Scribed Lecture - Week 1

FADML

IIT Kharagpur

January 2025

Anirudh A (24BM6JP06)

Foundations of Algorithm Design

To solve any given problem, we come up with solutions (in this case, algorithms) in a step-by-step manner, outlining the general framework initially by making use of constructs like branching, looping, etc. The step-by-step process that can be converted to code is called pseudocode. However, the algorithm we come up with has to be efficient in addition to being correct.

Outlined below is a structured method to come up with efficient algorithms.

1. Initial Solution

(a) Recursive Definition - A set of solutions
(b) Inductive Proof of Correctness
(c) Analysis using recurrence relations

2. Exploration of Possibilities

(a) Decomposition or unfolding of the recursion tree
(b) Examination of structures formed
(c) Re-composition properties

3. Choice of Solution & Complexity Analysis

(a) Balancing the split, choosing paths
(b) Identical sub-problems

4. Data Structures & Complexity Analysis
(a) Remembering past computations for future
(b) Space complexity

5. Final Algorithm & Complexity Analysis
(a) Traversal of the recursion tree
(b) Pruning

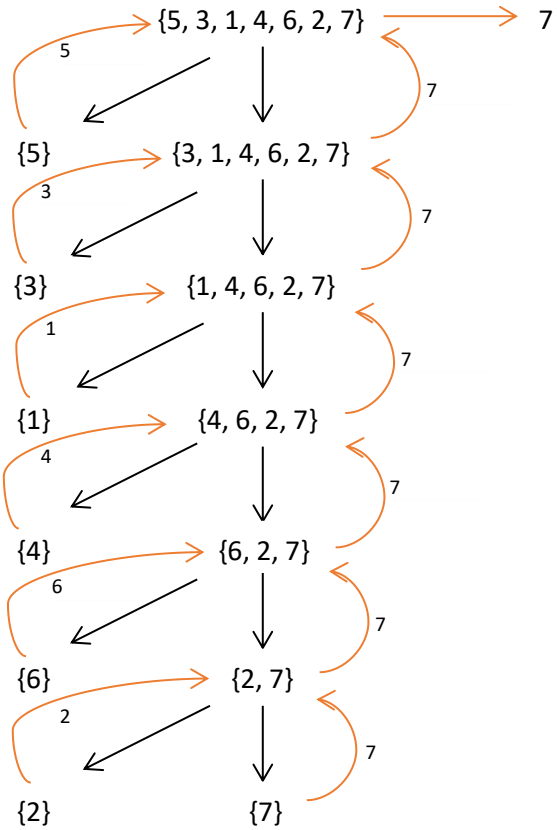
6. Implementation

(a) Available memory, time, quality of solution, etc.

It is imperative to get a precise definition of the problem before proceeding to solve it. We usually try to construct a simplified version of the problem, adding complexities later on.

Problem 1: Return max of L

```
def max(L):
{
if |L|=1 return x1
L' = L – {x1}
y=max(L')
if x ≥ y return x1
else return y
}
```

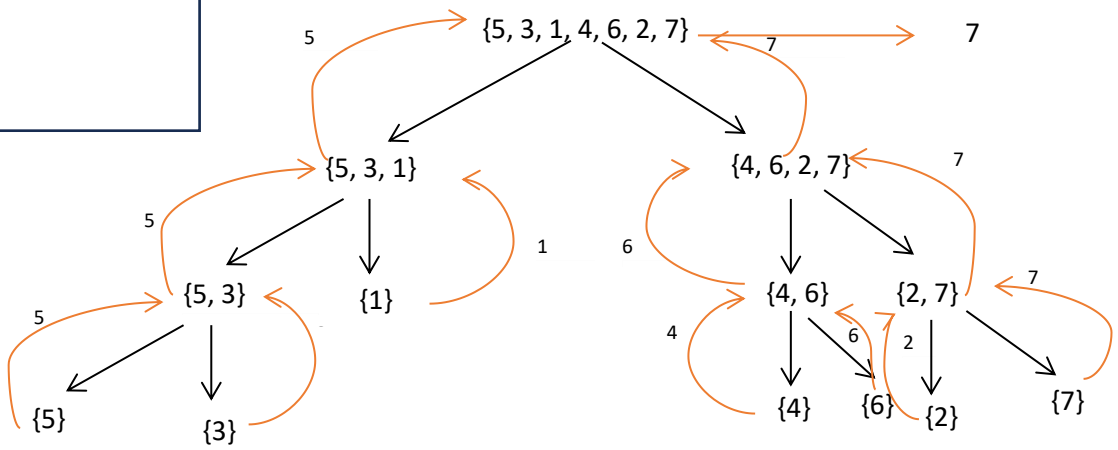


Time Complexity:

