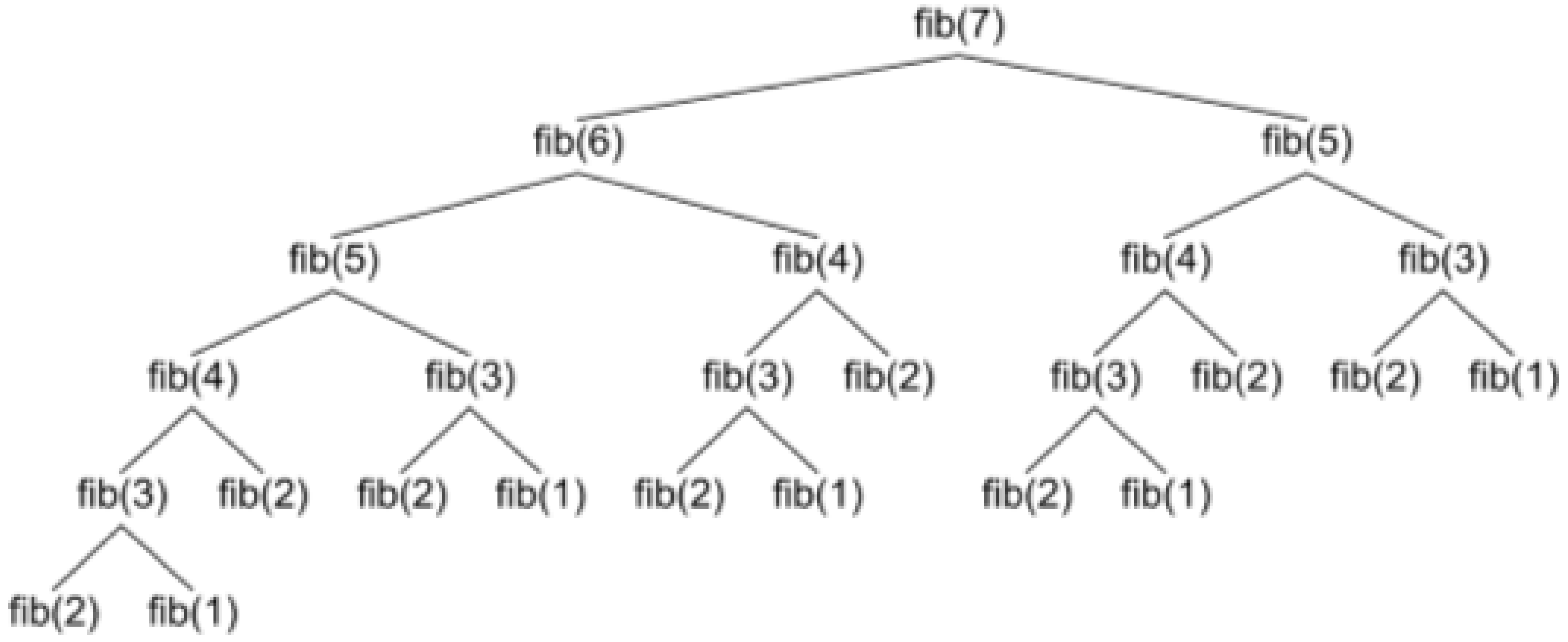
## Comparison of Divide and Conquer with DP

- **divide and conquer** combines solutions to sub problems, but ***applies when the sub problems are disjoint***. For example, here is the recursion tree for merge sort on an array A[1..8]. Notice that the indices at each level do not overlap



## Overlapping sub problems

- **Dynamic programming *applies when the sub problems overlap***. For example, here is the recursion tree for Fibonacci problem. Notice that not only do numbers repeat, but also that there are entire subtrees repeating. It would be redundant to redo the computations in these subtrees.



