Recursion & Backtracking

Agenda

1. Recursion
    - What is recursion?
        - Recursion flow
    - How Recursion Works
        - The base case
        - The recursive case
        - Stack Overflow
2. Backtracking
    - Warm-up
    - Motivation
    - What’s backtracking?
        - How it works
        - Common approach to implement a backtracking solution
        - Types of Backtracking Problems
    - Example 1: Subset sum
    - Try by yourself first: 0-1 Knapsack problem
    - Example 2: N-Queens problem
    - To solve

1. Recursion

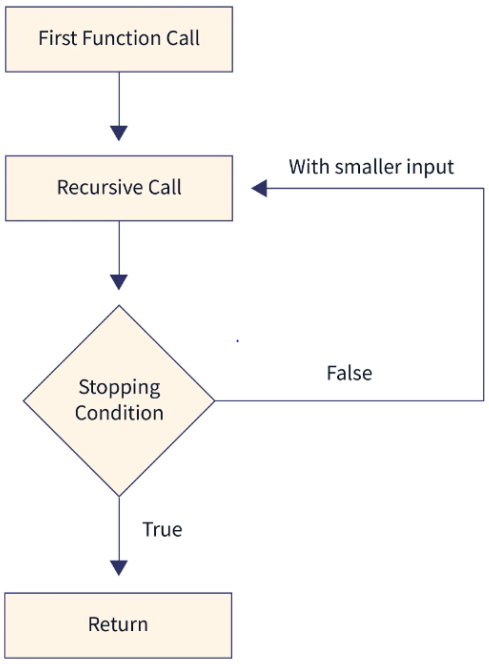
- ## What is Recursion:

  Recursion is a programming technique that involves breaking a problem down into smaller sub-tasks and repeatedly calling the function to solve them. It provides an alternative to the iterative method, which can be more time-consuming and require more effort.

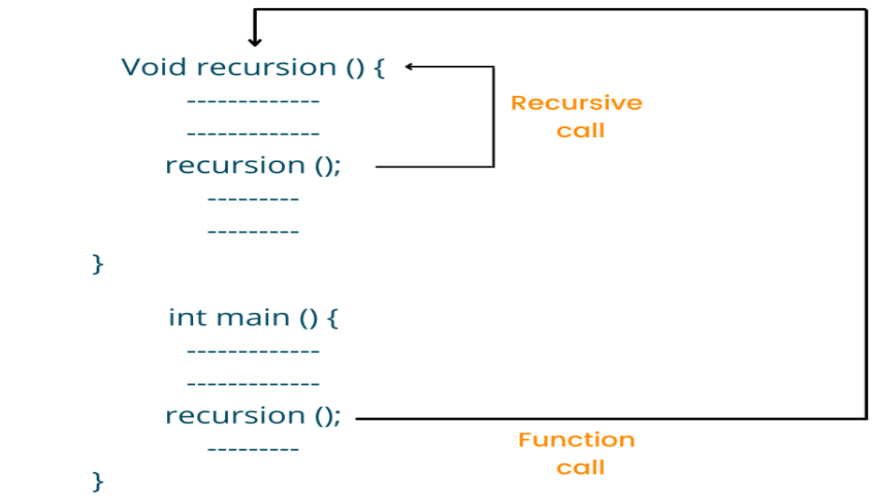
  It is a useful technique that makes code shorter and easier to read and write.

  - ### Recursion flow:

    The recursion process begins with the initial function call and continues until a specified condition is met, at which point the process stops and the function returns.



```
void recursion() {
    // base case
    if () {
        return;
     }
    recursion();
}

int main() {
    recursion();
}
```



- ## How Recursion Works

To Write a recursive function, you need to consider two things:

1. **The base case:**

   The base case in recursion is a fundamental condition that serves as the termination point for the recursive algorithm. It defines the scenario in which the recursive function stops calling itself and returns a specific result directly, without further recursion. Base cases are essential in recursion to prevent infinite recursion and to provide a final answer or outcome for the problem being solved.