$T(n) = T(n - 1) + 1 = n – 1$

This decomposition is not general enough. Let's check if it is the only way to find largest element.
Some observations that can be made are:
1. We can remove any element, not just the first.
2. We can bisect the array and compare.
Essentially, we can just divide the array into two non-empty sets L_1 and L_2.

```
def max2(L):
{
if |L|=1 return x1
Split L into 2 non-empty sets L1, L2
y1 = max2L1
y2 = max2L2
if y1 ≥ y2 return y1
else return y2
}
```



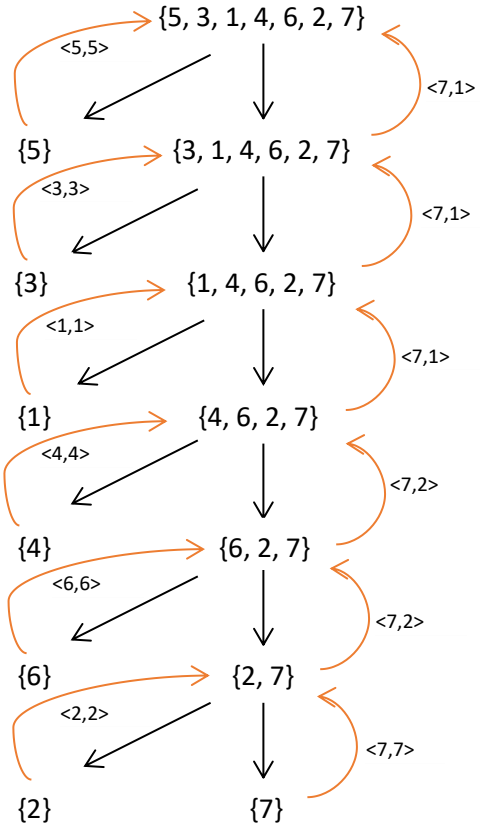
Time Complexity:

$T(n) = T(k) + T(n - k) + 1 = n - 1$

The above operation can't be performed in less than n - 1 terms. So, the above algorithm we have obtained is optimal. Since we are doing n – 1 comparisons anyway, we might as well take the first element for split instead of finding some other point to split, since it would be unnecessary additional effort.

<u>Problem 2:</u> Return max and min of L

```
def maxmin(L):
{
    if |L| = 1   return < x1, x1 >
    L' = L – {x1}
    < y1, y2 > = maxmin(L')
    if x1 ≥ y1 then m1 = x1
    else m1 = y1
    if x1 ≤ y2 then m2 = x1
    else m2 = y2
    return < m1, m2 >
}
```

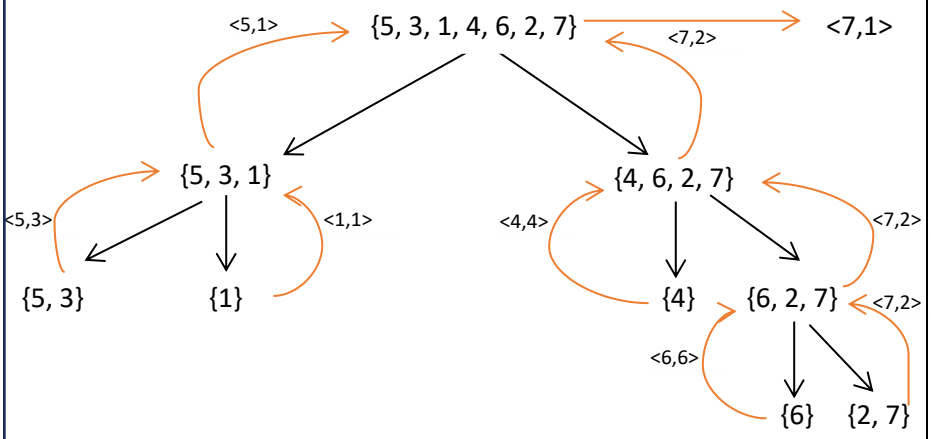


Time Complexity:

