

Pass argument by reference vs by value

Let's consider an example where you have three arrays, and you pass one by value, one by reference, and one by pointer to a function:

```
cpp
Copy code

#include <vector>
using namespace std;

void function(vector<int> x, vector<int>& y, int* z) {
    // Do something with x, y and z
}

int main() {
    vector<int> a = {1, 2, 3, 4, 5};
    vector<int> b = {6, 7, 8, 9, 10};
    int c[5] = {11, 12, 13, 14, 15};
    function(a, b, c);
    return 0;
}
```

```
assembly
Copy code

.section .data
a:
    .long 1, 2, 3, 4, 5    ; Define integer vector a
b:
    .long 6, 7, 8, 9, 10   ; Define integer vector b
c:
    .long 11, 12, 13, 14, 15 ; Define integer array c

.section .text
.extern function
.globl main

main:
    leal c, %eax           ; Load address of c into eax register
    pushl %eax             ; Push address of c to stack (pass pointer)


    leal b, %eax           ; Load address of b into eax register
    pushl %eax             ; Push address of b to stack (pass reference)

    movl a, %eax           ; Copy a to eax (pass by value, this is very si
    pushl %eax             ; Push copy of a to stack (pass by value)

    call function          ; Call the function
```

In C++, when you pass an array to a function, you're actually passing a pointer to the array's first element.

1. **Array of Buckets (or Slots):** At the core of the hash table, there is an array. Each element in this array is called a bucket (or slot). The size of this array usually increases as more elements are added to the hash table to reduce the likelihood of collisions and ensure constant-time average complexity for retrieval.
2. **Hash Function:** This is a function that takes a key as input and returns an integer, which is then used as the index to place the corresponding value in the array of buckets. This function is designed to distribute keys evenly across the array to minimize collisions (where two keys end up with the same index).
3. **Handling Collisions:** If two keys hash to the same index (a collision), we need a strategy to handle this. The two most common methods are separate chaining and open addressing.
 - **Separate Chaining:** Each bucket holds a linked list (or another kind of data structure). All the values that hash to the same bucket will be stored in this list. A disadvantage of this method is that if many keys hash to the same bucket, retrieving a value can devolve to a linear search through the list.
 - **Open Addressing:** Instead of storing multiple values at each index, open addressing proactively searches for another free slot in the array when a collision occurs. There are several methods of open addressing, such as linear probing (look for the next free slot in a linear fashion), quadratic probing (look for slots based on a quadratic function of the distance from the original hash), or double hashing (use a second hash function to determine how to probe for free slots).
4. **Rehashing:** As the hash table grows or shrinks, it may be necessary to adjust the size of the underlying array to ensure good performance. This process involves creating a new, (or smaller) array, and re-inserting all the existing keys/values into it. This is an exper...

 Regenerate

```
unordered_map<string, int> map; // Declare a hashmap
map["a"] = 10; map["b"] = 20; map["c"] = 30; // Insertions

if(map.count("a")) cout << "Key 'a' exists!" << endl; // Check if Key Exists

map.erase("b"); // Deletion

for(auto& pair : map) cout << pair.first << ": " << pair.second << endl; // Iterate
```

Vector

`std::vector` is a sequence container and also known as dynamic arrays or array lists.

Internally, a `std::vector` in C++ is implemented using a dynamically allocated array. It keeps track of its size (the number of elements currently in the vector) and its capacity (the current size of the allocated storage space in the array). This capacity grows as needed to accommodate adding new elements to the vector, and is typically implemented such that it grows in a way that amortizes the cost of resizing the array over a number of insertions.

Yes, you're correct. There is a difference between working with arrays and vectors in C++, especially when it comes to pointers.

In C++, arrays are a lower-level data structure compared to vectors. An array name is essentially a pointer to the first element of the array. So when you do `int* ptr = nums;` where `nums` is an array, you're creating a pointer that points to the first element of the array.

On the other hand, a `std::vector` is a more advanced container class that internally manages a dynamically allocated array and provides various high-level operations (like automatic resizing, etc.). When you create a pointer to a vector, like `vector<int> *ptr = &nums;`, the pointer `ptr` is pointing to the vector object `nums`, not to the internal array used by the vector.

If you want to create a pointer that points to the first element of the vector, similar to an array, you can use the `data()` member function of the vector, like so: `int* ptr = nums.data();`. This will give you a pointer to the first element of the vector's internally managed array, similar to how array pointers work.

