Essential algorithms for the coding interview

Paradigmatic coding problems and templates for preparing the coding interview

Ricardo Pallás Román

Contents

ISBN: 9798371892430.

Publisher: Independently published.

Cover image: Freepik.com.

https://rpallas.xyz.

1 Introduction

In this book, we will provide a comprehensive guide to the essential algorithms that every software engineer should know for their coding interviews.

The goal of the book is to provide a step-by-step guide for paradigmatic coding problems, including analysis of time and space complexities. These problems have been strategically selected according to the data structure or algorithm needed to solve each of them.

Throughout the book, we will use clear and concise explanations, accompanied by examples and step-by-step executions, to help readers understand the concepts and techniques involved in each algorithm.

Whether you are preparing for your first coding interview or looking to brush up on your algorithms skills, this book is for you. By the end of it, you will have a solid understanding of the essential algorithms and data structures asked in coding interviews.

2 First Bad Version (Binary Search)

You are responsible for managing the development of a new product and have recently discovered that the latest version of the product has failed a quality check. This means that all versions developed after this version are also of bad quality.

You have access to an API, `isBadVersion(version)` that returns true if the version is bad, and you want to determine the first version that is of bad quality (and therefore, all subsequent versions are also of bad quality).

Example 1:

> **Input:** n = 6, bad = 5
>
> **Output:** 4
>
> **Explanation:**
>
> call isBadVersion(2) -> false
> call isBadVersion(6) -> true
> call isBadVersion(5) -> true
> Then 5 is the first bad version.

As the problem states, we have a range of numbers from `1` to n, that represents the product versions. We can visualize them as an array:

> n=10
>
> [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

This problem requires us to do a search in that array until we find the first bad version. A naive approach to this would be to perform a linear search [1]:

Let's iterate from the beginning until we find the first bad version:

```
function findFirstBadVersion(n, isBadVersion) {
    for (let i=1; i<=n; i++) {
        if (isBadVersion(i)) {
            return i;
        }
    }
};
```

This is how this algorithm works for a given n = 10 and first bad version = 4:

We have 10 versions, that can be visualized as an array:

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

We start checking the first version:

```
i=1

[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
 i

isBadVersion(1) // false
```

Since the first version is not bad, we continue with the second version:

```
i=1

[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
    i

isBadVersion(2) // false
```

We haven't found the bad version yet, so we repeat:

```
i=1

[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
       i

isBadVersion(3) // false
```

Let's try with 4 and see if we are lucky:

```
i=1

[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
          i

isBadVersion(4) // true
```

Since `isBadVersion(4)` returned true, we know for sure it is the first bad version, therefore we can return 4 and finish the execution.

The time complexity of this solution is $O(n)$, where n is the number of versions.

This is because we are performing a **linear search** from `1` to n. The number of steps the algorithm needs to take is directly proportional to the number of versions. In the worst case (first bad version = n), it would need n steps to find the solution.

The space complexity is constant, $O(1)$, as we are not storing any data.

2.1 Introducing binary search

Notice an important property about the array: it is a **sorted array**; thus, values are always kept in order.

Taking this into account, is linear search the only possible solution? Can we do it better?

The answer is: yes, we can do it better using **binary search**.

2.1.1 What is binary search?

You can think of binary search [2] as the guessing game: I am thinking of a number between `1` and `20`, and I will tell you whether you need to guess lower or higher. You won't start with `1`; instead, you will say `10` because it is in the middle of `1` and `20`. If I answer "lower," you will discard half of the numbers as you will know the answer is a number between `1` and `9`. Then you will probably pick `5`, and I will reply "higher," so another half of numbers will be discarded. In the end, following this strategy, you will find out which number I was thinking of much faster.

Let's visualize it: imagine I am thinking of 6 and I ask you to guess it, from `1` to `10`:

> 1 2 3 4 5 6 7 8 9 10
>
> You: **5**
> Me: Higher

At this point, you can discard numbers between `1` and `5` (half of the possible numbers).

> ~~1 2 3 4 5~~ 6 7 8 9 10
>
> You: **8**
> Me: Lower

Then you discard numbers from `8` to `10`.

> ~~1 2 3 4 5~~ 6 7 ~~8 9 10~~

You now guess: `6`. You have found the number in only 3 steps.

2.1.2 How does binary search relate to the First Bad Version problem?

Let's try to solve the First Bad Version problem using binary search.

Coming back to our previous example, where we have `10` versions and the first bad version is 4:

> [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

Step 1: We begin our search from the version in the middle (5) and we check if it is a bad version:

> [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>
> isBadVersion(5) // true

Since 5 is a bad version, we can conclude that all subsequent versions are also bad. That means we can exclude them:

> [1, 2, 3, 4, 5, ~~6, 7, 8, 9, 10~~]

So far, 5 is the first bad version we have found.

Step 2: Let's continue now with the middle value, which in this case is 3:

> [1, 2, 3, 4, 5, ~~6, 7, 8, 9, 10~~]
>
> isBadVersion(3) // false

We checked the middle value for this step. Now, we know 3 is not a bad version; thus, we can discard it and the versions to its left, as they are also valid.

> [~~1, 2, 3~~, 4, 5, ~~6, 7, 8, 9, 10~~]

The only remaining possible candidates are 4 and 5.

Step 3: we repeat the previous steps, and we pick the version in the middle, 4, and check:

> [~~1, 2, 3~~, 4, 5, ~~6, 7, 8, 9, 10~~]
>
> isBadVersion(4) // true

We checked that 4 is a bad version, and given that we already discarded all the versions to its left (because they were all good versions), we can conclude that 4 is the first bad version.

We found the first bad version in 3 steps. Linear search would have taken 4, which is similar, but we will see the benefits of binary search in the next section.

2.1.3 Binary search vs. linear search

There is no significant difference between binary search and linear search when we apply them to small ordered arrays. However, if we try with larger arrays, we will notice that binary search is much more efficient.

Imagine we have a product with 50 versions instead of 10. Let's focus on the worst case: the last version (50) is the first bad version.

How many steps would each algorithm need for this worst case?

- Linear search: 50 steps.
- Binary search: 6 steps.

It is very intuitive to see why 50 steps are needed for linear search. It starts from the beginning and then checks the next version. Since the first bad version is the last version, linear search will check all 50 versions.

But why does binary search only need 6 steps? Let's visualize it.

> [0, 1, 2, 3, 4, 5, …, 48, 49, 50]

Step 1: We select `25` and verify it is a good version, so we can discard all versions from `1` to `25`.

> [26, 27, 28, 29, 30, …, 48, 49, 50]

Step 2: We select the middle (`38`), which is also a good version. Let's discard versions older than `38`.

> [39, 40, 41, 42, 43, 44, …, 48, 49, 50]

Step 3: The middle is `44` (good version). We discard versions older than `44`.

> [45, 46, 47, 48, 49, 50]

Step 4: The middle is `47` (good version). We discard versions older than `47`.

> [48, 49, 50]

Step 5: The middle is `49`. We can then discard `48` and `49`.

> [50]

Step 6: The middle is `50`, and it is a bad version. Since we have discarded all previous versions, we know it is the first bad version.

2.1.4 Time complexity of binary search

We previously mentioned that the time complexity of linear search is $O(n)$ (where n is the number of versions) because the number of steps it has to take is directly proportional to n.

In the worst case, where the first bad version is the last version, for `n=10` it will need 10 steps, and for `n=50` it will need 50 steps.

Binary search, however, is different.

How can we relate n to the number of steps for binary search? We can use the logarithm function.

> Steps = $log_2 n$

As a refresher, a logarithm is the inverse of an exponent. For example, 2^4 = 2 * 2 * 2 * 2 = 16.

So, $log_2 16$ is the inverse. In other words, how many times do you have to multiply 2 by itself to get a result of 16? The answer is $log_2 16$, or simply 4.

> $log_2 16$ = 4

Another way to understand $log_2 16$ is: how many times do we need to halve 16 until we get 1?

> 16 / 2 / 2 / 2 / 2 = 1

Since we need four 2s, $log_2 16$ = 4.

Do you see the relation between logarithms and binary search?

In binary search, we halve the array at each step and repeat the process until the array contains only 1 element.

That's why the number of steps binary search needs is $log_2 n$.

In the following graph, we can see the difference between $O(n)$ and $O(logn)$. The larger n is, the greater the difference:

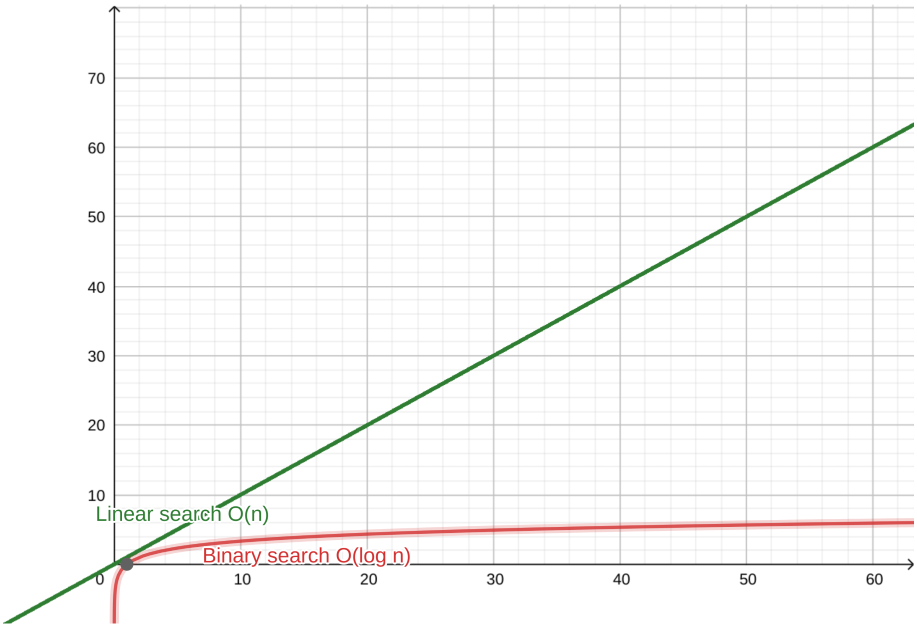

Figure 2.1: Linear vs logarithmic

For example, this table shows the number of steps needed for each algorithm for different values of n:

n	Linear search	Binary search
1	1	1
10	10	4
50	50	6
100	100	7
1000	1000	10

2.2 Binary search solution to First Bad Version problem

Now that we know how binary search works and that it is more efficient than linear search ($O(n)$ vs $O(logn)$), let's implement the solution:

```
function findFirstBadVersion(n, isBadVersion) {
    let left = 1;
    let right = n;

    while (left < right) {
        const mid = left + Math.floor((right-left) / 2);
        if (isBadVersion(mid)) {
            right = mid;
        } else {
            left = mid + 1;
        }
    }

    return left;
}
```

Using our previous example, let's visualize what the algorithm is doing step by step. Given n = 10 and the first bad version being 4:

We start by declaring two variables, `left` and `right`, and then assigning them to the first and last versions, respectively.

Then we start a `while` loop. Each execution represents one step of the binary search algorithm, where we check the middle element and discard the right/left elements depending on the result of `isBadVersion(mid)`.

We will repeat the steps until there is only 1 version left, or, in other words, `left == right`.

After exiting the while loop, **`left` will point to the first bad version.**

Now let's see the steps:

Step 1:

```
left = 1
```

```
right = 10
mid = 1 + floor(10-1/2) -> (5)

[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
 l          m              r

isBadVersion(5) // true

right = mid = 5
```

Step 2:

```
left = 1
right = 5
mid = 1 + floor(5-1/2) -> (3)

[1, 2, 3, 4, 5, 6, 7, 8, 9, 19]
 l     m     r

isBadVersion(3) // false

left = mid + 1 = 4
```

Step 3:

```
left = 4
right = 5
mid = 4 + floor(5-4/2) -> (4)

[1, 2, 3, 4   , 5, 6, 7, 8, 9, 10]
          l/m   r

isBadVersion(4) // true

right = mid = 4
```

Step 4:

Since both `left` and `right` point to the same element (4), the `left < right` condition is no longer `true`, and the loop is broken.

We return the number pointed to by `left`, which is 4, the first bad version.

2.3 Template for binary search problems

Now let's introduce a code template that can be used to solve many binary search problems:

```
function condition(element) {
    // returns true/false
}

function binarySearch(array) {
    let left = 0;
    let right = array.length - 1;

    while (left < right) {
        let middle = left + Math.floor((right-left)/2);
        if (condition(array[middle])) {
            right = middle;
        } else {
            left = middle + 1;
        }
    }

    return left;
}
```

This template is almost identical to the solution we provided for the First Bad Version problem. For each iteration, we halve the array, discarding elements to the right or left, depending on the condition.

Important: After exiting the loop, **left** points to the minimal value in the array that satisfies the `condition` function.

In the First Bad Version problem, the condition is `isBadVersion`, so in this case, `left` will point to the minimum version that is a bad version.

Another thing to consider about this template is how we calculate the middle:

```
let middle = left + Math.floor((right-left)/2)
```

There are other ways to calculate it like:

```
let middle = Math.floor((left + right) / 2)
```

The issue with that formula is that for large numbers, the sum of `left + right` can be greater than `Number.MAX_VALUE` and cause an overflow.

In conclusion, this template can be used for almost every binary search problem, but it might need some tweaks.

2.3.1 Template in action

Let's look at a quick problem where we can use this template:

Write a function that searches for a given integer, `target`, in a sorted array of integers, nums. If `target` is found in nums, the function should return its index. If `target` is not found in nums, the function should return `-1`.

Example:

> **Input:** nums = [-1,0,3,5,9,12], `target` = 9
> **Output:** 4
> **Explanation:** 9 exists in nums and its index is 4

Since the array is sorted, binary search is a great candidate to solve this problem. Let's see a possible solution using our template:

```
function search(nums, target) {
    let left = 0;
    let right = nums.length - 1;

    while (left < right) {
        let mid = left + Math.floor((right-left)/2);
        if (nums[mid] > target) {
            right = mid;
        } else {
            left = mid + 1;
        }
    }

    if (nums[left-1] === target) {
        return left-1;
    } else if (nums[left] === target) {
        return left;
    } else {
        return -1;
    }
}
```

At the end of the execution, our template guarantees the following:

> `left` is the minimum value satisfying condition (nums[mid] > target) This means `left` points to the minimum value that is greater than `target`. This is why we check `left-1` at the end of the algorithm. If `left` is the minimum value greater than `target`, the value to its left must be `target` (if it exists).

If `left-1` is equal to the target, we return `left-1` as it is the index that points to `target`. Otherwise, we return `-1` indicating that the target does not exist.

There is a special case where we check if `nums[left]` is equal to `target`. This check handles cases where `nums` contains only one number (`nums.length == 1`). In such cases, we don't execute any step, and `left` will point to the only existing element.

As you can see, we are halving the `nums` array until only one element is left. But should we continue iterating over `nums` if `nums[mid]` is equal to `target`?

We can tweak our template to avoid some extra steps:

```
function search(nums, target) {
    let left = 0;
    let right = nums.length - 1;

    while (left < right) {
        let mid = left + Math.floor((right-left)/2);

        if (nums[mid] === target) {
            return mid;
        } else if (nums[mid] > target) {
            right = mid;
        } else {
            left = mid + 1;
        }
    }

    return nums[left] === target ? left : -1;
}
```

We just added a new condition inside the loop:

`if (nums[mid] === target)`

If that happens to be `true`, we can simply return `mid` as it is already pointing to `target`. We don't need to continue execution if we have already found the solution.

After exiting the loop, we return `-1` as we haven't found any number equal to `target`. Notice that we check if `nums[left]` is equal to `target` before returning `-1`. This is done to handle the case where nums contains 1 element.

2.4 A Non-Trivial example

Before ending this chapter, let's take a look at another problem where binary search might not be obvious.

Given an array of package weights, `weights`, and a number of days, `days`, determine the minimum weight capacity of a ship that can transport all of the packages on a conveyor belt within the given number of days. The packages must be shipped in the order given by the array, and the ship must not exceed its weight capacity on any given day.

Example:

> **Input:** weights = [1,2,3,4,5,6,7,8,9,10], days = 5
>
> **Output:** 15
>
> **Explanation:** 15 is the minimum ship capacity to ship all the packages in 5 days like this:

> 1st day: 1, 2, 3, 4, 5
> 2nd day: 6, 7
> 3rd day: 8
> 4th day: 9
> 5th day: 10
>
> It is important to note that the packages must be shipped in the order specified and cannot be split into smaller groups. For example, using a ship with a capacity of 14 to transport the packages in the following groups is not allowed: (2, 3, 4, 5), (1, 6, 7), (8), (9), (10).

This is a problem that can be solved using binary search. It involves searching for the minimum capacity needed to ship all packages within `days` days.

We know the capacity we are looking for is within the following range:

> `[max(weights), sum(weights)]`

- **max(weights):** This is the minimum possible value; otherwise, we wouldn't be able to ship the heaviest package.
- **sum(weights):** This would be the maximum possible value, representing the capacity needed to ship all packages in 1 day.

Do you see the pattern? We just need to find the minimum capacity using binary search on that range.

The hard part is finding a good `condition` function. Once we have it, we can apply our template to solve the problem easily. The function will check if it is feasible to ship all packages in `days` days:

```
function isCapacityEnough(weights, days, capacity) {
    let currentDays = 1;
    let dayCapacity = capacity;

    for (let weight of weights) {
        dayCapacity = dayCapacity - weight;
        if (dayCapacity < 0) {
            dayCapacity = capacity - weight;
            currentDays = currentDays + 1;
            if (currentDays > days) {
                return false;
            }
        }
    }

    return true;
}
```

Now we can write a solution by just using our template:

```
function shipWithinDays(weights, days) {
    let left = max(weights);
    let right = sum(weights);

    while (left < right) {
        let mid = left + Math.floor((right-left)/2);
        if (isCapacityEnough(weights, days, mid)) {
            right = mid;
        } else {
            left = mid + 1;
        }
    }

    return left;
};

function isCapacityEnough(weights, days, capacity) {
    let currentDays = 1;
    let dayCapacity = capacity;

    for (let weight of weights) {
        dayCapacity = dayCapacity - weight;
        if (dayCapacity < 0) {
            dayCapacity = capacity - weight;
            currentDays = currentDays + 1;
            if (currentDays > days) {
                return false;
            }
        }
    }
    return true;
}

const max = (weights) => weights.reduce((a,b) => Math.max(a,b),
 ↪  -Infinity);
const sum = (weights) => weights.reduce((a,b) => a+b, 0);
```

2.5 Summary

- The template guarantees that, after exiting the while loop, `left` points to the minimum value that satisfies the condition.
- For each problem, the hardest part is to find a good condition function.
- You may need to tweak the template depending on the problem.
- This only works if the values are sorted.
- Time complexity is `O(log n)`, where n is the number of elements.

2.6 References

- [1] Linear search. https://www.geeksforgeeks.org/linear-search/.
- [2] Binary search. https://www.geeksforgeeks.org/binary-search/.

3 Valid Parentheses (Stack)

You are given a string `s` that only contains the characters `'('`, `')'`, `'{'`, `'}'`, `'['`, and `']'`. Write a function to determine if the input string is valid according to the following rules:

1. Open brackets must be closed by the same type of brackets.
2. Open brackets must be closed in the correct order.

Example 1:

> **Input:** s = "()"
>
> **Output:** true

Example 2:

> **Input:** s = "(){}[]"
>
> **Output:** true

Example 3:

> **Input:** s = "(]"
>
> **Output:** false

3.1 Easier version

Before focusing on the main problem, let's resolve an easier version where the string is only composed of '`(`' and '`)`' characters.

Problem description: Check if `s` is a valid string composed of parentheses: '`(`', '`)`'.

This version is easier because we don’t need to validate three types of parentheses. We only need to care about ‘(' and’)’ instead.

Let’s see some example of valid strings:

```
()       // valid
()()()   // valid
(()())   // valid
((()())) // valid
```

All of these strings have two things in common:

1. They have the same number of ‘(’ and ‘)’.
2. There is no close parenthesis ‘)’ if there is no open parenthesis ‘(’ first.

All strings that follow these two rules are valid. Otherwise, a string would be invalid if one or two of these rules were broken:

```
(()  // invalid as the first rule is broken
())  // invalid as the first and second rules are broken
)(   // invalid as the second rule is broken
```

In order to solve this problem, we need to iterate over the array and verify that both rules are satisfied:

- To check the first rule, we can use a `counter` that will be incremented each time we see an open parenthesis (and decremented if we see a closing parenthesis ‘)’. At the end of the iteration, if `counter` is equal to `0`, the first rule is met.
- For the second rule, we can take advantage of the `counter` from the first rule. How? If `counter` is less than `0` at some point, it means we have seen a closing parenthesis ‘)’ without its matching open parenthesis ‘(’. Remember, we are decrementing the counter each time we see ‘)’.

```
function isValid(s) {
    let counter = 0;

    for (let char of s) {
        if (char === '(') {
            counter++;
        } else if (char === `(`){
             counter--;
        }

        if (counter < 0) return false; // For each parenthesis,
         ↪  we check the second rule.
    }

    return counter === 0; // We check the first rule. Same
     ↪  number of ( and ).
}
```

This algorithm is pretty straightforward. We have a `for` loop to iterate over the array. We check if the current char is '`(`' and then we increment the counter; otherwise, if the char is '`)`', we decrement it.

After each iteration, we check the second rule: if `counter` is less than `0`, we return `false` and stop the execution as we already know the string is invalid.

At the end of the loop, we verify the first rule: if `counter` is `0`, we return `true` as the string has the same number of '`(`' and '`)`'.

Let's see the execution for a valid `s`, `(())`:

Step 1:

```
(())
^
char

counter = 1;
```

Step 2:

```
(())
 ^
 char

counter = 2;
```

Step 3:

```
(())
  ^
  char

counter = 1;
```

Step 4:

```
(())
   ^
   char

counter = 0;
```

Step 5:

We have exited the loop, so we execute the last check.

```
counter === 0; // true
```

Now, let's execute step by step when `s` is invalid, `())(`:

Step 1:

```
())(
^
char

counter = 1;
```

Step 2:

```
())(
 ^
 char

counter = 0;
```

Step 3:

```
())(
  ^
  char

counter = -1;

// return false
```

We return `false` on the third loop, as the second rule is broken since `counter` is less than `0`.