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

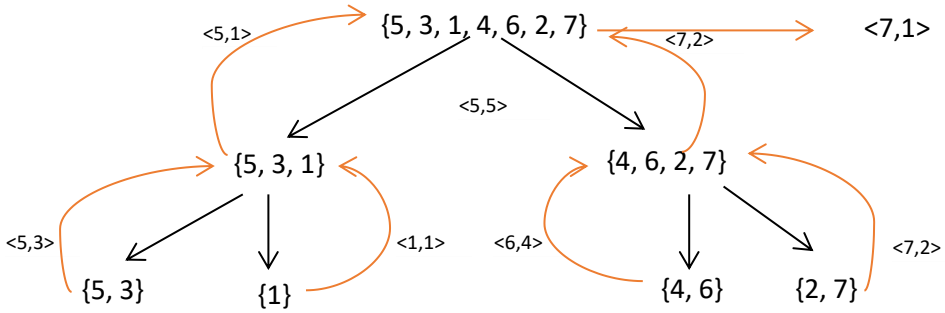
Let us look at an alternative solution where we recursively split L into 2 non-empty sets, and check how many comparisons are required.

```
def maxmin2(L):
{
 if |L| = 1   return < x1, x1 >
 if |L| = 2 {
 if x1 > x2 return < x1, x2 >
 else return < x2, x1 >
 }
 Split L into 2 non-empty sets L1, L2
 < y1, y2 > = maxmin2(L1)
 < z1, z2 > = maxmin2(L2)
 if y1 ≥ z1 then m1 = y1
 else m1 = z1
 if y2 ≤ z2 then m2 = y2
 else m2 = z2
 return < m1, m2 >
}
```



The above method requires **10 comparisons**, 1 each at nodes where there are 2 elements and 2 each at nodes where there are more than 2 elements.

In the above problem, had we gone for even sized splits at each step, we would have obtained the lowest possible cost.



The above method requires **only 9 comparisons**, 1 each at nodes where there are 2 elements and 2 each at nodes where there are more than 2 elements.

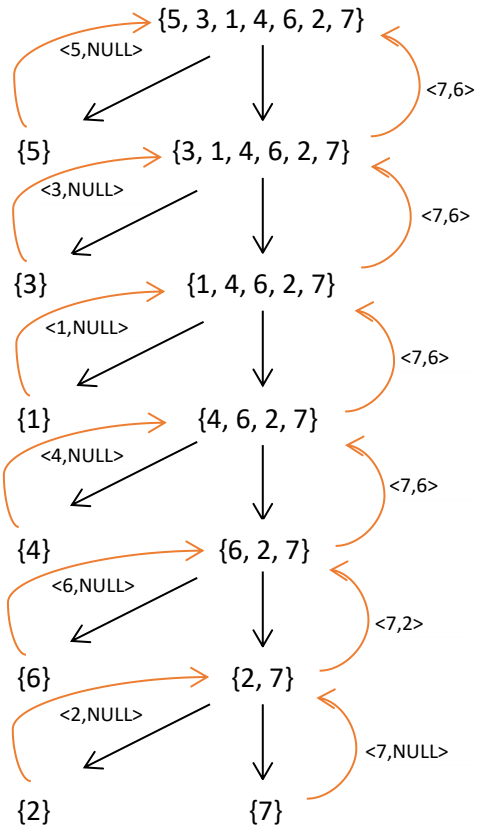
The number of comparisons in an algorithm where we split the array of size n into two arrays of size k and n − k would be given by,

$$T(n) = \begin{cases} 0 & n = 1 \\ 1 & n = 2 \\ T(n-k) + T(k) + 2 & n > 2 \end{cases}$$

If we choose k = 1, we get T(n) = 2n−3. Choosing k = 2, will result in T(n) = (3n/2) – 1, which is better.