2. **The recursive case:**

   The recursive case in recursion refers to a specific condition within a recursive function or algorithm where the function calls itself with a modified input, typically moving closer to a base case, in order to break down a complex problem into simpler, more manageable subproblems. This process continues until the base case is reached, allowing the algorithm to solve the problem by aggregating the results from all the recursive calls.

   What do you think is the output of the following functions if they are called with 5?

```
void fun(int i) {
    if(i == 0)
        return;
    fun(i - 1);
    cout << i <<'\n';
}
```

```
void fun(int i) {
    if(i == 0)
        return;
    cout << i <<'\n';
    fun(i - 1);
}
```

- **Stack Overflow:**
  - A stack overflow is a software error that occurs when a program attempts to use more memory than is available on the stack, causing the program to crash.
  - If a recursive function doesn't reach or define a base case, the recursive calls will never stop, which will cause a stack overflow problem.
  - To avoid infinite recursion, Ensure that your recursive function has well-defined base cases. These are the conditions under which the recursion terminates, preventing further recursive calls. Make sure your base cases are reachable and correctly designed to cover all scenarios.

What do you think is the output of the following function if it is called with 5?

```
int fact(int n) {
    return n * fact(n - 1);
}
```

2. Backtracking (Recursive complete search)

- **Warm-up:**

  Suppose you have N tasks, and for each task, you know its duration in minutes. You have only M free minutes, and you want to finish tasks as much as possible.

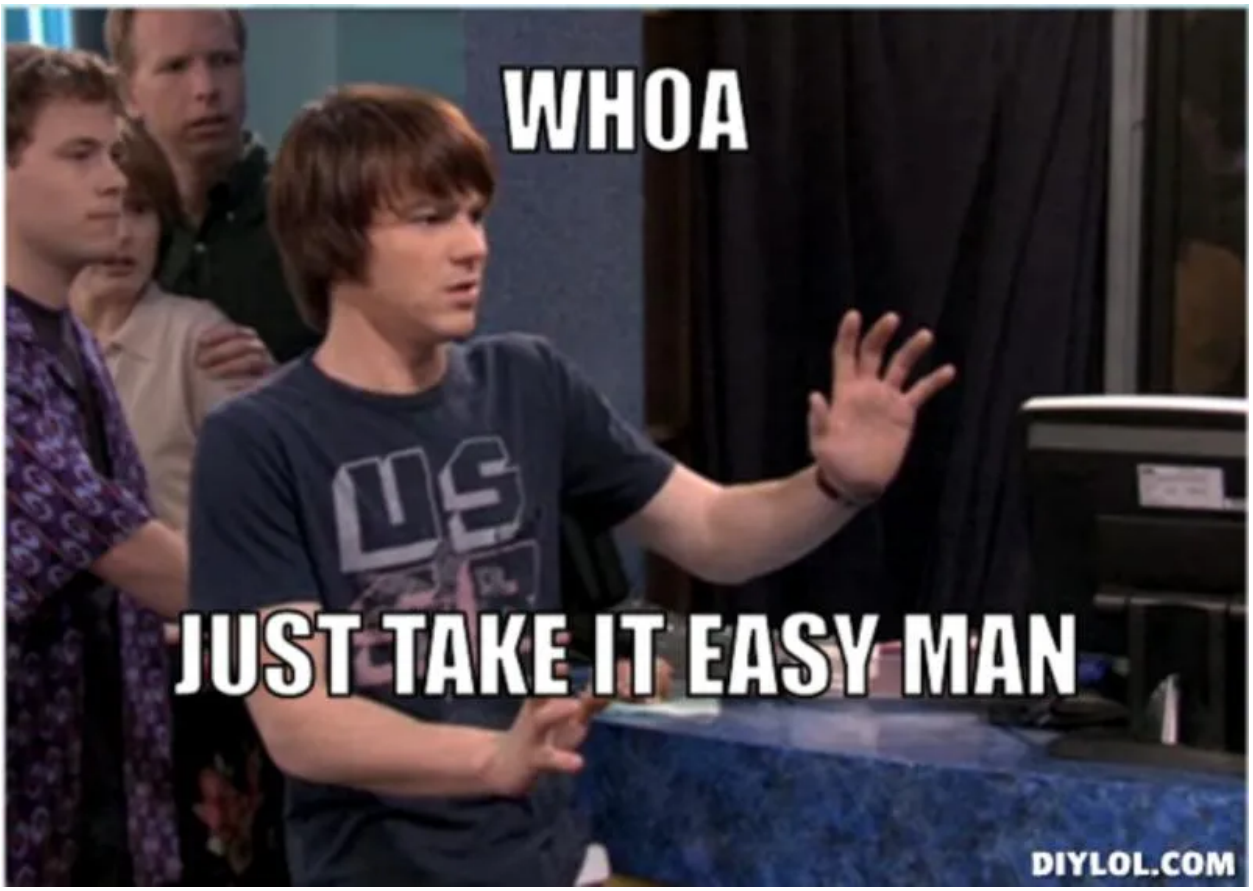
  Code:

```
int main(){
  int n, m;
  cin >> n >> m;
  vector<int> tasks(n);
  for(auto& task: tasks)
     cin >> task;
  sort(tasks.begin(), tasks.end());
  int count = 0;
  for(auto& task : tasks){
      if(task <= m){
            count++;
            m -= task;
      }
      else break;
  }
  cout << count <<'\n';
  return 0;
}
```

