# Introduction

## What Is Algorithm?

Let us consider the problem of preparing an omelet dish. To do that, we follow the steps given below:

1. Get the frying pan.
2. Get the oil.
   1. Do we have oil?
3. If yes, put it in the pan.
4. If no, do we want to buy oil?
5. If yes, then go out and buy.
6. If no, we can terminate.
7. Turn on the stove, etc...

What we are doing is providing a step-by-step procedure for solving it.

Definition of an algorithm: **An algorithm is the step-by-step unambiguous instructions to solve a given problem.**

## What Is Analysis of Algorithms?

Multiple algorithms are available for solving the same problem (for example, a sorting problem has many algorithms, like insertion sort, selection sort, quick sort, etc.). But only a few has an optimal performance. Thus, we need to analyze all of them to find the most suitable for our case.

Algorithm analysis helps us **determine which algorithm is most efficient in terms of time and memory consumed**.

## How to Compare Algorithms?

Do you think following measures are good to compare algorithms?

* Execution time? Not a good measure as they are specific to a particular computer.
* Number of statements executed? Not a good measure as it varies with the programming language and the style of the individual programmer.

Ideal solution? We express the running time of a given algorithm as a function of the input size of n: f(n). And compare the *rate of growth* of these functions. This kind of comparison is independent of machine time, programming style, etc.

## What Is Rate of Growth?

The rate at which the **running time increases as a function of input** is called *rate of growth*.

Below is the list of growth rates you will come across throughout this tutorial:

|  |  |  |
| --- | --- | --- |
| **Time Complexity** | **Name** | **Example** |
| 1 | Constant | Adding an element to the front of a linked list |
| loglogn |  |  |
| logn | Logarithmic | Finding an element in a sorted array |
| n | Linear | Finding an element in a unsorted array |
| lognlogn |  |  |
| nlogn | Linear Logarithmic | Sorting n items by 'Divide & Conquer' |
| n! |  | Finding all permutations in a string |
| n2 | Quadratic | Shortest path between 2 nodes in a graph |
| n3 | Cubic | Matrix Multiplication |
| 2n | Exponential | The Towers of Hanoi problem |

## How Many Types of Algorithm Analysis?

There are three types of analysis:

* **Worst case**
* Defines the input for which the algorithm takes **the longest time to complete**.
* **Best case**
* Defines the input for which the algorithm takes **the fastest time to complete**.
* **Average case**
* Provides a prediction about the running time of the algorithm.
* Run the algorithm many times, using many different inputs that come from some distribution that generates these inputs, compute the total running time (by adding the individual times), and divide by the number of trials.
* Assumes that the input is random.

Lower Bound <= Average Time <= Upper Bound

## What Are Algorithm Notations?

### Big-O Notation

This notation gives the tight upper bound of the given algorithm and we represent it as f(n) = O(g(n)).

For example, if f(n) = n4 + 100n2 + 10n + 50 is the given algorithm, then g(n) is n4 => O(n4).

**O(1)**

Time complexity of a function (or set of statements) is considered as O(1) if it doesn’t contain loop, recursion and call to any other non-constant time function. For example:

// set of non-recursive and non-loop statements

**O(loglogn)**

Time complexity of a loop is considered as O(loglogn) if the loop variables is reduced / increased exponentially by a constant amount. For example:

// Here c is a constant greater than 1

for (int i = 2; i <= n; i = pow(i, c)) {

    // some O(1) expressions

}

**O(logn)**

Time complexity of a loop is considered as O(logn) if the loop variables is divided / multiplied by a constant amount. For example:

for (int i = 1; i <= n; i \*= c) {

    // some O(1) expressions

}

**O(n)**

Time complexity of a loop is considered as O(n) if the loop variables is incremented / decremented by a constant amount. For example:

// Here c is a positive integer constant

for (int i = 1; i <= n; i += c) {

    // some O(1) expressions

}

**O(lognlogn)**

**O(nlogn)**

**O(nc)**

Time complexity of nested loops is equal to the number of times the innermost statement is executed. For example, the following sample loops have O(n2) time complexity:

for (int i = 1; i <= n; i += c) {

    for (int j = 1; j <=n; j += c) {

        // some O(1) expressions

    }

}

**O(2n)**

int fibo(int n) {

...

int result = fibo(n - 1) + fibo(n - 2);

}

### Omega-Ω Notation

This notation gives the tight lower bound of the given algorithm and we represent it as f(n) = Ω(g(n)).