When initializing a `std::vector` in C++, you can pass several different types of arguments to the constructor. Here are a few options: an example:

1. Size and fill value: `std::vector<int> v(n, value);` creates a vector of size `n` where all elements are initialized to `value`.
2. Size only: `std::vector<int> v(n);` creates a vector of size `n`. The initial value of the elements will be zero for fundamental types like `int`, `float`, etc.
3. Another vector (or a part of it): `std::vector<int> v(other_vector.begin(), other_vector.end());` creates a new vector that is a copy of `other_vector`.
4. Initializer list: `std::vector<int> v = {1, 2, 3, 4, 5};` creates a new vector with the specified elements.

For a 2D vector, you can use similar syntax:

1. `std::vector<std::vector<int>> matrix(m, std::vector<int>(n, value));` creates an `m` by `n` matrix where all elements are `value`.
2. `std::vector<std::vector<int>> matrix(m, std::vector<int>(n));` creates an `m` by `n` matrix where all elements are zero.

```
vector<vector<int>> matrix;

// Populate the matrix with some example data
// Here we're creating a 3x3 matrix, but you can change the
for (int i = 0; i < 3; ++i) {
    vector<int> row; // Create a row without an initial size
    for (int j = 0; j < 3; ++j) {
        row.push_back(i * j); // Add elements to the row
    }
    matrix.push_back(row); // Add the row to the matrix
}

// Print the matrix
for (const auto& row : matrix) {
    for (int val : row) {
        cout << val << " ";
    }
    cout << endl;
}
```

Range based for loop

```
for (const auto &val : *maxVec) {  
    std::cout << val << " ";  
}
```

This is a range-based for loop in C++. The range-based for loop is a feature added in C++11 to iterate over containers.

In the expression `for (const auto &val : *maxVec)`, the loop will iterate over each element in the container that `*maxVec` points to. `maxVec` is a pointer to a `std::vector<int>`, so `*maxVec` is the actual `std::vector<int>` it points to.

Here's a breakdown of each part of the for loop:

- `const`: This keyword means that the loop will not modify the elements of the vector.
- `auto`: This keyword tells the compiler to automatically deduce the type of the variable. In this case, it's deduced to be `int`, because the vector contains integers.
- `&val`: This means that `val` is a reference to the element of the vector. This allows the loop to access the actual element in the vector, not a copy. Because we've used `const`, we can't modify these elements.
- `*maxVec`: This specifies the container that the loop will iterate over. The `*` operator dereferences the pointer `maxVec`, getting the actual `std::vector<int>` it points to.

So, each time through the loop, `val` is a reference to the next integer in the vector. The loop continues until it has iterated over every element in the vector.

Reg

If you remove the `&` and use just `val` in your range-based for loop, it will still work, but there is a difference. Without the `&`, each `val` is a copy of the element in the vector, not a reference to it. This means that any changes you make to `val` inside the loop won't affect the original vector. However, for simply reading the vector values as in your code, there's no difference in output. If the elements of the vector are large (like large structures or classes) and/or the vector is large, it is more efficient to use references to avoid unnecessary copying.

When iterating through a container, you can choose to iterate by value or by reference.

However, in a single loop, you have to choose one way to iterate. The choice between value and reference depends on what you want to do with the elements.

Here's a brief summary of when you might use each:

- **Iterate by Value:** When you want to work with a copy of each element and don't want to modify the original elements in the container.

```
cpp Copy code  
  
for (auto value : container) {  
    // 'value' is a copy of the element  
    // Modifications to 'value' won't affect the original elements  
}
```

- **Iterate by Reference:** When you want to work directly with the original elements and might want to modify them.

```
cpp Copy code  
  
for (auto& reference : container) {  
    // 'reference' refers to the original element  
    // Modifications to 'reference' will affect the original elements  
}
```

Regenerate

The functions `std::lower_bound` and `std::upper_bound` used in the code are in fact implementations of binary search. Both functions operate on a sorted range of elements and perform a binary search in that range.

