**Coin change problem**:

Given a total and some coins. Find unique ways to make the total using these coins.

Infinite supply of coins.

When total is 0, then there is 1 way ( don’t take any coin ).

For the new coin, we can either use it or just use the previous coins.

We already have the answer for previous coins.

Time complexity: O(m\*n)

Space: O(m\*n)

m is the amount and n is number of coins.

Now find the ways to get the total using minimum number of coins.

Assume infinite supply of coins.

We need a base case in Top Down approach only

(that uses Recursion).

In bottom up, we fill the 2d array with the known values.

There are repeating subproblems.

In dynamic programming, know the subproblems.

In minimum number of coins for a change problem, if we cannot make the change using the given coins then answer for that subproblem will be 0.

There could be an assumption that the sum from 0 … total can be made using the given coins.

This can be done by having a coin of value 1.

**Longest Increasing Subsequence**:

Length of the longest increasing subsequence.

Longest non decreasing subsequence

Longest decreasing subsequence

Longest non increasing subsequence

Let’s discuss Longest Non decreasing subsequence problem first.

**Approach 1**: Generate all 2^n subsets.

Validate NDS property. Take longest.

O(2^n) at least.

**Approach 2**: Using DP

O(n^2)

**Approach 3**: O(n\*logn)

In dynamic programming, also learn how to build the complete solution rather than just finding an answer (single value).

* Brute Force
* Recursion with Memoization
* DP
* DP with Binary Search

Thinking:

Lets first try to create a particular subproblem as sequence[i]: length of the longest increasing subsequence starting at index i.

But taking this, we will be building the smaller subproblem by taking so many elements.

Rather focus on taking elements one by one and then building a particular subproblem.

We have to find an answer to a subproblem which is an array of length n.

So we take sequence[i] as subproblem which is equal to the length of the longest increasing subsequence ending at index i. Bottom Up approach.

Previous solution can be solved with the help of recursion and then we can convert that recursion into iteration.

In recursion we can go from forward to backwards or backwards to forwards. Our base case would be either the last element or the first element.

Minimum value of longest increasing subsequence will be 1.

**nlogn approach**:

Take 2 additional arrays.

T and R.

T stores intermediate/temporary results while R stores final results.

Another variable “**len**” which tells the length of lis up till now.

Initialise the R array (result array) with -1.

T stores minimum of the last value of an increasing subsequence of a particular length.

**Patience Sort**:

In the process of sorting, it also calculates the longest running sequence.

Suppose we have solitaire cards.

We create a pile of cards from the given cards.

Patience sort algorithm applies a greedy rule: Always choosing the leftmost eligible pile.

Each of the pile contains ordered cards. We can do a merge of these piles to sort the whole deck.

We can read off the longest increasing subsequence simply by adding pointer as each card is placed.

The pointer links the current card to the top card of the pile immediately on the left.

Now simply start at the rightmost pile and follow the pointers back.

This approach makes the fewest possible piles.

A card at any point of time is part of an increasing subsequence whose length is given by the pile index.

Cards within a pile form a decreasing sequence.

Any increasing sequence can use at most 1 card from each pile.

Length of the pile is always >= length of increasing subsequence.

Since patience sort’s greedy minimises the number of piles and piles are linked with pointers, following the pointers gives us the longest increasing subsequence.

Algorithm runs in nlogn, it examines every card only once. And using binary search it takes additional logn time per card to find the required card.

**Longest Common Subsequence**:

Length of longest common subsequence.

**Knapsack**:

Also see, Sliding Window Algorithm techniques.

**Backtracking**:

We make choices.

We have constraints on these choices.

At the end we are going to converge towards a goal.

**Choice**:

Think of the choice that we make at every step.

In a sudoku, we need to fill in the cells by making choices.

Our key choice will be made on a cell. A cell sits in a row and a column.

**How do we actually solve for a cell**?

Divide the problem into subproblems.

For a cell, we can **choose** a number from our decision space { 1, 2, …. 9 }

When we place a number in a cell,

we check whether it breaks the row or whether it breaks the column or whether it breaks the subgrid.

If a particular move is valid move, we recurse on it.

The **goal is reach the base cases**.

If the decision does not work out, then once we come back from our exploration we need to eject that decision.