For example, if f(n) = 100n2 + 10n + 50 is the given algorithm, then g(n) is n2 => Ω(n2).

### Theta-Θ Notation

This notation decides whether the upper and lower bounds of a given algorithm are the same.

IMPORTANT NOTE:

Throughout this tutorial, we generally **focus on the upper bound (O) because knowing the lower bound (Ω) of an algorithm is of no practical importance**, and we use the Θ notation if the upper bound (O) and lower bound (Ω) are the same.

# Supported Tools

## Visualization

Data Structures and Algorithms Visualization Tool: <https://csvistool.com/>

Tree Drawing and Visualization Tool: <https://tree-visualizer.netlify.app/>

Graph Drawing Tool: <https://csacademy.com/app/graph_editor/>

# Recursion Technique

## What Is Recursion?

Recursion is a technique where **a function calls itself** to solve a problem. This method is particularly useful for **problems that can be broken down into similar subproblems**.

### Key Concepts

* **Base case**: This is the condition under which the **recursion stops**. Without a base case, the function would call itself indefinitely, leading to a stack overflow.
* **Recursive case**: This is the part of the function where the **recursion occurs**. It typically involves calling the function with modified arguments that bring it closer to the base case.

### Usecases

Recursion is applied to a wide range of algorithms, such as backtracking, dynamic programming, etc.

## How Recursion Works

The call stack is a LIFO data structure that stores information about function calls in a program. It keeps track of active functions, their parameters, and local variables.

1. **When a function calls itself (in recursive case), a new frame is pushed to the call stack** for that new instance of the function. Each recursive call goes deeper into the stack until a base case is reached, at which point the function no longer calls itself.
2. **When a function completes its execution (after base case), its frame is popped off the stack. Then the call stack returns the call control back to the previous frame**. This is often referred to as "returning back up" the call stack. The program then continues executing the code that follows the function call in the previous frame.

TIP:

Push, pop and call control return. Mastering these keywords is the key to understand how recursion works.

IMPORTANT:

Deeply understanding of the **call stack is crucial** to undertsand how recursion works. The key to solve recursive problems is, thus, knows how your functions are managed in the call stack. For simple problems, there is no much thing to concern. But for complex problems, it’s almost impossible to mentally keep track of the recursion. Instead the **best advice is to draw the call stack in paper** and patiently tranverse it.

**TODO:**

<https://www.youtube.com/watch?v=AIwxbJ5mP8M&ab_channel=JAVAAID-CodingInterviewPreparation>

<https://launchschool.com/books/advanced_dsa/read/exploring_call_stack>

<https://www.freecodecamp.org/news/how-recursion-works-explained-with-flowcharts-and-a-video-de61f40cb7f9/>

## Pros and Cons

### Pros

* **Simplicity**: Recursive solutions can be more easier to understand than their iterative counterparts for certain problems.
* **Reduction of code**: It often results in less code as the recursive function can handle multiple cases in a single function call.

### Cons

* **Low performance**: Recursive functions can be less efficient than iterative solutions due to the overhead of multiple function calls and the **risk of stack overflow**.
* **Big memory usage**: Each recursive call consumes stack space, which can lead to increased memory usage, especially for deep recursion.

## Tips

### Recursion Illustration

Both tree and stack representations are useful for illustrating recursion, but they serve different purposes and highlight different aspects of the recursion process.

#### Stack

* Shows call stack behavior: A stack representation illustrates how function calls are managed in memory. It shows the most recent function calls at the top and highlights how functions are pushed and popped from the stack.
* Good for understanding return values: It clarifies how and when control returns to previous function calls.

#### Tree

* Visualizes function calls: A tree diagram shows the hierarchical structure of recursive calls and how they branch out. Each node represents a function call, and the branches indicate recursive calls. With Depth-First Traversal, it effectively illustrates the order of execution and the depth-first nature of recursion.
* Good for understanding complex recursions: If the recursion has multiple branches, a tree makes it easier to visualize how many times each function is called.

IN PRACTICE:

**Prefer tree to stack.** While tree illustration for recursion works in all cases, stack illustrationis very difficult to represents multiple-call recursion.

VERY HELPFUL:

In your code, **add debug logs wisely** to illustrate the call stack. Check below examples for more details.