<u>Problem 3:</u> Return first 2 max of L

```
def max1max2(L):
{
    if |L| = 1   return < x1, NULL >
    L' = L – {x1}
    < y1, y2 > = max1max2(L')
    if x1 ≥ y1 then m1 = x1, m2 = y1
    else if x1 ≥ y2 then m1 = y1, m2 = x1
    else m1 = y1, m2 = y2
    return < m1, m2 >
}
```

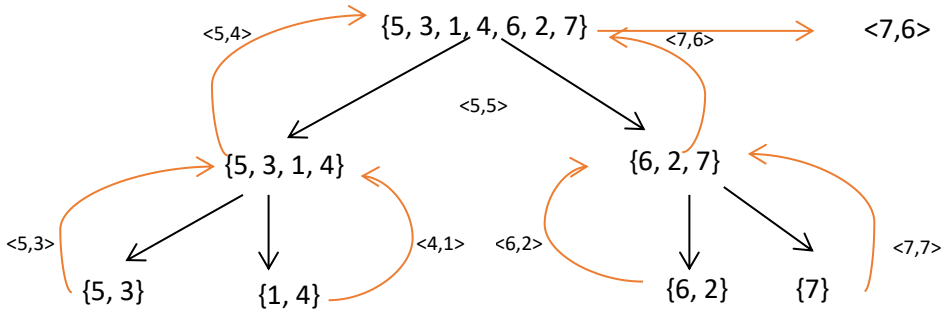


Time Complexity:

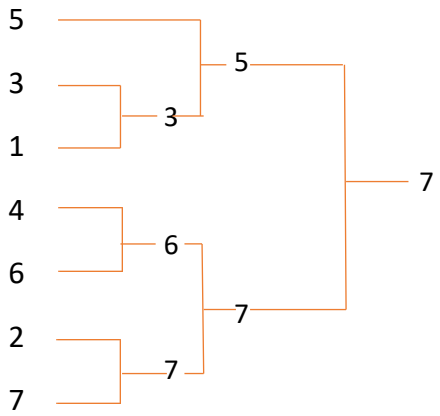
T(n) = T(n-1) + 2 = 2(n-1)

Let us look at an alternative solution where we recursively split L into 2 non-empty sets, similar to the maxmin problem.

```
def max1max2B(L):
{
if |L|=1 return < x1, NULL >
if |L|=2
  if x1 ≥ x2 return < x1, x2 >
  else return < x2, x1 >
Split L into two non-
empty sets L1 and L2.
< y1, y2 >= max1max2B(L1)
< z1, z2 >= max1max2B(L2)
if (y1 ≥ z1)
  m1 = y1
  if (z1 ≥ y2) m2 = z1
  else m2 = y2
else m1 = z1
if y1 ≥ z2 then m2 = y1
else m2 = y1
return < m1, m2 >
}
```



The same problem can be solved by designing an algorithm in which each level is a competition, also called as tournament structure. In such an approach, the structure needs to be created only once while finding the maximum element and stored in memory. Then, it can be traversed to find the subsequent maximum elements.



Finding the largest number will take n-1 comparisons. For the second largest number, we will just traverse the path of the largest element so it will take log n.

Final Algorithms and Data Structuring

Recursive definitions:

Max: Any split is fine. Choose 1, n − 1. The number of comparisons is n – 1.

MaxMin: At each level, split the array into two arrays of size 2, n−2. The number of comparisons is (3n/2) - 2.

Max1Max2: At each level, split the array into two arrays of size 2, n − 2. The number of comparisons is (3n/2) - 2.

Refined definitions:

Max: No further refinement. In order to find the maximum element, at least n − 1 comparisons have to be made. Hence, we have found an algorithm with best complexity possible.

MaxMin: Tournament structure for all the splits for optimal manner yields the same number of comparisons i.e., (3n/2) - 2.

Max1Max2: Balanced tournament split which minimized the height of the tournament allows a more efficient algorithm which can be extended to sorting. The number of comparisons is n - 1 + $\lceil \log_2 n \rceil$.