Every recursion call has a goal: Was the sudoku solvable given the placements that we just did?

**How to know whether we have a backtracking problem**?

We can know about it when it is easy to express the answer recursively.

“Generate All”

“Compute All”

Words implying exhaustion of decision space.

**Egg Dropping Problem:**

Given certain amount of floors (**k**) and certain amount (**n**) of eggs, what is the **least amount of**

**egg drops** that I need to perform to find the pivotal floor where right below it if I drop the egg, it won’t break and right above it, if I drop the egg, it will break.

If pivotal floor is 3, then egg breaks at floor > 3.

We have limited amount of eggs. And we have to minimize the amount of eggs we use.

Tell me the least drops to ensure the pivotal floor is found.

When we find the pivotal floor, we have to be guaranteed about the correctness of our answer.

Find the worst amount of eggs that I will have to drop to find a floor because that guarantees that we will be able to find the pivotal floor.

Let’s find the base cases first.

Suppose we only have 1 egg.

With 1 egg, start from floor 0.

If it breaks at some floor, then previous floor will be pivotal floor.

drop (totalEggs, totalFloors)

In worst case, 1 egg will break at the topMost floor. Suppose we have n floors.

drop(1, n) = n

This is the worst amount of drops I have to do in order to find the pivotal floor.

Now the 2nd base case relies on the amount of floors we have.

drop(n, 1) = 1 (worst amount of drops that I have to do in order to find the pivotal floor).

drop(n, 0) = 0

Suppose we have 3 eggs and 6 floors.

We act as if we drop the eggs from floor 1, then floor 2 and so on.

When we drop an egg, it either breaks or it does not break.

We want to do worst case simulation.

Repeating subproblems

If the egg did not break then the number of eggs stays the same.

If the egg breaks, then number of eggs reduces by 1.

If the egg does not break at the current floor, then number of floors which are left to be seen are

Total floors - currentFloor

If the egg does not break at the current floor, we go one floor upwards.

If the egg breaks at the current floor, we go one floor down.

Addition of 1 signifies that we are going to be dropping from this subproblem.

**Famous DP algorithms**:

Unix diff for comparing 2 files.

Bellman-Ford for shortest routing in networks

TeX the ancestor of LaTeX

WASP – Winning and Score Predictor

DP problems:

* Optimization problems
* Combinatorial problems

The optimization problems expect us to select a feasible solution, so that the value of the required function is minimized or maximized.

Combinatorial problems expect us to find out the number of ways to do something, or the probability of some event happening.

Try dropping an egg from every floor (from 1 to k) and recursively calculate the **minimum number of droppings needed in the** **worst case**.

The floor which gives the minimum value in the worst case is going to be part of the solution.

drop(n, k): This answer represents the minimum number of egg drops when there are n eggs and k floors remaining.

**Approach 2**:

Another approach is to divide the tower in blocks of size B.

If the egg breaks at top of block k\*B, then the pivotal tower is in [0, k\*B]

If it does not then the pivot block is in [k\*B, 2N]

In a block, we can use brute force approach.

**Range Sum Querying**:

Given a 1d array.

Sum of items in range [i, j]

If we use O(n) then we are wasting our time in dealing with items we have already seen.

When we are optimising algorithms, we think of Bottlenecks, Unnecessary Work and Duplicate Work.

If we save the result, then we can get the range sum in O(1).

**Maximum Sum Rectangle in a 2D Matrix**:

Brute Force: Explore every possible 2d rectangle.

Every rectangle has a Top left corner and bottom right corner.

Explore all bottom right corners for every top left corner.

Every cell in the matrix will have a chance to be a top left corner.

Number of top left corners = rows\*cols

O(rows\*cols)

Omega(rows^2 \* cols^2)

Kadane’s Algorithm

In the given 2d matrix, What is the rectangle with maximum sum ?

In brute force, for every row, right >= left

We take a left pointer and increment the right pointer.

Increment left pointer and then explore all the rights.

Our task is to try every single possibility of Left and Right boundary of a possible maximum rectangle.

To initialize the algorithm, left and right start at 0,0

We will store the running table of the row sums, that is keep them in an array.

Kadane’s algorithm: O(rows)

Linear with respect to rows.

We will keep the sum of left to right **for each row**.

