Try to solve simple problems using Recursion.

If the recusive call occurs at the end of a method, it is called a tail recursion.

The tail recursion is similar to a loop.

The method executes all the statements before jumping into the next recursive call.

If the recursive call occurs at the beginning of a method, it is called a head recursion.

The method saves the state before jumping into the next recursive call.

Recursion, Iteration

**Stack with Recursion**:

We have to track during recursion who called the given method and what arguments are to be handed over.

And we have to track the pending calls.

We just need a single stack data structure: the operating system does everything for us.

These important information are to be pushed to the stack.

Values are popped from the stack.

**Question**: Implement recursion using Stack data structure without making use of OS’s stack.

Recursion is atleast twice slower than iteration because first we unfold recursive calls (pushing them on a stack) until we reach the base case and then we traverse the stack and retrieve all recursive calls.

**Searching**:

Linear search: O(N)

Binary search: O(logN)

Interpolation search: O( log logN )

We can use it if the array is sorted. It is like we humans look for a name in a telephone book.

We make a guess where in the remaining search space the sought item might be.

In binary search, we jump to the middle index. Here this is not the case. We just make a **simple linear interpolation**.

O(logN) is faster than O(log logN) for sufficiently large input.

But Before this threshold O(loglogN) is faster.

**Tower of Hanoi**:

It consists of 3 rods and number of disks of different sizes which can slide onto any rod.

The puzzle starts with the disks in a neat stack in ascending order of size on one rod, the smallest at the top, thus making a conical shape.

The minimum number of moves required to solve a Tower of Hanoi problem is 2^n – 1.

// O(2^n) exponential time complexity.

We have some rules:

Only one disk can be moved at a time.

Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack 🡪 a disk can only be moved if it is the uppermost disk on a stack.

No disk may be placed on top of a smaller disk.

**Important step**: Put all the plates that are on the largest plate onto the auxiliary rod.

Then we are able to move the largest rod to its final position.

There will always be a situation like this: we have managed to shift **n-1** plates to the auxiliary rod.

Now we just have to put the largest to the last rod.

And now we have to put the plates from the auxiliary rod to the top of the biggest plate.

BUT it is same problem again, so we can use recursion.

The largest plate is at its final position.

Here, the starting rod is now the auxiliary rod(2) and auxiliary rod is now 1.

We are able to solve this problem using the auxiliary rod.

Recursion is like **some pattern**.

**Selection Algorithms**:

Selection algorithm is an algorithm for finding the kth-smallest/largest number in a list/array. Such a number is called the kth order statistic.

Maximum (first largest), minimum(first smallest) or median.

The aim is to achieve **O(N) linear time** complexity for this particular operation. ~not that easy.

Methods: quickselect, median or medians method.

First intuition is to use sorting.

Let’s sort the array in which we want to find the given item.

After sorting 🡪 We can access the item with the help of the index.

For example: If we sort an array in descending order, the array[0] yields the maximum item.

**Inefficient approach**: If we just want to find a single item (maximum, minimum or median)

**Efficient approach**: If we want to find several items at the same item.

Why? **O(N\*logN)** versus **O(N)**

Intuition: Selection can be reduced to sorting and vice versa.

If we need to find kth largest/smallest item, then we should go for quickselect algorithm.

If we use sorting then we cannot do better than O(N\*logN).

Sorting gives all the kth largest/smallest items.

k belongs to [0, n-1]

quickselect has **O(N)** running time which is better than linearithmic running time O(N\*logN).

Another intuition is to use a data structure.

Sublinear time can be reached: **O(logN)**

For example: construct a balanced binary search tree or a heap.

Problem: it has some memory complexity, we have to construct the tree structure first.

So it is not the best solution.

We can construct a O(N) solution without the need for extra memory.

It will be an in place algorithm.

We can make it fast by using memory or we don’t need to use any memory but it would be a bit slower.

**Online Selection**:

We want to find a given item (maximum, minimum or median) of a stream.

We keep downloading data and we want to find items at runtime.

**Problem**: We do not know all the values in advance.

We will not be able to construct an algorithm that finds the best solution: we can have a good guess, a value that probably the one we are looking for.

“**Secretary problem**”

**Quickselect**: Hoare algorithm

It is a selection algorithm to find the kth-smallest/largest item in an unordered array.

Hoare constructed the algorithm 🡪 “Hoare algorithm”

It has a very good average case running time 🡪 **O(N)**.

Worst case scenario: O(N^2)

**In-place algorithm**

Concept is similar to quicksort.

Choose a pivot element at random.

Partition the array.

Instead of recursing into both sides as we would do for quick sort, we just take one side.

O(N\*logN) 🡪 O(N)

1. Partition
2. Select

The partition method is just for partitioning the array according to the pivot.

* Choose a pivot value at random. We generate a random number in the range [firstIndex, lastIndex] or we can choose the middle item to be the pivot.
* Rearranges the list in a way that all elements less than the pivot are on left side of the pivor and others are on the right. It then returns index of the pivot element.

**Important**: We just need one half of the array.

**Left side**: If we want to find the small items. For example: third smallest value, etc.

**Right subarray**: We want the large items. For example: second largest value, etc.

Partition places the pivot at its correct index.

Best case: O(N)

Worst case: O(N^2)

Average: O(N)

**Worst**: when we want to find the maximum in a sorted array and we always choose the first element to be the pivot.

There are more advanced selection algorithms.