The time complexity of this solution is $O(n)$, where n is the number of characters `s` has. This is because we have a single loop that iterates over all `s` characters.

On the other hand, the space complexity is constant $O(1)$ as we don't store anything but a single variable.

Can we solve the original problem with a similar approach?

The difference from the original problem is that `s` contains `'('`, `')'`, `'{'`, `'}'`, `'['`, and `']'`. You may be thinking of using three different counters, one for each type of parenthesis. However, it would be much more difficult as we also need to keep track of the order between parentheses. Let's illustrate this with a simple example: `s` equals to `([)]`.

The parentheses `()` counter will keep track of all parentheses satisfying both rules (same amount of open and close parentheses, and not closing a parenthesis if there is no open parenthesis before). In the same way, the brackets `[]` counter will do the same.

Both counters will say `s` is valid separately because they are not taking into account the order between different types of parentheses.

We can easily solve this problem with the help of the **Stack** [1] data structure.

3.2 Stack

As a refresher, we are going to quickly discuss the Stack data structure.

We can think of a stack as an array with some constraints. It stores data in the same way arrays do; it contains a list of elements. However, it has three restrictions:

1. Elements can be inserted only at the top of a stack.
2. Elements can be deleted only from the end of a stack.
3. Only the element at the top can be read.

In other words, stack operations follow the LIFO (Last In, First Out) approach: The last element pushed onto a stack is always the first element popped from it.

We can visualize stacks as vertical arrays:

```
------
- 25 -
------
-  6 -
------
- 16 -
------
```

We can push a new element (15):

```
------
- 15 -
------
- 25 -
------
-  6 -
------
- 16 -
------
```

We can push other element (1):

```
------
-  1 -
------
- 15 -
------
- 25 -
------
-  6 -
```

```
------
- 16 -
------
```

And then pop the element (1) from the top of the stack:

```
------  // 1 is popped
- 15 -
------
- 25 -
------
-  6 -
------
- 16 -
------
```

Now that we remember what a stack is and how it works, let's use it to solve the original problem.

3.3 Solution to the parentheses problem

Remember, in the original problem, we need to determine if a string `s` containing just the characters `'('`, `')'`, `'{'`, `'}'`, `'['`, and `']'`, is valid.

As we said in the previous section, a valid string must satisfy two rules:

1. They have the same number of `(` and `)`
2. There is no close parenthesis `)` if there is no open parenthesis `(` first.

But in this case, we need a third rule, as we have different types of parentheses. Without a third rule, this string would be considered valid:

```
([)] // This would be valid as it satisfies rules #1 and #2
```

Let's introduce a third rule:

3. Open brackets must be closed by the same type of brackets and in the correct order.

A stack comes in handy for solving this problem, as it can help us check whether all three rules are met. It can act as a counter while tracking the parentheses types and order.

This is a possible solution:

```
const counterparts = {
    "(" : ")",
    "{" : "}",
    "[" : "]"
}

function isValid(s) {
    const stack = [];

    for (let char of s) {
        if (isOpenParenthesis(char))
         ↪  stack.push(counterparts[char]);
        else if (stack.pop() != char) return false;
    }

    return stack.length === 0;
};

const isOpenParenthesis = (char) => char === "(" || char === "{"
 ↪  || char === "[";
```

How does this code work?

We have the main function `isValid` that receives the string s as input and returns `true/false`.

> **Note:** In JavaScript, arrays have both push and pop methods. The push method adds a new element at the end of the array, and pop removes an element from the end of the array.
>
> If we think of the end of the array as the top of the stack, we can use arrays as stacks.

The `isValid` function declares a new array `stack` that will be used as a stack. Then, we iterate over the array, character by character.

Within each iteration, we execute one of two exclusive actions:

1. If the current `char` is an open parenthesis, we add its counterpart (close parenthesis) to the stack.

2. If the current `char` is a close parenthesis, we pop the element at the top from the stack, and we check if it is the same element.

Each time we see an open parenthesis, we store its counterpart in the stack, and we use it to keep track of how many parentheses we expect to close, which type of parenthesis, and in which order.

After exiting the loop, we check if the stack size is equal to `0`. If so, we return `true` because the first rule is met.

Let's execute the algorithm step by step for different inputs to better understand how it works.

In the first execution, `s` will be a valid string `({}[])`:

Step 1:

We see a '`(`', so we push '`)`' to the stack. This is its matching parenthesis that we expect to find in order to have a valid pair '`()`' in the correct order.

```
({}[])
^
char

-----
- ) -
-----
stack
```

Step 2:

Another open parenthesis, we repeat the same as in Step 1.

```
({}[])
 ^
 char

-----
- } -
-----
- ) -
-----
stack
```

Step 3:

We see a close parenthesis. To check if it is valid, we pop from the stack and compare if that is the close parenthesis we were expecting.

Implicitly, the stack keeps the order of closing parentheses we need to ensure the string is valid. As it is a valid string, we first close the last open pair.

Do you see the relation between this problem and stacks now? In valid strings, parentheses are opened and closed in LIFO order. The last open parenthesis '{' is the first we need to close '}'.

Since the element that was at the top of the stack is the closing parenthesis we were expecting, we continue as s is still valid.

```
({}[])
  ^
  char

----- // stack.pop() returns '}'
- ) -
-----
stack
```

Step 4:

```
({}[])
   ^
   char

-----
- ] -
-----
- ) -
-----
stack
```

Step 5:

```
({}[])
    ^
    char
```

```
----- // stack.pop() returns ']', as we expected.
- ) -
-----
stack
```

Step 6:

```
({}[])
     ^
     char

--------- // stack.pop() returns ')', which is equals to char.
- empty -
---------
stack
```

Step 7:

We have now finished the `for` loop, so we check if the stack is empty to ensure rule #1 is satisfied.

We return `true` as the stack is empty.

Now let's see the execution for an invalid string `s`, `([]{)}`:

Step 1:

```
([]{)}
^
char

-----
- ) -
-----
stack
```

Step 2:

```
([]{)}
 ^
 char

-----
- ] -
```

```
-----
- ) -
-----
stack
```

Step 3:

```
([]{)}
  ^
  char

----- // stack.pop() returns ']', which is equal to char.
- ) - // so we continue as s is still valid
-----
stack
```

Step 4:

```
([]{)}
   ^
   char

-----
- } -
-----
- ) -
-----
stack
```

Step 5:

```
([]{)}
    ^
    char

----- // stack.pop() returns '}', which is not equal to char.
- ) - // We were expecting `{}` pair to be closed before.
-----
stack
```

We return false, as the parentheses were not closed in the correct order.

The time complexity of this solution is $O(n)$ where n is the number of characters s has. This is because we have a single loop that iterates all over s characters. On each loop, we call to push and pop, which both are $O(1)$.

On the other hand, the space complexity is $O(n)$ as we are storing n/2 elements in the stack. The bigger s is, the more space we need.

3.4 Summary

- We identified the 3 rules that need to be satisfied for a string to be valid.
- We used a stack to keep the count of open parentheses, its type and the order.
- The problem was solved in linear time and space, $O(n)$, where n is the number of characters s has.

3.5 References

- [1] Stack. https://www.geeksforgeeks.org/stack-data-structure/.

4 Valid Palindrome (Two Pointers)

Determine if a given string, `s`, can be transformed into a palindrome by removing at most one character. Return `true` if it is possible, `false` otherwise.

> A palindrome [1] is a word, number, phrase, or other sequence of characters that reads the same backward as forward, such as `madam` or `racecar`.

Example 1:

> **Input:** s = “aba”
>
> **Output:** true

Example 2:

> **Input:** s = “abca”
>
> **Output:** true
>
> **Explanation:** You could delete the character ‘c’.

Example 3:

> **Input:** s = “abc”
>
> **Output:** false

4.1 Simplified version of the problem

Before solving the problem, let’s first focus on an easier version of it:

> Given a string `s`, return `true` if `s` is a palindrome.

This is a classic problem that can be easily solved using two pointers. One pointer points to the first character, and the other points to the last character. If both are the same character, s is a potential palindrome. We move the first pointer to the right and the second one to the left, repeating until all characters have been checked.

We will understand it better by examining the code:

```
function isPalindrome(s) {
    let start = 0;
    let end = s.length-1;

    while (start <= end) {
        if (s.charAt(start) != s.charAt(end)) {
            return false;
        }

        start++;
        end--;
    }

    return true;
}
```

Let's see the step-by-step execution for two different inputs. First, s equals abcba, which is a palindrome.

Step 1:

```
a    b    c    b    a
^                   ^
start               end

s.charAt(start) === s.charAt(end)
```

First and last characters are the same, so we continue with the loop execution as s is a palindrome so far.

Step 2:

```
a    b    c    b    a
```

```
     ^         ^
     start     end

s.charAt(start) === s.charAt(end)
```

Both `start` and end point to characters equal to b. We continue.

Step 3:

```
a    b    c    b    a
          ^
          start

          ^
          end

s.charAt(start) === s.charAt(end)
```

Same character. We have checked all the characters from `s`, so we exit the loop and return `true` as it is a palindrome.

Let's see the step-by-step execution for `s` equal to `abcsba`, which is not a palindrome.

Step 1:

```
a    b    c    s    b    a
^                        ^
start                    end

s.charAt(start) === s.charAt(end)
```

Step 2:

```
a    b    c    s    b    a
     ^              ^
     start          end

s.charAt(start) === s.charAt(end)
```

Step 3:

```
a    b    c    s    b    a
          ^
          start
               ^
               end

s.charAt(start) !== s.charAt(end)
```

Since `s.charAt(start)` is not equal to `s.charAt(end)`, we can return `false` as we know `s` is not a palindrome.

The time complexity of this algorithm is `O(n)`, where `n` is the length of `s`, as we iterate over the entire array.

The space complexity is `O(1)` as we don't store data in any data structure.

4.2 Solution to the original problem

Now that we know how to implement the `isPalindrome` function to check if a given string `s` is a valid palindrome, we can solve the original problem.

The difference between the original problem and the easier version is that we can try removing any character from `s` to make it a palindrome. For example:

> a b c j b a

It is not a palindrome, but we can remove `j` to make it a valid palindrome: `abcba`.

The function we need to implement will be very similar to the `isPalindrome` one we just implemented in the previous section. We will start with two pointers and continue checking two characters on each iteration. The difference is that when we find different letters, we should not return `false` right away; instead, we should skip that different character and continue.

But which character should we remove? The one pointed to by `start` or the one pointed to by `end`? We know both are different, but we don't know which one we should remove to make s a valid palindrome. We will try removing the one pointed to by `start` and check if s is a palindrome. If it is not, we will do the same with the character pointed to by end.

We are going to understand it better by examining the code and some step-by-step executions:

```
function validPalindrome(s) {
    let start = 0;
    let end = s.length - 1;

    while (start <= end) {
        if (s.charAt(start) != s.charAt(end)) {
            return isPalindrome(s, start+1, end)
                || isPalindrome(s, start, end-1)
        }
        start++;
        end--;
    }

    return true;
};

function isPalindrome(s, start, end) {
    while (start <= end) {
        if (s.charAt(start) != s.charAt(end)) {
            return false;
        }
        start++;
        end--;
    }

    return true;
}
```

The main function, `validPalindrome`, is very similar to the `isPalindrome` function from the previous section. The only difference is that if we find a different character, instead of returning `false`, we are going to try skipping the character pointed to by `start` and the one pointed to by end. If, after removing one of these two characters, s becomes a valid palindrome, we will return `true`; otherwise, `false`.

The `isPalindrome` is the same function from the previous section. We just added two new arguments, `i` and `j`, that indicate from where we should continue checking `s`.

Let's start by looking at a step-by-step execution for `s` equal to `abcchba`, which is a valid palindrome after removing h from it.

Step 1:

```
validPalindrome("abcchba")

a   b   c   c   h   b   a
^                       ^
start                   end

start = 0
  end = 6

s.charAt(start) == s.charAt(end) // a = a, continue
```

Step 2:

```
validPalindrome("abcchba")

a   b   c   c   h   b   a
    ^               ^
start               end

start = 1
  end = 5

s.charAt(start) == s.charAt(end) // b = b, continue
```

Step 3:

```
validPalindrome("abcchba")

a   b   c   c   h   b   a
        ^       ^
        start   end
start = 2
  end = 4

s.charAt(start) != s.charAt(end) // c != h, call helper
 ↪  function
```

Since c != h we will return the result of calling the helper function `isPalindrome` twice:

- One time with `start=3` and `end=4`, so we skip the c
- Another time with `start=2` and `end=3`, so we skip the h

In other words, we are going to try if, after removing `c`, `s` becomes a palindrome, and if not, we will try the same removing h. Because we know that both letters are different, but we don't know which one prevents s from being a palindrome, so we try both ways.

Step 4:

```
isPalindrome("abcchba", 3, 4)

a   b   c   c   h   b   a
            ^
            start
                ^
                end

start = 3
  end = 4

s.charAt(start) != s.charAt(end) // c != h, return false
```

We removed `c`, but `s` is not a palindrome; then we return `false`. We still need to try removing h.

Step 5:

```
isPalindrome("abcchba", 2, 3)

a   b   c   c   h   b   a
        ^
        start
```

```
          ^
          end

start = 2
  end = 3

s.charAt(start) != s.charAt(end) // c == c, continue
```

Step 6:

```
isPalindrome("abcchba", 2, 3)

a   b   c   c   h   b   a
            ^
            start
        ^
        end

start = 3
  end = 2

// Exit loop and return true
```

Since `start > end`, we exit the loop and return `true`. Consequently, `validPalindrome` also returns `true`.

Now let's see an example where `s` is not a valid palindrome after removing one character: `abcka`.

Step 1:

```
validPalindrome("abcka")

a   b   c   k   a
^               ^
start           end

start = 0
  end = 4

s.charAt(start) != s.charAt(end) // a == a, continue
```

Step 2:

```
validPalindrome("abcka")

a    b    c    k    a
     ^         ^
     start     end

start = 1
  end = 3

s.charAt(start) != s.charAt(end) // b != k, call helper
 ↪  function
```

Since we have found a mismatch, we are going to call the helper function:

- `isPalindrome("abcka", 2, 3)`
- `isPalindrome("abcka", 1, 2)`

Step 3:

```
isPalindrome("abcka", 2, 3)

a    b    c    k    a
          ^
          start
               ^
               end

start = 2
  end = 3

s.charAt(start) != s.charAt(end) // c != k, return false
```

It is not a palindrome after removing b; we still have a chance if we remove k.

Step 4:

```
isPalindrome("abcka", 1, 2)

a    b    c    k    a
     ^
     start
          ^
```

```
        end

start = 1
  end = 2

s.charAt(start) != s.charAt(end) // b != c, return false
```

The second call to `isPalindrome` also returned `false`, so our main function `validPalindrome` will return `false` as well.

The time complexity of this algorithm is `O(n)`, where `n` is the length of `s`, as we iterate over the entire array. Even if we might be iterating two times in the worst case, we can simplify it: `O(2n) == O(n)`.

The space complexity is `O(1)` as we don't store data in any data structure.

4.3 Summary

- We can check if a given string `s` is a palindrome by using two pointers.
- We modified the original `isPalindrome` function to solve the original problem.
- When the characters pointed to by `start` and `end` are not the same, we try removing one of them, and then we try removing the other.
- We solved it with `O(n)` time complexity.

4.4 References

- [1] Palindrome. https://en.wikipedia.org/wiki/Palindrome.

5 House Robber (Recursion)

You are a professional robber planning to rob houses along a street. Each house has a certain amount of money stashed. The only constraint stopping you from robbing each of them is that adjacent houses have security systems connected, and **it will automatically contact the police if two adjacent houses are broken into on the same night.**

Given an integer array `nums` representing the amount of money of each house, return *the maximum amount of money you can rob tonight* ***without alerting the police***.

Example 1:

> **Input:** nums = [1,2,3,1]
>
> **Output:** 4
>
> **Explanation:** Rob house 1 (money = 1) and then rob house 3 (money = 3).
>
> Total amount you can rob = 1 + 3 = 4.

Example 2:

> **Input:** nums = [2,7,9,3,1]
>
> **Output:** 12
>
> **Explanation:** Rob house 1 (money = 2), rob house 3 (money = 9) and rob house 5 (money = 1).
>
> Total amount you can rob = 2 + 9 + 1 = 12.

5.1 Introducing recursion

The House Robber problem can be solved using recursion [1]. Before jumping to the solution, we are going to refresh how recursion works.

Recursion is the term for a function calling itself.

```
function foo() {
    foo(); // It will call itself infinitely
}
```

Iterative algorithms can be written recursively and vice versa. The following function prints all numbers from n to 1:

```
function printCountdown(n) {
    for (let i=n; i <=1; i++) {
        console.log(i)
    }
}

// printCountdown(5) will print:
// 5
// 4
// 3
// 2
// 1
```

We can write the same function using recursion, without using any loop:

```
function printCountdown(n) {
    console.log(n)

    printCountdown(n-1);
}
```

Let's execute that function step by step, `printCountdown(5)`:

- Step 1: n=5, we print 5, and then call `printCountdown(4)`
- Step 2: n=4, we print 4, and then call `printCountdown(3)`
- Step 3: n=3, we print 3, and then call `printCountdown(2)`
- …

You can see we don't need a loop to implement this function. We just use recursion and change the input.

The issue with the recursive `printCountdown` function is that it will never stop. We want it to stop after printing `1`.

We need to define the *base case*. The base case is a conditional statement that ensures if n is 1, we don't call `printCountdown` anymore.

```
function printCountdown(n) {
    console.log(n)

    if (n > 1) {
        printCountdown(n-1);
    }
}
```

Now, when the number is `1`, we will print it, and we won't call `printCountdown` again as n>1 will be false.

Now, let's write a recursive function that prints the numbers in the opposite order, from 1 to n.

```
function printNumbers(n) {
    if (n == 0) return; // Base case

    printNumbers(n-1); // Recursive call

    console.log(n); // Prints the number
}
```

If we call `printNumbers(3)` it will print:

```
> printNumbers(5)

1
2
3
```

To have a better understanding of how recursion works, we need to find out how computers handle it. We call `printNumbers(3)`, and the computer calls `printNumbers(2)` from within the `printNumbers(3)` function itself. So, `printNumbers(2)` is called before `printNumbers(3)` has finished its execution.

The computer uses the *call stack* to keep track of all executions. That stack is the same data structure we already introduced in the “Valid Parentheses” chapter.

This is how the stack looks for our example:

The computer starts executing `printNumbers(3)`, and in the middle of the execution, it then calls `printNumbers(2)`. In order to keep track of the unfinished execution of `printNumbers(3)`, the computer pushes that information onto the call stack:

```
-------------------
- printNumbers(3) -
-------------------
Call stack
```

Now, the computer continues executing `printNumbers(2)`, which calls `printNumbers(1)`:

```
-------------------
- printNumbers(2) -
-------------------
- printNumbers(3) -
-------------------
Call stack
```

After that, `printNumbers(1)` is executed, which calls `printNumbers(0)`.

```
-------------------
- printNumbers(1) -
-------------------
- printNumbers(2) -
-------------------
- printNumbers(3) -
-------------------
Call stack
```

Since `0` is the base case, `printNumbers(0)` completes without calling the `printNumbers` function anymore.

Following, the computer checks if there is something pending on the call stack. It pops the first element, which is the last function called, and continues its execution. In this case, the element popped is `printNumbers(1)`.

```
-------------------
- printNumbers(2) -
-------------------
- printNumbers(3) -
-------------------
Call stack

Element popped: printNumbers(1)
```

The computer completes the execution of the element popped from the stack (`printNumbers(1)`):

```
console.log(n); // code pending to be executed of function(1) -
 ↪  it prints 1
```

This process is repeated until there are no elements left on the stack.

Let's visualize the whole process when we call `printNumbers(3)`:

```
printNumbers(3)
    printNumbers(2)
        printNumbers(1)
            printNumbers(0) // Base case, it returns.
        console.log(1) // printNumbers(1) popped from the stack
    console.log(2)
console.log(3) // No more elements on the call stack. It
 ↪  finishes the execution.
```

In the end, it will print:

```
> printNumbers(3)

1
2
3
```

This was a brief introduction to how recursion works. For more details about recursion, I recommend reading the guide referenced in [1].

5.2 Solution to the house robber problem

After this recursion refresher, we can start solving the problem recursively.

For each house, the robber has 2 options:

- Rob the house
- Skip the house

If they rob the house, they won't be allowed to rob neighboring houses, which might have more money, as can be seen in this example:

```
nums = [2, 20, 6]
```

For the first house (money = 2), the robber can rob it or skip it. If they rob it, the maximum amount of money that can be robbed is 8 (house 1 + house 3). If they don't rob it, they can rob house 2 (money = 20). In this example, it is better not to rob house 1 so they can rob house 2.

We need to write an algorithm that, for each house, takes into account both options: rob the current house or skip it, and then compares which alternative yields more money.

Let's write a recursive algorithm to solve this problem:

```
function rob(nums) {
    return robHouse(nums, 0);
};

function robHouse(nums, house) {
    if (house >= nums.length) {
        return 0;
    }

    let totalAmountRobbingHouse = nums[house] + robHouse(nums,
     ↪  house+2);
    let totalAmountSkippingHouse = robHouse(nums, house+1);

    return Math.max(totalAmountRobbingHouse,
     ↪  totalAmountSkippingHouse);
}
```

We have the main function, called `rob`; its only instruction is to call `robHouse`, which is our helper recursive function.

It will return the maximum amount of money that can be robbed when starting from the first house.

> *Note that starting from the first house does not mean that we rob it; we will only do so if that yields the maximum amount of money.*

The function `robHouse` receives two arguments: `nums`, which contains the array of money each house has, and `house`, which represents the current house we are considering. The `rob` function calls `robHouse` with `house = 0` as we want to start robbing from the first house.

The first `if` condition is the base case that stops the recursion. When `house >= nums.length`, it means we have already considered all houses, so we don't want to call `robHouse` again, as there are no more houses left. Instead, we return `0` because we can't rob any money from a non-existing house.