Here is what they do:

- `std::lower_bound(first, last, val)`: Returns an iterator pointing to the first element in the range `[first, last)` that is not less than (i.e., greater or equal to) `val`. If all elements in the range compare less than `val`, the function returns `last`. If the function can find more than one element matching `val`, it returns the position of the first match found in the sorted range. This function utilizes binary search.
- `std::upper_bound(first, last, val)`: Returns an iterator pointing to the first element in the range `[first, last)` that compares greater than `val`. If no such element is found (i.e., all elements compare less or equal), the function returns `last`. If the function can find more than one element not matching `val`, it returns the position of the first such element found in the sorted range. This function also utilizes binary search.

In the provided code, `std::lower_bound` is used to find the first occurrence (if any) of `nums1[i]` in `nums2`, and `std::upper_bound` is used to find the first element in `nums2` that is greater than `nums1[i]`. The difference between the two iterators (`upper - lower`) is therefore the number of occurrences of `nums1[i]` in `nums2`.

Repe

Lower upper

```
int main() {
    std::vector<int> v = {1, 2, 4, 4, 5, 6};
    int target = 4;

    auto it = std::lower_bound(v.begin(), v.end(), target);

    if (it != v.end()) {
        std::cout << "Lower bound of " << target << " is at index " << std::
    } else {
        std::cout << "No elements are not less than " << target;
    }
}
```

- **Lower Bound (`right`)**: This is the last value that was smaller than the target. Since we set `right = mid - 1` when `vec[mid] > value`, `right` will end up pointing to the largest element less than the target. If the target is less than the first element, `right` will be -1.
- **Upper Bound (`left`)**: This is the next value that is greater than the target. Since we set `left = mid + 1` when `vec[mid] < value`, `left` will end up pointing to the smallest element greater than the target. If the target is greater than the last element, `left` will be equal to the size of the vector.

Binary search

```
std::pair<int, int> find_bounds(const std::vector<int>& vec, int value) {  
    int left = 0;  
    int right = vec.size() - 1;  
    int mid;  
  
    // Binary search loop  
    while (left <= right) {  
        mid = left + (right - left) / 2;  
  
        if (vec[mid] == value) {  
            return {mid, mid}; // Value found  
        } else if (vec[mid] < value) {  
            left = mid + 1; // Narrow search to right half  
        } else {  
            right = mid - 1; // Narrow search to left half  
        }  
    }  
  
    // When value is not found  
    int lower_bound = right;  
    int upper_bound = left;  
  
    return {lower_bound, upper_bound};  
}
```

Assigning just `mid` to `right` or `left` would be incorrect in the context of a standard binary search for a target value within a sorted array. Here's why:

- If you set `right = mid` when `vec[mid] > value`, you are not excluding the midpoint itself from the search range, even though you know that the target value cannot be at the midpoint (because `vec[mid] > value`). This could lead to an infinite loop if the target value is not in the array.

Uniform Initialization

Certainly! Uniform initialization introduced in C++11 allows you to use braces `{}` to initialize objects in a consistent way. Here are some examples using different types of objects:

1. Basic Data Types:

```
cpp  Copy code

int a{5};
double b{3.14};
```

2. Arrays:

```
cpp  Copy code


int arr[]{1, 2, 3, 4, 5};
```

3. Vectors:

```
cpp  Copy code

#include <vector>
std::vector<int> v{1, 2, 3};
```


4. Classes with Constructors:

```
cpp  Copy code

class MyClass {
public:
    int a;
    double b;
    MyClass(int x, double y) : a(x), b(y) {}
};

MyClass obj{5, 3.14};
```

5. std::pair and std::tuple:

```
cpp  Copy code

#include <utility>
std::pair<int, double> p{5, 3.14};

#include <tuple>
std::tuple<int, char, double> t{5, 'c', 2.71};
```

Uniform initialization helps in making code more readable and consistent across different parts of the program.