We will find the maximum sum subarray in the temporary array which will help us to find the maximum sum rectangle.

The 2d rectangle must have atleast 1 positive number for this algorithm to work.

L

R

currentSum

maxSum

maxLeft

maxRight

maxUp

maxDown

maxLeft is the current L, maxRight is the current R.

maxUp is the starting index of the maximum continuous subarray in the temporary array.

maxDown is the ending index of the maximum continuous subarray in the temporary array.

Row wise: we are finding normal sums and Column wise, we are finding the maximum continuous subarray sum.

**Max Contiguous SubArray Sum**:

Kadane’s Algorithm

When a problem asks us about maximum of an array or a group of items, then we can think of heap if we want to think of quantity or larginess of value but we can also think of Dynamic Programming to find a global solution based on smaller subproblems.

O(n^3) approach:

We can have Left and Right pointers.

Take left pointer and iterate right pointer from left to the end.

Storing the sum reduces the time to O(n^2).

Avoid one for loop.

Now, think in terms of subproblems and DP.

At each point of our iteration, **what does each item contribute**?

What choices do we have at each element?

Choice 1 is we can start a new subarray at a certain item which means that subarray at that item ends at itself.

Second choice is we can continue the maximum subarray coming before us with the item we are sitting at.

If we had to find maximum subsequence then we had the iterate over all the previous ones.

We store the solution of smaller subproblems in a cache.

As we have to find the maximum of different subarrays, building solution along with the algorithm may be a difficult task.

So we can build it separately in O(n)

We can build solution of Kadane’s algorithm (optimal left and right pointers) in a much simpler way.

**NOTE**:

Kadane’s Algorithm requires atleast one positive number.

When all elements are negative, the maximum subarray is the empty subarray, which has sum 0.

**Recursive Staircase**:

We have to climb some steps and we can take 1 step or 2 steps.

Find the total number of ways in which we can climb n steps.

Repeating Subproblems

Top Down

Bottom Up

**N matched Parentheses problem**:

Generate all strings with n matched parentheses

Backtracking

This problem is key in understanding how recursion can express our decisions and how constraints can change how a recursion happens through the recursion tree.

Input is a number n.

It is the number of open brackets that we need to generate all of our parentheses.

N open and closed parentheses.

In this, backtracking will be helpful.

**Brute Force**: Exponential

We will make decisions based on 3 key things.

* Our choice
* Our constraints
* Our goal

These are the 3 keys to backtracking.

**Variables**: Number of left brackets and Number of right brackets.

These narrow it down to the choice we make at each stack frame of our recursion.

Every stack frame has a state and that state dictates how we act.

Our choice at every stage is, do I open a bracket or do I close a bracket?

We cannot have more open brackets than the number of brackets we are given.

Number of right brackets >= Number of left brackets

I cannot close a bracket that has not been opened. We cannot have ), right bracket as the first bracket.

Our goal is to place n\*2 characters.

N=3

State of topmost stack frame: 3, 3

(can place 3 open and 3 closed brackets).

At every stage, we think whether we can place a left or right bracket or not.

As we are traversing the tree, we are starting to get the output that we desire.

Code will do a depth first search.

Base cases will be our answers.

Every node in the tree denotes a state. And a state decides how we can continue in our recursion.

**Variation**:

Given a string with brackets.

Check whether it is balanced or not.

We cannot use DP here since there are no repeating subproblems.

left = 0, right = 2 might represent different subproblems.

**Count Total Unique Binary Search Trees**:

The nth Catalan Number

Given set of numbers from [ 1, 2, …, n ]

How many unique binary search trees can we construct from a set of numbers from 1 to n.

How many structurally **unique** binary search trees can we create ?

First step is to choose the root node.

With a particular number as the root, we explore all possibilities of generating a tree.

n=3

1, 2, 3

By placing 1 at the root, we get 2 trees.

2+1+2 = 5

3 cases: Placing every element at the root and finding number of distinct binary search trees.

**G(n)**: The nth Catalan number

Given [1, 2, 3, …. , n]

How many structurally unique BSTs?

**F(i, n)**: With i at the root and #’s [1, 2, 3, …., n]

How many structurally unique BST’s ?

G(3) = F(1, 3) + F(2, 3) + F(3, 3)

Valid only when we have numbers [1, 2, 3,…n]

in sorted sequence

**Permute a String:**

Fundamental backtracking problem. Given a string, print all permutations of that string.