When the quickselect algorithm is done, we can find the kth largest item at index k-1.

After randomly finding pivot element using Random(), we swap lastIndex and pivot so that we can iterate easily.

If we want bigger items on the left of the pivot, then search for elements bigger than the pivot and when we find one such element at index i, then swap elements at index i and firstIndex.

**Advanced Selection**: Median of Medians, Introselect

Sometimes generating random pivot indices is not going to help. It can lead to quadratic running time.

Boosting up the selection algorithm has to do with choosing the pivot.

If we want to make sure that running time will be linear, then we should make sure that we discard half of the array on every iteration.

We have to pick a good pivot. If we pick the median as a pivot then there will be approximately same amount of items on the left and right subarrays.

Approximated median: good enough to make sure we discard more items.

**Median of medians select**:

It is basically the same as quickselect, the only difference is how we get the pivot value.

* quickselect: We generate a random index.
* Median of medians: We calculate the approximated median.

O(N) running time guaranteed.

O(logN) worst case memory complexity.

In-place (does not use memory)

If we want to make it in place then it could happen that it may not be the fastest algorithm possible.

There is some Trade off between running time and memory complexity.

**Introselect**: it is like introsort.

It is a hybrid algorithm. It combines two algorithms in order to take advantage of the best feature.

For introsort, we combine quick sort and heap sort.

For introselect, we combine quick select and median of medians select.

Quick select is in place. Median of medians select is always fast: O(N)

We take advantage of their best features.

Introselect starts with quickselect in order to obtain good average performance, and then falls back to median of medians if progress is too slow.

**Secretary problem**:

Online algorithm related problem.

We want to find kth smallest/largest item of a stream.

We keep downloading data from web.

We do not know data in advance. So partition algorithm cannot be used.

We have to make some optimal decisions in advance.

The problem is to select (under these constraints) a specific element of the input sequence of data with largest probability.

Secretary problem is very important problem of optimal stopping theory.

Also known as “best choice problem”

**Problem**:

We have to hire the best secretary out of N applicants. Applicants are interviewed one by one + after rejecting, the applicant cannot be recalled.

We can rank the applicant among all applicants interviewed so far, but we are unaware of the quality of yet unseen applicants.

What is the optimal strategy?

We want to maximize the probability of selecting the best applicant.

NP-Hard problem.

When to sell car, when to sell stock, etc.

If we can make the decision at the end: we just have to make a maximum finding.

It can be done in O(N).

But we have to make the decision immediately.

It is not so realistic.

In real life, we interview all the applicants and make a decision about the best applicant.

Here, suppose we have 10 applicants and after interviewing the 4th one we make the decision to hire the 4th applicant or the 6th applicant.

And we are unaware that the best candidate would be in the upcoming applicants.

**Solution**:

Always reject the first n/e applicants and then we have to stop at the one who is better than all the previous ones.

e: natural logarithm ~ 2.718

* It is very popular problem because it has a well defined solution.
* The probability of choosing the best appicant is 1/e.
* So 37% chance to find the optimal one.

This can be proved using Series Analysis, Integration, etc.

It is a very popular problem because it has a well defined solution.

**What if we don’t find a candidate better than the first n/e applicants** ?

**Backtracking**:

It is a form of Recursion.

General algorithm for finding all solutions to some computational problems 🡪 “constraint satisfaction problems”

We incrementally build candidates to the solutions.

If partial candidate A cannot be completed to a valid solution: we abandon A as a solution.

For example: eight-queens problem or Sudoku.

Backtracking is often faster than brute force enumeration of all complete candidates, because it can eliminate a large number of candidates with a single test.

Backtracking is an important tool for solving constraint satisfaction problems 🡪 combinatorial optimization problems.

**The method**:

The partial candidates are represented as the nodes of a tree structure.

“potential search tree”

Each partial candidate is the parent of the candidates that differ from it by a single extension step.

The leaves of the tree are the partial candidates that cannot be extended any further.

The backtracking algorithm traverses this search tree recursively, from the root node in a top down manner (like DFS).

We can represent the whole structure with a tree like structure.

This is why backtracking is sometimes called depth first search.

1. For every node the algorithm checks whether the given node can be completed to a valid solution.
2. If it cannot 🡪 the whole subtree is skipped.
3. Recursively enumerates all subtree of that given node.

We have several options. After every choice, we have a new set of options.

If we make good choices 🡪 we end up with a **GOOD** state.

If not: we have to **backtrack**.

States are the leaf nodes.

**N-queens problem**:

The problem of placing N chess queens on an NxN chessboard so that no two queens threaten each other (they will not be able to attack each other).

We have to consider: Queens can move diagonal directions too.

The original problem was designed for 8 queens. The general form is about N queens.

Gauss worked on this problem.

Dijkstra used this problem to illustrate the power of what he called **structured programming**.

We are able to solve this for >= 4 queens.

**Empty board**: Root node of the tree.

We place a queen in every column.

No feasible solution: We have to step back to the previous column and increment the position of the queen there ~ **BACKTRACK**.

A single node can have a huge subtree.

When pruning 🡪 We get rid of the whole subtree.

(Bad state)

In game theory, it is called alpha beta pruning.

We eliminate huge branches as we come to the conclusion that there is not going to be a feasible solution for sure.