LinkedList

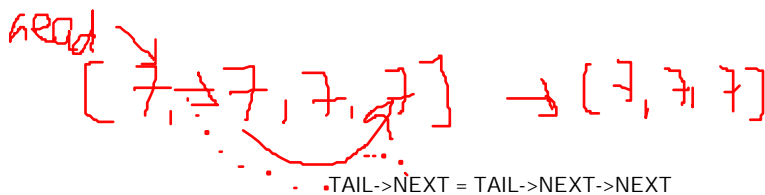
removeElements

1)

```
ListNode *dummy = new ListNode(0, head);
// tail->next = head;
ListNode *tail = dummy;
return dummy->next;
```

VS

```
ListNode *tail = new ListNode(0, head); return head
```



EDGE CASE : REMOVING FIRST ELEMENT

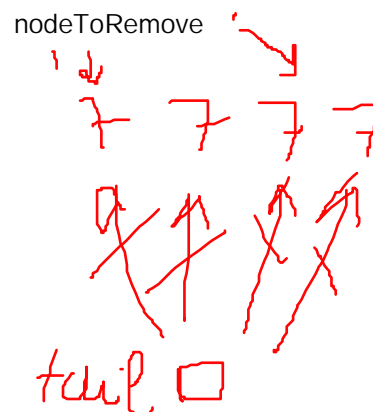
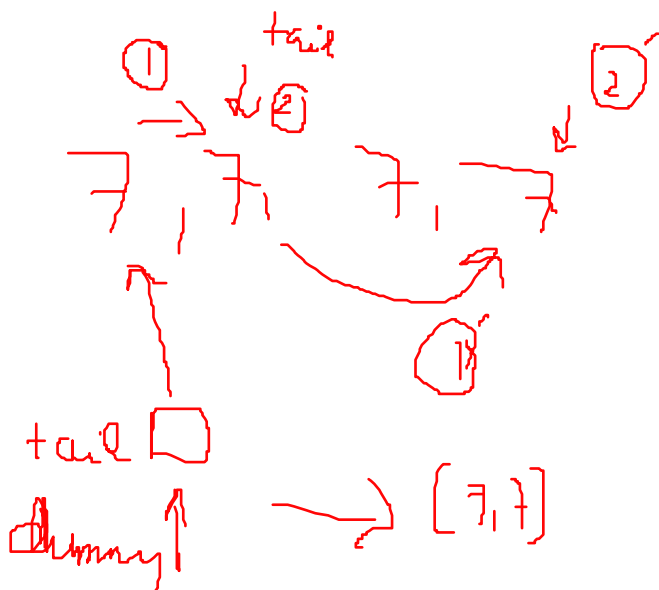


2)

```
if (tail->next->val == val)
{
    // ListNode *nodeToRemove = tail->next;
    // tail->next = nodeToRemove->next;
    // delete nodeToRemove;
    tail->next = tail->next->next; ①
}
// tail = tail->next->next;
// else
tail = tail->next; ②
```

VS

```
if (tail->next->val == val)
{
    1 ListNode *nodeToRemove = tail->next;
    2 tail->next = nodeToRemove->next;
    delete nodeToRemove;
    // tail->next = tail->next->next;
}
// tail = tail->next->next;
else
    tail = tail->next; }
```



passing function calls directly as arguments or storing the results in variables before passing

When you removed the `int left` and `int right` variables and directly passed the arguments, you likely changed the function to something like this:

```
cpp Copy code

int maxDepth(TreeNode* root) {
    if (root == nullptr)
        return 0;
    return 1 + max(maxDepth(root->left), maxDepth(root->right));
}
```

Now, let's explain the observed changes:

1. **Runtime Improvement:** By removing the local variables `left` and `right`, you reduce the number of operations in the function (assignments and intermediate storing of values). This makes the function slightly more efficient in terms of runtime.
2. **Memory Increase:** In the modified version, the call stack might be growing slightly differently due to the way the `max` function evaluates its arguments. In some compilers and environments, this might lead to a temporary increase in memory consumption.

Here's a detailed explanation of the memory aspect:

 Regenerate

Here's a detailed explanation of the memory aspect:

- When you call `max(maxDepth(root->left), maxDepth(root->right))`, both function calls are passed as arguments to the `max` function.
- Depending on the order of evaluation, this may lead to both recursive calls being pushed onto the call stack before evaluating the `max` function, causing a temporary increase in memory usage.

In the original version, by storing the results in variables `left` and `right`, you are controlling the order of evaluation, possibly leading to a more consistent pattern of call stack growth and shrinkage.


- If you have a simple call like `max(f(x), g(y))`, passing the function calls directly is often fine.
- If you have a more complex call like `max(f(x, g(y, z)), h(a, b, c))`, storing intermediate results in variables can enhance readability and maintainability.

In your specific example of the `maxDepth` function, either approach can be justified. If you find that one version consistently performs better in terms of runtime or memory, you might choose that one. Otherwise, consider readability and maintainability in choosing between the two.