Exact Time Analysis

Exact Time Analysis is the process of precisely calculating the number of operations an algorithm performs on an input of size n, providing a detailed breakdown of the execution time based on the specific operations involved.

Examples:

1. Summing an array:

- **Algorithm:** Iterates through an array to calculate the sum of elements.
- **Operations:** Initialization(1), loop comparisons(n+1), additions inside the loop(n), return statement(1).
- **Exact Time:** 2n + 3
- **Time Complexity:** O(n)

2. Matrix Addition:

- **Algorithm:** Adds two n×n matrices element by element using nested loops.
- **Operations:** Outer loop(n+1), inner loop (n(n+1)), additions(n×n).
- **Exact Time:** $2n^2 + 2n + 1$
- **Time Complexity:** $O(n^2)$

3. Matrix multiplication:

- **Algorithm:** Multiplies two n×n matrices using three nested loops.
- **Operations:** Three nested loops for n time each, resulting in n^3.
- **Exact Time:** $n^3 + 3n^2 + 2n + 1$
- **Time Complexity:** $O(n^3)$

4. Printing statements:

- For a simple loop printing 'IITKGP' n times, the complexity is O(n).
- If the loop increments by 2, the number of iterations becomes approximately n/2, but the complexity remains O(n).

5. Incremental updates:

- Example: Incrementing a variable p by successive integers until p > n.
- The loop runs approximately $\sqrt{n}$ times, resulting in a time complexity of $O(\sqrt{n})$.

Some more cases:

Statement	Time Complexity
for(i = 0, i < n, i++)	O(n)
for(i = 0, i < n, i = i+2)	O(n)
for(i = n, i > 1, i--)	O(n)
for(i = 1, i < n, i = i*2)	$O(\log_2 n)$
for(i = 1, i < n, i = i*3)	$O(\log_3 n)$
for(i = n, i > 1, i = i/2)	$O(\log_2 n)$

Asymptotic Analysis

Asymptotic analysis is used to evaluate the efficiency of algorithms by studying their growth rates as the input size(n) approaches infinity. It helps compare algorithms by abstracting their behavior for large inputs.

Key Notations:

- **Big-O(O):** Represents the upper bound of a function. For example, if $f(n) \leq c.g(n)$ for large n, then $f(n) = O(g(n))$.
- **Big-Omega(Ω):** Represents the lower bound of a function. For example, if $f(n) \geq c.g(n)$ for large n, then $f(n) = \Omega(g(n))$.
- **Big-Theta(Θ):** Represents the tight bound. $c_1 g(n) \leq f(n) \leq c_2 g(n)$.

Function Growth Comparison

$$1 < \log_2 n < \sqrt{n} < n < n \log n < n^2 < n^3 < \ldots < 2^n < 3^n < \ldots < n^n$$

Examples:

1. $f(n) = n^2 \log(n) + n$

Upper Bound: $f(n) = O(n^2 \log(n))$
Lower Bound: $f(n) = \Omega(n^2 \log(n))$
Tight Bound: $f(n) = \Theta(n^2 \log(n))$

2. f(n) = n!

Upper Bound: $f(n) = O(n^n)$
Lower Bound: $f(n) = \Omega(1)$

Examples of Recursive Relations

Recursive relations describe the relationship between the solution to a problem and its subproblems. Solving these relations involves finding the time complexity using substitution, recursion trees or Master theorem.

1. Linear Recursion:

$T(n) = T(n-1) + O(1)$
Solves to $T(n) = O(n)$

2. Exponential Recursion:

$T(n) = 2T(n-1) + O(1)$
Solves to $T(n) = O(2^n)$

3. Logarithmic Recursion:

$T(n) = T(n/2) + O(1)$
Solves to $T(n) = O(\log(n))$

4. Merge Sort:

$T(n) = 2T(n-/2) + n$
Solves to $T(n) = O(n \log(n))$

Fibonacci Numbers

Definition

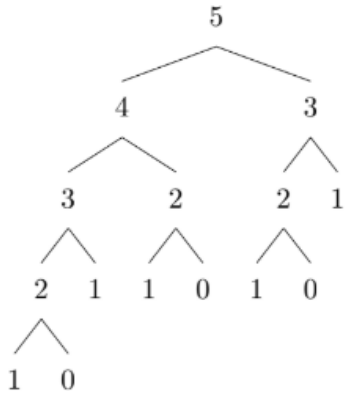
$$f(n) = \begin{cases} 0 & \text{if } n \leq 0 \\ 1 & \text{if } n = 1 \\ f(n-1) + f(n-2) & \text{if } n \geq 1 \end{cases}$$