Now let's try to solve a harder version..

- **Motivation**

We are given N items where each item has some weight and profit associated with it. We are also given a bag with capacity W, [i.e., the bag can hold at most W weight in it. The target is to put the items into the bag such that the sum of profits associated with them is the maximum possible**.**



In this version the last solution won't bring the optimal answer, so what should we do?

- **What is Backtracking?**

  Backtracking is a recursive algorithm that incrementally builds solution candidates and excludes those with invalid solutions early. It works by testing the validity of the candidate at each step and backtracking if it is found to be invalid. This continues until a valid solution is found or all candidates have been explored.

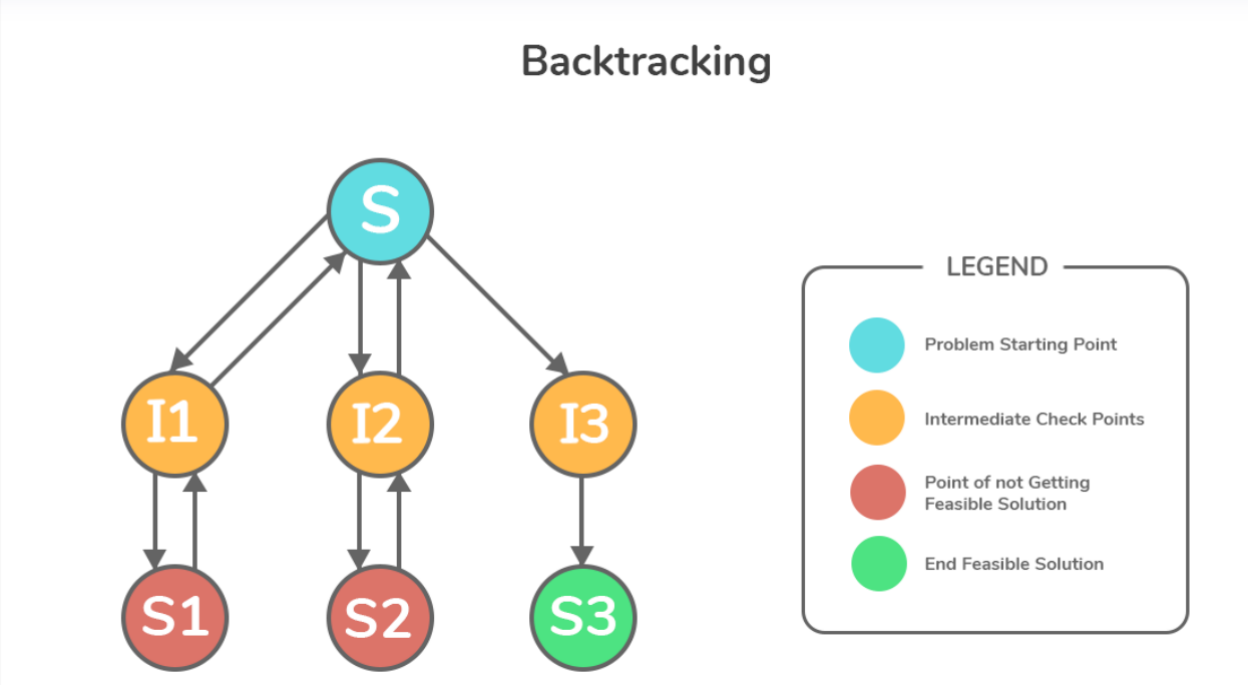
  Backtracking solves problems recursively by constructing potential solutions and removing those that violate the problem constraints.

