Backtracking

Where does Backtracking come from?

Before we dive into backtracking, let's take a step back and talk about **recursion** — because backtracking is really just a specialized form of recursion.

So what is recursion?

At its core, recursion is just a function **calling itself** to solve a smaller version of the same problem. We keep breaking the problem down until we hit a base case — the smallest, simplest version — and then we return.

A classic example is the Fibonacci sequence:

```
def fib(n):
    if n <= 1:
        return n
    return fib(n - 1) + fib(n - 2)
```

This is **single-path recursion** — we go down one path (or two in this case), all the way to the base case, and then return.

What is Backtracking?

Backtracking is a strategy for **exploring multiple possible paths**, built on top of **recursion** — but with a twist: it adds a mechanism for **trying**, **failing**, and **undoing** choices.

So at its core, you can think of backtracking as:

> *Backtracking = recursive calls + making choices + undoing those choices*

It's not a separate algorithm from recursion — it's really just **a way of using recursion** when the problem involves: "Trying out all possible paths, and finding all the correct or optimal ones."

At every level of recursion in backtracking:

- You make a choice;
- You continue down that path recursively;
- If it works, great — you keep going;
- If it doesn't, you **backtrack** — undo that choice, and try something else.

```
         []
       /  |  \
     1    2   3
    /     |    \
  2 3    1 3   1 2
...
```

Each level in this tree represents a decision point.
Each branch is a possible choice.
You go down one path, and if you reach a valid solution — you record it.
If you hit a dead end, you **back up** and try the next branch.

This is the core idea of backtracking — it's a **systematic way of exploring all options** using recursion and a controlled way to undo steps.

Pruning Strategies

Backtracking, by nature, is about brute-force searching all possible solutions.
But let’s be honest — not every path is worth exploring.

In fact, many branches of the recursion tree can be ruled out early because we know they won’t lead to a valid result. So if we can skip those branches ahead of time, we can save a lot of unnecessary computation.

That’s what we call **pruning** — cutting off bad paths before wasting time on them.

Common strategies

Sorting-based pruning

> *used to eliminate duplicates*

If the input has duplicate numbers and the problem says **“no duplicate results allowed”**,
we can **sort the array first**, and then **skip repeated elements** in the same recursion level. This is especially useful in problems like combinations or subsets where order doesn't matter.

Example: If you’ve already used the number `2` at this level, skip any more `2`s that come right after.

Constraint-based pruning

> *(early stopping based on problem limits)*

Sometimes, the problem gives constraints like:

- the current sum must not exceed a target,
- the combination must have exactly k elements,
- or the placement must not break certain rules (like in N-Queens or Sudoku).

In these cases, if a recursive path already violates a rule, you can immediately return — no need to explore further.

Prediction-based pruning

> *math-based or sorted-based shortcut*

Sometimes you can look ahead and predict that a path won’t work out —
even if it hasn't broken the rules yet.

For example, if you're trying to reach a target sum and your numbers are sorted,
then as soon as one number is greater than the remaining target, you can safely `break` out of the loop — because all the numbers after it will be even larger and won’t help.

Practical tips

1. **Start simple** — always write the basic working backtracking solution first, even if it's not optimized.
2. **Study the recursion tree** — look for repetitive branches or clearly invalid paths.

3. **Add pruning step-by-step** — insert `if` checks inside your `for` loop to skip or `return` when necessary.

This gradual process helps you understand both the algorithm and how to optimize it without losing correctness.

Types of Problems

Backtracking is a perfect fit for problems where you need to **explore multiple possible paths** — usually when each step involves making a **choice**.

Here are some classic types of problems where backtracking really shines:

- **Subset problems** – Given a list of numbers, find all possible subsets (combinations of any size).
- **Permutation problems** – Find all the possible ways to reorder the numbers.
- **Combination problems** – Pick `k` numbers from `n`, under certain rules.
- **String segmentation problems** – Cut a string in all valid ways (like palindrome partitioning).
- **Board problems** – Solve constraint-based puzzles like N-Queens or Sudoku.

> *These problems all have something in common:*
>
> - *At each step, there are **multiple choices** you can make.*
> - *You often need to **try out all possible paths**.*
> - *Each path might be a full or partial **valid solution**.*
> - *You can naturally model the process as a **tree of states** — where each node is a choice, and each branch is a path you're exploring.*

So anytime you're faced with “try all options and find all valid answers”, and especially when the order or constraints matter — that’s your cue: this is probably a backtracking problem.

Template

Here’s the core structure of a backtracking function in Python:

```
def backtrack(parameters):
    # Base case: when to stop recursion — depends on the problem
    if meet_end_condition:
        save path as a valid result
        return

    for option in options:
        make a choice
        backtrack(path + [option], updated_options)
        undo the choice (if necessary)
```

Function arguments and return value The return value of the function in the backtracking algorithm is generally None. The other parameters are not defined until we write the logic.

Base case / end condition This is what stops the recursion.

The loop This is where you **try all possible next steps** from the current state.