After the base case, we have two recursive calls to `robHouse` itself:

- The first call robs the house and then visits the `house+2` house, as we can't rob neighboring houses.
- The second call skips the house (does not sum the money) and tries with the next house. We can visit the next house if we don't rob the current one.

Once we have the amount of money returned by both possibilities, we return the one with the maximum amount, as the problem asks.

Let's visualize in form of tree the recursive calls that will be done for `nums = [3, 20, 15]`:

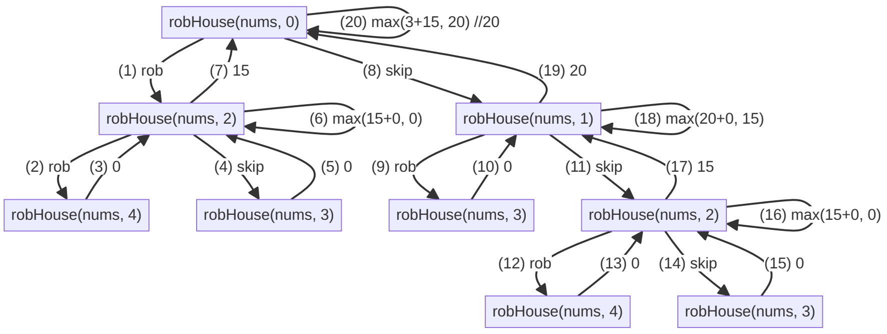

Figure 5.1: Execution for nums = [3, 20, 15]

We are going to explain what each step in the diagram does.

Step 1

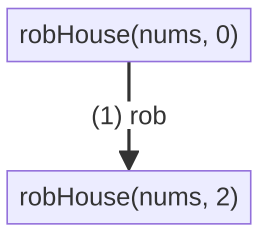

Figure 5.2: Step 1

`robHouse(nums, 0)` robs the current house and calls `robHouse(nums, 2)`

Step 2

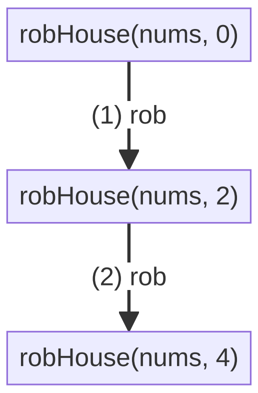

Figure 5.3: Step 2

`robHouse(nums, 2)` robs the current house and calls `robHouse(nums, 4)`

Step 3

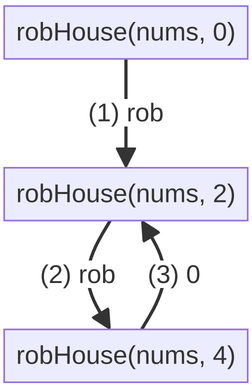

Figure 5.4: Step 3

`robHouse(nums, 4)` is executed and returns `0` without calling itself again, as it is the base case since `4 > nums.length`.

Step 4

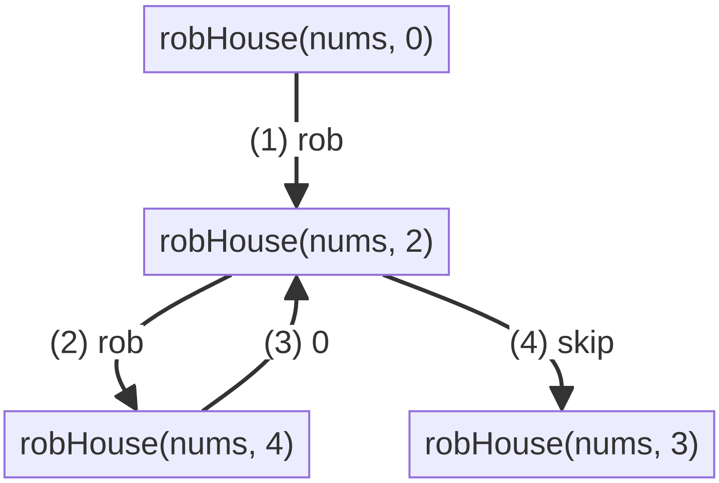

Figure 5.5: Step 4

`robHouse(nums, 2)` executes the other choice: it skips robbing the current house (2) and calls `robHouse(nums, 3)`.

Step 5

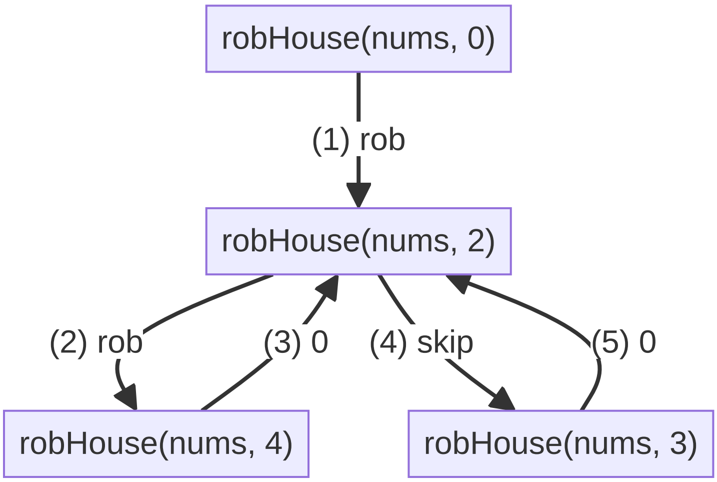

Figure 5.6: Step 5

`robHouse(nums, 3)` returns 0 as it is the base case.

Steps 6 and 7

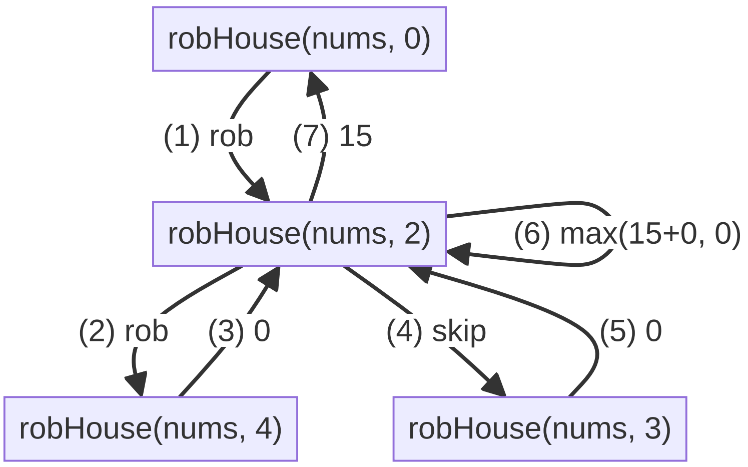

Figure 5.7: Steps 6 and 7

`robHouse(nums, 2)` has its 2 variables calculated:

- `totalAmountRobbingHouse` = 15 + 0
- `totalAmountSkippingHouse` = 0;

It returns, 15 as it is the maximum for the 2 options. This means that when we start robbing from `house=2` (the third house), the best choice a robber can make is to rob it. If they don't rob it, they won't be able to rob more houses because it is the last. The amount of money is 0 if they don't rob any house.

`robHouse(nums, 2)` returns 15.

Step 8

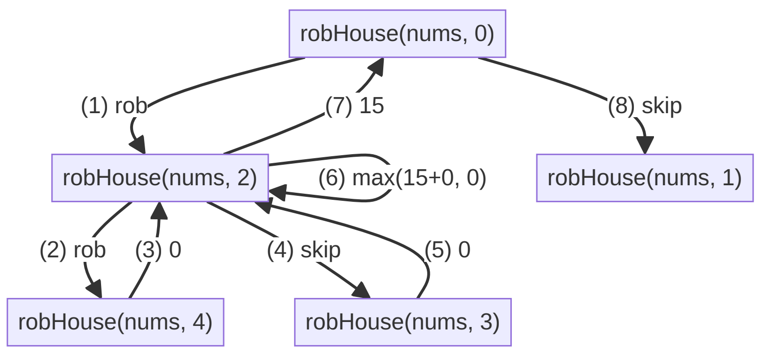

Figure 5.8: Step 8

We are back on `robHouse(nums, 0)` execution. All the recursive calls to calculate `totalAmountRobbingHouse` have been made. Now, we know that `totalAmountRobbingHouse` for `robHouse(nums, 0)` is `nums[house] + robHouse(nums, 2)`, so `3 + 15`, which is 18.

The maximum amount of money we can get if we rob the first house is 18.

The algorithm is now going to execute the other choice: skipping the first house. So, we call `robHouse(nums, 1)` to get the result.

Steps 9 and 10

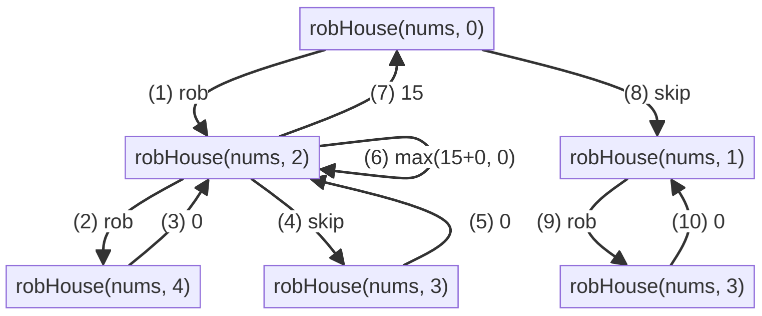

Figure 5.9: Steps 9 and 10

`robHouse(nums, 1)` checks the first option, robbing the second house

(`house=1`), and for that, it calls `robHouse(nums, 3)`. We already executed `robHouse(3)` earlier. It returns `0` as it is the base case.

Steps 11 to 19

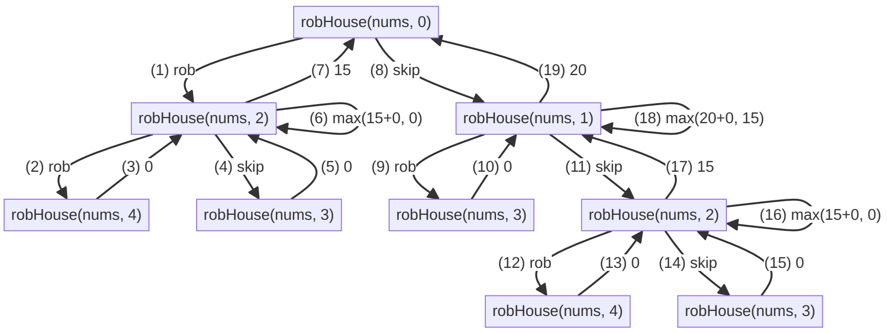

Figure 5.10: Steps 11 to 19

At this point, `robHouse(nums, 1)` calculates the second option: not robbing the second house (`house=1`) and going to the next one. It calls `robHouse(nums, 2)`.

We already executed `robHouse(nums, 2)` in previous steps, and we know it returns `15`.

Now, at Step 18 we know the values for `robHouse(nums, 1)` variables:

- `totalAmountRobbingHouse` = `nums[house] + robHouse(nums, 3)` = `20 + 0`;
- `totalAmountSkippingHouse` = `robHouse(nums, 2)` = `15`;

Since `totalAmountRobbingHouse` contains the maximum amount of money (20), it is the one returned by `robHouse(nums, 1)` in Step 19.

Step 20

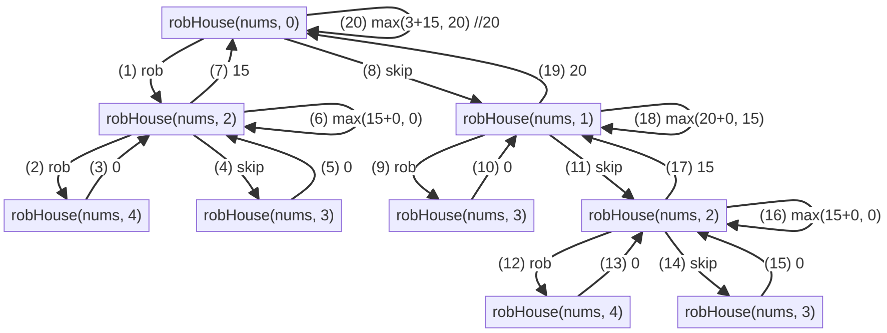

Figure 5.11: Step 20

All the recursive calls have been completed. We are back at `robHouse(nums, 0)` execution. We know both variables:

- `totalAmountRobbingHouse` = `nums[house] + robHouse(nums, 2)` = `3 + 15` = `18`;
- `totalAmountSkippingHouse` = `robHouse(nums, 1)` = `20`;

If we rob the first house, we will get `18` units of money, but if we skip it, we will get `20`. It will return `20`, which is the correct solution, because it is better to rob the second house instead of the first to get `20` units of money.

We have seen how the recursive algorithm works and solves the problem, but how efficient is it?

5.2.1 Time and space complexity

Time and space complexity it's sometimes challenging to compute for recursive algorithms. Next, we are going to introduce some heuristics that can help us.

Time complexity

Generally, the time complexity will depend on 2 variables:

- Number of recursive calls
- Work per recursive call

In other words, the time complexity will be:

> `O(number of recursive calls * work for each recursive call)`

In our example, work for each recursive call is constant, so $O(1)$.

We need to first find the number of recursive calls. On **Figure 11** we see the diagram has 9 nodes. The first node `robHouse(nums, 0)` is the initial call, and it makes 8 recursive calls.

Here is a heuristic to find the number of recursive calls:

> Number of nodes = O(b^d) Where b is the branching factor, and d is the depth of recursion.

As we can see, the number of recursive calls depends on two variables:

- **Branching factor**: This is how many times we make a recursive call.
  - In our example we make 2 recursive calls:
    * `nums[house] + robHouse(nums, house+2);` - we rob the house
    * `robHouse(nums, house+1);` - we skip the house
- **Depth of recursion**: This is how deep our recursion goes.
  - In our example we will go until the base case (`house >= nums.length`).
  - Our recursion will go from `0` to `nums.length`
  - So, the depth of recursion in our case is `nums.length` (number of houses).

With this information, we can conclude that the time complexity is:

> $O(b^d$ * work_per_recursive_call) = O(2^{nums.length})$.

Space complexity

The space complexity is a little easier to compute, but not trivial. It depends on the depth of recursion and the space used by each recursive call. The branching factor does not affect on space, as we can reuse space, since we execute one branch at a time. (It affects during time complexity calculation, as we can't reuse time.)

This is the formula:

> $O(d * s)$ Where d is the depth of recursion, and s is the space per recursive call.

For our problem:

> $O(nums.length * 1) = O(nums.length)$

5.3 Dynamic Programming

In the previous point, we calculated the time and space complexity for our problem. We saw that the time complexity is:

> $O(2^{nums.length})$

Which is an exponential function depending on the number of houses. This may be reasonable for a small number of houses, but it becomes huge for a greater number of houses:

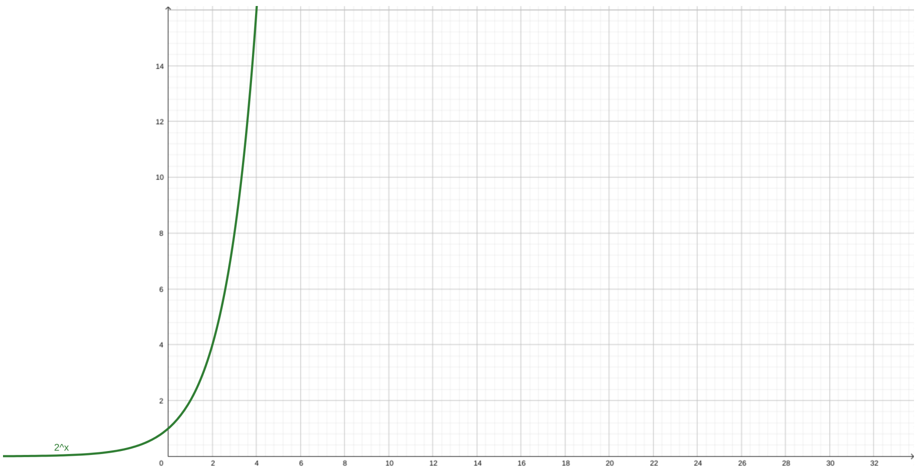

Figure 5.12: Exponential function

This is not efficient at all. Can we do it better? The answer is yes, with dynamic programming [2]. If we take a look again at the recursion tree for `nums = [3, 20, 15]` you will notice that we are making some calls twice:

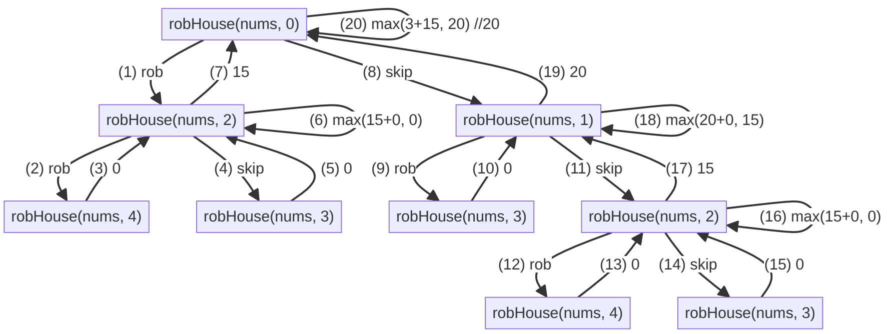

Figure 5.13: Execution for nums = [3, 20, 15]

Let's list the number of calls for each house:

- `robHouse(nums, 0)`: 1
- `robHouse(nums, 1)`: 1
- `robHouse(nums, 2)`: 2
- `robHouse(nums, 3)`: 3
- `robHouse(nums, 4)`: 2

We are calling `robHouse(nums, 2)` and `robHouse(nums, 4)` twice. And what is worse: we are calling `robHouse(nums, 3)` three times. We don't need to call them more than once, because we already know what are they going to return:

- `robHouse(nums, 0)`: it will always return `20`.
- `robHouse(nums, 1)`: it will always return `20`.
- `robHouse(nums, 2)`: it will always return `15`.
- `robHouse(nums, 3)`: it will always return `0`.
- `robHouse(nums, 4)`: it will always return `0`.

We are wasting time calling the same function with the same arguments more than once. Those extra calls are not needed. Dynamic programming can help us to optimize our recursive algorithm.

> *Dynamic programming* is the process of optimizing recursive problems that have overlapping subproblems.

For this optimization, we are going to use a cache in our algorithm. This cache will store the result of calling `robHouse` for a given house. If `robHouse` is called again with the same arguments, we will return the result from the cache, instead of calculating it again. This technique is also called ***memoization*** [3], [4].

As you can notice, this is a tradeoff between time and space. We will need more space (as we have to store the results on the cache), but we will improve time.

5.4 Dynamic programming solution to the house robber problem

This is the same solution as in previous sections, but using dynamic programming, or in other words, using a cache:

```
function rob(nums) {
    const cache = new Array(nums.length);
    return robHouse(nums, 0, cache);
};

function robHouse(nums, house, cache) {
    if (house >= nums.length) {
        return 0;
    }

    if (cache[house] != undefined) {
        return cache[house];
    }

    let totalAmountRobbingHouse = nums[house] + robHouse(nums,
     ↪  house+2, cache);
    let totalAmountSkippingHouse = robHouse(nums, house+1,
     ↪  cache);

    const result = Math.max(totalAmountRobbingHouse,
     ↪  totalAmountSkippingHouse);
    cache[house] = result;
    return result;
}
```

We introduced a new array that is used as cache. We could use any other data structure like a map or hash table, but I used an array for efficiency.

We introduced a new array that is used as a cache. We could have used any other data structure such as a map or hash table, but an array was chosen for efficiency.

Likewise, we introduced an `if` condition that checks if the value is present in the cache, and if so, it returns it, instead of calculating it. If it does not exist, we calculate it, and before returning it, we cache it. And that's it.

Now, we are going to see the new recursion tree:

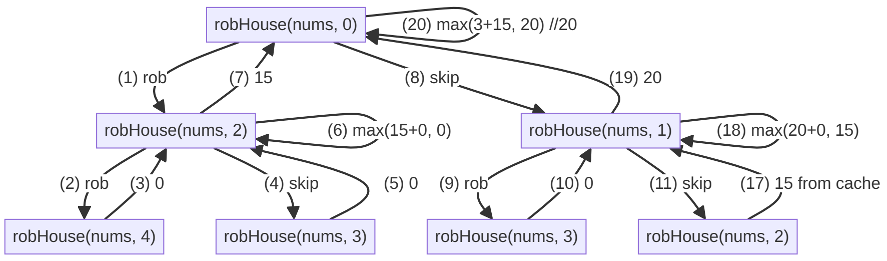

Figure 5.14: Execution for nums = [3, 20, 15] using dynamic programming

As we can see, the second call to `robHouse(nums, 2)` does not make any calculation, it just returns 15 from the cache. We are reducing the number of recursive calls.

5.4.1 Time and space complexity

The cache has a huge impact on improving the time complexity.

Time complexity

We already mentioned that the time complexity will depend on 2 variables and can be calculated using this formula:

> $O(numberofrecursivecalls * workforeachrecursivecall)$

The big difference with dynamic programming is the number of calls. In the previous algorithm we were doing $O(2^{nums.length})$ calls, due to we were making repeated and unnecessary calls.

But now, for each input, we only calculate it once. In other words, the number of function calls we do is `nums.length`. For each house, we don't call it again. We can conclude that the new time complexity is linear on the number of houses:

> $O(numberofrecursivecalls*workforeachrecursivecall) = O(nums.length)$

This is a significant improvement.

Space complexity

We will need more space because of the cache. The size of it will be `nums.length`. In terms of complexity, it will still be `O(nums.length)` because the recursion space complexity was `O(nums.length)`, and we introduce the cache which also has `O(nums.length)` complexity.

The final space complexity is:

> $O(nums.length + nums.length) = O(nums.length)$

Despite using more space, the complexity remains linear.

5.5 Summary

- We introduced how recursion works.
- We talked about how to compute time and space complexity for recursive algorithms.
- Finally, we learned to improve efficiency with dynamic programming.

5.6 References

- [1] Recursion. https://www.byte-by-byte.com/recursion/.

- [2] Dynamic programming.
  - https://en.wikipedia.org/wiki/Dynamic_programming.
- [3] Memoization. https://en.wikipedia.org/wiki/Memoization.
- [4] Memoization vs Dynamic programming.
  - https://stackoverflow.com/a/6185005/.1320113.