**Backtracking:**

Print all

Generate all

Compute all

n=3

Input Output

‘abc’ ‘abc’

‘acb’

‘bac’

‘bca’

‘cab’

‘cba’

Both array and string are collection of stuff in sequence. So we can interchange the way we work with them.

**3 Keys to Recursive problems in Backtracking**:

Our Choice

The character we place and recurse on.

Our Constraints

None really but at each point we have less characters to work with

Our Goal

n placements. n choices. A permuted string.

We choose a character and place it at some position.

Our choice is going to reduce the decision space.

Each one of the empty slots is a recursive decision.

A decision will be expressed for them by a frame in the call stack that the recursion decides on.

Every time we make a decision, we reduce the space of characters that we have left to place in the string.

Our goal is to make n decisions. Recursion tree will go at max n calls deep.

When we have made our first permutation, say

“boat”

Then the algorithm backtracks (by function returns)

Then we have

“boa” and t in the decision space.

But we have already used ‘t’ as a decision, so the algorithm backtracks further.

“bo” ,Now both ‘a’ and ‘t’ are back in the decision space.

Now we can choose ‘t’

“bota”

For the first call, we have to choose one of the characters and then recurse on characters that are left to choose.

Blue clouds: these are the possibilities we can choose from each position. Each position’s goal is to exhaust the possibilities that are available.

Backtrack when all the possibility space has been explored for a recursion branch.

If we are just printing the permutations, then we are not storing auxiliary space.

But the call stack is taking O(n) space because we are going to make n decisions.

We have a recursion tree with maximum depth of n.

When we talk about space complexity in relation to the call stack, we care about the maximum amount of stack frames that get on the call stack at one time.

When we backtrack, we are popping frames off.

We can print the permutations in lexicographic order.

**CODE THIS**: See Tushar Roy

**Another Important Case**: There could be duplicate characters.

We can create a **count array** which stores the count of each of the characters.

Count array is of size equal to number of distinct characters.

We have result array of size equal to size of the string.

Go from left to right and see the count of character. A character is available when its count is greater than 0.

**Compute the Next Permutation of a Numeric Sequence**:

If we have some input sequences,

Output

[1, 2, 3] [1, 3, 2]

[3, 2, 1] **[ ]**

[1, 5, 2] [2, 1, 5]

[3, 2, 1] is the last permutation for the numbers 1, 2 and 3.

For [1, 2, 3] the next permutation is [1, 3, 2]

2 approaches to build permutations.

1. Generate (n+1)th permutation
2. Case Analysis

We generate every single permutation until we hit the permutation we are given.

And then we go one step forward. That will give us the next permutation.

Problem with this approach is that we will be doing O(n!) work.

We should do an in depth case analysis.

The basic is about building permutations.

To decide the next permutation, we start from the rightmost element, backtrack one step and the next combination is the answer.

(We also see the any further possible placements, if possible. For example, if we have total 4 elements but an array of 3 elements)

We look for section that gets exhausted.

**NOTE**: A strictly decreasing section is on its last permutation.

We need to modulate what is right before the strictly decreasing section.

We can do this in linear time.

When the sequence is decreasing, it is maximized.

When increasing, it is minimized (first permutation).

**Total Ways to Decode A String**:

Input Output

“12” 2 ways

“1”, “2”, “12” 3 ways

1 = “A”

2 = “B”

3 = “C”

…

26 = “Z”

Can solve this both recursively and iteratively.

Can use a decoding pointer.

Can recursively decompose the string.

Recursion is a great way to express tons of decisions.

We have a call stack to remember our progress.

“2263”

At every point in the recursion, we make a decision.

We can decode 1 or 2 characters out and have to see if they are valid decodings or not.

Recursion takes depth first path.

**Brute Force**: 2^n

Recursion is expressing all of the ways that we can decode this string.

We have duplicate subproblems here.

For Dynamic programming, we prune the tree.

(Remove the duplicate subproblems).

Time becomes O(n).

Using Recursion, we get to know whether a particular decoding is valid or not only when we reach the base case.

Base case will be when the decoding pointer reaches the last character of the string.

When we reach the base case then we return 1.