### Avoiding Stack Overflow

#### Good Base Case

Always ensure that your recursive function has a well-defined base case that terminates the recursion. If the base case is never reached, the function will continue to call itself, leading to a stack overflow.

void recursiveFunction(int n) {

if (n <= 0) return; // Base case

recursiveFunction(n - 1);

}

#### Tail Recursion

Tail recursion is a special case where **the recursive call is the last operation in the function**. Some compilers can optimize tail-recursive functions to avoid increasing the call stack.

How tail recursion works?

Recursion uses a stack to keep track of function calls. With every function call, a new frame is pushed onto the stack which contains local variables and data of that call. Let’s say one stack frame requires O(1) memory space, then for N recursive calls, memory required would be O(N).

Tail call elimination reduces the space complexity of recursion from O(N) to O(1). Since the recursive call is the last statement, there is nothing left to do in the current function. As function call is eliminated, no new stack frames are created and the function is executed in constant memory space.

Code:

int tailRecursiveFactorial(int n, int accumulator = 1) {

if (n == 0) {

return accumulator;

}

return tailRecursiveFactorial(n - 1, n \* accumulator);

}

#### Limit Recursion Depth

You can implement a depth limit to **prevent deep recursion**. If the recursion exceeds a certain depth, you can return an error or handle it appropriately.

void limitedDepthRecursion(int n, int depth = 0) {

if (depth > MAX\_DEPTH) return; // Limit depth

// Recursive call...

limitedDepthRecursion(n - 1, depth + 1);

}

#### Convert to Iteration

If recursion depth is a concern, consider converting your recursive algorithm to an iterative one **using loops** and a stack data structure. This way, you can manage memory usage more effectively.

int iterativeFactorial(int n) {

int result = 1;

for (int i = 2; i <= n; ++i) {

result \*= i;

}

return result;

}

#### Use a Stack Data Structure

If you need to maintain the recursive structure but avoid stack overflow, use your own stack data structure (like a std::stack in C++) to simulate recursion. Push function parameters onto the stack and pop them off as you would in a recursive call.

#include <stack>

void iterativeDFS(Node\* root) {

std::stack<Node\*> stack;

stack.push(root);

while (!stack.empty()) {

Node\* node = stack.top();

stack.pop();

// Process the node...

// Push children onto the stack

if (node->right) stack.push(node->right);

if (node->left) stack.push(node->left);

}

}

## Examples

### [Easy] One Function Call

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| #include <iostream>  using namespace std;    void fn(int n) {  if (n > 2) {  fn(n - 1);  }  cout << n << " ";  }    int main() {  fn(5);  return 0;  } | **Output**:  2 3 4 5  **Explanation**:  fn(5) is called:  Since 5 > 2, it calls fn(4).  fn(4) is called:  Since 4 > 2, it calls fn(3).  fn(3) is called:  Since 3 > 2, it calls fn(2).  fn(2) is called:  Since 2 not > 2, it reaches the base case.  It prints **2**.  Control returns to fn(3), which then prints **3**.  Control returns to fn(4), which then prints **4**.  Control returns to fn(5), which then prints **5**. | **Stack illustration**:   |  |  |  | | --- | --- | --- | | Main call | Sub call | Output | | fn(5) | fn(4) | Print 5 | | fn(4) | fn(3) | Print 4 | | fn(3) | fn(2) | Print 3 | | fn(2) | Base case | Print 2 |   Push: down arrow  Pop: up arrow  fn(5)  |\_\_fn(4)  |\_\_fn(3)  |\_\_fn(2) | **Tree illlustration**: |

Another version with **debug logs** to illustrate call stack:

|  |  |
| --- | --- |
| #include <iostream>  using namespace std;    void fn(int n, string recDepth) {      printf("%sCall fn(%d)\n", recDepth.c\_str(), n);      if (n > 2) {          fn(n - 1, recDepth + "  ");          printf("%sReturn call to fn(%d)\n", recDepth.c\_str(), n);      }     printf("%sBase case --> print %d\n", recDepth.c\_str(), n);  }    int main() {      fn(5, "");      return 0;  } | Call fn(5)  Call fn(4)  Call fn(3)  Call fn(2)  Base case --> print 2  Return call to fn(3)  Base case --> print 3  Return call to fn(4)  Base case --> print 4  Return call to fn(5)  Base case --> print 5  **Basically, you’ll need 3 logs:**   * **Beginning of the function** * **After the function call** * **Base case** |

Time complexity: O(n)

Space complexity: O(n)

### [Easy] Factorial Calculation

**Problem**:

Write a function that calculates the factorial of a non-negative integer n using the following constraints:

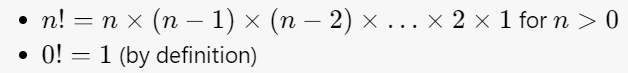
* If n is less than 0, return -1 (indicating an invalid input).
* If n is 0, return 1 (by definition).
* For n greater than 0, use a recursive approach to calculate the factorial.