6 Combination Sum (Backtracking)

Find all unique combinations of elements from the array `candidates` that add up to the target integer `target`. You may use each element from `candidates` any number of times and may return the combinations in any order. Two combinations are considered unique if the frequency of at least one element is different.

Example 1:

> **Input:** candidates = [2,3,6,7], target = 7
>
> **Output:** [[2,2,3],[7]]
>
> **Explanation:** 2 and 3 are candidates, and 2 + 2 + 3 = 7. Note that 2 can be used multiple times. 7 is a candidate, and 7 = 7.
>
> These are the only two combinations.

Example 2:

> **Input:** candidates = [2], target = 3
>
> **Output**: []

6.1 Introducing backtracking

Backtracking [1] is a technique for solving constraint satisfaction problems using recursion. The idea behind it is to eliminate partial solutions that are not valid for solving the problem.

It is similar to brute force, but it has the ability to discard invalid solutions early. As soon as the algorithm identifies a candidate solution as invalid, it will reject it and "backtrack."

It is also similar to recursion, but the difference is that in recursion, the function keeps calling itself until a base case is reached. However, backtracking utilizes recursion to explore all possible solutions until we solve the problem.

In summary, backtracking, at each step, removes those choices that cannot give us a final solution and continues with the choices that could result in a potential valid solution.

6.1.1 What type of problems can be solved using backtracking?

Typically, backtracking can solve problems that have clear constraints to reach a solution. Also, it can be applied to problems where it is possible to test the validity of partial solutions in order to discard them early.

Some of the problems that can be resolved using backtracking can also be solved by dynamic programming or greedy algorithms in a more efficient way. Note that backtracking algorithms are usually exponential in time and space.

6.1.2 Backtracking by example

Let's see how backtracking works by using a simple problem:

> Given the following array of numbers `numbers = [1, 3, 5]`, return all its subsets, without duplicates.

We know that the number of subsets is 2^N, where N is the number of elements in the set. In our case, it is 8 subsets: - $[], [1], [3], [5], [1, 3], [1, 5], [3, 5], [1, 3, 5]$.

> Why 2^N subsets? By using our intuition: If we want to get a subset, for each element we have to decide if to include it or not. All these decisions are independent. In other words, for each number, you have two options: include it or not. And you have to make N decisions. So, for `[1, 3, 5]` you have: $2 * 2 * 2 = 2^3 = 8$.

Let's try to find a rule to calculate subsets. We know that the possible subsets for $[1, 3]$ are:

- $[], [1], [3], [1, 3]$

If we add 5 to all subsets of $[1, 3]$ we will have:

- $[5], [1, 5], [3, 5], [1, 3, 5]$

If you merge these subsets with the subsets of $[1, 3]$, you will notice that the result is equal to the subsets of $[1, 3, 5]$.

Do you see the recursive pattern?

- Subsets of $[1, 3, 5]$ = subsets of $[1, 3]$ and subsets of $[1, 3]$).append(5).

Then, subsets of $[1, 3, 5]$ can be calculated from $[1, 3]$. And subsets of $[1, 3]$ can be calculated from subsets of $[1]$. The base case is the empty set, whose subset is also the empty set.

Let's code the algorithm using backtracking:

```
function getSubsets(numbers) {
    const subsets = [];
    backtrack(numbers, subsets, [], 0);
    return subsets;
}

function backtrack(numbers, subsets, path, start) {
    subsets.push([...path]);

    for (let i=start; i<numbers.length; i++) {
        path.push(numbers[i]);
        backtrack(numbers, subsets, path, i+1);
        path.pop();
    }
}
```

This would be the resultant recursion tree after invoking `getSubsets([1, 3, 5])`:

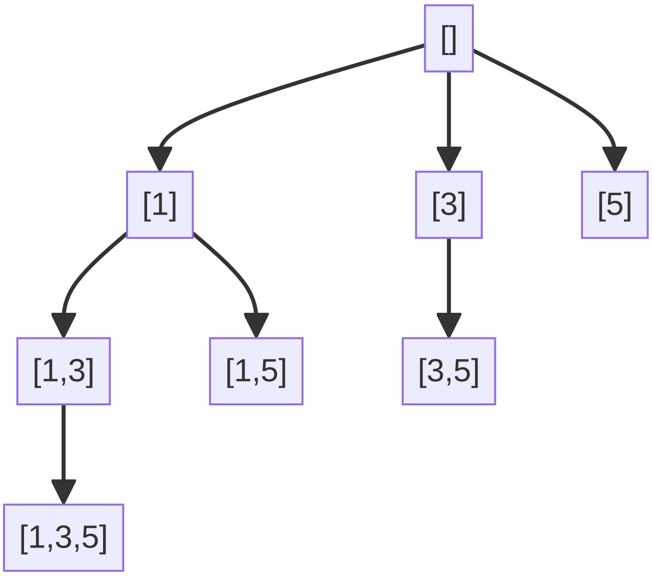

Figure 6.1: getSubsets([1, 3, 5])

Apart from the tree above, the execution can be visualized step by step:

Step 1

- Solution found: **[]**
- Adding number 1
- Calling backtrack with candidate: *[1]*, and start = 1 (next number = 3)

Step 2

- Solution found: **[1]**
- Adding number 3
- Calling backtrack with candidate: *[1,3]*, and start = 2 (next number = 5)

Step 3

- Solution found: **[1,3]**
- Adding number 5
- Calling backtrack with candidate: *[1,3,5]*, and start = 3 (next number = out of bounds)

Step 4

- Solution found: **[1,3,5]**
- Returning, as `start` (3) is greater than or equal to `numbers.length` (3).
- Removing 5, new candidate [1,3]
- End of while loop, returning.
- Removing 3, new candidate [1]
- Adding number 5
- Calling backtrack with candidate: *[1,5]*, and start = 3 (next number = out of bounds)

Step 5

- Solution found: **[1,5]**
- Returning, as `start` (3) is greater than or equal to `numbers.length` (3).
- Removing 5, new candidate [1]
- End of while loop, returning.
- Removing 1, new candidate []
- Adding number 3
- Calling backtrack with candidate: *[3]*, and start = 2 (next number = 5)

Step 6

- Solution found: **[3]**
- Adding number 5
- Calling backtrack with candidate: *[3,5]*, and start = 3 (next number = out of bounds)

Step 7

- Solution found: **[3,5]**
- Returning, as `start` (3) is greater than or equal to `numbers.length` (3).
- Removing 5, new candidate [3]
- End of while loop, returning.
- Removing 3, new candidate []
- Adding number 5
- Calling backtrack with candidate: *[5]*, and start = 3 (next number = out of bounds)

Step 8

- Solution found: **[5]**
- Returning, as `start` (3) is greater than or equal to `numbers.length` (3).
- Removing 5, new candidate []
- End of while loop, returning.

6.1.3 Template for backtracking problems

The following structure applies to most backtracking problems:

```
function backtrack(candidates, path, result, start) {
    if (isLeaf(start)) {
        result.push(path);
        return;
    }

    for (candidate in candidates) { // candidate starts from
     ↪   `start`
        path.push(candidate);
        backtrack(candidates, path, result, start+1)
        path.pop(); // removes candidate
    }
}
```

Where:

- `path` = current path we are trying to get a solution.
- `candidates` = input for the algorithm.
- `result` = variable that will store all solutions found.
- `start` = first candidate to start with.

6.2 Solution to the Combination Sum problem

We are going to use backtracking to solve the Combination Sum problem by applying the template we introduced in the previous section.

We said that a problem can be resolved with backtracking if it has clear constraints to reach a solution and if it is possible to test the validity of partial solutions.

For the Combination Sum problem, constraints are:

- Find a subset whose elements add up to `target`.
- Repeated elements can be used.

And partial solutions can be early rejected by:

- Checking if the current partial solution is greater than `target`; if so, reject the solution and backtrack.

As a conclusion, a backtracking solution fits this problem; hence, let's solve it by applying the backtracking template:

```javascript
function combinationSum(candidates, target) {
    const combinationsResult = [];
    backtrack(candidates, combinationsResult, [], target, 0);
    return combinationsResult;
}

function backtrack(candidates, combinationsResult,
 ↪  candidateCombination, currentTarget, start) {
    if (currentTarget < 0) {
        return; // Partial solution (candidateCombination) is
         ↪  not valid.
    }

    if (currentTarget === 0) { // Solution found
        combinationsResult.push([...candidateCombination]);
        return;
    }

    for (let i=start; i<candidates.length; i++) {
        candidateCombination.push(candidates[i]);
        backtrack(candidates, combinationsResult,
         ↪  candidateCombination, currentTarget-candidates[i],
         ↪  i); // no i+1 as repeated candidates are allowed
        candidateCombination.pop();
    }
}
```

The code is very similar to the subsets problem's code, as both use the backtracking template. However, there are some differences shown in the following table:

Difference	Subsets problem	Combination Sum
`isLeaf` / `isSolution` function	It adds every candidate to the solutions as it searches all possible subsets.	It checks if the candidates sum to `target`.
Partial solutions	No partial solutions discarded, as all subsets are wanted.	If the current candidates sum is greater than `target` the partial solution is discarded.
Backtrack call	Each backtrack function is called with `i+1` as repeated numbers are not allowed.	Each backtrack function is called with `i` as duplicates are allowed.

We are going to see the step-by-step execution for `combinationSum([2,6], 6)`:

Step 1

- Current target: 6.
- Adds candidate number 2.
- Calls backtrack with candidate solution: *[2]*, and start = `0` (next number = 2).

Step 2

- Current target: 4.
- Adds candidate number 2.
- Calls backtrack with candidate solution: *[2,2]*, and start = `0` (next number = 2).

Step 3

- Current target: 2.
- Adds candidate number 2.
- Calls backtrack with candidate solution: *[2,2,2]*, and start = `0` (next number = 2).

Step 4

- Solution found: **[2,2,2]**. Function returns.
- Removes 2, new candidate [2,2].
- Adds candidate number 6.
- Calls backtrack with candidate solution: *[2,2,6]*, and start = 1 (next number = 6).

Step 5

- Current target (-4) is less than zero. Partial solution discarded. Function returns.
- Removes 6, new candidate [2,2].
- End of while loop, function returns.
- Removes 2, new candidate [2].
- Adds candidate number 6.
- Calls backtrack with candidate solution: *[2,6]*, and start = 1 (next number = 6).

Step 6

- Current target (-2) is less than zero. Partial solution discarded. Function returns.
- Removes 6, new candidate [2].
- End of while loop, function returns.
- Removes 2, new candidate [].
- Adds candidate number 6.
- Calls backtrack with candidate solution: *[6]*, and start = 1 (next number = 6).

Step 7

- Solution found: **[6]**. Function returns.
- Removes 6, new candidate [].
- End of while loop, function returns.

Solutions: *[[2,2,2], [6]]*

6.2.1 Time and space complexity

The **time complexity** for backtracking algorithms is usually exponential. For this problem, it is: $O(C^K)$, where C is the number of candidates and K is the length of the longest possible combination. Additionally, we can define K as $target/min\{candidates\}$.

For our previous example:

- Candidates = [2,6] (Number of candidates = 2)
- Target = 6

Hence, C would be equal to 2, and K would be equal to 3, as $target/min\{candidates\} = 6/2 = 3$.

As introduced in the previous chapter, the **space complexity** can be calculated using the formula:

> $O(d * s)$ Where d is the depth of recursion, and s is the space per recursive call.

For our problem:

> $O(K * 1) = O(K)$, where K is the length of the longest combination.

6.3 Summary

- Backtracking can be used in problems that have clear constraints to reach a solution and where it is possible to test the validity of partial solutions in order to discard them early.
- Some of the problems that can be resolved using backtracking can also be solved by dynamic programming or greedy algorithms in a more efficient way.
- Usually, backtracking solutions are exponential in time and space.

6.4 References

- [1] Backtracking. http://www.cs.toronto.edu/~fbacchus/Papers/uniform-backtracking.pdf.

7 Lowest Common Ancestor (Binary Trees)

Determine the lowest common ancestor (LCA) [1] of two given nodes in a binary tree [2].

The LCA of nodes p and q is defined as the lowest node in the tree that has both p and q as descendants (with the exception that a node is considered a descendant of itself).

Example 1:

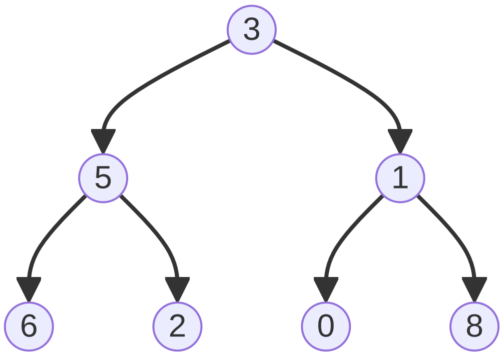

Figure 7.1: Example 1 LCA

> **Input:** root = [3,5,1,6,2,0,8,null,null], p = 5, q = 1
>
> **Output:** 3
>
> **Explanation:** The LCA of nodes 5 and 1 is 3.

7.1 Introducing binary trees

A binary tree [2] is a non-linear data structure used to store data in a hierarchical structure. They are made of nodes, where each node has a *left* pointer, a *right* pointer, and a *data* element. A null node represents the empty tree.

The left and right pointers recursively point to smaller *"subtrees"*.

The formal recursive definition is: a binary tree is either empty (represented by a null pointer) or is made of a single node, where the left and right pointers (recursive definition ahead) each point to a binary tree.

Binary tree example:

```
    3
   /  \
  2    1
 / \   /
4   7  5
```

In the example above, the root node is the top one, whose data element is 3. Its left pointer references node 2, which is itself another binary tree. We can say that subtree is its left child:

```
    2
   /  \
  4    7
```

This left subtree is a binary tree whose root is node 2. This root node also has two subtrees, the left one and the right one. Both of them are root nodes that don't have children.

You can see how a binary tree is a recursive data structure:

> A binary tree is either empty (represented by a null pointer) or is made of a single node, where the left and right pointers (recursive definition ahead) each point to a binary tree.

This is why using recursive algorithms to traverse binary trees is natural and easy to reason about.

This introduction about binary trees, along with the tree traversals we will see in this section, should be enough to solve the exercise. However, for more details about binary trees, read reference [2].

7.2 Binary tree traversals

A traversal is a process that visits all the nodes in the binary tree. Since a tree is a non-linear data structure, there are different types of traversals.

All traversal algorithms have to:

- Process node.
- Process left subtree (recursion).
- Process right subtree (recursion).

Each traversal type does these operations in a different order. The most common ones are:

- Pre-order traversal: node, left, right.
- In-order traversal: left, node, right.
- Post-order traversal: left, right, node.

In an example, consider this tree:

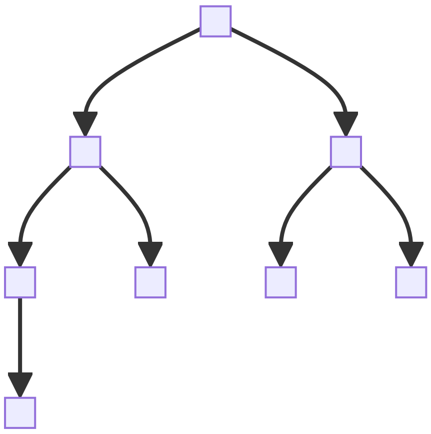

Figure 7.2: Sample tree to be traversed

Pre-order traversal

The order of operations is:

1. Visit the *node* first.
2. Visit the *left* subtree.
3. Visit the *right* subtree.

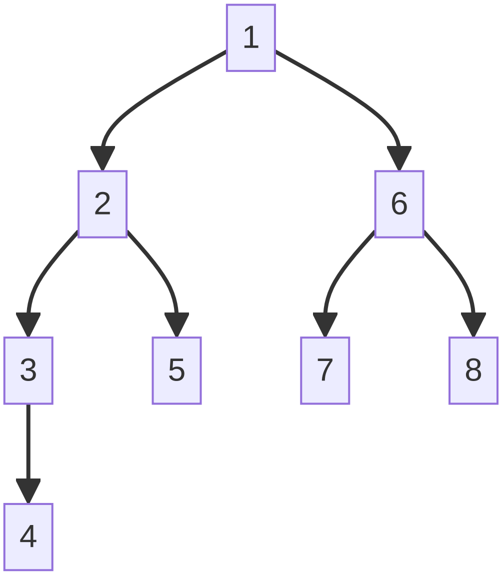

Figure 7.3: Sample tree traversed in order

The algorithm in JavaScript:

```
function preorderTraversal(root) {
    if (root !== null) {
        console.log(root.val);
        preorderTraversal(root.left);
        preorderTraversal(root.right);
    }
};
```

In-order traversal

The order of operations is:

1. Visit the *left* subtree.
2. Visit the *node* first.
3. Visit the *right* subtree.

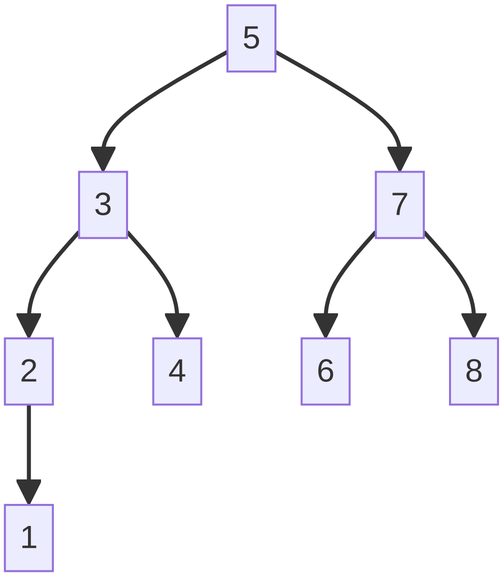

Figure 7.4: Sample tree traversed pre order

The algorithm in JavaScript:

```
function inorderTraversal(root) {
    if (root !== null) {
        inorderTraversal(root.left);
        console.log(root.val);
        inorderTraversal(root.right);
    }
};
```

Post-order traversal

The order of operations is:

1. Visit the *left* subtree.
2. Visit the *right* subtree.
3. Visit the *node* first.

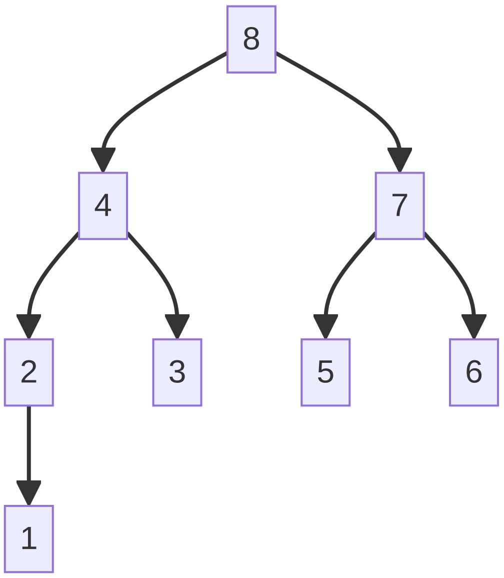

Figure 7.5: Sample tree traversed post order

The algorithm in JavaScript:

```
function postorderTraversal(root) {
    if (root !== null) {
        postorderTraversal(root.left);
        postorderTraversal(root.right);
        console.log(root.val);
    }
};
```

7.3 Solution to Lowest Common Ancestor problem

There are different ways to solve this problem; some are shorter than the one we are going to discuss here. But this solution is very intuitive, even if it requires more code.

We can divide the problem into two subproblems:

1. Find the path from root to q and p.
2. Compare both paths at the same time. We know that their LCA is the node before their paths start diverging.

7.3.1 Algorithm to find the path from root to a child node.

This is the first subproblem we want to solve. Consider this example: find the path from root to *q=3*:

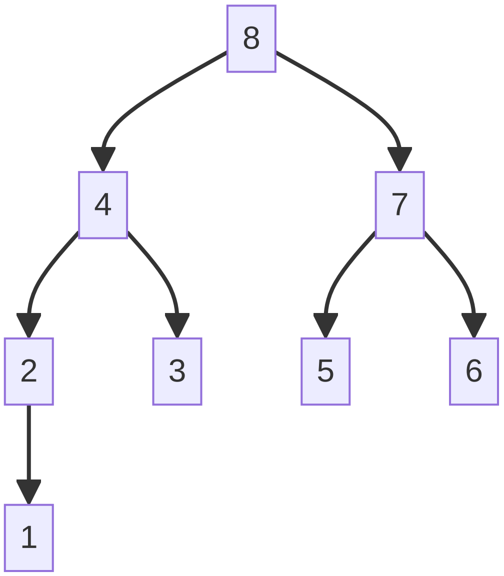

Figure 7.6: Find the path from root to 3

We know the path is `8 -> 4 -> 3`. From the root node, 8, we go to 4, and then we find 3.

We can write an algorithm that recursively traverses the tree until it finds node 3. Also, the algorithm should keep track of the path and then return it.

```
function getPath(root, target) {
    if (root == null) {
        return null;
    }

    if (root.val == target.val) {
       return [root];
    }

    const leftPath = getPath(root.left, target);

    if (leftPath != null) {
        leftPath.push(root);
        return leftPath;
    }

    const rightPath = getPath(root.right, target);

    if (rightPath != null) {
        rightPath.push(root);
        return rightPath;
    }
}
```

This recursive algorithm has two base cases and two recursive calls.

The base cases are:

1. If the current node is null, it means there are no more nodes to explore from the previous node. We return null.
2. If the current node is equal to the node we are looking for, then we don't have to continue searching. We return an array that represents the path.

There are also two recursive calls. Each node can have up to two child nodes. As we don't know what path we should take to find the target node, we have to try both:

1. Call `getPath` trying with the left child. If the result is not null, we have found the target node. Then we add the current node to the path and return it.
2. If the `leftPath` is null, we didn't find the target node. We do the same with the right child.

In the end, if the target node exists, the algorithm will return the path; otherwise, it will return null.

Here is the step-by-step execution for the following tree and target node = 3.