We are just counting the number of decodings and not building the actual decodings.

Repeating subproblems.

Particular position of the decoding pointer is a subproblem.

**Edit Distance between 2 strings:**

By finding the edit distance between string that we are typing and the string in the dictionary, the computer can find what word we are trying to type.

It can sort according to edit distance and then give us suggestions.

Lower the edit distance, closer we are to that word.

We have to find the cost of converting 1 string to another string.

Levenshtein distance

The minimum edit distance between 2 strings is the minimum number of editing operations, insertion, deletion and substitution that are needed to transform one string into the other.

We can replace a single character, delete a single character and can insert a single character.

Suppose we have a string A = “horse” and another string B = “ros”

We need to transform string A into string B.

The minimum number of operations we need to do in order to transform A into B are 3.

Replace

horse 🡪 rorse (Replace “h” with “r”)

Remove

rorse 🡪 rose (Remove “r”)

Remove

rose 🡪 ros (Remove “e”)

“benyam”: Including both indices.

(-1, 0) “”

(0, 0) “b”

(0, 1) “be”

(0, 2) “ben”

(0, 3) “beny”

(0, 4) “benya”

(0, 5) “benyam”

There can 4 situations.

A = “benyam”

B = “ephrem”

Situation 1: Do Nothing

A[0, 5] 🡪 B[0, 5]

“benyam” “ephrem”

A[0, 4] 🡪 B[0, 4]

“benya” “ephre”

We check transformation distance for every cross section of sub strings between the 2 strings.

Suppose we need to transform subtring A[0, 5] to substring B[0, 5]

(Top Down)

We care about the last character in the sequence because that is what we want to check whether we are going to replace it, remove it or do any operation on it.

A[0, 5] and B[0, 5] have ‘m’ in the final position.

This means that the substring tranformation from A[0, 5] to B[0, 5] is going to have same edit distance as A[0, 4] transformed to B[0, 4]

Matching characters won’t contribute to the edit distance. We do not need to take action on them.

In next 3 cases, characters do not match.

**Case 2**: Replace

A[0,3] 🡪 B[0,2]

“beny” “eph”

y does not match h. We can either do a replace, insertion or deletion.

With replace, drop the ‘y’ and the ‘h’ and now all we need to do is convert A[0, 2] into B[0, 1]

“ben” “ep”

After we turn “ben” to “ep”, we add an ‘h’ to “ep”.

Now we have what we wanted in the beginning.

We transform everything before the mismatch and at the mismatch, we perform a replacement.

**Insertion**:

A[0, 3] 🡪 B[0, 2]

“beny” “eph”

A[0, 3] 🡪 B[0, 1]

“beny” “ep”

We transform the whole original A[0, 3] string into B[0, 1] and after this transformation we perform an insertion at the end of transformation result, “ep”.

**Deletion**:

A[0, 3] 🡪 B[0, 2]

“beny” “eph”

A[0, 2] 🡪 B[0, 2]

“ben” “eph”

We take the minimum of these 3 operations.

Build the globally optimal solution with help of subproblems.

We can solve this problem through bottom up DP.

**What does a cell represent in the DP table**?

Each of the cell is a subproblem.

“benyam” 🡪 “ephrem”

**Base subproblems**:

When we have to transform an empty string into ‘e’, ‘p’, ….

We just do insertions. These are single insertions

If we have to transform a substrings into empty string, then we do deletions.

Single character deletions.

When the 2 characters for a subproblem do not match, then we have to take minimum of replace, insert and delete.

When we have a matching character, we do nothing and the answer for that subproblem is substring without that matching character.

**IMP NOTE:**

Suppose we have to change string A to string B.

In Replacement, we think of replacement as replacing a character in string A with the required character in string B.

In Insertion, we insert a character into string B after carrying out a transformation.

abc 🡪 def

Suppose we need to insert ‘f’, then we carry out a transformation from “abc” 🡪 “de”

transformation + 1

In Deletion, we delete the last character from string A and then carry out the remaining transformation.

1 + transformation

**Partition to K equal subset sums**:

K buckets

Number of subsets = Total sum / k

We can use an element only once and must use all the elements of the array.

This looks like a problem in which we **try placements** and backtrack when certain placements do not work.

We might add an element to a bucket and then remove it.