Example:

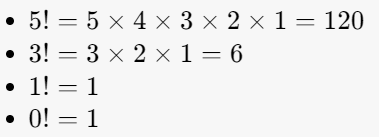
* Input: n = 4
* Output: 24 (Since 4! = 4×3×2×1 = 24)

What is factorial?

The factorial of a non-negative integer n is the product of all positive integers from 1 to n. It is denoted by n! and is defined as follows:



Examples**:**



**Implementation**:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| #include <iostream>  using namespace std;    int fac(int n) {      if (n < 0) {          return -1;      } else if (n == 0) {          return 1;      } else {          return n \* fac(n - 1);      }  }    int main() {      int n = 4;      int result = fac(n);      cout << result << endl;      return 0;  } | Output:  24  Explanation:   * Call: fac(4**)** * A stack frame is created for fac(4). * n is 4. * The function needs to calculate 4 \* fac(3). It can't do the multiplication yet, so it pushes fac(3) onto the stack. * Call: fac(3) * A stack frame is created for fac(3). * n is 3. * The function needs to calculate 3 \* fac(2). It pushes fac(2) onto the stack. * Call: fac(2) * A stack frame is created for fac(2). * n is 2. * The function needs to calculate 2 \* fac(1). It pushes fac(1) onto the stack. * Call: fac(1) * A stack frame is created for fac(1). * n is 1. * The function needs to calculate 1 \* fac(0). It pushes fac(0) onto the stack. * Call: fac(0) * A stack frame is created for fac(0). * n is 0. * This is the base case! The function returns 1. * Returning from fac(0) * The stack frame for fac(0) is removed (popped). * Execution returns to fac(1). * fac(1) can now calculate 1 \* 1 = 1. It returns 1. * Returning from fac(1) * The stack frame for fac(1) is removed. * Execution returns to fac(2). * fac(2) can now calculate 2 \* 1 = 2. It returns 2. * Returning from fac(2) * The stack frame for fac(2) is removed. * Execution returns to fac(3). * fac(3) can now calculate 3 \* 2 = 6. It returns 6. * Returning from fac(3) * The stack frame for fac(3) is removed. * Execution returns to fac(4). * fac(4) can now calculate 4 \* 6 = 24. It returns 24. * Returning from fac(4) * The stack frame for fac(4) is removed. * The final result, 24, is returned to the caller. | **Stack illustration**:   |  |  |  | | --- | --- | --- | | Main call | Sub call | Return | | fac(4) | 4 x fac(3) | 4 x 6 = 24 | | fac(3) | 3 x fac(2) | 3 x 2 = 6 | | fac(2) | 2 x fac(1) | 2 x 1 = 2 | | fac(1) | 1 x fac(0) | 1 x 1 = 1 | | fac(0) | *Base case* | 1 |     **Tree visualization**: |

Another version with **debug logs** to illustrate call stack:

|  |  |
| --- | --- |
| #include <iostream>  using namespace std;    int fn(int n, string recDepth) {      printf("%sCall fn(%d)\n", recDepth.c\_str(), n);      if (n < 0) {          printf("%sBase case\n", recDepth.c\_str());          return -1;      } else if (n == 0) {          printf("%sBase case\n", recDepth.c\_str());          return 1;      } else {          int result = n \* fn(n - 1, recDepth + "  ");          printf("%sReturn call to fn(%d)\n", recDepth.c\_str(), n);          printf("%sReturn value: %d x fn(%d - 1) = %d\n", recDepth.c\_str(), n, n, result);          return result;      }  }    int main() {      int n = 4;      int result = fn(n, "");      cout << "Output: " << result << endl;      return 0;  } | Call fn(4)  Call fn(3)  Call fn(2)  Call fn(1)  Call fn(0)  Base case  Return call to fn(1)  Return value: 1 x fn(1 - 1) = 1  Return call to fn(2)  Return value: 2 x fn(2 - 1) = 2  Return call to fn(3)  Return value: 3 x fn(3 - 1) = 6  Return call to fn(4)  Return value: 4 x fn(4 - 1) = 24  Output: 24 |