Empirical Formula

$$f(n) = \frac{\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n}{\sqrt{5}}, \forall n \in N$$

Initial Solution:

```
fib(n)
{
  if(n = 0 or n = 1) return n
  m = fib(n-1) + fib(n-2)
  return m
}
```



Time Complexity:

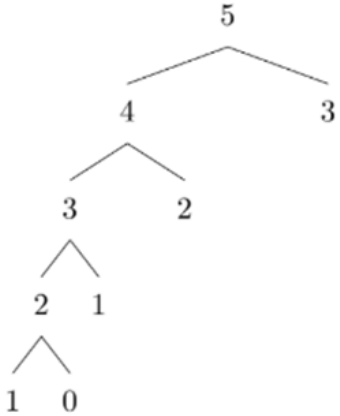
$T(n) = T(n-1) + T(n-2) + 1 = O(2^n)$

We are solving the same problem repeatedly. It would be better if we remembered past computations and compute f(n) only once for every n. One way to optimize such a structure is to store the values of computed nodes in a table. When an already solved subproblem is encountered, the value from the table can be reused. This is called **memoization.**

Improved solution:

```
fib2(n)
{
  if(DONE[n] = 2)
    fib(n) = b(n)
    return(fib(n))
  if(DONE[n] = 1)
    "CYCLE ENDS"
  DONE[n] = 1
  x = g(n), y = h(n)
  fib(n) = fib2(x) + fib2(y)
  DONE[n] = 2
  return fib(n)
}
```

DONE = 0	Computation is yet to be done
DONE = 1	Computation cycle has been encountered
DONE = 2	Computation is completed



FIB	0	1	1	2	3	5
DONE	1	1	1	1	1	1

Time Complexity:

$T(n) = O(n)$

The chart above shows the memoization table and pruned recursive structure. The Fibonacci sequence can also be solved using an iterative method as shown below:

```
fib[0] = 0
fib[1] = 1
fib3(n)
{
  for i = 2 to n do
    fib[i-1] + fib[i-2]
}
```

Coin Selection Problem

Given a set C of n coins having denomination values $\{C_1, C_2, \ldots, C_n\}$ 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 chosen 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 U
y	Remaining value desired to be chosen from P
n	Number of coins selected

Output: ⟨U, n⟩ = Coins(U, P, x, y, n)
U is the coins we used and n is the number of coins used.

If (y = 0) return <U, n>
If (y < 0) return <NULL, ∞>
If (P = NULL) return <NULL, ∞>
P_{min} = NULL, d_{min} = ∞

For the initial solution, we can use brute force recursion and try out every possible combination of taking coins equivalent to the desired amount, adding them all up to determine how many ways there are to get the desired amount.