78. Subsets

The first problem we are going to talk about is the **subset problem**, which is actually a very good entry point for us to get started with backtracking algorithms. Because in this problem, you can **start from the simplest recursive thinking and deduce the backtracking structure step by step**, and the whole process is very natural and clear.

Let's look at the question first: given an integer array `nums` without repeated elements, return all its possible subsets, including the empty set and itself.

This is actually asking us: In a bunch of elements, how many combinations do we have to choose or not? You can choose each number or not.

recursion

```
# initialization

# define recursion function
def recursion(index, subset):

  # do not choose nums[index]
  recursion(index+1, subset)

  # choose nums[index]
  subset.append(nums(index))
  # record one subset
  self.append(subset)
  recursion(index + 1, subset)
  subset.pop()

# use the recurtsion
backtracking(0, [])

# get the result
```

```python
        return self.result
```

Let's first think about this problem recursively.

We start from the first number and look forward. For each number, we can either **choose it** or **skip it**, right? So let's just handle both cases at every position:

- **Don't take** the current number: skip it and recursively process the rest.
- **Take** the current number: add it to the `subset`, and continue recursion on the remaining elements.

So we define a recursive function `backtracking(index, subset)`,
which means: **we're currently looking at `nums[index]`, and we've already picked a prefix called `subset`** — now we decide whether to take this number.

Starting from index 0, we branch the recursion into two cases:

- Skip the current number and recurse.
- Take the current number, add it to `subset`, record the path, and recurse further.

When we reach the end of the list, we stop the recursion.

> *Here, we're recording the current path immediately after making a "pick",*
> *which means we're using pre-order traversal to build the result.*
> *That's why we manually add an empty subset at the very beginning.*

```python
class Solution(object):
    def subsets(self, nums):
        """
        :type nums: List[int]
        :rtype: List[List[int]]
        """
        # input: array -> unique element
        # output: all subsets
        self.result = []
        self.result.append([])

        def backtracking(index, subset):
            if index == len(nums):
                return

            # 不选 nums[index]
            backtracking(index + 1, subset)

            # 选 nums[index]
            subset.append(nums[index])
            self.result.append(subset[:])
            backtracking(index+1, subset)
            subset.pop()

        backtracking(0, [])
        return self.result
```

`for`-loop in Backtracking Mode

You may have noticed that in our earlier version, we made **two recursive calls** for each position: one for “not choosing” the number, and one for “choosing” it.

But what if the problem changes — says, instead of making a yes/no decision at each position, you’re allowed to choose any number from a set of unused options at each step?

This is no longer a binary decision — it becomes a matter of selecting from multiple available choices. This type of situation comes up in **permutation problems, combination problems**, or even more complex backtracking problems.

In those cases, writing two separate DFS calls won't cut it. So what do we do?

That's when we turn tothe classic for-loop version of the backtracking framework.

Let's rewrite our previous logic using this idea. Instead of manually branching into "choose" and "don't choose", we now use a `for` loop to try each number starting from position `start`:

```python
class Solution(object):
    def subsets(self, nums):
        """
        :type nums: List[int]
        :rtype: List[List[int]]
        """
        self.result = []
        self.n = len(nums)

        def backtrack(start, path):
            self.result.append(path[:])

            for i in range(start, self.n):
                path.append(nums[i])
                backtrack(i + 1, path)
                path.pop()

        backtrack(0, [])
        return self.result
```

Now, what we're doing is: At each level of recursion, we loop through all choices starting from `start`, choose a number, go deeper via recursion, and then backtrack by popping the number off. We record every valid path, including the empty subset, because we call `result.append(path[:])` at the start of every recursive call.

46. Permutations

This problem is quite similar to the **subsets problem** — we're still asked to **generate all possible results**. But this time, instead of subsets, the task is to return **all permutations** of a list of distinct numbers.

That means the requirements have changed:

- Every number must be used exactly once
- No duplicates allowed
- And — most importantly — **order matters**. Different orders mean different permutations.

So when building a solution, we can't skip any number, we can't use the same number twice, and we have to treat different arrangements as different results.

Now, compared to the subsets problem, there are three key differences in the implementation:

- Termination condition: We stop the recursion when `subset` has the same length as `nums`.
- When to record results: Only when the path is complete (length == n), we record it.
- How to make choices: Each time, we try every number in `nums` — but only if it hasn't been used yet.

You can imagine we're building a permutation tree using recursion. We start with an empty list and, at each step, add one unused number to the current path.

- Each level of recursion represents one position in the final permutation.
- Each node is a partial solution — a prefix of the full permutation.
- Each complete path from root to leaf is a valid permutation.

Define a recursive function `backtracking(subset)` that represents: “Given this partial arrangement `subset`, try to build a full permutation from here.”

```python
class Solution(object):
    def permute(self, nums):
        """
        :type nums: List[int]
        :rtype: List[List[int]]
        """
        # input: array -> distict nums
        # output: all permutations
        self.result = []

        def backtracking(subset):
            if len(subset) == len(nums):
                self.result.append(subset[:])
                return

            for i in range(len(nums)):
                if nums[i] not in subset:
                    subset.append(nums[i])
                    backtracking(subset)
                    subset.pop()

        backtracking([])
        return self.result
```