Pair

Yes, the `std::pair` is defined inside the `<utility>` header in C++. So you would include it with:


cpp

 Copy code

```
#include <utility>
```

Here's a brief example again:

cpp

 Copy code

```
#include <utility>
#include <iostream>

int main() {
    std::pair<int, double> p = std::make_pair(5, 3.14);
    std::cout << p.first << ", " << p.second; // Outputs: 5, 3.14
    return 0;
}
```

Structured binding

Structured binding is a feature introduced in C++17 that allows you to decompose objects into their individual parts, binding the names of the variables to the members of the object.

Here are some examples:

1. With Pairs:

```
cpp Copy code

std::pair<int, double> p = {42, 3.14};
auto [a, b] = p;
std::cout << a << ", " << b; // Outputs: 42, 3.14
```

2. With Tuples:

```
cpp Copy code

std::tuple<int, char, float> t = {5, 'A', 2.71f};
auto [x, y, z] = t;
std::cout << x << ", " << y << ", " << z; // Outputs: 5, A, 2.71
```

3. With Arrays:

```
cpp Copy code

int arr[] = {1, 2, 3};
auto [first, second, third] = arr;
std::cout << first << ", " << second << ", " << third; // Outputs: 1, 2, 3
```

4. With Custom Structs:

```
cpp Copy code

struct Point {
    int x, y, z;
};

Point p = {10, 20, 30};
auto [x, y, z] = p;
std::cout << x << ", " << y << ", " << z; // Outputs: 10, 20, 30
```

Structured binding provides a convenient way to handle multiple return values and makes the code cleaner and more readable. It works with standard containers like pairs and tuples, as well as custom structs with non-static data members.

No, you cannot use a specific type like `int` with structured binding for arrays. Structured binding requires the use of `auto` so that the compiler can deduce the types of the individual elements.

Here's an example of correct usage with an array:

```
cpp Copy code

int arr[] = {1, 2, 3};
auto [first, second, third] = arr; // OK
```

Trying to use `int` instead of `auto` will result in a compilation error:

```
cpp Copy code

int [first, second, third] = arr; // Error
```

Reference = label

Using the `&` symbol before `[]` in a structured binding declaration creates references to the elements, rather than copying them. This means that any changes made to the bound variables will also change the original data.

Here's an example to illustrate the difference:

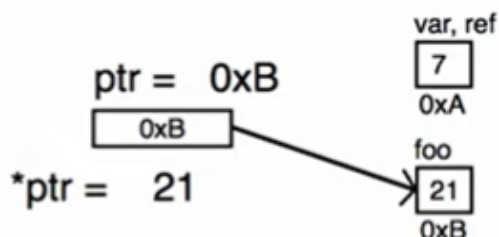
1. Without `&`:

```
cpp
std::pair<int, double> p = {42, 3.14};
auto [a, b] = p;
a = 100; // This does not change p.first
```

2. With `&`:

```
cpp
std::pair<int, double> p = {42, 3.14};
auto &[a, b] = p;
a = 100; // This changes p.first to 100
```

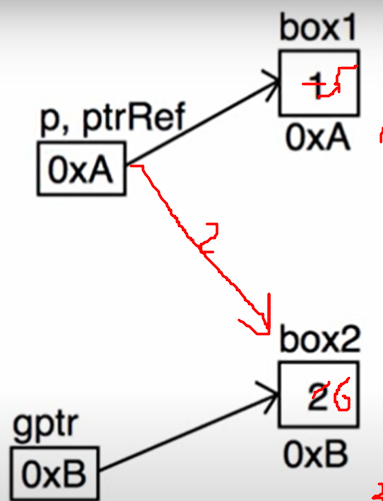
```
ptr_vs_ref.cpp
1 int* ptr;
2 int var = 7;
3 int foo = 21;
4 ptr = &var;
5 ptr = &foo;
6 int& ref = var;
```



```
return 0;
}

void passByPtr(int * ptr)
{
    *ptr = 3;
    ptr = gpPtr;
    *ptr = 4;
    cout << "\n_____ passByPtr has been called __\n";
}

void passByPtrRef(int * & ptrRef)
{
    *ptrRef = 5;
    ptrRef = gpPtr;
    *ptrRef = 6;
    cout << "\n_____ passByPtrRef has been called\n";
}
```