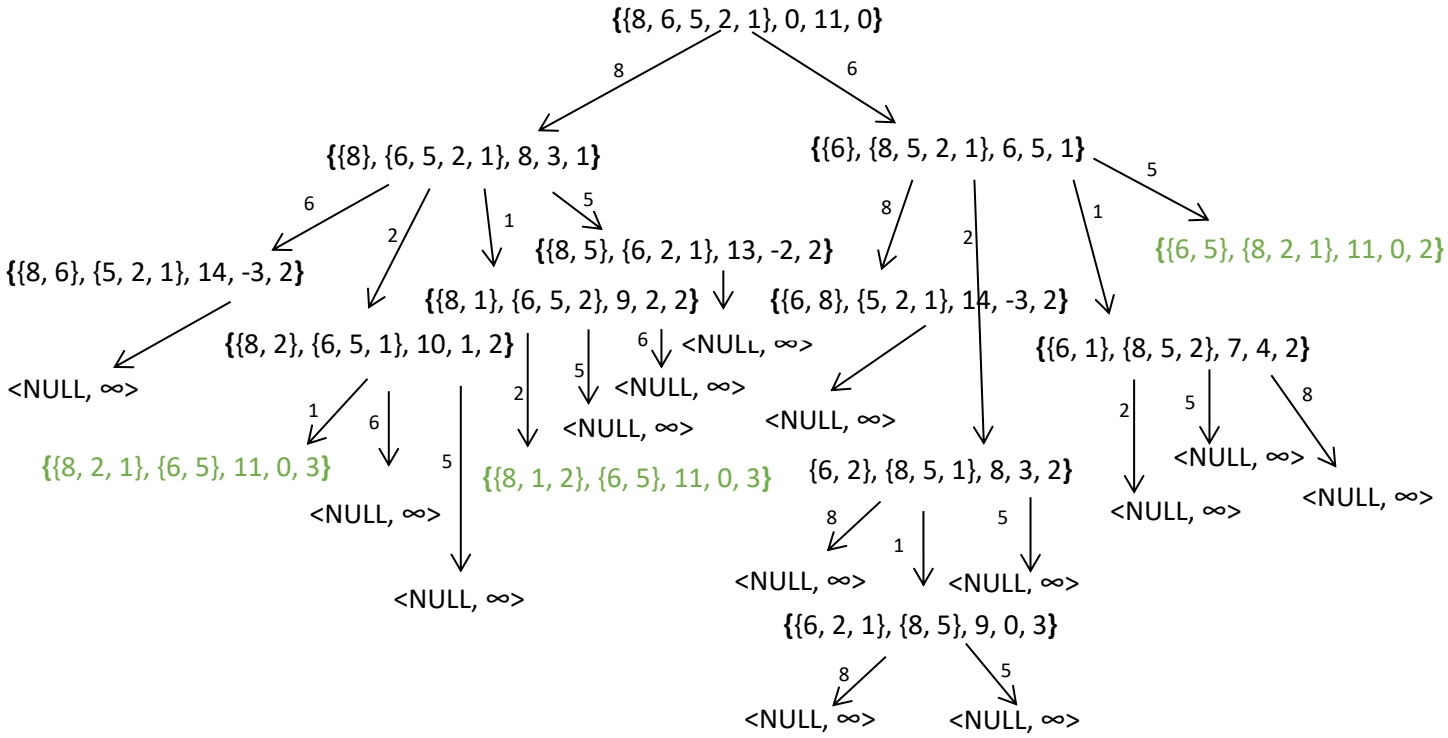
Initial Solution:

```
Coins(U, P, x, y, n)
{
  if (y = 0) return <U, n>
  if (y < 0) return <NULL, ∞>
  if (y < 0) return <NULL, ∞>
  P_min = NULL, d_min = ∞
  for(i = 1 to n)
  {
    U' <- U ∪ {C_i}
    P' <- P - {C_i}
  }
  <s, m> = Coins(U', P', x + C_i, y-C_i, n+1)
  if (m < d_min)
  {
    d_min <- m
    U_min <- s
  }
return  <U_min, d_min>
}
```