Time complexity: O(n)

Space complexity: O(n)

### [Easy] Multiple Function Calls

|  |  |  |
| --- | --- | --- |
| #include <iostream>  using namespace std;    void fn(int n) {  if (n > 2) {  fn(n - 1);  fn(n - 2);  fn(n - 3);  }    cout << n << " ";  }    int main() {  fn(5);  return 0;  } | **Output**:  2 1 0 3 2 1 4 2 1 0 3 2 5  **Explanation**:  call fn(5)  call fn(5-1)=fn(4)  call fn(4-1)=fn(3)  call fn(3-1)=fn(2)  base case --> print 2  return call to fn(3)  call fn(3-2)=fn(1)  base case --> print 1  return call to fn(3)  call fn(3-3)=fn(0)  base case --> print 0  return call to fn(3)  base case --> print 3  return call to fn(4)  call fn(4-2)=fn(2)  base case --> print 2  return call to fn(4)  call fn(4-3)=fn(1)  base case --> print 1  return call to fn(4)  base case --> print 4  return call fn(5)  call fn(5-2)=fn(3)  call fn(3-1)=fn(2)  base case --> print 2  return call to fn(3)  call fn(3-2)=fn(1)  base case --> print 1  return call to fn(3)  call fn(3-3)=fn(0)  base case --> print 0  return fn(3)  base case print 3  return call fn(5)  call fn(5-3)=fn(2)  base case print 2  return call fn(5)  base case print 5 | **Stack illustration** (from top to bottom):  fn(5)  ├── fn(4)  │ ├── fn(3)  │ │ ├── fn(2) (prints 2)  │ │ ├── fn(1) (prints 1)  │ │ └── fn(0) (prints 0)  │ │ └── prints 3  │ ├── fn(2) (prints 2)  │ └── fn(1) (prints 1)  │ └── prints 4  ├── fn(3)  │ ├── fn(2) (prints 2)  │ ├── fn(1) (prints 1)  │ └── fn(0) (prints 0)  │ └── prints 3  └── fn(2) (prints 2)  └── prints 5  **Tree illustration**: |

Another version with **debug logs** to illustrate call stack:

|  |  |
| --- | --- |
| #include <iostream>  using namespace std;    void fn(int n, string recDepth) {  printf("%sCall fn(%d)\n", recDepth.c\_str(), n);  if (n > 2) {  fn(n - 1, recDepth + " ");  printf("%sReturn call to fn(%d)\n", recDepth.c\_str(), n, n - 1);  fn(n - 2, recDepth + " ");  printf("%sReturn call to fn(%d)\n", recDepth.c\_str(), n, n - 2);  fn(n - 3, recDepth + " ");  printf("%sReturn call to fn(%d)\n", recDepth.c\_str(), n, n - 3);  }  printf("%sBase case", recDepth.c\_str());    cout << " ==> Print " << n << endl;  }    int main() {  fn(5, "");  return 0;  } | Call fn(5)  Call fn(4)  Call fn(3)  Call fn(2)  Base case ==> Print 2  Return call to fn(3)  Call fn(1)  Base case ==> Print 1  Return call to fn(3)  Call fn(0)  Base case ==> Print 0  Return call to fn(3)  Base case ==> Print 3  Return call to fn(4)  Call fn(2)  Base case ==> Print 2  Return call to fn(4)  Call fn(1)  Base case ==> Print 1  Return call to fn(4)  Base case ==> Print 4  Return call to fn(5)  Call fn(3)  Call fn(2)  Base case ==> Print 2  Return call to fn(3)  Call fn(1)  Base case ==> Print 1  Return call to fn(3)  Call fn(0)  Base case ==> Print 0  Return call to fn(3)  Base case ==> Print 3  Return call to fn(5)  Call fn(2)  Base case ==> Print 2  Return call to fn(5)  Base case ==> Print |

Time complexity: O(3n)

Space complexity: O(n)

### [Easy] Fibonacci Sequence

**Problem:**

Write a function that calculates the **n-th** Fibonacci number (start index is 0) using the following constraints:

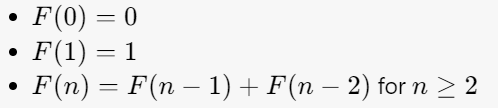
* If n is less than 0, return -1.
* If n is 0, return 0.
* If n is 1, return 1.
* For n greater than 1, use an iterative approach to calculate the Fibonacci number.