Lambda functions: can be defined within a function

1. Basic Syntax:

```
cpp Copy code

auto myLambda = [] { std::cout << "Hello, Lambda!\n"; };
myLambda(); // Outputs: Hello, Lambda!
```

2. With Parameters:

```
cpp Copy code

auto add = [](int a, int b) { return a + b; };
int sum = add(3, 4); // sum is 7
```

3. Capturing Variables:

```
cpp Copy code

int x = 5;
auto addX = [x](int a) { return a + x; };
int result = addX(3); // result is 8
```

4. Capturing by Reference:

```
cpp Copy code

int y = 10;
auto multiplyByY = [&y](int a) { return a * y; };
int product = multiplyByY(2); // product is 20
```

5. Capturing All Variables by Value or Reference:

```
cpp Copy code

int a = 5, b = 10;
auto captureAllByValue = [=] { return a + b; }; // Captures all by value
auto captureAllByReference = [&] { return a * b; }; // Captures all by reference
```

6. Specifying Return Type:

```
cpp Copy code

auto divide = [](double x, double y) -> double {
    if (y != 0.0) return x / y;
    return 0.0;
};
```

7. Mutable Lambdas (if you want to modify captured variables by value):

```
cpp Copy code

int z = 3;
auto incrementZ = [z]() mutable { z += 1; };
incrementZ(); // z inside lambda is modified, but outer z remains 3
```

What's Better?

- If you value **explicitness and readability**, especially for those who may be less familiar with C++ idioms, then `while (root != nullptr)` may be preferable.
- If you prefer **conciseness and are working with others who are familiar with C++ conventions**, then `while (root)` might be the better choice.

In practice, both are correct, and the choice may come down to personal or team coding style preferences.

Backtracking

1. **Choose:** Select an option from the remaining choices.
2. **Explore:** Move forward with that choice and explore further options.
3. **Check:** If a solution is found, save it. If the current path leads to an invalid solution or if you've explored all possibilities, go to step 4.
4. **Backtrack:** Undo the last choice (or choices) and explore the next option in the previous step.
5. **Repeat:** Continue the process until all possible solutions have been explored.