**2 approaches**:

We can explicitly make an array of buckets, try placements in the bucket, backtrack if the placements do not work.

And when fill all the buckets and exhaust all the items, we are done.

Another approach is we simulate how we fill the buckets. We don’t actually create k buckets.

We do a recursion and work on a certain bucket and we are done with a certain bucket, we reduce k.

There is also a **DP based solution** for this.

**Backtracking approach**:

We fill the k buckets that we need to fill.

We will be simulating filling buckets here.

Number of subsets has to be an integer.

Key parameters to recursion where we will be simulating filling the buckets.

We start by choosing items in the array from index 0 in the array.

“iterationStart” tells us the point where we start choosing the items from in the array.

We want to know what items we have already used and we cannot use a single item more than once.

We must know what item we have used.

We keep track of the sum of the current bucket we are working on.

We also have to keep track of target sum we want a bucket to sum to.

Each time we do a recursion, we reduce k by 1.

If we have 1 bucket left to fill or we have filled k-1 buckets, then we are done.

When we finish a bucket, we need to reduce k and need to set the sum of the bucket we are working on to 0 because we are working on a new bucket.

We now continue to work on the next bucket.

We explore all the possible placements. If a placement fails and we reach a position that we cannot partition from, we backtrack and we try another item in the same stack frame and we unmark the item as false for that it was not used.

The iteration continues and we try another item.

Input for this problem is actually limited.

It scales very fast as the input gets larger.

We are simulating filling of the k-1 buckets by placing n elements. Nearly n choices.

n: items in the array.

We will not fill the last bucket

**Space**: O(n)

The worst case space complexity is controlled by the call stack in the recursion.

In the code there is recursion as well as iteration.

**Sudoku Solver**:

3 keys to backtracking:

* Our Choice
* Our Constraints
* Our Goal

Choice on the constraints and goal that we want to reach at the end.

9x9

We should have numbers 1-9 in every row, every column and every subgrid.

There are 9 subgrids here.

We have make choices and find a globally correct answer.

**Approach 1** (Brute Force):

* Generate all boards
* Validate all boards
* Return valid board.

Every possible board. This will generate exponential amount of boards.

**Our Choice**: Place 1-9 in an empty cell (Decision Space)

**Our Constraints**: Placement cant break board.

**Our Goal**: Fill the board.

Before placing, check whether the number will break the board or not.

Fill the board row wise. If a cell already has a number, we skip it.

**NP Complete**: Time is atleast exponential.

int to char

char(2+ ‘0’) 🡪 ‘2’

**Generate All Palindromic Decompositions of a String**:

Break a string into pieces where each piece would be a palindrome.

(All decompositions)

**Input**: “aab”

**Output**: [“aa”, “b”], [“a”, “a”, “b”]

**Generate All**: Backtracking

We explore a space of decisions or ways to decompose the string.

Make choices, backtrack on these choices and collect all possibilities when we reach our goal.

**Brute Force**: Check each decomposition if it is a palindrome or not. 2^(n-1)

(Wastes time)

**Backtracking**: “aab”

1. Choice: Snippet to recurse on.
2. Constraints: Snippet must be a palindrome
3. Goal: Decompose whole string. **Decomposition pointer** runs over and is equal to length of string.

Every snippet we recurse on must be a palindrome.

Array 🡨🡪 String

Taking snippet from [0, 0], [0, 1] and then [0, 2]

Every snippet we recurse on must be a palindrome.

**IP Address Decomposition**:

Given a string with only digits, restore it by returning all possible valid IP combinations.

Input “25525511135”

Output: “255.255.11.135”, “255.255.111.35”

**Compute All**, **Generate All**: Backtracking

Read the problem statement properly and then decide which tool to use.

**Approaches**:

* Nested for loops
* Backtracking

**3 keys to backtracking**:

* Our Choice:

We take snippets 1-3 digits long

* Our Constraints:

Value between 1 and 255 (snippet)

No Leading 0s

* Goal:

4 valid subsections. Build pointer is at the end.

Take various snippets/snapshots of the string.

Backtracking is replacement to nested for loops.

Key is taking snippets within our constraints.

We are finished when we have 4 valid subsections and we have reached the end of the string.

If the string is “25525511135”

Our choice will be “2”, “25”, “255”

Taking the solution to the code.

