Aniket Thakur (24BM6JP05) **Date 09-jan-2025**

RECAP

We solved and designed algorithms for various problems step by step in previous classes, and while solving "Fibonacci Numbers," we found **Identical subproblems** during solution refinement.

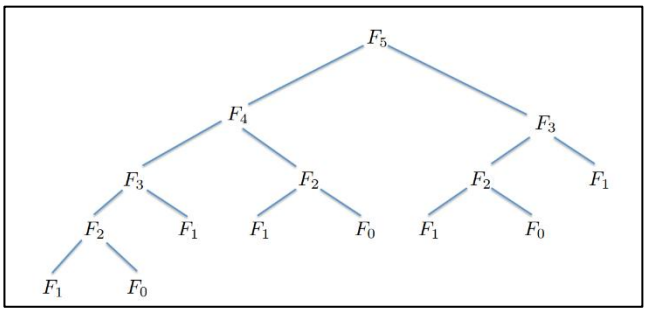


Fig1.1 Recursion without memoization

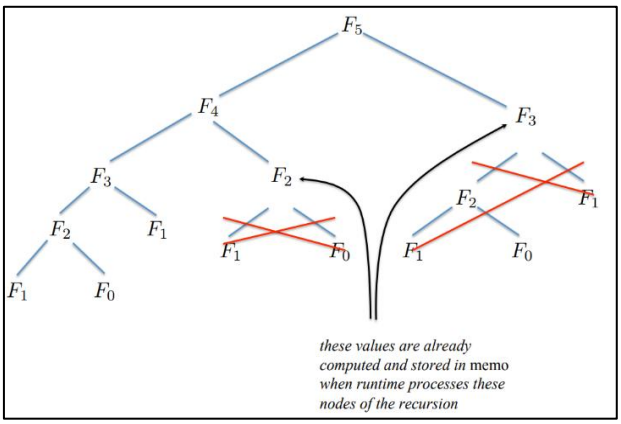


Figure 1.2 Recursion if memoization is used

We found that we were computing $F(n)$ each time from scratch which was not needed.

Never recompute a subproblem $F(k), k \leq n$ if it has been computed before.

This technique of remembering previously computed values is called "MEMOIZATION."

And this technique gives rise to

Dynamic Programming (DP) Algorithm

DP ≈ recursion + memoization (i.e. re-use)

It is mainly an optimization over plain recursion. Wherever we see a recursive solution that has repeated calls for the same inputs, we can optimize it using Dynamic Programming.

The idea is to store the results of subproblems so that we do not have to re-compute them when needed later. This simple optimization typically reduces time complexities from exponential to polynomial.

We will use this in our Fibonacci numbers problem and solve it. We will do it in two ways- recursive version and iterative version.

Fibonacci numbers

Recursive version (Top Down)

Declare a global array of size(n) and call it FIB[ ].

Create a Boolean array, DONE[ ], where DONE[0] = 1, DONE[1] = 1, all others are 0.

n	0	1	2	3	4
DONE[n]	1	1	0	0	0
FIB[n]	0	1	0	0	0

```
Function eval_f(n):
    If DONE(n) = 1 Then
        Return Fib(n)
    Else
        Fib(n) = eval_f(n - 1) + eval_f(n - 2)
        DONE(n) = 1
        Return Fib(n)
```

This comes with initial conditions(base conditions)

$Fib(0) = 0\,,\ DONE(0) = 1\ and\ Fib(1) = 1 \quad DONE(1) = 1$

Time complexity $\boldsymbol{O(n)}$: as each Fibonacci number is computed once.

Space complexity $\boldsymbol{O(n)}$: due to the memoization table and recursion call stack.

Iterative version (Bottom-Up)

(calculates Fibonacci numbers starting from smallest values)

```
Function eval_f(n):
    Fib(0)=0    Fib(1)=1
    for i=2,3,…..,n
      { z ⟵ Fib(0) +Fib(1)
        Fib(0)⟵ Fib(1)
        Fib(1)⟵ z
       }
    Return Fib(1)
```

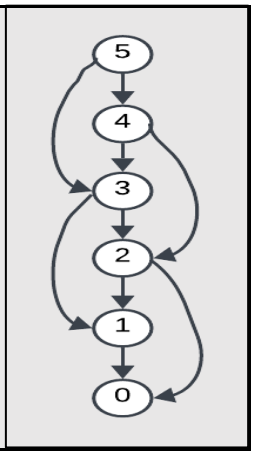


Figure 1.3 Bottom-Up approach

Time complexity $\boldsymbol{O(n)}$: loop runs (n-1) times to calculate F(n).