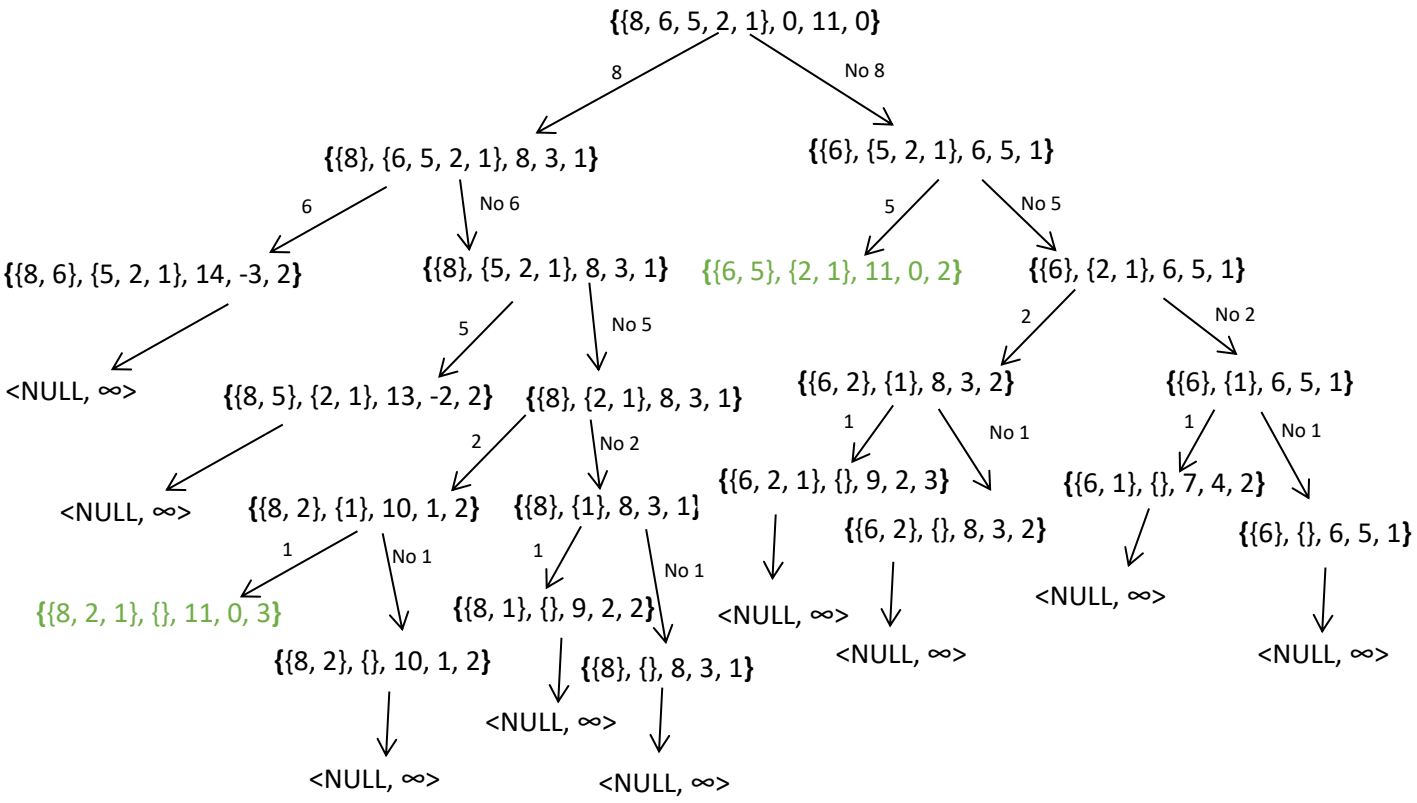
Time Complexity:

$T(n) = nT(n-1) + O(n) = O(n^n)$

Note: Branching done below only for 6 and 8, it will happen for all the elements present.



The above approach encounters identical sub-problems, which makes it inefficient and time consuming. To avoid this problem, we can create strictly two cases each step: one where we include the current coin and the other where we exclude it. The same approach is applied to all the coins. This ensures that once a coin is included in one branch, it is excluded from the other, thereby eliminating formation of identical sub-problems and eliminating redundant computations. This is called the **inclusion-exclusion principle**.



At each node, the algorithm makes a binary decision, either to include or exclude the coin. This process continues till a valid solution is found or the path leads to an invalid state, which are the base conditions.

Time Complexity:

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

Although this is better than the initial solution, the time complexity is still exponential. There are no same snapshots of problems that we are solving repeatedly, but there are cases that can come under identical subproblems. For instance, we use different coins to get the solution but the number of coins remains the same, then it is redundant.

To solve this problem, we tweak the number of coins and sum in the recursive solution.

We create a 2D matrix of size n*(sum+1) for memoization. Each cell in i^{th} row and j^{th} column gives (U_{min}, n_{min}) using coins till i and making value sum j. We initialized the matrix with (NULL, ∞).

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 obtaining our desired value.

Time Complexity:

O(n*V)

n - number of coin denominations
V - Target sum

Space Complexity:

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 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).