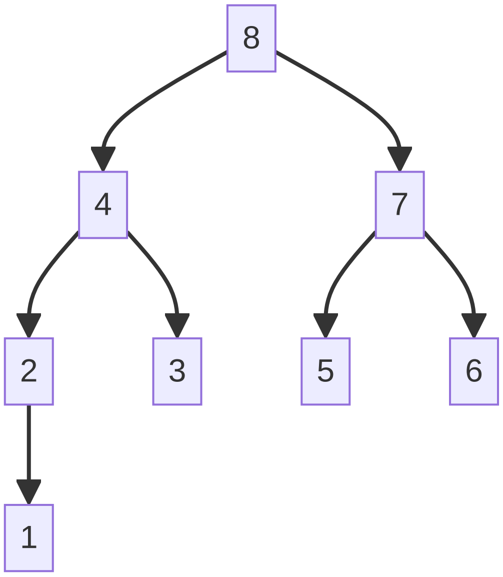

Figure 7.7: Find the path from root to 3

Step 1

```
getPath(Node(8), Node(3))
```

Has it reached a base case? No.

- Current root not null.
- Current root not Node(3).

It tries with left child first.

Step 2

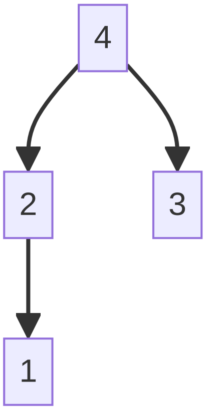

Figure 7.8: Step 2

```
getPath(Node(4), Node(3))
```

Has it reached a base case? No.

- Current root not null.
- Current root not Node(3).

It tries with left child first.

Step 3

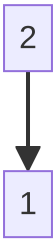

Figure 7.9: Step 2

```
getPath(Node(2), Node(3))
```

Has it reached a base case? No.

- Current root not null.
- Current root not Node(3).

It tries with left child first.

Step 4

The left child is 1, which is a leaf. It's not null, and it's not the target. So the algorithm will try with its children, but all of them are null; then it will return null.

Step 5

We are back to node 4. The left path returned null; thus, the algorithm will try with the right child.

```
getPath(Node(4), Node(3))
```

Has it reached a base case? No.

- Current root not null.
- Current root not Node(3).

Did it find the target node on the left child? No.

Then, try with the right child.

Step 6

Figure 7.10: Step 2

```
getPath(Node(3), Node(3))
```

Has it reached a base case? Yes.

- Current root not null.
- Current root is Node(3).

We found the target. Return `[3]`.

Step 7

We are back to node 4.

- `rightPath = [3]`

It pushes itself and returns:

- returns `[3, 4]`

Step 8

We are back to node 1.

- `leftPath = [3, 4]`

It pushes itself and returns:

- returns `[3, 4, 8]`

The algorithm is finished and it has returned the path to node 3:

- `[3, 4, 8]`

We are using the array as a stack. Its bottom (first element) contains the target node. The element above is its parent node, and so on.

7.3.2 Main algorithm.

We wrote a helper function to get the path from root to a target node. Our main algorithm can use it to get the paths for p and q, and then compare them to find the LCA.

```
function lowestCommonAncestor(root, p, q) {
    const pathToP = getPath(root, p);
    const pathToQ = getPath(root, q);

    let lca = root;

    while (pathToP.length > 0 && pathToQ.length > 0) {
        const pAncestor = pathToP.pop();
        const qAncestor = pathToQ.pop();

        if (pAncestor.val === qAncestor.val) {
            lca = pAncestor;
        } else {
            break;
        }
    }

    return lca;
};
```

These are the steps for the main algorithm:

1. Calculates the path for both p and q.
2. Declares a `lca` variable that points to root. It will reference the LCA for p and q.
3. Iterates over the two stacks that represent the paths.
    1. Gets the elements from the top of the stacks.
    2. If they are the same, then updates the `lca`.
    3. If they are not the same, it breaks the loop.
4. Returns the LCA.

This would be the step-by-step execution for the following tree with p=`3` and q=`1`.

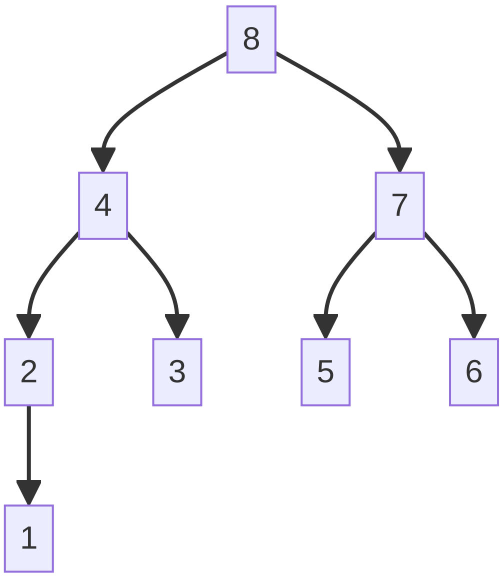

Figure 7.11: LCA where p=3 and q=1

Step 1

We calculate the paths for Node(3) and Node(1):

- `pathToP`: [3, 4, 8].
- `pathToQ`: [1, 2, 4, 8].

Step 2

We declare `lca = 8` and start the loop.

- `pAncestor`: 8
- `qAncestor`: 8
- `lca`: 8

Since both ancestors are the same, we continue after updating `lca`.

Step 3

- `pAncestor`: 4
- `qAncestor`: 4
- `lca`: 4

Both ancestors are the same, we update `lca` and continue.

Step 4

- `pAncestor`: 3
- `qAncestor`: 2
- `lca`: 4

Ancestors are not the same anymore, so we break the loop without updating `lca`. Finally, we return 4 as the LCA.

7.3.3 Time and space complexity

The time complexity is $O(N)$, where N is the number of nodes. The tree is traversed twice, and then we iterate over the paths.

The space complexity is also $O(N)$ as we are storing the paths, and in the worst-case scenario, the path could be composed of all nodes.

7.4 Summary

- We saw different ways to traverse a binary tree.
- There are other ways to solve this problem, but we chose one that is intuitive and easy to reason about.
- We divided the problem into two problems: divide and conquer.
- We traversed the tree to find the path.

7.5 References

- [1] Lowest common ancestor.
  - https://en.wikipedia.org/wiki/Lowest_common_ancestor.
- [2] Binary trees. http://cslibrary.stanford.edu/110/BinaryTrees.pdf.

8 Binary Tree Level Order Traversal (Tree and BFS)

Return the level-order traversal [1] of the values of the nodes in a binary tree, starting from the root. The traversal should go from left to right, level by level.

Example 1:

```
    3
   / \
  9  20
    /  \
   15   7
```

> **Input:** root = [3,9,20,null,null,15,7]
>
> **Output:** [[3],[9,20],[15,7]]

Example 2:

> **Input:** root = [1]
>
> **Output:** [[1]]

Example 3:

> **Input:** root = []
>
> **Output:** []

8.1 Introduction

In the previous chapter, we talked about tree traversals. The exercise for this chapter is about implementing another type of tree traversal. To do that, we will use the breadth-first search algorithm intuitively to become familiar with it. In the next chapter, we will introduce graphs and provide a formal definition of the breadth-first search (**BFS**) and depth-first search (**DFS**) algorithms.

The tree traversals we covered in the last chapter (preorder, inorder, and postorder) are DFS traversals as they start at the root and then explore each branch, from the root to every leaf. However, BFS traversal, or level traversal, begins at the root and then visits all nodes level by level. After starting at the root, it explores all of its children. Then, for each child, it repeats the algorithm.

8.2 Queues

Usually, the data structure used for implementing BFS is a queue. A queue is a linear data structure in which the first element is inserted from one end (called the tail) and elements are removed from the other end (called the head).

That's why queues are first-in, first-out (FIFO) data structures. The first element that is inserted will be the first one to be removed. It works the same as a real-world queue.

```
Head <- Element 1 <- Element 2 <- Tail
```

If we add a new element, Element 3, it will be inserted from the tail:

```
Head <- Element 1 <- Element 2 <- Element 3 <- Tail
```

And the first element to be removed will be Element 1:

```
Head <- Element 2 <- Element 3 <- Tail
```

We can quickly implement a queue in JavaScript by leveraging the native array structure:

```
class Queue {
    constructor() {
        this.array = [];
    }

    enqueue(element) {
        this.array.push(element);
    }

    dequeue() {
        return this.array.shift();
    }

    isEmpty() {
        return this.array.length === 0;
    }
}
```

The `enqueue` method adds new elements at the end of the array, while the `dequeue` method removes the first element from the array. This implementation satisfies the FIFO property, the first element added is the first to be removed.

It is a simple implementation; however, it is not efficient, as in the worst case, its time complexity is $O(N)$, depending on the implementation (The ECMA specification does not specify time complexity [2]).

We can implement a queue without relying on the JavaScript array in order to ensure that all its methods have a time complexity of $O(1)$. We can meet this requirement by using a doubly linked list.

8.3 Doubly linked lists

A doubly linked list is a linear data structure that consists of a series of linked nodes. Each node contains two links (or pointers): one to the next node and another to the previous node in the list. The list has a head node that is at the beginning of the list and a tail node that is at the end of the list.

For example:

```
 Node(null) <--> Node(A) <--> Node(B) <--> Node(null)
   ^                                         ^
   Head                                      Tail
```

Removing a node from the head or tail is $O(1)$ in time complexity. For example, if we want to remove Node(A), we just need to update Head and Node(B) pointers.

- Head's next pointer will point to Node(B).
- Node(B)'s previous pointer will point to Head.

```
Node(null) <-> Node(B) <-> Node(null)
   ^                          ^
   Head                       Tail
```

The same applies for adding an element to the head or tail. If we add Node(C) to the tail:

- Node(B)'s next pointer will point to Node(C).
- Tail's previous pointer will point to Node(C).
- Node(C)'s next pointer will point to Tail.
- Node(C)'s previous pointer will point to Node(B).

Since both adding and removing operations have a constant time complexity, we can leverage a doubly linked list to easily implement a queue:

- Elements will be added to the tail.
- Elements will be removed from the head.

By using this implementation, all queue operations have a constant time complexity.

8.4 Implementing an efficient queue

Here is an implementation for a queue using a doubly linked list:

```
class QueueNode {
    constructor(val) {
        this.val = val;
        this.next = null;
        this.prev = null;
    }
}
```

```
class Queue {
    constructor() {
        this.size = 0;
        this.head = new QueueNode(null);
        this.tail = new QueueNode(null);
        this.head.next = this.tail;
        this.tail.prev = this.head;
    }

    enqueue(val) {
        const node = new QueueNode(val);
        node.next = this.tail;
        node.prev = this.tail.prev;
        node.prev.next = node;
        this.tail.prev = node;
        this.size++;
    }

    dequeue(){
        if (this.size === 0) {
            return null;
        }

        const result = this.head.next.val;
        this.head.next = this.head.next.next;
        this.head.next.prev = this.head;
        this.size--;
        return result;
    }

    isEmpty() {
        return this.size === 0;
    }
}
```

First, we define a `QueueNode` class that has three attributes: `val` that stores any data; `next` that points to the next node; and `prev` that points to the previous node in the queue.

The `Queue` class has three attributes too: `size` that is used to track the number of elements the queue has; `head`, an empty node that represents the head of the queue; and `tail`, another empty node that represents the tail of the queue.

Its three methods have a constant time complexity:

- `enqueue`: creates a new node to be added to the tail. The new node's `next` points to Tail and `prev` points to the previous node. It also updates the previous node's `next` and the Tail's `prev`, so both point to the new node. It increments the `size` attribute.
- `dequeue`: removes the first element from the head. It updates Head's `next` pointer to point to the second node, and updates the second node's `prev` to point to Head. It decrements the `size` attribute.
- `isEmpty`: returns if `size` is equal to 0.

This efficient queue built from scratch will help to solve the level order traversal problem.

8.5 Solution to the Binary tree level order traversal problem

The exercise asks to return an array of arrays. For each level, an array with all nodes has to be created.

See the following example:

```
  3
 / \
9  20
  /  \
 15   7
```

This tree has three levels: 0, 1, and 2. Each of them has different nodes:

- Level 0: 3
- Level 1: 9 and 20.
- Level 2: 15 and 7.

The result we should return is: `[[3],[9, 20],[15, 7]]`. Position 0 in the array corresponds to level 0; that's why it contains another array with all nodes belonging to level 0: `[3]`. The same reasoning applies to the rest of the levels.

Our algorithm needs to keep track of the level for each node in order to add it to the correct position in the array. A pair `(Node, Level)` can be used to achieve this.

We will start by adding the pairs to the queue and then adding them to the array to be returned as a result.

```
/**
 * Definition for a binary tree node.
 * function TreeNode(val, left, right) {
 *     this.val = (val===undefined ? 0 : val)
 *     this.left = (left===undefined ? null : left)
 *     this.right = (right===undefined ? null : right)
 * }
 */
/**
 * @param {TreeNode} root
 * @return {number[][]}
 */
function levelOrder(root) {
    if (!root) {
        return [];
    }

    const result = [];
    const queue = new MyQueue();
    queue.enqueue({node: root, level: 0})

    while (!queue.isEmpty()) {
        const {node, level} = queue.dequeue();

        if (result.length < level + 1) {
            result[level] = [];
        }

        result[level].push(node.val);
        if (node.left) {
            queue.enqueue({node: node.left, level: level + 1});
        }

        if (node.right) {
            queue.enqueue({node: node.right, level: level + 1});
        }

    }

    return result;
}
```

The algorithm starts by creating a queue and adding the pair (root, level 0) to it. Afterward, it loops until the queue is empty and then returns the result.

The loop dequeues the first pair from the queue. It adds the node to the array in the correct position, according to its level. Finally, it adds its right and left children to the queue, if they exist.

Let's visualize the algorithm by going through a step-by-step execution for the following tree:

```
  3
 / \
9   20
   /  \
  15   7
```

Step 1

The algorithm creates a new queue and adds the root node. Then, it starts the loop.

- Queue: `Head <-> (Node 3, 0) <-> Tail`
- Result: `[]`

Step 2

The first iteration is executed. Node 3 is removed from the queue and its value is added to the result. Both left and right children are added to the queue.

- Queue: `Head <-> (Node 9, 1) <-> (Node 20, 1) <-> Tail`
- Result: `[[3]]`

Step 3

Node 9 is removed from the queue and since it does not have any children, nothing is added to the queue.

- Queue: `Head <-> (Node 20, 1) <-> Tail`
- Result: `[[3], [9]]`

Step 4

Node 20 is removed, and its children are added.

- Queue: `Head <-> (Node 15, 2) <-> (Node 7, 2) <-> Tail`
- Result: `[[3], [9, 20]]`

Step 5

Next node is removed, and nothing is added.

- Queue: `Head <-> (Node 7, 2) <-> Tail`
- Result: `[[3], [9, 20], [15]]`

Step 6

The last node is removed. Since the queue is empty, the loop is finished, and the algorithm returns the result.

- Queue: `Head <-> Tail`
- Result: `[[3], [9, 20], [15, 7]]`

8.6 Time and space complexity

The time complexity is $O(N)$, where N is the number of nodes the tree has. This is because:

- We are exploring all nodes.
- Time complexity for queue operations is $O(1)$.
- We are doing two queue operations for each node: `enqueue` and `dequeue`. The total number of queue operations is $2N$.

In conclusion, the total time complexity is equal to:

- Total queue operations * Time complexity of each operation = $2N * O(1) = O(N)$.

The space complexity is the same as the number of nodes the queue has. Since we process the tree level by level, at some point the queue contains all nodes for a given level. Then, the maximum number of nodes is equal to the number of nodes for the level with the most nodes.

Worst case scenario, space complexity is $O(N)$.

8.7 Summary

- There are different types of traversals: DFS and BFS.
- We implemented a doubly linked list.
- On top of that, we implemented an efficient queue.
- We used the queue to solve the problem using BFS.

8.8 References

- [1] Level order traversal. https://www.geeksforgeeks.org/level-order-tree-traversal/.
- [2] Array.prototype.shift. https://tc39.es/ecma262/multipage/indexed-collections.html#sec-array.prototype.shift.

9 Course Schedule (Graphs)

You have a total of `numCourses` courses to take, numbered from `0` to `numCourses - 1`.

The array `prerequisites` indicates that certain courses must be taken before others. For example, `prerequisites[i] = [ai, bi]` means that you must take course `bi` before course `ai`.

Determine if it is possible to complete all courses. Return `true` if it is possible, `false` otherwise.

Example 1:

> **Input:** numCourses = 2, prerequisites = [[1,0]]
>
> **Output:** true
>
> **Explanation:** There are a total of 2 courses to take. To take course 1 you should have finished course 0. So it is possible.

Example 2:

> **Input:** numCourses = 2, prerequisites = [[1,0],[0,1]]
>
> **Output:** false
>
> **Explanation:** There are a total of 2 courses to take. To take course 1 you should have finished course 0, and to take course 0 you should also have finished course 1. So it is impossible.

9.1 Introducing graphs

A graph [1] is a non-linear data structure composed of vertices and edges. Sometimes, vertices are also named nodes. Edges are lines that connect any two vertices in the graph.

It is a way to represent a network of relationships between objects. In a graph, the vertices represent the objects, and the edges represent the connections between them. Graphs can be used to represent many real-world phenomena, such as networks of friends on a social media site or the connections between cities on a map.

Example of graph:

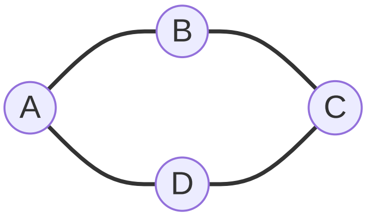

Figure 9.1: Example of graph

At this point, you might have noticed graphs are similar to trees: both have nodes and lines that connect nodes. Actually, trees are a type of graph.

What's the difference? All trees are graphs, but not all graphs are trees. A tree is a graph without cycles, and all nodes must be connected.

A cycle occurs when different nodes are connected circularly. In the graph from the example, A is connected to B, B is connected to C, C is connected to D, and D to A. These four nodes form a cycle.

The other rule specific to trees is that every node is connected to every other node, even if these connections are not direct.

See this graph example:

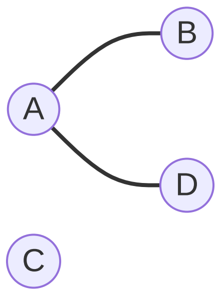

Figure 9.2: Example of not fully connected graph

It cannot be considered a tree as it's not fully connected.

There are 2 types of graphs, directed or undirected.

A directed graph is a type of graph in which the edges have a specific direction. This means that the edges connect one vertex to another in a specific way, and it is possible to follow the edges in a particular order.

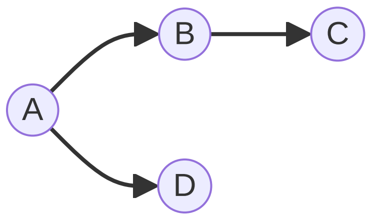

Figure 9.3: Example of directed graph

In contrast, an undirected graph is a type of graph in which the edges do not have a specific direction. This means that the edges can be traversed in either direction, and it does not matter which vertex is the starting point or the ending point.

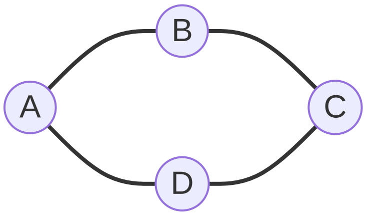

Figure 9.4: Example of undirected graph

In general, directed graphs are used to represent relationships or connections that have a specific direction, such as one-way streets or flows of information, while undirected graphs are used to represent relationships or connections that do not have a specific direction, such as friendships or airline routes.

9.2 How to represent a graph in code

There are several ways to represent a graph in code. One common way to represent a graph is to use an adjacency list, which is a collection of arrays that represents the connections between vertices. In this approach, each vertex in the graph is associated with an array that contains the vertices that are connected to it.

Let's represent the following graph in code using an adjacency list.

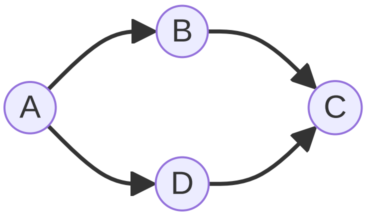

Figure 9.5: Directed graph

We can create a map whose keys are nodes and whose values are arrays that contain the nodes that are connected to it:

Key	Value
A	[B, D]
B	[C]
C	[]
D	[C]

9.3 Graph search

A common graph operation is searching for a particular vertex. It can be done by traversing a graph.

Graph search algorithms [2] explore the vertices and edges in the graph to find a path from a starting vertex to a goal vertex, or to perform other operations on the graph.

As we introduced in the previous chapter, there are two main search algorithms: depth-first search (DPS) and breadth-first search (BFS).

On one hand, Depth-first search is a type of graph search algorithm that explores the graph by following one path as far as possible before backtracking and exploring other paths. It is useful for searching large or complex graphs, as it can often find a solution quickly.

On the other hand, breadth-first search is a type of graph search algorithm that explores the graph by visiting all the vertices at a given depth before moving on to the next depth. It is useful for finding the shortest path between two vertices in a graph.

Graph search algorithms are an important tool for exploring and analyzing graphs and can be used in a variety of applications, such as route planning, computer vision, and natural language processing.

9.3.1 Depth-first search (DFS) example

As we said, DFS explores the graph by following one path as far as possible before exploring other paths.

These are the steps for the DFS algorithm:

1. Select a node and start at it.
2. Add the current vertex to a hash table to mark it as visited. This is done in order to avoid visiting the same vertex more than once.
3. Loop through the adjacent vertices.
4. For each adjacent vertex, if it was already visited, ignore it.
5. If the adjacent vertex was not visited, recursively perform DFS on that vertex.

Let's see a step-by-step example for the following graph:

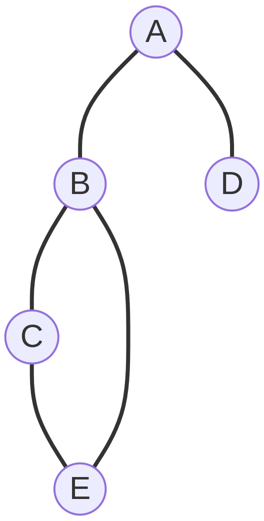

Figure 9.6: DFS traversal

We will start with Node A. Nodes in pink with a thick stroke mean they have been visited.