## Memoization

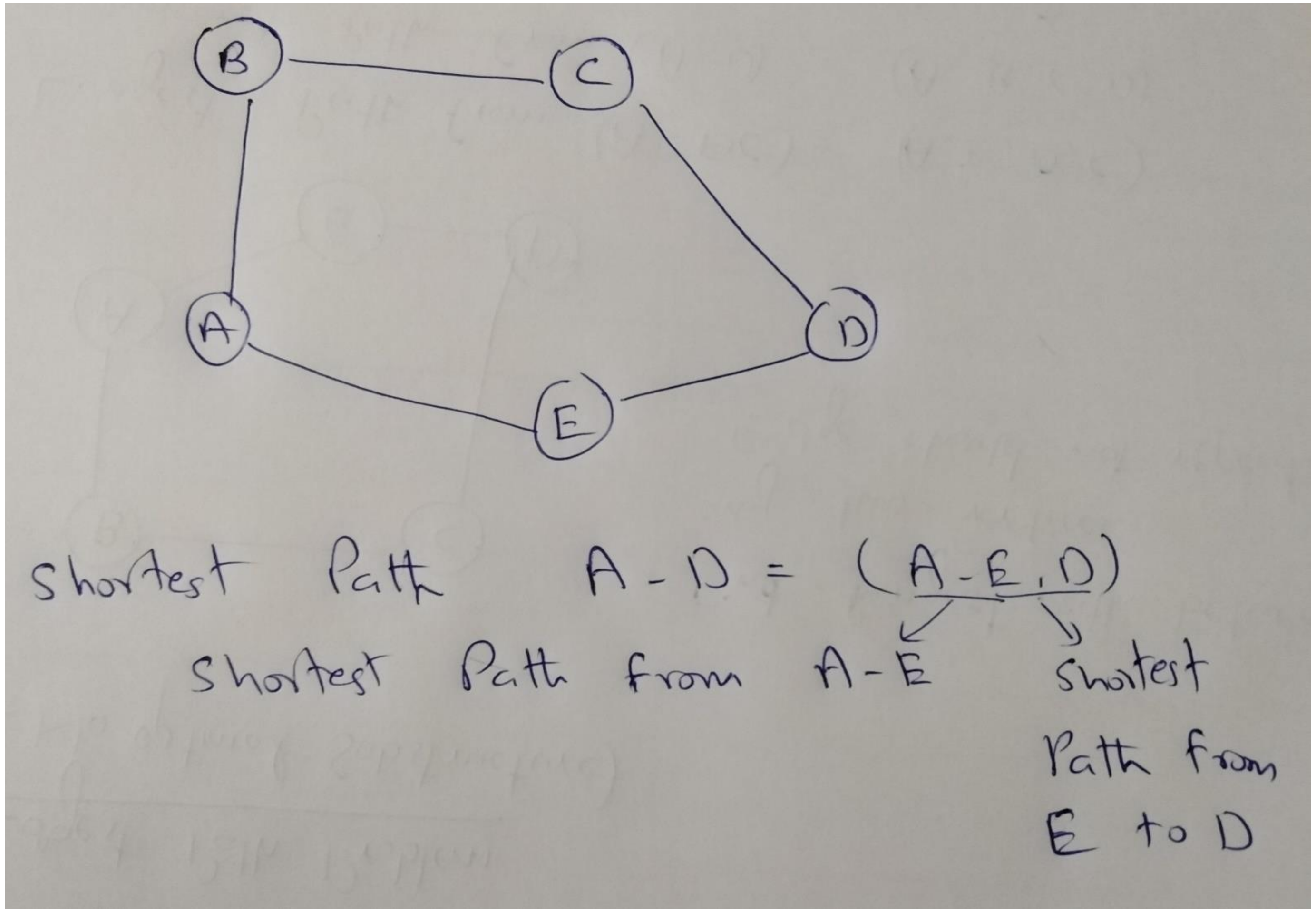
- Dynamic programming *solves each sub problem just once, and saves its answer in a table*, to avoid the re computation. This idea is also called **memoization**