  The backtracking method tries all possible solutions and selects the best ones by discarding those that don't work.

  Backtracking is efficient for problems with multiple possible solutions.

  Backtracking searches for a solution that includes checkpoints. If a checkpoint fails, it tries another path. This continues until a solution is found or all options are exhausted.

  - **How it works:**



- **Common approach to implement a backtracking solution:**
  1. You are currently in some state.
  2. You have some options that move you to other states.
  3. Check for the invalid states.
  4. For the valid states: **do - recurse - undo**.
- **Types of Backtracking Problems:**

  Problems associated with backtracking can be categorized into 3 categories:
  1. **Decision Problems:** We search for a feasible solution.
  2. **Optimization Problems:** We search for the best solution.
  3. **Enumeration Problems:** We find the set of all possible feasible solutions to the problems of this type.

Example 1: Subset sum

Given an array A of N unique numbers and a target sum T. You are asked to print all the subsets that sum up to the target sum.

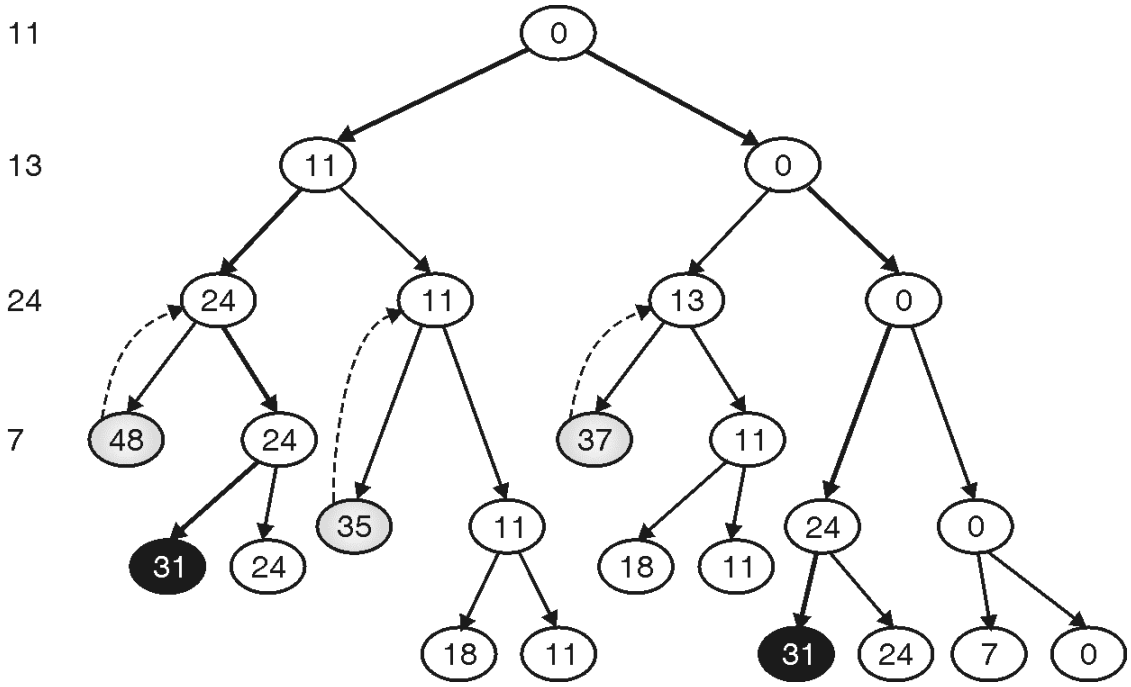
```
Input:
N = 5, T = 6
A = [1, 5, 2, 7, 3]

Output:
1 5
1 2 3
```