Space complexity $\boldsymbol{O(1)}$: only uses two values to store current and previous Fibonacci numbers.

Generalization

In general, we can say the following

$$f(n) = \begin{cases} f\big(g(n)\big) + f\big(h(n)\big), if\ c(n) = false \\ b(n) \qquad\qquad\quad , if\ c(n) = true \end{cases}$$

Where $c(n)$ is BASE condition.

and include one more case, if it's **a cycle, i.e.**, **F(n) calling F(n) again.** In that case we tell it to exit.

DONE(n) → 0 *if not computed*

DONE(n) → 1 *if it's a CYCLE*

DONE(n) → 2 *if already computed*

For Fibonacci numbers

```
Function eval_f(n):
   if(DONE(n)=2){
      Fib(n)←b(n)
      Return Fib(n)
      }
    if(DONE(n)=1){'CYCLE' exit}
    DONE(n)=1
    X←g(n), y←h(n)
    Fib(n)←eval_f(x) + eval_f(y)
    DONE(n)=2
    return Fib(n)
```

We will design an algorithm for another interesting problem, the **coin selection problem.**

Coin Selection Problem

Given a set C of n coins having denomination values $\{C_1, C_2, \cdots, C_n\}$ and a desired final value of **V**, find the minimum number of coins to be chosen from **C** to get an exact value of **V** from the sum of denominations of the selected subset.

`Coins (U,P,x,y,n)`

U: set of coins selected till now

P: remaining set of coins from which we can select

x: value of set S

y: remaining value desired to chosen from T

n: number of coins selected

⟨U, n⟩ = `Coins(U,P,x,y,n)`

This is the output where **U** is the coins we used, and **n** is the number of coins used.

Base conditions

• If (y = 0) return ⟨U, n⟩ *(as it's a solution or at least initial solution)*

• If (y < 0) return ⟨NULL, ∞⟩ *(it means we have gone more than desired value and so no solution)*

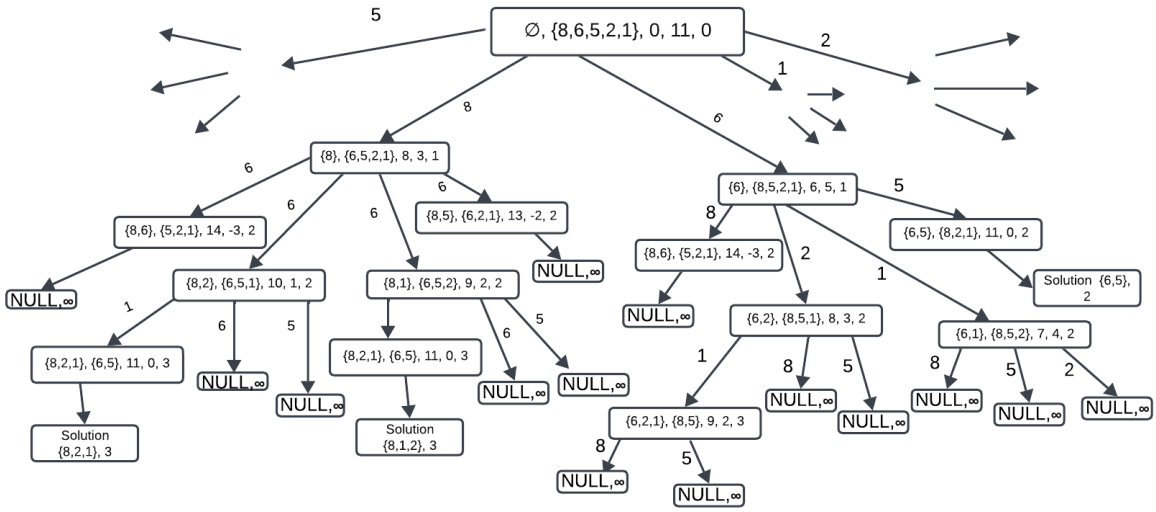
• If (P = NULL) return ⟨NULL, ∞⟩ *(as no coin is left and $y \neq 0$ as we have checked that first)*

• P_{min} = NULL, $d_{min} = \infty$

From naïve approach, We can use a brute force recursion and try every conceivable combination of taking coins to equal the desired amount, adding them all up to determine how many ways there are to get the desired amount. We will take each element and then next till we reach our base conditions and we will do it for all the elements.