## Optimal Substructure

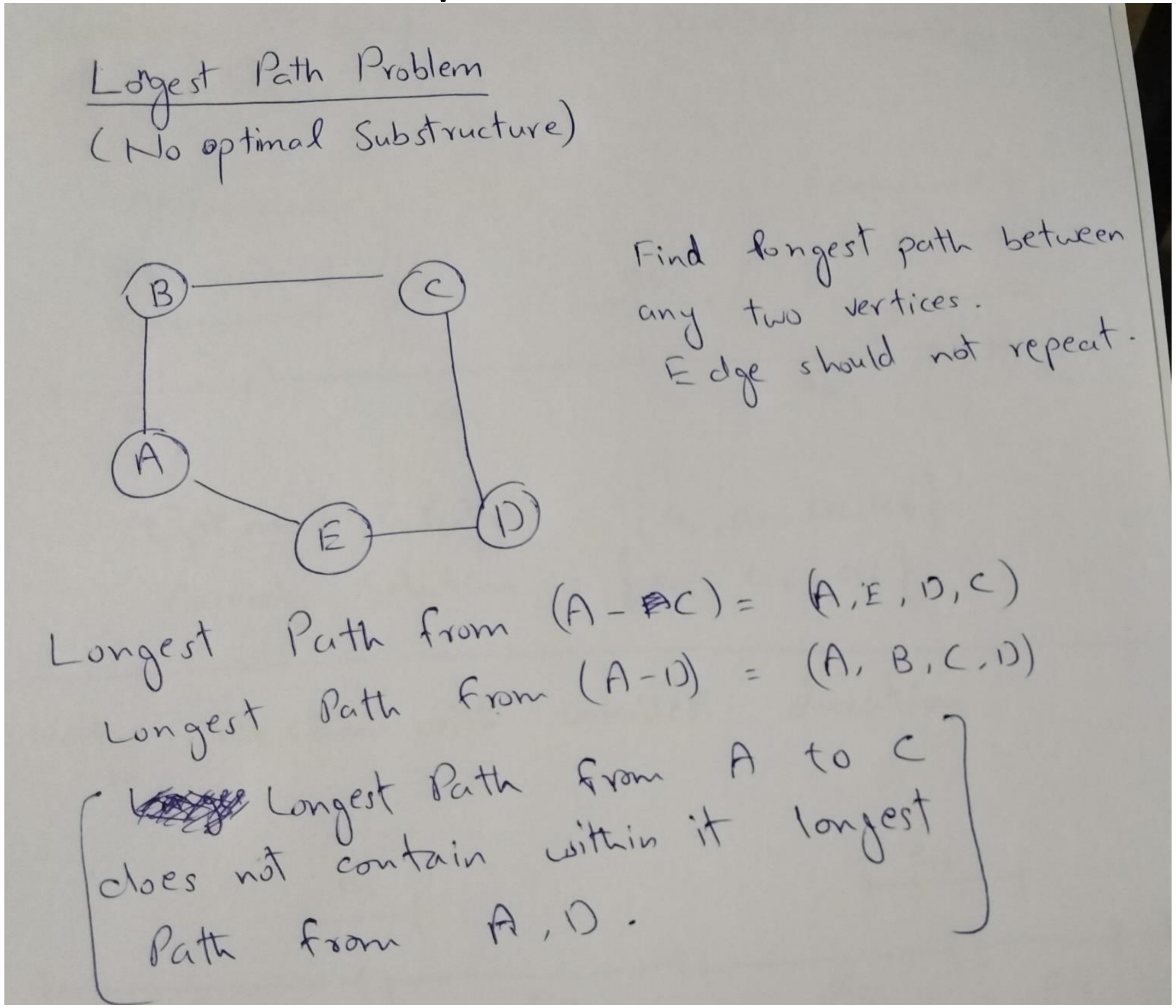
- DP is applicable to problems that have optimal substructure
- **Optimal Substructure:** An optimal solution contains within it optimal solution to its sub problems.

# Optimal Substructure

- Shortest Path Problem has optimal substructure

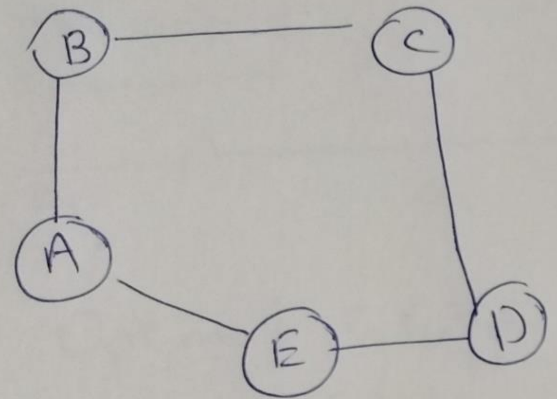


# Problem without optimal substructure



Logest Path Problem

(No optimal Substructure)



Find longest path between any two vertices.
Edge should not repeat.

Longest Path from (A - C) = (A, E, D, C)

Longest Path from (A - D) = (A, B, C, D)

[ Longest Path from A to C does not contain within it longest Path from A, D. ]

## DP is used for Optimization problems

- Problems have many solutions; we want the best one

## Steps in DP

A typical application of the dynamic programming is for optimization problems.

1. Characterize the structure of an optimal solution.
2. Recursively define the value of an optimal solution.
3. Compute the value of an optimal solution, typically in a bottom-up fashion.
4. Construct an optimal solution from computed information.