Step 1

We start with A and mark it as visited:

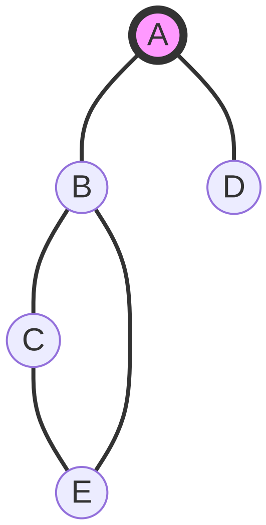

Figure 9.7: DFS traversal

Next, we will iterate through its adjacent nodes: B and D.

Step 2

Now, we perform DFS on each of A's adjacent nodes, for example, B. We mark it as visited.

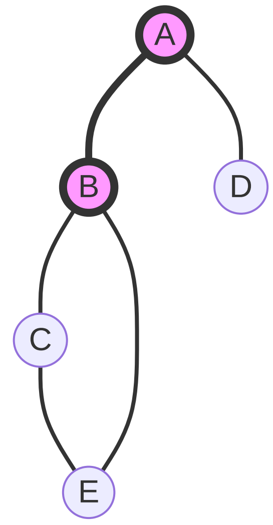

Figure 9.8: DFS traversal

We now loop over B's adjacent nodes: A and C.

Step 3

We ignore A as it was already visited, so we go with C. We call DFS on C. We mark it as visited:

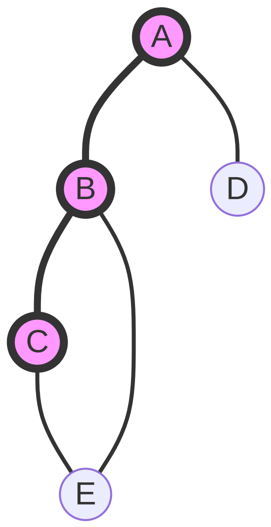

Figure 9.9: DFS traversal

Then, we iterate over C's neighbors: B and E.

Step 4

Since B was already visited, we skip it. We apply DFS on E as it was not visited:

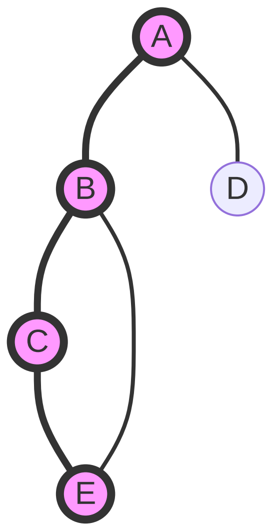

Figure 9.10: DFS traversal

Step 5

Let's iterate over E's adjacent nodes: B. But B was already visited, so we skip it.

We have visited all nodes on this path; now the algorithm will backtrack and explore another path: the rest of A's adjacent nodes: D.

Step 6

We are at node A again; we will explore the rest of the adjacent nodes, in this case, it is only D. We perform DFS on it.

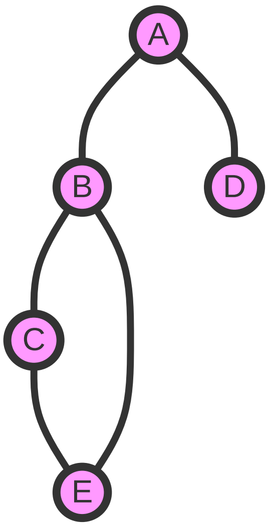

Figure 9.11: DFS traversal

D's only adjacent node is A, which is already visited.

Since there are no more nodes to visit, we have finished the algorithm.

9.3.2 Breath-first search (BFS) example

Let's explore the same graph using BFS. As we said before, BFS explores the graph by visiting all nodes at a given depth before moving on to the next depth.

This is the algorithm for BFS:

1. Select a node and start at it.
2. Add that node to a hash table to mark it as visited.
3. Add that node to a queue.
4. Start a loop that runs until the queue is empty.
5. Within the loop, dequeue the first node from the queue. We call it the "current node".
6. Iterate over all the adjacent nodes of the current node.
7. If the adjacent node is already visited, skip it.

8. If not, mark it as visited and add it to the queue.
9. Repeat the loop (from step 4) until the queue is empty.

Let's see the step-by-step execution:

Step 1

We start with A and mark it as visited:

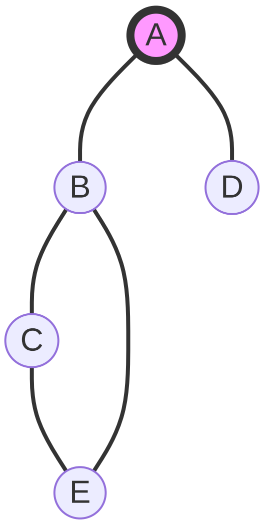

Figure 9.12: BFS traversal

We add it to the queue.

Queue:

```
Head <-> A <-> Tail
```

Then, we start the loop until the queue is empty.

Step 2

We remove the first element from the queue: A, and we proceed to iterate over its adjacent nodes.

Queue:

```
Head <-> A <-> Tail
```

Current node: A

Step 3

We start with B; we mark it as visited and add it to the queue.

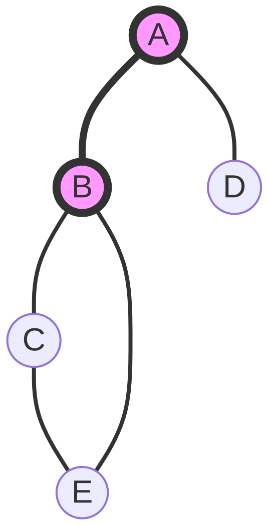

Figure 9.13: BFS traversal

Queue:

```
Head <-> B <-> Tail
```

Current node: A

Note that A is still the current node.

Step 4

We continue with A's next adjacent node: D. We mark it as visited and add it to the queue:

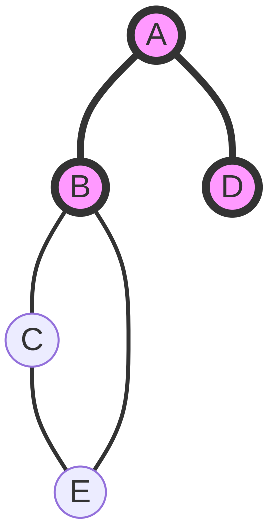

Figure 9.14: BFS traversal

Queue:

```
Head <-> B <-> D <-> Tail
```

Current node: A

Step 5

Now, we have finished iterating over all adjacent nodes of the current node (A). We remove the first one from the queue and make it the current node, in this case, B. Then we continue the loop.

Since B is now the current node, we will iterate over its adjacent nodes.

Queue:

```
Head <-> D <-> Tail
```

Current node: B

Step 6

We proceed to iterate over B's adjacent nodes; let's start with C. We mark C as visited and add it to the queue:

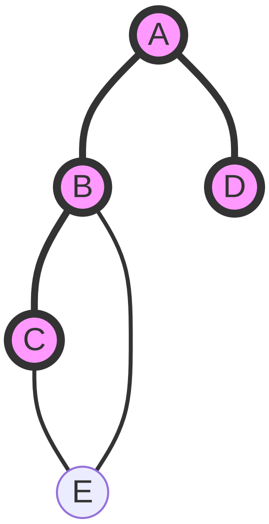

Figure 9.15: BFS traversal

Queue:

```
Head <-> D <-> C <-> Tail
```

Current node: B

Step 7

We continue with the next and last adjacent node of B: E. We mark it as visited and add it to the queue.

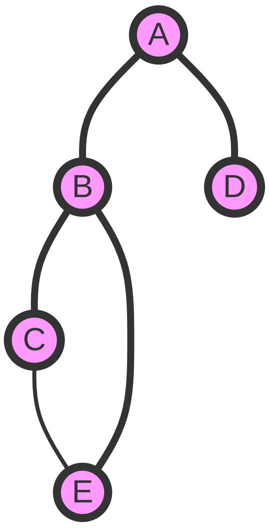

Figure 9.16: BFS traversal

Queue:

```
Head <-> D <-> C <-> E <-> Tail
```

Current node: B

Step 8

We have finished iterating over B's neighbors, as A was already visited. Let's get the next current node from the queue: D.

Queue:

```
Head <-> C <-> E <-> Tail
```

Current node: D

Step 9

Since D does not have any unvisited neighbor (A is already visited), we will dequeue the next current node: C.

We iterate over its adjacent nodes, but all of them are already visited.

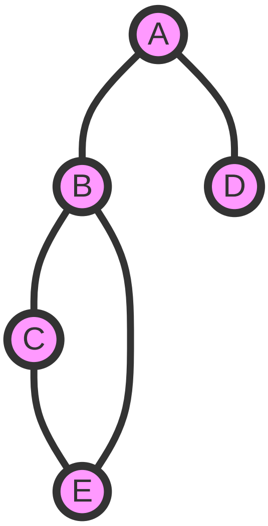

Figure 9.17: BFS traversal

Queue:

```
Head <-> E <-> Tail
```

Current node: C

Step 10

We dequeue the next node: E. All its neighbors are visited:

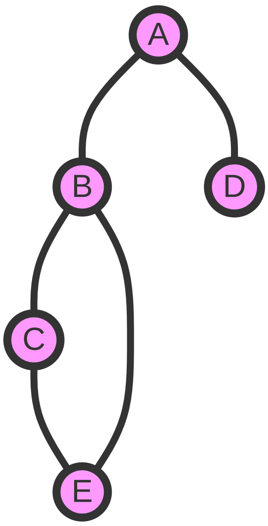

Figure 9.18: BFS traversal

Queue:

```
Head <-> Tail
```

Current node: E.

Since the queue is now empty, we finish the algorithm as we have traversed the graph.

9.4 Solution to the Course Schedule problem

We are given two inputs: the array of courses and the array of prerequisites. The prerequisites indicate what courses need to be taken first as a requisite to take others. There is a better way to model the courses and their relationships: we can use a graph. The vertices are the courses, and the edges are the relationships. Also, notice that this is a DAG (Directed Acyclic Graph). A DAG is a directed graph that does not contain any cycles.

How can we represent the following example using a DAG?

- `numCourses` = 4
- `prerequisites` = [[2,0], [2,1], [3,2]]

We have three courses (0, 1, 2, 3) and three prerequisites. The first one tells us that to take course 2, we have to finish course 1. The second one means we also need to complete course 1 to take course 2. And the last one indicates we have to finish course 2 before starting course 3.

This is a DAG that models our example:

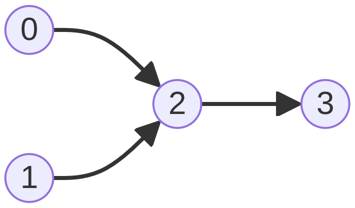

Figure 9.19: DAG to model course schedule problem

The graph shows that courses 0 and 1 does not have any prerequisite and can be taken right away. After these are completed, we can move to course 2, and then to course 3. Notice that we need to take courses 0, 1 and 2 before taking 3.

The first part of the algorithm would be to create a DAG like this one for the given inputs. Then we will apply the topological sort algorithm to see if it is possible to take all courses.

9.4.1 Topological sort

Topological sort [3] is a sorting algorithm for DAGs. It arranges the nodes in a DAG such that for every edge from node A to node B, node A appears before node B in the ordering. In other words, a topological sort is a linear ordering of the nodes in a DAG such that all the directed edges in the graph go from left to right.

For solving the course schedule problem, topological sort can be used to find an order in which the courses can be taken such that all the prerequisites are satisfied.

Since the graph from the example above is small, it is easy to sort it topologically by-hand, but more complex graphs require topological sorting algorithms to process them.

9.4.2 Kahn's algorithm

Kahn's algorithm [4] is a topological sorting algorithm, proposed by Arthur B. Kahn in 1962. It's time complexity is linear, so it is an efficient way of sorting DAGs topologically.

It is important to note that a DAG may have multiple valid topological sortings, so the specific ordering of nodes produced by Kahn's algorithm may vary. However, the algorithm is guaranteed to produce a valid topological sorting if one exists.

Before going to the algorithm steps, let's define what is the ***indegree*** of a node:

> The indegree of a node is the number of incoming edges it has. Namely, it is the number of edges pointing towards that node from other nodes in the graph. A node with no incoming edges is called a source node.

These are the steps of Kahn's algorithm:

1. Choose a source node to be used as the first element of the topological sorting.
2. Remove all outgoing edges from that node.
3. Repeat this with all nodes in the graph.

Usually, this is implemented using a queue. First we add the source nodes. At each step, we remove nodes from the queue and mark it as the next element in the topological sorting, and remove all of its outgoing edges. If a node has no incoming edges after we remove its outgoing edges (it has indegree = 0), we add it to the queue. We repeat the process until the queue is empty. If after finishing the algorithm, all nodes have been ordered, then it is possible to topological sort the graph.

9.4.3 Code for solving the course schedule problem

As we said, we will create the DAG with the courses and their prerequisites, and then use the Kahn's algorithm to topological sort it. If the sorting is possible, we will return true (as we can finish all the courses), otherwise we will return false.

```
/**
 * @param {number} numCourses
 * @param {number[][]} prerequisites
 * @return {boolean}
 */
function canFinish (numCourses, prerequisites) {
    // It will contain the nodes sorted topologically.
    const topologicalSort = [];

    // Graph represented using adjacency list: array of arrays.
    // Position n in the array corresponds to node n.
    const graph = new Array(numCourses).fill().map(() => []);

    // Indegrees of each node.
    // Position n in the array corresponds to node n.
    const indegrees = new Array(numCourses).fill(0);

    // We build the graph and get the indegrees of nodes
    // from the prerequisites array.
    for (let prerequisite of prerequisites) {
        const [course, prereq] = prerequisite;
        indegrees[course]++;
        graph[prereq].push(course);
    }

    // Queue that will contain nodes with indegree = 0
    // In other words, courses that don't have prerequisites,
    // so it can be taken right away.
    const queue = new Queue();

    // We add the source nodes to the queue.
    for (let course = 0; course < numCourses; course++) {
        if (indegrees[course] === 0) {
            queue.enqueue(course);
        }
    }

    // While the queue is not empty, do the following:
    // 1. Remove the first element from the queue and append it
     ↪  to the topological sort list.
    // 2. For each edge going out from the removed element,
     ↪  decrease the indegree of the destination node by 1.
    // If the destination node now has an indegree of 0, add it
     ↪  to the queue.
    while (!queue.isEmpty()) {
        const course = queue.dequeue();
        const adjCourses = graph[course];

        topologicalSort.push(course);

        for (let adjCourse of adjCourses) {
            indegrees[adjCourse]--;
            if (indegrees[adjCourse] == 0) {
                queue.enqueue(adjCourse);
            }
        }
    }

    // If all nodes have been sorted, then it is possible
    // to take all the courses.
    return topologicalSort.length === numCourses;
};
```

Let's see a step-by-step execution for the following inputs:

- `numCourses` = 4
- `prerequisites` = [[2,0], [2,1], [3,2]]

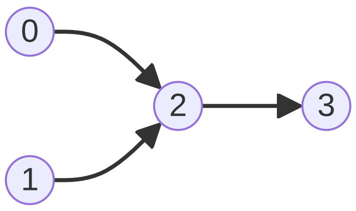

Figure 9.20: DAG to model course schedule problem

Step 1

We declare 4 variables:

- `topologicalSort`: Array containing the nodes sorted topologically.
- `graph`: Graph represented using an adjacency list.
- `indegrees`: The indegree of each node.
- `queue`: Queue that will contain nodes with indegree equals to 0.

Next, we iterate over the list of prerequisites to build the adjacency list and calculate the indegrees.

After, we add the nodes with indegree = 0 to the queue.

The graph variable will remain unchanged during all the execution:

Graph:

```
[
    [2],
    [2],
    [3],
    []
]
```

The rest of variables:

- Topological sort: `[]`.
- Indegrees: `[0, 0, 2, 1]`.
- Queue: `Head <-> 0 <-> 1 <-> Tail`.

Step 2

We start iterating until the queue is empty. We dequeue the first element: 0.

We add course 0 to the topological sort list and then decrease the indegrees for each of its adjacent nodes, in this case course 2. Since course 2 still does not have indegree equal to 0, we don't add it to the queue.

Current status:

- Topological sort: `[0]`.
- Indegrees: `[0, 0, 1, 1]`.
- Queue: `Head <-> 1 <-> Tail`.

Step 3

We remove the next node from the queue: course 1. We decrease the indegrees of course 2, and since it now has indegrees equal to 0, we add it to the queue.

Current status:

- Topological sort: `[0, 1]`.
- Indegrees: `[0, 0, 0, 1]`.
- Queue: `Head <-> 2 <-> Tail`.

Step 4

We repeat the process. The next course to dequeue is course 2, and we decrease the indegrees of its adjacent node, course 3. We also add it to the queue.

Current status:

- Topological sort: `[0, 1, 2]`.
- Indegrees: `[0, 0, 0, 0]`.
- Queue: `Head <-> 3 <-> Tail`.

Step 5

We remove course 3 from the queue and add it to the topological sort list. As the queue is empty, we exit the loop.

We finish the algorithm by returning `true` as the Kahn's algorithm was able to sort all nodes topologically!

9.5 Time and space complexity

The time complexity is $O(V+E)$ where V is the number of nodes and E the number of edges. This is because the algorithm processes each node and each edge once.

Regarding space complexity, it's $O(V)$, as we use a queue to store the nodes. In the worst case scenario, all the nodes in the graph could be added to the queue.

9.6 Summary

- We introduced graphs and how to represent them.

- We talked about DFS and BFS traversals.
- We saw how to use topological sort to solve the problem.
- We used Kahn's algorithm to implement topological sort.

9.7 References

- [1] Graph. https://en.wikipedia.org/wiki/Graph_(discrete_mathematics).
- [2] Graph search algorithms. https://cs.stanford.edu/people/abisee/gs.pdf.
- [3] Topological sorting. https://en.wikipedia.org/wiki/Topological_sorting.
- [4] Kahn's algorithm. https://www.geeksforgeeks.org/topological-sorting-indegree-based-solution/.

10 Minimum Window Substring (Sliding Window)

Find the smallest possible substring in string `s` that contains every character in string `t`, including duplicates; the order does not matter.

Return an empty string, `""`, if no such substring exists. Both strings `s` and `t` have lengths `m` and `n`, respectively.

A substring is a contiguous sequence of characters within a larger string.

Example 1:

> **Input:** s = "ADOBECODEBANC", t = "ABC"
>
> **Output:** "BANC"
>
> **Explanation:** The minimum window substring "BANC" includes 'A', 'B', and 'C' from string t.

10.1 Sliding windows

The aim of this problem is to iterate through the string 's' from start to end and determine the smallest substring within 's' that contains string 't'.

Instead of using brute force to resolve this problem, we can use a more efficient algorithm called **sliding window** [1]. Sliding windows are appropriate for problems that involve sequences like lists or arrays.

A sliding window is a contiguous sequence that is part of the list and slides over it.

For example, given an array `[1, 2, 3, 4, 5]`, a sliding window of size 3 would slide over the array like this:

```
1. [1, 2 ,3]
2.    [2, 3, 4]
3.        [3, 4, 5]
```

A sliding window can both increase and decrease its size:

```
1. [1, 2 ,3]
2.    [2, 3]
3.    [2, 3, 4]
4.    [2, 3, 4, 5]
5.       [3, 4, 5]
```

At any given time, a sliding window contains some elements that satisfy a certain condition.

Whenever the sliding window breaks this rule, the algorithm will try to recover it by shrinking or expanding it, until all elements meet the condition. The process continues until the problem is solved.

10.2 Solution to the Minimum window substring problem

The first solution that might come to our mind is brute force: we could have 2 nested loops that try all possible substrings ($O(N^2)$) and check if the pattern is included in the substring ($O(N)$). This algorithm would have a time complexity of $O(N^3)$.

But we can massively improve the time complexity if we go with a sliding window solution. The idea is to start with a window containing the first element of the array. Then, we can expand it until it meets the condition: the sliding window must contain a substring that includes the pattern.

Once we have expanded it until the condition is met, we could continue by shrinking while the condition is still satisfied. We repeat this algorithm until there is no more string to iterate over.

Let's see how the algorithm works with a simple example, where `s = "javascript"` and `t = acr`.

Step 1

We initialize the sliding window, containing the first element:

```
"[j]avascript"
```

- Sliding window: *“j”*.
- Minimum window substring: *null*.

Step 2

Does the current window meet the condition? No, as it does not contain `t`. Then, we expand it.

```
"[ja]vascript"
```

- Sliding window: *“ja”*.
- Minimum window substring: *null*.

Step 3 to 7

We expand the window by one character at each step. The current window looks like this:

```
"[javascr]ipt"
```

- Sliding window: *“javascr”*.
- Minimum window substring: *“javascr”*.

The window meets the condition as *“javascr”* contains `t` (*acr*). The current minimum window substring is then *”javascr”*, for now.

The next step of the algorithm is to shrink the window while the condition is still satisfied.

Step 8

We shrink the window by one character:

```
"j[avascr]ipt"
```

- Sliding window: *“avascr”*.
- Minimum window substring: *“avascr”*.

Does the current sliding window meet the condition? Yes. Thus, we compare if the current window (*"avascr"*) is shorter than the previous minimum window substring (*"javascr"*). Since it is, we update the current minimum substring.

Step 9 and 10

We continue shrinking the window while the condition is still satisfied:

```
"jav[ascr]ipt"
```

- Sliding window: *"ascr"*.
- Minimum window substring: *"ascr"*.

We update the minimum substring as *"ascr"* contains `t` and is shorter than the previous window.

Step 11

Let's keep shrinking the sliding window to see if we can get a shorter substring:

```
"java[scr]ipt"
```

- Sliding window: *"scr"*.
- Minimum window substring: *"ascr"*.

The current sliding window does not contain `t` (*acr*), so we don't update the minimum substring.

Step 12 to 14

At this point, we need to repeat the algorithm and try to expand the sliding window until it meets the condition again:

```
"java[script]"
```

- Sliding window: *"script"*.
- Minimum window substring: *"ascr"*.

None of the new sliding windows contain `t`, and we have reached the end of `s`. We can finish the algorithm and return *ascr* as the minimum window substring.

And all of this in $O(N)$ time complexity!

10.2.1 Coding the solution

Now that we understand the sliding window algorithm, let's code it. We can divide the solution into two parts: the main algorithm and a `SlidingWindow` object that encapsulates the sliding window functionality.