In the subjects problems:

```
void generateBinaryStrings(int n, vector<int>& current) {
    // Base Case: If the current binary string is of length n, print it
    if (current.size() == n) {
        for (int x : current) cout << x << ' ';
        cout << '\n';
        return;
    }

    // Choose, Explore, and Backtrack for 0 and 1
    for (int i = 0; i <= 1; i++) {
        // Choose: Add the current bit
        current.push_back(i);

        // Explore: Recursively call the function to explore further bits
        generateBinaryStrings(n, current);

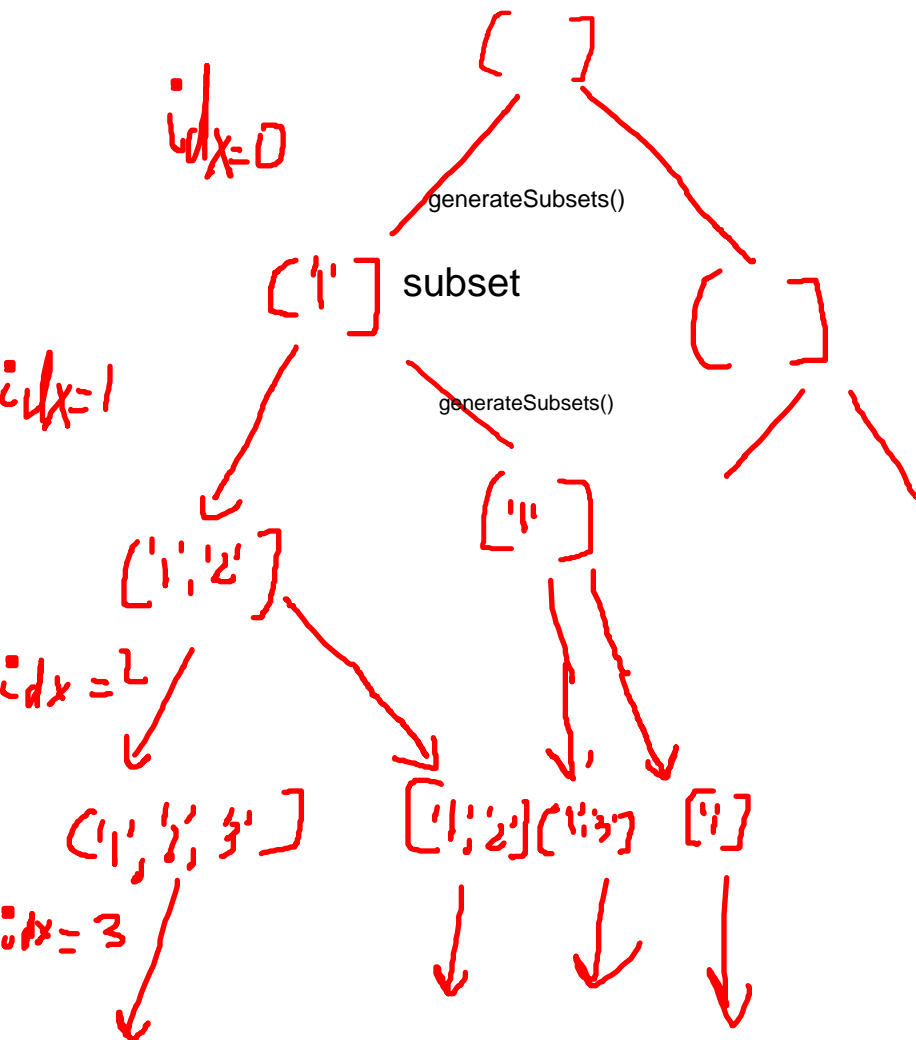
        // Backtrack: Remove the last added bit to explore the next possibility
        current.pop_back();
    }
}
```

```
1. `(3, [])`
1.1. `(3, [0])`
1.1.1. `(3, [0, 0])`
1.1.1.1. `(3, [0, 0, 0])`
1.1.1.2. `(3, [0, 0, 1])`
1.1.2. `(3, [0, 1])`
1.1.2.1. `(3, [0, 1, 0])`
1.1.2.2. `(3, [0, 1, 1])`
1.2. `(3, [1])`
1.2.1. `(3, [1, 0])`
1.2.1.1. `(3, [1, 0, 0])`
1.2.1.2. `(3, [1, 0, 1])`
1.2.2. `(3, [1, 1])`
1.2.2.1. `(3, [1, 1, 0])`
1.2.2.2. `(3, [1, 1, 1])`
```