# Dynamic Programming

- Top Down Approach
- Bottom Up Approach

## Fibonacci Numbers

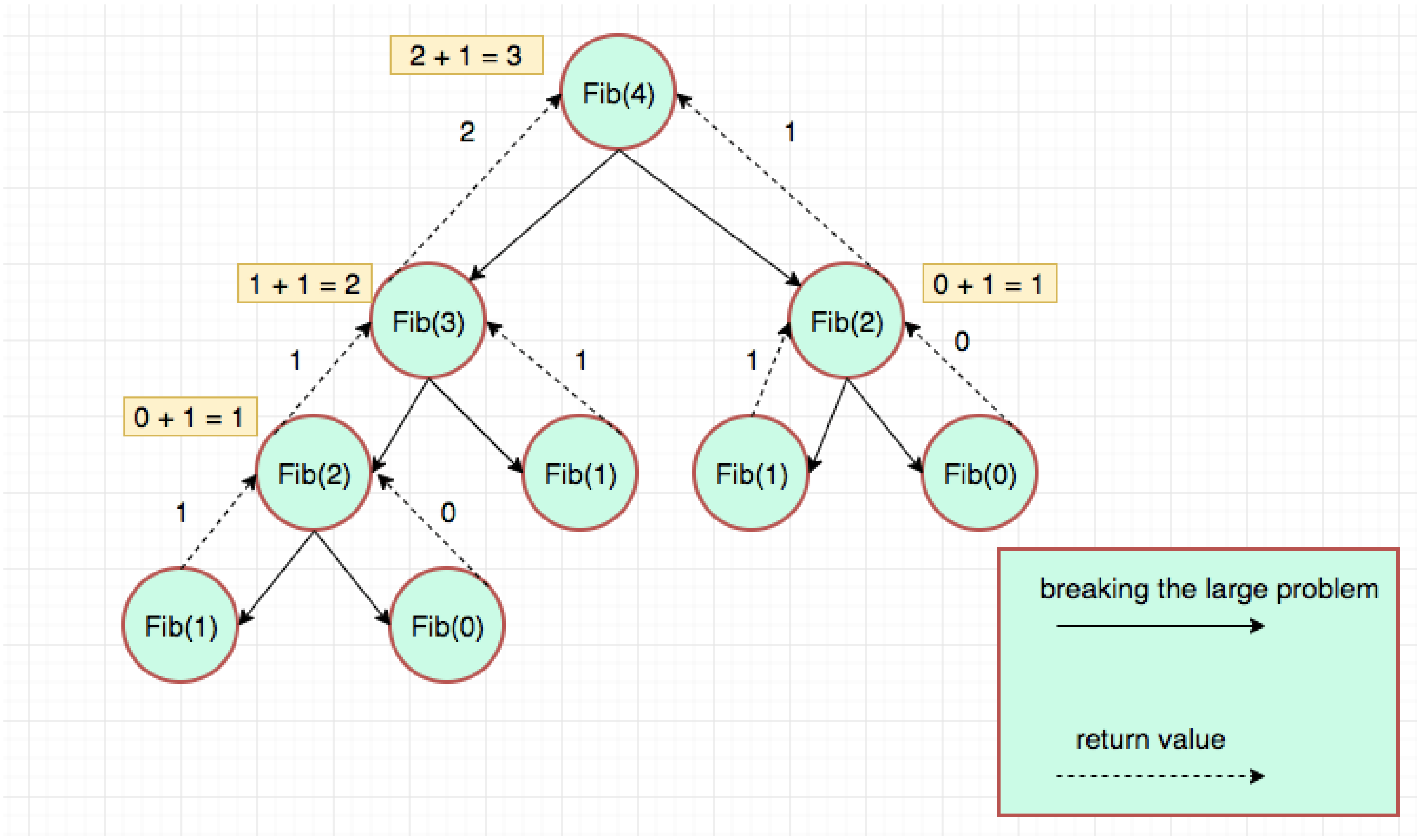
1, 1, 2, 3, 5, 8, 13, …

fib(n) = fib(n-1) + fib(n-2)