Starting with the full set of coins {8, 6, 5, 2, 1} and a target sum of 11, the algorithm recursively selects one coin and then continues exploring combinations with the remaining coins. For example, starting with coin 8, it explores all combinations with the remaining coins {6, 5, 2, 1}, then moves to coin 6, and so on. This exhaustive search continues until we reach the BASE conditions.

In the diagram, it has been done for only two branches, but it will happen for all the branches.



The algorithm for this approach is:

```
Coins(U,P,x,y,n){

if (y = 0) return (U, n)

if (y < 0) return (NULL, ∞)

if (y < 0) return (NULL, ∞)

Pmin = NULL, dmin = ∞

for(i=1 to n){

  U'⟵ U ⋃ {Ci}

  P'⟵ P - {Ci}

       }

⟨s,m⟩ = Coins(U',P', x+Ci, y-Ci, n+1)

if (m<dmin) {

  dmin ⟵ m

  Umin ⟵ s

        }
```

```
}

return (U_min, d_min)
```

Time Complexity

For this, let's take the worst case in which our value decreases by only 1 each time, and also, in each branch, that is, for every single cell, there are n calls(for loop). Therefore, the recurrence relation is :

$$\boldsymbol{T(n) = nT(n-1) + O(n)}$$

And if we solve this either by substitution. Substituting recursively, it expands to:

$using\ T(n-1) = (n-1) \cdot T(n-2) + O(n-1)$

$put\ it\ back\ in\ and\ we\ got$

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

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

Continuing this pattern leads to:

$\boldsymbol{T(n) = n! + \ lower\ order\ terms}$.

$O(n^n)$ is sometimes used as an upper bound for n! because:

$n! \leq n^n$

This simplification avoids dealing with Stirling's approximation and factorials. Thus, the **time complexity is** $O(n^n)$

We can also see from the **recursive tree**.

1st level - n nodes, then (n-1) then (n-2) and so on...

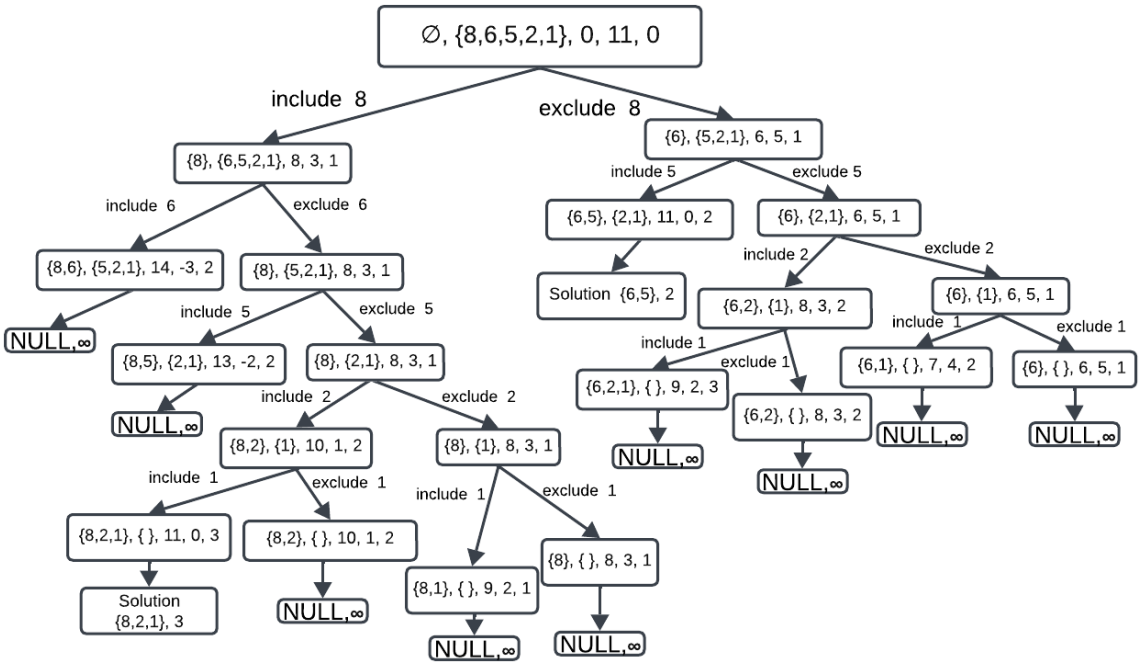
Total nodes - n * (n-1) * (n-2) ..... = **n! (Permutation)**