generateSubsets

main

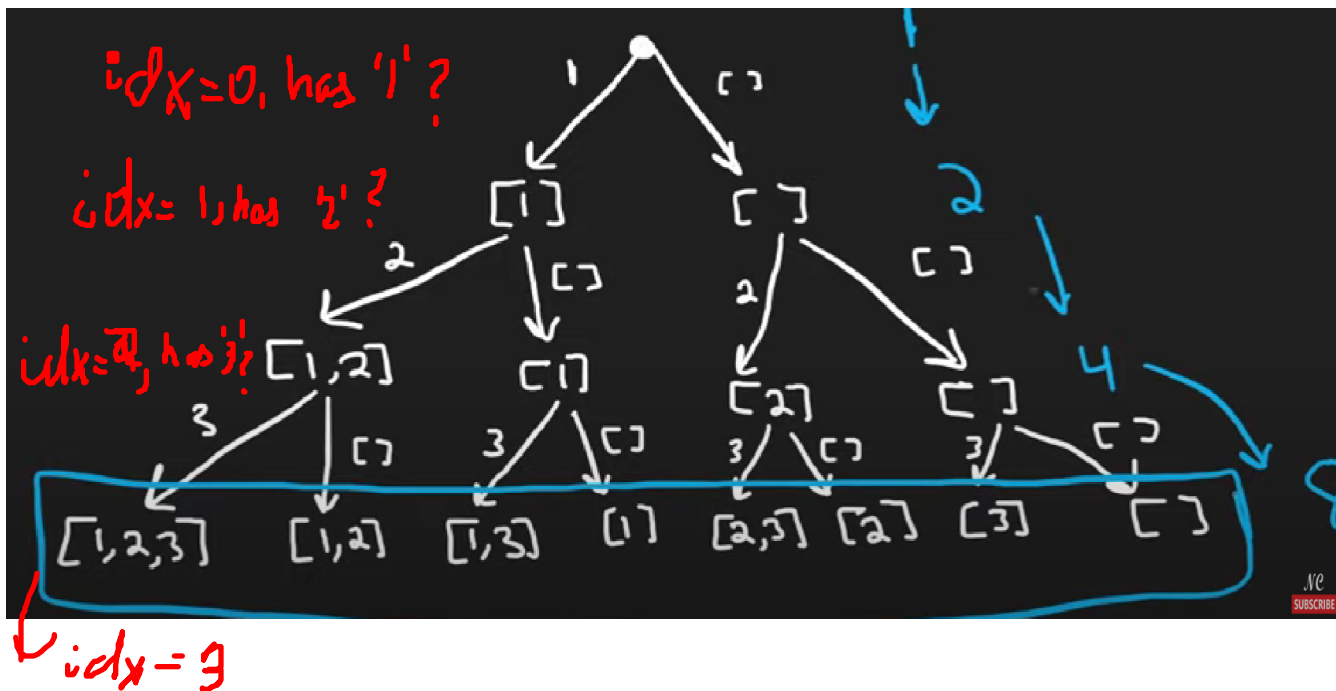
idx subset



1. `(0, [])`
 - 1.1. `(1, [1])`
 - 1.1.1. `(2, [1, 2])`
 - 1.1.1.1. `(3, [1, 2, 3])` ✓
 - 1.1.1.2. `(3, [1, 2])` ✓
 - 1.1.2. `(2, [1])` ✓
 - 1.1.2.1. `(3, [1, 3])` ✓
 - 1.1.2.2. `(3, [1])` ✓
 - 1.2. `(1, [])`
 - 1.2.1. `(2, [2])`
 - 1.2.1.1. `(3, [2, 3])` ✓
 - 1.2.1.2. `(3, [2])` ✓
 - 1.2.2. `(2, [])`
 - 1.2.2.1. `(3, [3])` ✓
 - 1.2.2.2. `(3, [])` ✓

$idx == \text{nums.size}() \checkmark$

$$8 = 2^3$$



Dynamic Programming

Fibonacci

```
memo[n] = fib(n - 1, memo) + fib(n - 2, memo);  
return memo[n];
```

memo
{
→ 3: 2,
 4: 3
}

fib(6)

A recursion tree for fib(6). The root is 6, which branches into 5 and 4. Node 5 branches into 4 and 3. Node 4 branches into 3 and 2. Node 3 branches into 2 and 1. Some nodes and edges are highlighted in orange, and an arrow points to node 3.

2n

A recursion tree for fib(9). The root is 9, which branches into 8 and 7. Node 8 branches into 7 and 6. Node 7 branches into 6 and 5. Node 6 branches into 5 and 4. Node 5 branches into 4 and 3. Node 4 branches into 3 and 2. Node 3 branches into 2 and 1. The tree is highlighted in orange.

fib memoized
complexity
O(n) time
O(n) space

CanSum

m = target sum
n = array length

O(n^m) time
O(m) space

m {

A recursion tree for CanSum. The root branches into three children. Each child branches into three children. The tree is highlighted in orange. A bracket on the left indicates the depth is m. Labels *n are placed near the nodes.

brute force

O(n^m) time
O(m) space

→

memoized

O(m * n) time
O(m) space

2 in fib



Priority Queue

3) Implement Priority Queue Using Heaps:

Binary Heap is generally preferred for priority queue implementation because heaps provide better performance compared to arrays or LinkedList.

Considering the properties of a heap, The entry with the largest key is on the top and can be removed immediately. It will, however, take time $O(\log n)$ to restore the heap property for the remaining keys. However if another entry is

Binary Heap	insert()	remove()	peek()
Time Complexity	$O(\log n)$	$O(\log n)$	$O(1)$

<https://www.geeksforgeeks.org/priority-queue-using-binary-heap/>

A priority queue can be implemented using a binary heap, and a binary heap can be efficiently represented as an array rather than using a tree with separate `TreeNode` objects.

Here's why using an array can be advantageous, with an example in C++ for implementing a max-heap as a priority queue:

1. **Space Efficiency:** Arrays eliminate the need for pointers, which can save space.
2. **Cache Locality:** Arrays provide better cache locality, making access faster.
3. **Easy Indexing:** In a binary heap, the parent, left child, and right child of any node can be found using simple arithmetic on the indices, making the operations efficient.