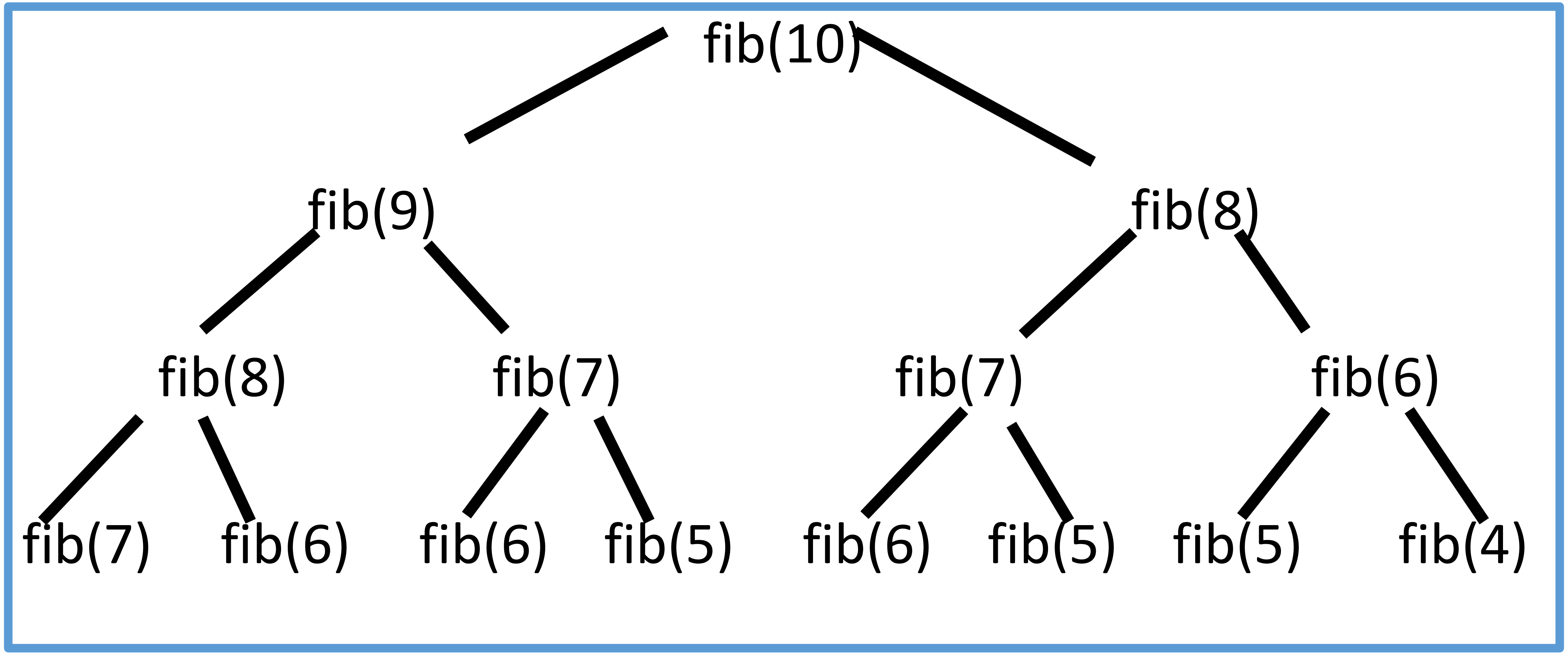
fib(0) = fib(1) = 1

## Top Down Approach (Brute Force Exponential)

```
int Fibonacci(int N)
{
          if(N <= 1)
             return N;
          return Fibonacci(N-1) + Fibonacci(N-2);
}
```


2 + 1 = 3
Fib(4)
2
1
1 + 1 = 2
Fib(3)
0 + 1 = 1
Fib(2)
1
1
1
0
0 + 1 = 1
Fib(2)
Fib(1)
Fib(1)
Fib(0)
1
0
Fib(1)
Fib(0)
breaking the large problem
return value

## Fibonacci Numbers



# Top Down Approach (with Memoization)

```
int result[size];
 //initially set the result array to -1
void init_result()
{
		int i;
		for(i = 0; i < size; i++)
				result[i] = -1; //-1 indicates that the subproblem result needs to be computed
}

 int Fibonacci(int N)
{
		//if the subproblem is not computed yet, //recursively compute and store the result
		if(result[N] == -1)
		{
				if(N <= 1)
						result[N] = N;
				else
						result[N] = Fibonacci(N-1) + Fibonacci(N-2);
		}
		//Otherwise, just return the result
		return result[N];
 }
```

# Fibonacci Numbers: Bottom Up Appraoch

| n | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| fib(n) | | | | | | | | | | | | | |

- We did this by hand in much less than exponential time. How?
- We looked up previous results in the table, re-using past computation.
- Big Idea: Keep an array of **sub-problem solutions**, use it to avoid re-computing results!

## Fibonacci – Bottom Up

```
public int fibonacci(int n) {
    fib = new int[n];
    fib[0] = fib[1] = 1;
        for (int i = 2; i < n; ++i) {
        fib[i] = fib[i - 1] + fib[i - 2];
    }
    return fib[n];
}
```

## Fibonacci – Bottom Up

```
public int fibonacci(int n) {
    fib = new int[n];
    fib[0] = fib[1] = 1;
        for (int i = 2; i < n; ++i) {
        fib[i] = fib[i - 1] + fib[i - 2];
    }
    return fib[n];
}
```

## Assignment Question

- Here's a recurrence. Write the brute force solution and also the dynamic programming solution (top down and bottom up) for this recurrence,

$$C(N) = \frac{2}{N}(\sum_{i=0}^{N-1} C(i)) + N$$

$$C(0) = 1$$