But it is very bad as it's **exponential** and also notice Given C= {8,6,5,2,1} V =11. The minimum solution {6,5} will come from path 6 -> 5 and path 5 -> 6, and also, we are calculating for the same case again and again in different branches. This is going to generate a graph, and identical nodes will be coming. So, we have identified **identical sub-problems** from our algorithm.

Solution Refinement

To avoid solving identical subproblems in the coin selection problem, we can apply strategy by creating two cases at each step: one where we include the current coin in the solution and another where we exclude it. For each subsequent branch, we apply the same approach to the remaining coins. This method ensures that once a coin (e.g., 8) is included in one branch, it is excluded from the other. As a result, if we choose coin 6 in the branch without coin 8, it prevents the formation of identical subproblems and eliminates redundant computations.

The **Recursive Tree** will now look like following



At each node, the algorithm makes a binary decision: either include the current coin in the solution or exclude it. This decision branches the recursion, progressively reducing the target sum. For example, starting from the full set, one branch includes coin 8 (reducing the sum to 3), while the other excludes it and continues with the remaining coins. This process continues until either a valid solution is found (e.g., {8, 2, 1}) or the path leads to an invalid state (e.g., a negative sum or no remaining coins) (BASE conditions). This is called as **Inclusion-Exclusion Principle.**

This ensures we don't have variable branching like in the previous case. It will have a binary branch, one for the included list and one for the excluded list. This will also generate all subsets.

Time Complexity

For this, let's take the worst case in which our value decreases by only 1 each time, and also, from each branch there are 2 branches. Therefore, the recurrence relation is :

$$\boldsymbol{T(n) = 2T(n-1) + O(1)}$$

Solving it by substitution

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

Substituting $T(n-1)$

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

$T(n) = 2^2T(n-2) + 2 \cdot O(1) + O(1)$

$Continuing\ this\ expansion$:

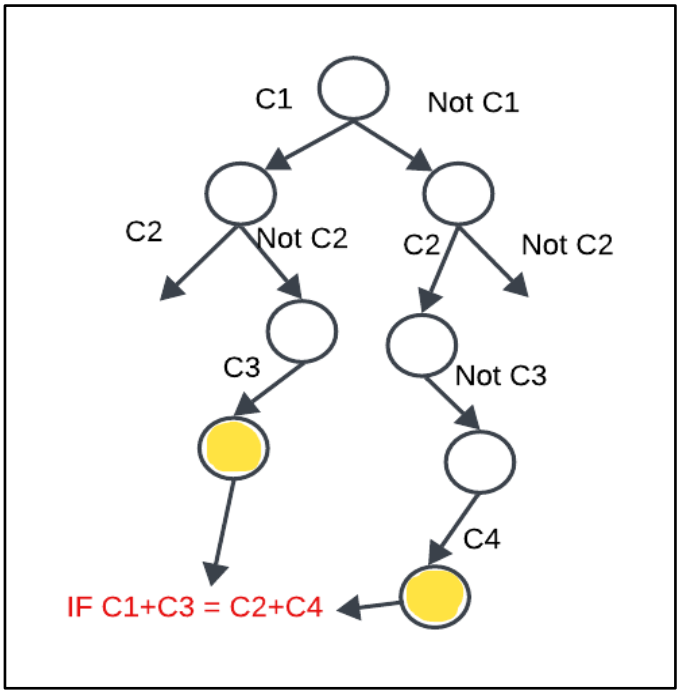
$$T(n) = 2^kT(n-k) + \sum_{i=0}^{k-1} 2^i\ O(1)$$

Let $k = n$, so $T(n-n) = T(0)$ and $T(0) = 1$.

$$T(n) = 2^n T(0) + \sum_{i=0}^{n-1} 2^i \, O(1)$$

$$T(n) = 2^n + O(2^n) \qquad using \sum_{i=0}^{n-1} 2^i = 2^n - 1$$

We get time complexity $\boldsymbol{O(2^n)}$
This is better than the previous case, but the time complexity is still exponential.
Notice that now there is no same snapshot of problems that we are solving again and again, but there can be cases that do come under **identical subproblems.**
When **the same number of coins have already been given the same amount of value.**



If C1+C3 = C2+C4 and we used two coins in both cases, we do not need to do it repeatedly.