We will be working on segments 0, 1, 2 and 3.

Our Goal defines our base case.

Our base case catches our answers or it cuts off a dead path.

Base Case 🡨🡪 Our Goal

* Catches Answers
* Kill “overshoots”

With recursion: Think of base cases.

In Backtracking,

Choose 🡪 Explore 🡪 Unchoose } **Choice**

**0/1 Knapsack Problem**:

Write a program that selects a **subset of items** that has maximum value and satisfies a given weight constraint.

0/1 means no splitting.

Brute Force: Consider all subsets of items within weight constraint. Take subset with largest value.

Complete Search is always there in our tool kit.

Give and take

If we choose an item, we get its value but we lose the amount of items we can choose.

We get closer to that maximum weight.

Number of items and total weight are variables.

We can pick the items in any order but we do have to follow some order in order to easily write code.

(Final subset answer won’t change)

Do we choose the item in the row we are sitting in or do we not choose it.

We have 2 choices to make: Either include the item or don’t include it.

Many problems can be solved through Inclusion-Exclusion.

Recurrence Relation

Recursion is a beautiful way to express decisions as we go downwards and go towards our answer.

n: Total items, m: Max weight

Time: O(mn), Space: O(mn)

**Compute All Mnemonics of a Phone Number**:

Backtracking problem

We have a dialer pad. If we hit numbers 2 and 3,

“23” returns all of the possible combination or possible letter arrangements that can originate from these 2 numbers.

When I am at digit 2, I have 3 choices.

When we choose, we branch off.

Choose, Explore, UnChoose

No constraints are there.

“234”

Each of the recursion levels represent a choice.

After an item is placed, we are exploring the possibilities for the next digit.

We have 3 to 4 letters to choose from every digit.

To copy string: O(n)

Our goal is to place n letters.

Number of iterations on a recursive level will depend on the number of letters a digit has.

We can use array of strings as a hash table.

**IMP Thing To NOTE**:

* Using Backtracking, we can generate a list of subsets with each subset having 1 or more elements.
* Generate a complete list of individual elements.

In Backtracking, when there is a constraint, then we only add the current snippet and recurse only when this constraint satisifes else we backtrack.

We can write an if for which works on satisfying the constraint and also can write an if for which processes when the constraint does not satisfies.

**N Queens Placement Problem**:

Return all non-distinct placement of n queens on a NxN board, N is an input.

Backtracking is a method to solve problems by making a series of choices that we can return or backtrack to.

Where does this memory or remembrance comes from ?

Call stack remembers our choices and knows what to choose next.

Decision points 🡪 Backtracking

**3 Keys**:

* **Our Choice**

What choice do we make at each call of the function?

Recursion expresses decision

* **Our Constraints**

When do we stop following a certain path?

Why do we not even go one way?

* **Our Goal**

What is our target?

**What are we trying to find**? Is it something 10 characters long, etc.

Recursion represents decisions and these decisions are remembered on the call stack.

Base Case comes from the Goal.

Trust the recursion to do its work.

When filling the queens row wise, For optimisations we do not place 2 queens in the same row and can avoid checking this in the code.

In backtracking problems, we generate all components.

In Dynamic programming problems, we find some minimum or maximum value or total ways of doing something.

**IMP NOTE IN DP**:

With reference to min edit distance,

When using Bottom up approach, we include the base case subproblems in the 2d array

In top down, we return a value on these base cases and do not include them in the 2d array.

We first check for base cases before checking whether we know the answer to this subproblem or not.

**Permute a String with Duplicate Characters**:

The technique which we will see handles the output in lexicographically sorted order.

“AABC”

ABC

211

count[] array

Go from left to right and look for the first available character.

A character is available when it’s count > 0

result[] array

We put a character into its corresponding position in the result[] array by looking at the depth of the recursion.

If we are at the third level of the recursion, then we are gonna put that character at the 3rd position in the result[] array.

And we go into the recursion again after decreasing the count of this character because we have used this character.

When we come back from the recursion, we restore the count back to its original value and then look for the next available character on the right side of this character and keep repeating this process. (Backtrack)

Recursion: DFS

We backtrack when count of each distinct character becomes 0.

Question: Given 2 strings, write a function to decide if one string is a permutation of the other string.