One way to find subsets that sum to T is to consider all possible subsets. A power set is a set containing all possible subsets of a given set. The size of such a power set is 2^n.

Let’s visualize how Backtracking works with this example:

```
Input:
N = 4, T = 31.
A = [11, 13, 24, 7]
```

State-space tree for a given problem is shown here:



Solution:

Items in sub set	Condition	Comment
{ }	0	Initial condition
{ 11 }	11 < 31	Add next element
{ 11, 13 }	24 < 31	Add next element
{ 11, 13, 24 }	48 < 31	Sub set sum exceeds, so backtrack
{ 11, 13, 7 }	31	Solution Found

Code:

```
void solve(int i){
    if(tmp > sum)
        return;
    if(i == n){
        if(tmp == sum){
            for(int x : path) cout << x << ' ';
            cout << '\n';
        }
    }
    else{
        // option 1: pick
        tmp += arr[i];
        path.push_back(arr[i]);
        solve(i + 1);
        tmp -= arr[i];
        path.pop_back();

        //option 2: leave
        solve(i + 1);
    }
}
```

- **Try by yourself first: 0-1 Knapsack problem**

  You are given N items where each item has some weight and profit associated with it. You are also given a bag with capacity W, [i.e., the bag can hold at most W weight in it]. The target is to put the items into the bag such that the sum of profits associated with them is the maximum possible.

Code:

```
int knapSack(int W, int wt[], int val[], int n) {
    // Base Case
    if (n == 0 || W == 0)
        return 0;
    if (wt[n - 1] > W)
        return knapSack(W, wt, val, n - 1);
    else
        return max(
            val[n - 1]
                + knapSack(W - wt[n - 1], wt, val, n - 1),
            knapSack(W, wt, val, n - 1));
}
```

Time Complexity: O(2^n).

- ## Example 2: N-Queens problem

We want to arrange N queens on an NxN chessboard such that no queen can attack any other queen. A queen can attack horizontally, vertically, or diagonally.

How to solve this problem:

- We know that for each row it must contain one queen so we can use this to go through the N rows.
- We start from the first row, and for each cell in the row if it's a valid position for the ith queen, do the following (transition):
    1. **Do**: place the queen in this cell.
    2. **Recurse**: Solve for the next row.
    3. **Undo**: Remove the queen from the cell.
- The cell is valid if there are no queens that share the same column or same diagonals.

- When we finally place all the N queens, we have reached the base case so we return.

Code:

```
int n;
vector<vector<char>> grid;
bool valid(int r, int c) {
    for (int i = 0; i < r; i++)
        if (grid[i][c] == 'Q')
            return 0;
    for (int i = r - 1, j = c + 1; i >= 0 && j < n; i--, j++)
        if (grid[i][j] == 'Q')
            return 0;
    for (int i = r - 1, j = c - 1; i >= 0 && j >= 0; i--, j--)
        if (grid[i][j] == 'Q')
            return 0;
    return 1;
}

void solve(int r) {
    if (r == n) {
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++)
                cout << grid[i][j];
            cout << '\n';
        }
        cout << "----------------------------\n";
    } else {
        for (int c = 0; c < n; c++) {
            if (valid(r, c)) {
                grid[r][c] = 'Q';
                solve(r + 1);
                grid[r][c] = '.';
            }
        }
    }
}
```

To solve:

- All Possible Combinations
- Maximal String
- Back to the 8-Queens
- Tavas and SaDDas
- C - Many Requirements
- 167 - The Sultan’s Successors