To solve this problem, there are only two parameters, **the number of coins(n) and sum(amount),** that change in the recursive solution. So, we create a 2D matrix of size $\boldsymbol{n} * (\boldsymbol{sum} + \mathbf{1})$ for **memoization**.

Each cell in $\boldsymbol{i^{th}}$ *row and* $\boldsymbol{j^{th}}$ *column* gives $(\boldsymbol{U_{min}}, \boldsymbol{n_{min}})$ using coins till $\boldsymbol{i}$ and making value sum $\boldsymbol{j}$. We initialized the matrix with $(\boldsymbol{NULL}, \infty)$.
That is, (2,3) cell is giving a set of coins from {1,2} to make the sum of value 4, and our solution will be last cell (**intersection of last row and last column)** that is using all coins and getting our desired value.
The memoization matrix for our problem looks like following:

W coins	0	1	2	3	4	5	6	7	8	9	10	11
Ø	Ø 0	NULL, ∞	NULL, ∞	NULL, ∞	NULL, ∞	NULL, ∞	NULL, ∞	NULL, ∞	NULL, ∞	NULL, ∞	NULL, ∞	NULL, ∞
1	Ø 0	{1} 1	NULL, ∞	NULL, ∞	NULL, ∞	NULL, ∞	NULL, ∞	NULL, ∞	NULL, ∞	NULL, ∞	NULL, ∞	NULL, ∞

2	∅ 0	{1} 1	{2} 1	{1,2} 2	NULL, ∞	NULL, ∞	NULL, ∞	NULL, ∞	NULL, ∞	NULL, ∞	NULL, ∞	NULL, ∞
5	∅ 0	{1} 1	{2} 1	{1,2} 2	NULL, ∞	{5} 1	{1,5} 2	{2,5} 2	{1,5,2} 3	NULL, ∞	NULL, ∞	NULL, ∞
6	∅ 0	{1} 1	{2} 1	{1,2} 2	NULL, ∞	{5} 1	{6} 1	{1,6} 2	{2,6} 2	{1,2,6} 3	NULL, ∞	{6,5} 2
8	∅ 0	{1} 1	{2} 1	{1,2} 2	NULL, ∞	{5} 1	{6} 1	{1,6} 2	{8} 1	{1,8} 2	{2,8} 2	{6,5} 2

H.W Do the above problem if you have infinite coins of all denominations. Make the table like the above, too.

Following is the way to fill the table's cells: to fill, we will either include the coin of that row or not include the coin of that row. **W** is target value.

- If **coin(i) > w**, then coin(i) will not be included and we will just take value from upper cell.
- If **coin(i) ≤ w** then we will consider both cases, including the coin as well as excluding it and filling the $minimum\ of\ them$.

Time Complexity

$$\boldsymbol{O(n * V)}$$

This is called pseudo-polynomial complexity due to the presence of V, as it depends on the target value.

- **n**: Number of coin denominations.
- **V**: Target sum.
- Every subproblem (combination of coin index and remaining sum) is solved once due to memoization.

Space Complexity:

$$\boldsymbol{O(n * V)}$$

- Space for the 2D memoization table of size $(n + 1) * (V + 1)$.
- Additional space for the recursion stack (maximum depth N), but it's absorbed in $O(n * V)$

Now during implementation, the time complexity can be $O(V)$.

Each DP state only depends on the previous row (i.e., results from the previous coin). Instead of storing the entire $\boldsymbol{n * V}$ table, use only **two rows**:

1. **Current row** for the current coin.
2. **Previous row** for the previous coin.

We just need to know two cases, including the coin(current row) or excluding it(previous row). This reduces space usage from **O(n × V)** to **O(2 × V)**, which simplifies to **O(V)**.

A set that forms a matroid can be used to solve the coin-changing problem by using a ***greedy approach*** ***(it always selects the coin with the largest denomination not exceeding the remaining sum).***

H.W Search about matroid property and greedy algorithm will be discussed in next session.