Main algorithm

```
function getMinimumSlidingSubstring(s, t) {
    const slidingWindow = new SlidingWindow(s, t);

    let minStart = 0;
    let minLength = Number.MAX_VALUE;

    while (slidingWindow.currentEnd < s.length) {
        slidingWindow.expandToRight();

        while (slidingWindow.containsTargetWord()) {
            const currentLength = slidingWindow.getLength();

            if (currentLength < minLength) {
                minStart = slidingWindow.currentStart;
                minLength = currentLength;
            }

            slidingWindow.shrinkFromLeft();
        }
    }

    if (minLength == Number.MAX_VALUE) {
        return "";
    }

    return s.substring(minStart, minStart + minLength);
};
```

First, we initialize a `SlidingWindow` object, which we will discuss next. We also declare two variables to keep track of the minimum window substring:

- `minStart`: It points to the start of the current min substring.
- `minLength`: It has the length of the current min substring.

With these 2 variables, we can know what is the minimum substring in `s` that contains `t`, as you can see at the end of the algorithm:

```
return s.substring(minStart, minStart + minLength);
```

Next, we can see two `while` loops: the first one expands the window until the end of the window has reached the end of `s`; the second one tries to shrink the window while the condition is satisfied. In other words, we expand the window until the condition is met. Then, if the current window length is less than the minimum window length so far, we update it. After this, we try to shrink the window while the condition is still satisfied, and we keep updating the minimum sliding window. When the condition is not met anymore, we try to expand the window again. This is basically the sliding window algorithm we already saw before.

At the end of the algorithm, if `minLength` is still `Number.MAX_VALUE`, this means we have not updated it, so we didn't find any window substring; thus, we return an empty string. Otherwise, we return the minimum sliding window substring.

The second part of the algorithm is the `SlidingWindow` class:

```
class SlidingWindow {
    constructor(baseWord, targetWord) {
        this.currentStart = 0;
        this.currentEnd = 0;
        this.baseWord = baseWord;
        this.targetWord = targetWord;
        this.numberOfCharsToFind = targetWord.length;
        this.countByChar = this._createMap(targetWord);
    }

    expandToRight() {
       const charToAdd = this.baseWord.charAt(this.currentEnd);
        const charBelongsToTargetWord =
         ↪  this._charBelongsToTargetWord(charToAdd);

        if (charBelongsToTargetWord &&
         ↪  this.countByChar[charToAdd] > 0) {
           this.numberOfCharsToFind--;
        }
        if (charBelongsToTargetWord) {
            this.countByChar[charToAdd]--;
        }

        this.currentEnd++;
    }

    shrinkFromLeft() {
        const charToRemove =
         ↪  this.baseWord.charAt(this.currentStart);
        const charBelongsToTargetWord =
         ↪  this._charBelongsToTargetWord(charToRemove);

        if (charBelongsToTargetWord) {
            this.countByChar[charToRemove]++;
        }

        if (charBelongsToTargetWord &&
         ↪  this.countByChar[charToRemove] > 0) {
            this.numberOfCharsToFind++;
        }

        this.currentStart++;
    }

    containsTargetWord() {
        return this.numberOfCharsToFind === 0;
    }

    getLength() {
        return this.currentEnd - this.currentStart;
    }

    _createMap(word) {
        const map = {};
        for (const char of word) {
            if (map[char] == undefined) {
                map[char] = 0;
            }
            map[char]++;
        }
        return map;
    }

    _charBelongsToTargetWord(char) {
        return this.countByChar[char] !== undefined;
    }
}
```

The constructor declares two pointers to track the window position within `s`. From the beginning, the window has `size = 0`. It also stores *s* as `baseWord` and *t* as `targetWord`. The last two variables are for storing the number of characters that `t` has and a hash map containing the number of occurrences for each character of `t`.

This map is used to efficiently identify if the current sliding window meets the condition, namely whether it contains `t`. It keeps track of the characters belonging to `t` that are included in the current window. We will see how this works with a step-by-step example later on.

The `SlidingWindow` object exposes different methods to expand, shrink, get the length, and check if it contains `t`.

10.2.2 Solution step-by-step

Let's see the solution for `s = "richi"` and `t = "hi"`.

Step 1

```
"[]richi"
```

We initialize the `SlidingWindow` object and the variables for tracking the minimum sliding substring.

Delivery Window status:

- `currentStart`: 0.
- `currentEnd`: 0.
- `numberOfCharsToFind`: 2.
- `countByChar`: `[h: 1, i: 1]`.

Minimum sliding substring status:

- `minStart`: 0.
- `minLength`: `Number.MAX_NUMBER`.

Step 2

`"[r]ichi"`

We enter the first `while` loop and expand the window by one character. Since *r* is not part of `t`, we still have two characters to find, and we don't update the `countByChar` map.

Delivery Window status:

- `currentStart`: 0.
- `currentEnd`: 1.
- `numberOfCharsToFind`: 2.
- `countByChar`: `[h: 1, i: 1]`.

Minimum sliding substring status:

- `minStart`: 0.
- `minLength`: `Number.MAX_NUMBER`.

Step 3

`"[ri]chi"`

We didn't enter the second loop because the condition was not met, so we continued with a second iteration on the first loop. Now, the window contains *i*, which is part of `t`. This is why we decrease `numberOfCharsToFind` (we already found one char), and we decrease the count for `i` in the map.

Delivery Window status:

- `currentStart`: 0.
- `currentEnd`: 2.
- `numberOfCharsToFind`: 1.
- `countByChar`: `[h: 1, i: 0]` (`i<= 0` means the current window contains all *i* characters present in `t`).

Minimum sliding substring status:

- `minStart`: 0.
- `minLength`: `Number.MAX_NUMBER`.

Step 4-5

```
"[rich]i"
```

On Step 4, we expanded the window again, and since *c* is not part of `t`, we executed Step 5. On Step 5, we expanded the window, and now it contains h, which is the last character from `t` that we had to find. This means the current window (*"rich"*) is the latest minimum window substring the algorithm observed. We decrease `numberOfCharsToFind` and update `countByChar`. As long as `numberOfCharsToFind` is equal to 0, the current sliding window will contain `t`.

Delivery Window status:

- `currentStart`: 0.
- `currentEnd`: 4.
- `numberOfCharsToFind`: `0` (Current sliding window contains `t`, nothing else to find).
- `countByChar`: `[h: 0, i: 0]`.

Minimum sliding substring status:

- `minStart`: 0.
- `minLength`: 4.

Step 6-7

Steps 6 and 7 represent the second part of the algorithm: shrinking the window to check if we can get a smaller substring that contains `t`. We will reduce the window until the condition is no longer satisfied.

Step 6

```
"r[ich]i"
```

The reduced window still contains `t`, so we update the minimum substring variables:

- `minStart`: 1.
- `minLength`: 3.

Step 7

`"ri[ch]i"`

We keep shrinking the window, but now `t` is not contained anymore. We don't update the minimum substring variables, but we do for the internal `SlidingWindow` ones:

Delivery Window status:

- `currentStart`: 2.
- `currentEnd`: 4.
- `numberOfCharsToFind`: 1 (We increase the count as the window does not contain *i* anymore.).
- `countByChar`: `[h: 0, i: 1]` (We increase the count for *i*).

Minimum sliding substring status:

- `minStart`: 1.
- `minLength`: 3.

The current `countByChar` value indicates the current window does not need to find more *h*, but still needs one *i*. Also, the reader will notice that `numberOfCharsToFind` is equal to 0 when the count for each char on `countByChar` is less than or equal to 0.

Step 8

Since the current window does not contain `t`, we come back to the first part of the algorithm (the first `while` loop) to expand it again to try to find the missing characters.

`"ri[chi]"`

We discovered a new *i* char. We have a sliding window containing `t` again. Is it shorter than our current minimum substring? It's not, since *ich* has the same length as *chi*. Thus, we don't update the variables.

Delivery Window status:

- `currentStart`: 2.
- `currentEnd`: 5.
- `numberOfCharsToFind`: 0 (We found `t` chars again.).
- `countByChar`: `[h: 0, i: 0]`.

Minimum sliding substring status:

- `minStart`: 1.
- `minLength`: 3.

Step 9

Let's now try to shrink the window and see if the condition is satisfied after it:

```
"ric[hi]"
```

Bingo! After reducing the window, the condition is still met, and the substring is shorter than our previous minimum substring!

Delivery Window status:

- `currentStart`: 3.
- `currentEnd`: 5.
- `numberOfCharsToFind`: 0.
- `countByChar`: `[h: 0, i: 0]`.

Minimum sliding substring status:

- `minStart`: 3 (New min start).
- `minLength`: 2 (New min length).

We reached the end of `s`; the algorithm is finished. We return `s`'s substring starting from 3 with a length of 2: `hi`. That's the minimum substring containing `t` characters.

10.2.3 Time and space complexity

The time complexity for this algorithm is $O(S)$, where S is the length of `s`. This is because there are two `while` loops. The first one expands the window, and the second one shrinks it. In the worst-case scenario, we iterate over `s` twice: once for expanding and once for shrinking. This is $O(2 * S)$, so $O(S)$.

Regarding space complexity, the algorithm needs to keep a map to store all `t` characters. Hence, the complexity is $O(T)$, where T is the number of `t` distinct characters.

10.3 Summary

- We introduced the sliding window technique.
- We used a sliding window to efficiently solve the algorithm.
- By resizing the window, we searched for the minimum window substring.
- The sliding window algorithm used a character count map, which contained all of the characters in `t`, to keep track of the characters included in the current window.

10.4 References

- [1] Sliding window. https://www.geeksforgeeks.org/window-sliding-technique/.

11 Merge K Sorted Lists (Heaps)

You are given an array of k linked lists, where each linked list is sorted in ascending order. Your goal is to merge all the linked lists into a single sorted linked list and return it.

Example 1:

> **Input:** lists = [[1,5],[1,4,5],[6,8]]
>
> **Output:** [1,1,4,5,5,6,8]

11.1 Introduction

The first solution that comes to mind is the brute force solution.

We can summarize its steps:

- We initialize an empty array, `result`, to store the final merged list.
- We declare an array, `positions`, where `positions[i]` points to the current element in the `i`-th list.
- We repeat this loop until we have processed all elements from all lists.
  - We get the minimum element among the current elements of the lists. We use `positions` to check the current elements.
  - We append that element to the result.
  - Then, we update the current index for that list in `positions`.
- Finally, we return the result.

This simple solution can solve the problem, but what is its time complexity? It is $O(kn)$, where k is the number of lists and n is the number of elements in the longest list.

This is because, for each element we add to the result, we iterate over the current elements of the k lists. In other words, every time we want to add the next element to the result, we need to make k checks (one per list), to see what is the minimum next element.

Also, regarding space complexity, it is $O(k)$ as we need an array to store the current index for each list.

Can we do better than $O(kn)$ time complexity? Yes, we will see it in the next sections.

11.2 Introducing heaps

A heap [1] is a tree-based data structure that satisfies the heap property: parent nodes have a higher *priority* than any of their children. A Binary Heap is a heap where there are at most 2 children of a node. There are two types of heaps: max heaps and min heaps.

Min heaps

Higher priority means a smaller number. Every parent node is smaller than any of its children.

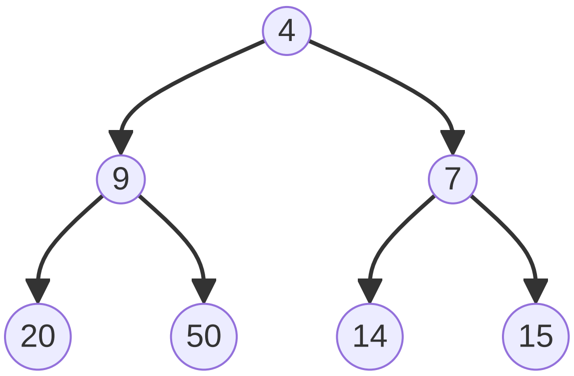

Figure 11.1: Min heap

Max heaps

Higher priority means a greater number. Every parent node is greater than any of its children.

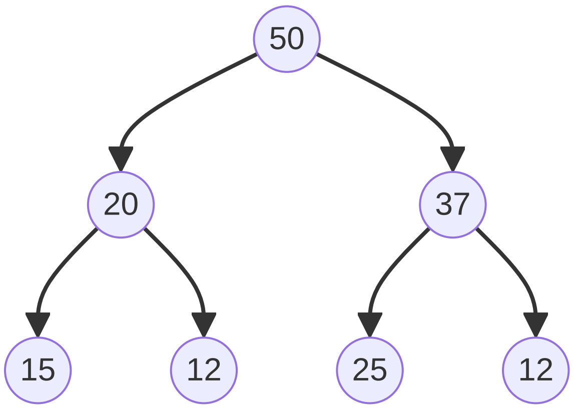

Figure 11.2: Max heap

Why are heaps interesting?

Because they allow you to find the largest/smallest element in the heap in $O(1)$ time (while in arrays these operations usually take $O(N)$ time). You can now see how heaps can help us solve the problem. We can get the minimum element from all sorted lists by using a heap.

Heaps are **completely filled**, except for the bottom level. Therefore, they are complete trees:

> Complete tree: all levels are filled from right to left except the bottom. The bottom level gets filled from left to right.

Height

The **height** of a heap is the number of levels it has (or rows); for example, the following heap has a height of 3:

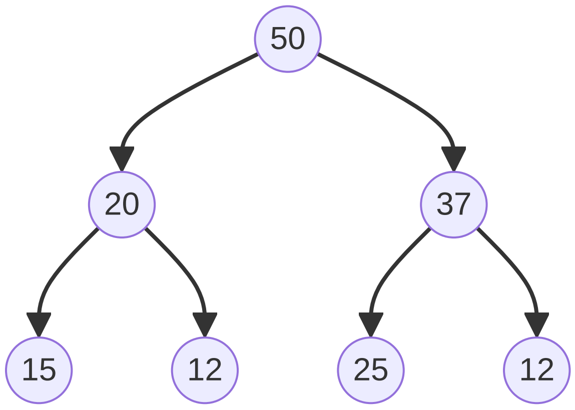

Figure 11.3: Heap with height of 3

A heap of n nodes has a height of about log_2n. This property also applies to balanced binary trees in general.

To understand this, we can think of a heap completely filled with 2 levels (height of 2):

```
  4 // Level 1: 1 node
 / \
5   6 // Level 2: 2 nodes
```

This heap has 3 nodes in total (1 + 2). If we add a third level, how many nodes are we adding? 4 nodes. The new heap will have now 7 nodes in total (1 + 2 + 4). Notice that, after adding a level, we have doubled the number of nodes the heap has (Actually, we are doubling the nodes and adding one).

A heap with 2 levels has 3 nodes, while a heap of 3 levels has 7 nodes (3*2 + 1).

```
    4 // Level 1: 1 node
   / \
  5    6 // Level 2: 2 nodes
 /\   / \
8 10  9 15 // Level 3: 4 nodes.
```

Does the same happen if we add a fourth level? Yes, as the level 4 would add 8 new nodes, so the total number of nodes would be 15 (7*2 + 1).

Every time we add a new level, we are roughly doubling the number of nodes of the heap.

> **Recap from chapter 1:**
>
> As a refresher, logarithm is the inverse of exponents. For example, 2^4 = 2 * 2 * 2 * 2 = 16.
>
> So, $log_2 16$ is the inverse. In other words, how many times do you have to multiply 2 by itself to get a result of 16? The answer is $log_2 16$ or just 4.
>
> $log_2 16$ = 4
>
> Another way to understand $log_2 16$ is: how many times do we need to halve 16 until we get 1?
>
> 16 / 2 / 2 / 2 / 2 = 1
>
> Since we need four 2s, $log_2 16$ = 4.

Do you see the relationship between the number of nodes and the height?

Height is about $log_2 n$ as at each level we double the number of nodes.

Heap operations

The main 2 operations of a heap are inserting and deleting.

Heaps are *weakly* ordered compared to binary search trees because the only order requirement is that descendants cannot be greater or smaller than their ancestors. This is why searching is not an interesting operation for heaps, as we would need to explore all nodes, thus it would take $O(N)$. On the other hand, binary search trees are better for searching because they offer $log(N)$ time complexity, as they are totally ordered.

In a heap, both inserting and deleting operations can be implemented in $log(N)$ time complexity.

Insertion

Let's assume we are talking about a max heap.

These are the high-level steps we should follow when inserting a new value:

1. Create a new node containing the value and insert it at the next rightmost available position at the bottom level. This new node would be the heap's last node.
2. We compare this new node value with its parent.
3. If the new node's value is greater than its parent, then swap both nodes.
4. Repeat the next step until the new node's value is not greater than its parent.

Let's see the step-by-step execution for adding a new node, 3, to the following heap:

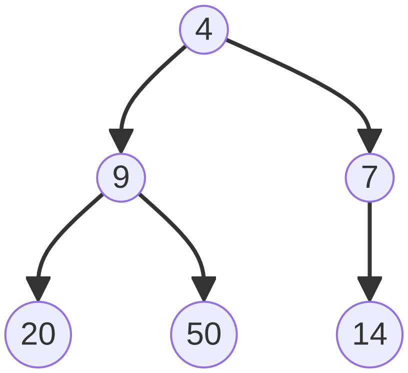

Figure 11.4: Min heap - Insertion

Step 1

We add the new node, 3, as the heap's last node:

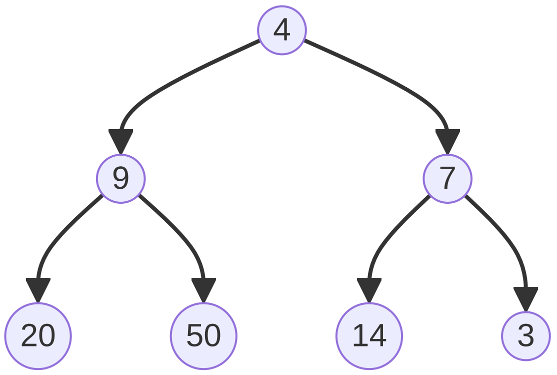

Figure 11.5: Min heap - Insertion

Step 2

We compare 3 with its parent node, 7. Since 3 is less than 7, we swap the nodes.

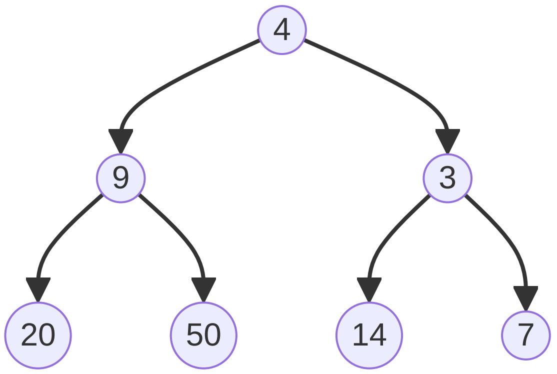

Figure 11.6: Min heap - Insertion

Step 3

We compare 3 with its new parent. Given that 4 is greater than 3, we swap the nodes.

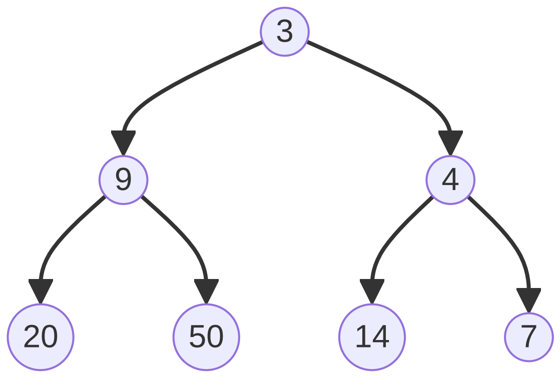

Figure 11.7: Min heap - Insertion

Step 4

Now, 3 is the parent node, so we are done!

Deletion

In heaps, we only delete the root node because it is the node that is most interesting for heap use cases, as it is the smaller/greater one.

These are the high-level steps for deleting the root node of a heap:

1. Move the last node to the root node position.
2. Sift that node down until it is located in its proper position.

Let's see the step-by-step execution for removing the root node of the following heap:

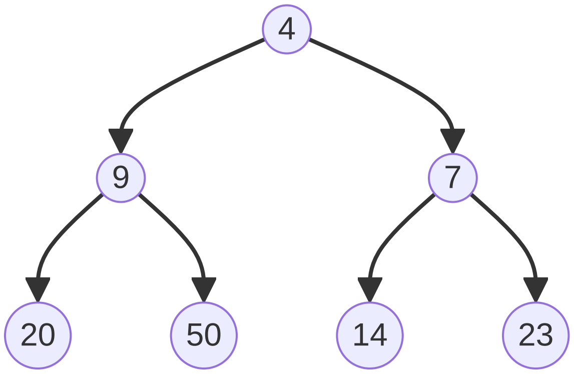

Figure 11.8: Min heap - Deletion

Step 1

We want to remove the root node, 4. We move the last node, 23, to the root's node position:

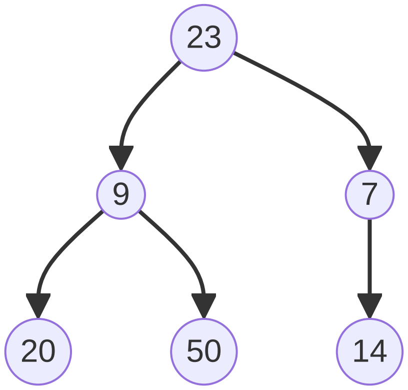

Figure 11.9: Min heap - Deletion

Step 2

The heap condition is not satisfied anymore, as 23 is greater than its children. This is why we have to sift this node down.

We have 2 options to swap the node: we can either swap it with its left child or its right child. We will swap the node with the smallest child, only if the parent is greater than it.

In this case, we swap 23 with 7, since 7 is the smallest child and is less than 23.

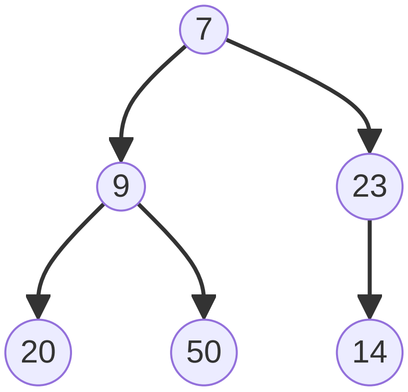

Figure 11.10: Min heap - Deletion

Step 3

The heap condition is still not satisfied because 14 is smaller than 23. We need to swap them:

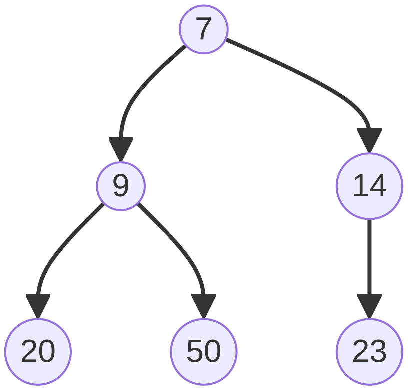

Figure 11.11: Min heap - Deletion

The node 23 does not have more children and the heap condition is satisfied, we are done!

Note that the reason behind swapping the parent node with the smaller of its two children is to ensure the heap condition is not broken.

11.3 Implementing a heap

Since accessing the heap's last node is fundamental for both insertion and deletion operations, heaps are usually implemented using arrays. Using an array to represent a heap has the advantage of being simple and easy to implement, and it allows for efficient node access. It also makes it easy to add and remove nodes from the heap, as the heap property can be maintained by swapping nodes within the array.

The following example shows how to represent a heap (max heap in this case) using an array:

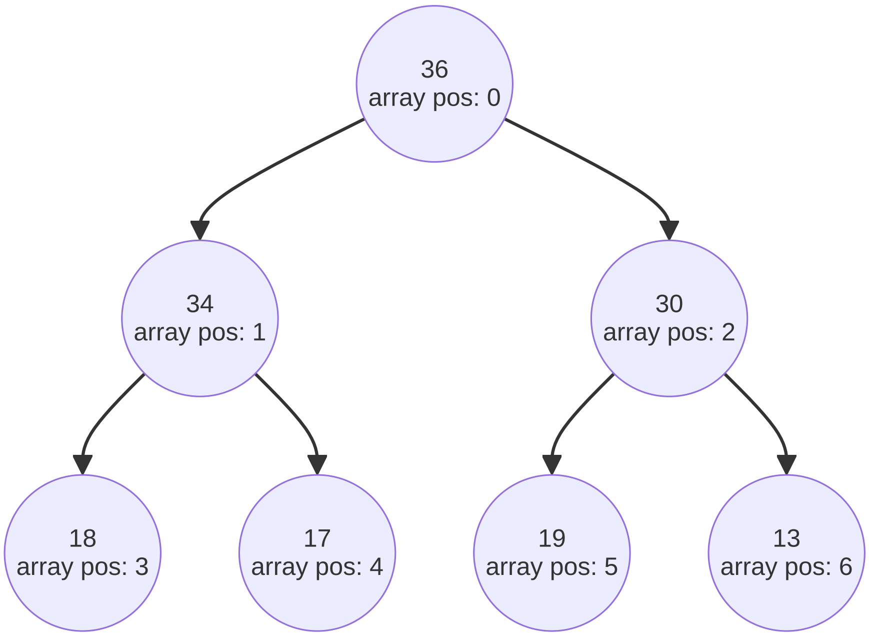

Figure 11.12: Max heap array representation

The array to represent the heap above is:

```
[36, 34, 30, 18, 17, 19, 13]
```

What is the pattern here? The root node is always stored at index 0. Its left child is stored at the next available position (index 1), and the same for its right node (index 2). We repeat this pattern for the following nodes.

The rules are:

- The left child is always stored at (parent index * 2) + 1.
- The right child is always stored at (parent index * 2) + 2.

Then, we know that the root node is stored at index 0, the last node is stored at the last index, and we also know how to access every node's children.

We can write the following functions:

```
getLeftChildIndex(parentIndex) {
    return 2 * parentIndex + 1;
}

getRightChildIndex(parentIndex) {
    return 2 * parentIndex + 2;
}
```

And, if we want to access the parent node, we can use the formula `(childIndex - 1) / 2` (note it uses integer division).

```
getParentIndex(childIndex) {
    return Math.floor((childIndex + 1) / 2) - 1;
}
```

Now, let's implement a full heap [2]:

```
class BinaryHeap {
    constructor(sortFunction) {
        this.items = [];
        this.sortFunction = sortFunction;
    }

    push(item) {
        this.items.push(item);
        this.siftUp(this.length() - 1);
    }

    pop() {
        const [item] = this.items;
        const lastItem = this.items.pop();
        if (this.length() > 0 && lastItem) {
            this.items[0] = lastItem;
            this.siftDown(0);
        }
        return item;
    }

    length() {
        return this.items.length;
    }

    siftUp(index) {
        let currentIndex = index;
        let item = this.getItem(currentIndex);

        while (currentIndex > 0) {
            const parentIndex =
             ↪  this.getParentIndex(currentIndex);
            const parent = this.getItem(parentIndex);

            if (this.sortFunction(parent, item)) {
                break;
            }

            this.swap(parentIndex, currentIndex);
            currentIndex = parentIndex;
        }
    }

    siftDown(index) {
        const item = this.getItem(index);
        let currentIndex = index;

        while (true) {
            let indexToSwap = null;
            const leftChildIndex =
             ↪  this.getLeftChildIndex(currentIndex);
            const rightChildIndex =
             ↪  this.getRightChildIndex(currentIndex);

            if (this.indexExists(leftChildIndex)) {
                const leftChild = this.getItem(leftChildIndex);
                indexToSwap = this.needsToSwap(item, leftChild)
↪   ? leftChildIndex : null;
            }

            if (this.indexExists(rightChildIndex)) {
               const rightChild = this.getItem(rightChildIndex);
                const itemOrLeftChild =
                    indexToSwap === null ? item :
↪   this.getItem(leftChildIndex);
```

```
                indexToSwap = this.needsToSwap(itemOrLeftChild,
↪   rightChild)
                    ? rightChildIndex
                    : indexToSwap;
            }

            if (indexToSwap == null) {
                break;
            }

            this.swap(currentIndex, indexToSwap);
            currentIndex = indexToSwap;
        }
    }

    swap(indexA, indexB) {
        const a = this.getItem(indexA);
        const b = this.getItem(indexB);
        this.items[indexB] = a;
        this.items[indexA] = b;
    }

    needsToSwap(parent, child) {
        return !this.sortFunction(parent, child);
    }

    getItem(index) {
        return this.items[index];
    }

    getParentIndex(childIndex) {
        return Math.floor((childIndex + 1) / 2) - 1;
    }

    getLeftChildIndex(parentIndex) {
        return 2 * parentIndex + 1;
    }

    getRightChildIndex(parentIndex) {
        return 2 * parentIndex + 2;
    }

    indexExists(index) {
        return index < this.length();
    }
}
```

The `BinaryHeap` class has a constructor that takes one argument: `sortFunction`. This function will determine whether it is a min or max heap.

For example, if we want a min heap:

```
const minFunction = (a, b) => a < b;
const minHeap = new BinaryHeap(minFunction);
```

Or, if we need a max heap:

```
const maxFunction = (a, b) => a > b;
const maxHeap = new BinaryHeap(maxFunction);
```

The class also implements the 2 main operations: `push` and `pop`.

The push operation just adds the new item at the end of the array and calls the helper method `siftUp`, which implements the behavior we previously introduced in the previous sections.

The same for `pop`: it deletes the root and puts the last node in the root position. Then it calls `siftDown` to place the value in the correct index.

The rest of the methods are helpers for getting the parent and child indexes, getting an item, checking if two items need to be swapped, etc.

11.4 Solution to the Merge k sorted lists problem

Let's solve the problem using the heap we implemented to achieve a better time complexity than the brute force solution.

The idea is to create a min heap that contains the first element from each of the k sorted lists. Then, we pop the minimum element from the heap and add it to the result list. After removing an element from the heap, we add the next element from the same list to the heap. We repeat these steps until the heap is empty.

This is the code for the sorted list (singly-linked list):

```
function ListNode(val, next) {
    this.val = val === undefined ? 0 : val;
    this.next = next === undefined ? null : next;
 }
```

And here is the solution:

```
/**
 * @param {ListNode[]} lists
 * @return {ListNode}
 */
var mergeKLists = function(lists) {
    if(!lists || lists.length == 0) {
        return null;
    }

    const minHeap = new BinaryHeap((a, b) => a.val < b.val);

    for (let listNode of lists) {
        if (listNode) {
            minHeap.push(listNode);
        }
    }

    const head = new ListNode(0);
    let tail = head;

    while (minHeap.length() > 0) {
        const root = minHeap.pop();

        if (root.next) {
            minHeap.push(root.next);
        }

        tail.next = root;
        tail = root;
    }

    return head.next;
};
```

Let's see the step-by-step execution for the given input:

- List 1: `2 -> 5 -> 7`.
- List 2: `0 -> 3`.
- List 3: `1 -> 4`.

Step 1

We create a new min heap and initialize it with the first element of each list.

Min heap:

```
  0
 / \
2   1
```

Step 2

We initialize two variables, `head` and `tail`, which are used to track the merged linked list we will return as a result.

Then, we start the loop until the heap is empty.

Step 3

We pop the root element from the heap (0). Then, we add the next element (3) from the same list (list 2) to the heap.

We add the popped element to the result list, and update the tail reference.

Min heap:

```
  1
 / \
2   3
```

Result: `0`

Step 4

We repeat the process, we pop the root (1), add the next element to the heap (4, from list 3) and update the result.

Min heap:

```
   2
  / \
3     4
```

Result: `0 -> 1`

Step 5

We loop again.

Min heap:

```
   3
  / \
4     5
```

Result: `0 -> 1 -> 2`

Step 6

Heap still not empty, we keep iterating. In this case, the element popped is 3. Since it does not have a next element, we don't add anything to the heap.

Min heap:

```
   4
  /
5
```

Result: `0 -> 1 -> 2 -> 3`

Step 7

Heap still not empty, we repeat.

Min heap:

```
   5
```

Result: `0 -> 1 -> 2 -> 3 -> 4`

Step 8

the heap still has one element left, let's pop it and add the last element.

Min heap:

```
 7
```

Result: `0 -> 1 -> 2 -> 3 -> 4 -> 5`

Step 9

Last element in the heap and no more elements to merge.

Result: `0 -> 1 -> 2 -> 3 -> 4 -> 5 -> 7`

Since the heap is empty, we return the result list.

11.5 Time and space complexity

The time complexity for this solution is $O(n * log(k))$, where n is the total number of elements in all the lists, and k is the number of lists. This is because we are adding each element to the heap once, and then we also remove them once. So, we are doing $2n$ operation, and each operation has a time complexity of $log(k)$.

The space complexity is $O(k)$, since the heap needs to store at most k elements at any given time.

11.6 Summary

- We introduced a new data structure, heaps.
- We explained its properties and common operations.
- We saw how to implement a heap from scratch using arrays.
- We solved the problem using heaps, improving the time complexity of the brute force solution.

11.7 References

- [1] Heap (data structure). https://en.wikipedia.org/wiki/Heap_(data_structure).

- [2] TypeScript/JavaScript heap implementation.
  - https://github.com/RPallas92/heap-ts.

12 Sudoku Solver (Backtracking)

To practice what we learned about backtracking in the 'Combination Sum' chapter and for fun, we will write a Sudoku solver in this final chapter of the book.

Sudoku [1] is a popular number puzzle that requires players to fill in a grid of squares with numbers such that each row, column, and 3x3 subgrid (also known as a "box") contains all the numbers from 1 to 9. The objective is to fill in the grid such that no number is repeated within a row, column, or box.

Write a program that takes in a partially completed Sudoku grid and returns a complete, valid solution.

The input to the solver is a 2D array representing the grid, with empty cells represented by zeros.

The program does not return anything; it just modifies the input in place by filling the cells with the correct numbers.

Example 1:

Input:

```
[
 [0, 0, 0, 2, 6, 0, 7, 0, 1],
 [6, 8, 0, 0, 7, 0, 0, 9, 0],
 [1, 9, 0, 0, 0, 4, 5, 0, 0],
 [8, 2, 0, 1, 0, 0, 0, 4, 0],
 [0, 0, 4, 6, 0, 2, 9, 0, 0],
 [0, 5, 0, 0, 0, 3, 0, 2, 8],
 [0, 0, 9, 3, 0, 0, 0, 7, 4],
 [0, 4, 0, 0, 5, 0, 0, 3, 6],
 [7, 0, 3, 0, 1, 8, 0, 0, 0]
]
```

Output

```
[
  [4, 3, 5, 2, 6, 9, 7, 8, 1],
  [6, 8, 2, 5, 7, 1, 4, 9, 3],
  [1, 9, 7, 8, 3, 4, 5, 6, 2],
  [8, 2, 6, 1, 9, 5, 3, 4, 7],
  [3, 7, 4, 6, 8, 2, 9, 1, 5],
  [9, 5, 1, 7, 4, 3, 6, 2, 8],
  [5, 1, 9, 3, 2, 6, 8, 7, 4],
  [2, 4, 8, 9, 5, 7, 1, 3, 6],
  [7, 6, 3, 4, 1, 8, 2, 5, 9]
]
```

12.1 Solution to the Sudoku solver problem

There are different approaches to solve a Sudoku, like brute force. We are going to write a backtracking solution to practice what we learned in the 'Combination Sum' chapter.

Our approach will try different numbers in the empty cells of the grid and check if they lead to a valid solution. If a given number leads to an invalid solution, we will backtrack and try a different candidate. We will repeat this algorithm until we find a valid solution.

These are the high-level steps we will use to solve the problem:

1. Identify the next empty cell in the grid.
2. Try filling in the cell with each of the digits from 1 to 9, one at a time.
3. For each number, check if it is a valid choice for the cell by verifying that it does not violate the constraints of the puzzle (i.e., it does not appear in the same row, column, or subgrid as any other occurrences of the same number).
4. If the number is valid, move on to the next empty cell and repeat from number 2.
5. If the number is not valid, backtrack and try a different number for the current cell.
6. Repeat this process until all cells have been filled in with a valid number.

Before we write the main algorithm, let's code a helper function `isValid` that will check whether a given value is valid for a given cell in the puzzle. It will verify the candidate against the values in the same row, column, and 3x3 subgrid, and will return true if the candidate is unique in all of these areas, or false otherwise.

```
function isValid(board, row, column, candidate) {
    for (let index = 0; index < board.length; index++) {
        let currentRowValue = board[row][index];
        let currentColumnValue = board[index][column];
        let squareRowOffset = 3 * Math.floor(row/3);
        let squareColumnOffset = 3 * Math.floor(column/3);
        let currentSquareRow = squareRowOffset +
         ↪  Math.floor(index/3);
       let currentSquareColumn = squareColumnOffset + index % 3;
        let currentSquareValue =
         ↪  board[currentSquareRow][currentSquareColumn];

        if (currentRowValue == candidate
            || currentColumnValue == candidate
            || currentSquareValue == candidate)
        {
            return false;
        }
    }

    return true;
}
```

The function receives four arguments:

1. `board`: the Sudoku board.
2. `row`: the row index of the cell being checked.
3. `column`: the column index of the cell being checked.
4. `candidate`: the candidate number being considered for the cell.

The function consists of a loop that iterates from 0 to the board length - 1 (8). Each of the steps will check if the candidate already exists in the row, column, or subgrid. Note that rows, columns, and subgrids contain 9 cells; that's why we iterate from 0 to 8.

The idea is to check the 3 areas using a single loop. The alternative is to write 3 loops:

one for checking the row cells, another for the column cells, and the last one for the subgrid cells.

Obtaining the current row and column values is quite straightforward. We simply fix the row or column and use the current index.

Getting the current subgrid cell is a little bit trickier. We will understand it better after seeing a step-by-step execution. For now, let's explain what each variable is used for:

- `squareRowOffset`: This points to the first row of the correct subgrid. There are three rows of subgrids. The first one starts at row 0, the second one at row 3, and the last one at row 6.
- `squareColumnOffset`: Same as the previous variable, but for columns.

The combination of these 2 variables allows us to identify the first cell of the correct subgrid. Remember that there are 9 subgrids. Also, these 2 variables will always remain the same during the entire loop.

Now that we know the correct subgrid (by knowing its first cell), we need 2 more variables to explore all its cells:

- `currentSquareRow`: points to the current cell's row in the subgrid.
- `currentSquareColumn`: points to the current cell's column in the subgrid.

If we know the current row and column, we also know the current subgrid cell.

Let's see a step-by-step execution for the `isValid` function, given the following parameters:

- `row`: 3
- `column`: 6
- `candidate`: 6

And for the following board:

```
[
  [0, 0, 0, 2, 6, 0, 7, 0, 1],
  [6, 8, 0, 0, 7, 0, 0, 9, 0],
  [1, 9, 0, 0, 0, 4, 5, 0, 0],
  [8, 2, 0, 1, 0, 0, 0, 4, 0],
  [0, 0, 4, 6, 0, 2, 9, 0, 0],
  [0, 5, 0, 0, 0, 3, 0, 2, 8],
  [0, 0, 9, 3, 0, 0, 0, 7, 4],
  [0, 4, 0, 0, 5, 0, 0, 3, 6],
  [7, 0, 3, 0, 1, 8, 0, 0, 0]
]
```

Step 1

We start the loop and these are the current values:

- `index`: 0
- `currentRowValue`: 8
- `currentColumnValue`: 7
- `squareRowOffset`: 3 (target subgrid's first row is 3)
- `squareColumnOffset`: 6 (target subgrid's first column is 6)
- `currentSquareRow`: 3
- `currentSquareColumn`: 6
- `currentSquareValue`: 0

We continue as we haven't found any '6'.

Step 2

- `index`: 1
- `currentRowValue`: 2
- `currentColumnValue`: 0
- `squareRowOffset`: 3
- `squareColumnOffset`: 6
- `currentSquareRow`: 3
- `currentSquareColumn`: 7
- `currentSquareValue`: 4

No '6', we continue.

Step 3

- `index`: 2
- `currentRowValue`: 0
- `currentColumnValue`: 5
- `squareRowOffset`: 3
- `squareColumnOffset`: 6

- `currentSquareRow`: 3
- `currentSquareColumn`: 8
- `currentSquareValue`: 0

No ‘6’, we continue.

Step 4

- `index`: 3
- `currentRowValue`: 1
- `currentColumnValue`: 0
- `squareRowOffset`: 3
- `squareColumnOffset`: 6
- `currentSquareRow`: 4
- `currentSquareColumn`: 6
- `currentSquareValue`: 9

Notice how we are exploring all subgrid cells by calculating the subgrid’s row and column from the current index.

Let’s skip until the last step, 9.

Step 9

- `index`: 8
- `currentRowValue`: 0
- `currentColumnValue`: 0
- `squareRowOffset`: 3
- `squareColumnOffset`: 6
- `currentSquareRow`: 5
- `currentSquareColumn`: 8
- `currentSquareValue`: 8

We have finished the loop, and we didn’t find any other ‘6’; thus, we return true. We know ‘6’ is a valid candidate for the current board.

Now, we can write the main backtracking algorithm to solve Sudoku puzzles:

```
/**
 * @param {number[][]} board
 * @return {void} Do not return anything, modify board in-place
 ↪  instead.
 */
function solveSudoku(board) {
    if (board == null || board.length == 0) {
        return;
    }

    solve(board);
};

function solve(board) {
    const candidates = [1, 2, 3, 4, 5, 6, 7, 8, 9];

    for (let row = 0; row < board.length; row++) {
        for (let column = 0; column < board[0].length; column++)
         ↪  {
            if (board[row][column] !== 0) {
                continue;
            }

            for (let candidate of candidates) {
                if (isValid(board, row, column, candidate)) {
                    board[row][column] = candidate;

                    if (solve(board)) {
                        return true;
                    }

                    board[row][column] = 0;
                }
            }

            return false;
        }
    }

    return true;
}
```

The `solveSudoku` function is the main entry point. It takes the Sudoku board and checks whether it is empty or null. If it is not, it calls the `solve` function to find a valid solution.

The `solve` function implements the backtracking algorithm. It has 2 loops used to iterate over the cells of the board. If the current cell is not empty, we skip it, as we don't need to find a value. But if the cell is empty, we try all possible values (from 1 to 9).

For each value or candidate, we first check if it is valid by calling our `isValid` helper function. If it is valid, we fill the current cell with that candidate, and we recursively call `solve` passing the updated board in order to try to find a solution for the rest of the board. If the recursive call returns true, then we also return true. In case no valid value is found for the current cell, the function backtracks by setting the cell back to 0.

If the function returns true after finishing the loops, it means a valid solution has been found for the Sudoku.

12.2 Time and space complexity

In the "House robber" chapter, we introduced a formula to calculate the time complexity for recursive algorithms:

> $O(b^d$ Where b is the branching factor, and d is the depth of recursion.

Remember that the branching factor is the average number of possibilities at each step (number of candidates), and the depth of recursion is the depth of the search tree (number of empty cells in the board).

We can conclude the time complexity is $O(9^m)$, where m is the number of cells that are empty.

Remember the formula from the "House robber" chapter for calculating the space complexity:

> $O(d * s)$ Where d is the depth of recursion, and s is the space per recursive call.

Then, the space complexity for our solution is $O(m)$.

12.3 Summary

- We wrote a Sudoku solver using backtracking.
- We practiced what we learned about backtracking in the 'Combination Sum' chapter.
- We used the formulas presented in the "House robber" chapter to calculate both time and space complexities.

12.4 References

[1] Sudoku. https://en.wikipedia.org/wiki/Sudoku.

13 Afterword

As the author of "Essential algorithms for the coding interview", I am grateful to have had the opportunity to share how to solve some of the paradigmatic problems that are, in my opinion, important for preparing code interviews.

I tried to create a short but straight to the point book, where the reader can understand all solutions easily. I hope you had some "aha" moments when we introduced concepts and data structures, or when we explained how to calculate time and space complexities.

Whether you are just starting out in the world of software development or are an experienced programmer looking to refresh your skills, I hope that the book has provided you with the knowledge and tools you need to start preparing for coding interviews.

As a personal recommendation, I would like to suggest one more extensive book about algorithms, *A common sense guide DS and algorithms.*, that can be read after this book. Also, to practice for the coding interview, I recommend this 12-weeks study plan https://www.techinterviewhandbook.org/best-practice-questions/, that includes many coding questions grouped by week.

Happy coding!

Ricardo.