Example:

* Input: n = 6
* Output: 8 (the Fibonacci sequence is 0, 1, 1, 2, 3, 5, 8 ...)

What is Fibonacci?

In the Fibonacci sequence, each number is the sum of the two preceding ones, starting from 0 and 1 (or sometimes 1 and 1). The sequence is typically defined as follows:



Examples

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, …

**Implementation**:

|  |  |
| --- | --- |
| #include <iostream>  using namespace std;    int fibo(int n, string depth) {      printf("%sCall fibo(%d)\n", depth.c\_str(), n);      if (n < 0) {          printf("%sBase case\n", depth.c\_str());          return -1;      } else if (n == 0) {          printf("%sBase case\n", depth.c\_str());          return 0;      } else if (n == 1) {          printf("%sBase case\n", depth.c\_str());          return 1;      }      int result = fibo(n - 1, depth + "  ") + fibo(n - 2, depth + "  ");      printf("%sReturn call to fibo(%d)\n", depth.c\_str(), n);      printf("%sReturn value: fibo(%d) + fibo(%d) = %d\n", depth.c\_str(), n - 1, n - 2, result);      return result;  }    int main() {      int n = 6;      int fibo\_num = fibo(n, "");      printf("%dth number in Fibo sequence: \n", n, fibo\_num);      return 0;  } | Call fibo(6)  Call fibo(5)  Call fibo(4)  Call fibo(3)  Call fibo(2)  Call fibo(1)  Base case  Call fibo(0)  Base case  Return call to fibo(2)  Return value: fibo(1) + fibo(0) = 1  Call fibo(1)  Base case  Return value: fibo(2) + fibo(1) = 2  Call fibo(2)  Call fibo(1)  Base case  Call fibo(0)  Base case  Return call to fibo(2)  Return value: fibo(1) + fibo(0) = 1  Return call to fibo(4)  Return value: fibo(3) + fibo(2) = 3  Call fibo(3)  Call fibo(2)  Call fibo(1)  Base case  Call fibo(0)  Base case  Return call to fibo(2)  Return value: fibo(1) + fibo(0) = 1  Call fibo(1)  Base case  Return call to fibo(3)  Return value: fibo(2) + fibo(1) = 2  Return call to fibo(5)  Return value: fibo(4) + fibo(3) = 5  Call fibo(4)  Call fibo(3)  Call fibo(2)  Call fibo(1)  Base case  Call fibo(0)  Base case  Return call to fibo(2)  Return value: fibo(1) + fibo(0) = 1  Call fibo(1)  Base case  Return call to fibo(3)  Return value: fibo(2) + fibo(1) = 2  Call fibo(2)  Call fibo(1)  Base case  Call fibo(0)  Base case  Return call to fibo(2)  Return value: fibo(1) + fibo(0) = 1  Return call to fibo(4)  Return value: fibo(3) + fibo(2) = 3  Return call to fibo(6)  Return value: fibo(5) + fibo(4) = 8  Output: 8 |

Time complexity: O(2n)

Space complexity: O(n)

### [Medium] Factorial Calculation Using Tail-Recursion

Compared to the non-tail recursion, it's easily to see that the value returned by fact(n - 1) is used in fact(n), and thus, call to fact(n - 1) is not the last thing done by fact(n).

This is the tail recursion:

|  |  |  |
| --- | --- | --- |
| #include <iostream>  using namespace std;    // A tail recursive function to calculate factorial  unsigned fac(unsigned int n, unsigned int a) {      if (n <= 1) {          return a;      }        return fac(n - 1, n \* a);  }    // A wrapper over fac  unsigned int factorial(unsigned int n) {      return fac(n, 1);  }    int main() {      cout << factorial(4);      return 0;  } | Output:  24 |  |

Time complexity: O(n)

Space complexity: O(1) – Compared to non-tail version which is O(n)

## Practices

### [Easy] Sum of Natural Numbers

**Problem**:

Write a function that calculates the sum of the first n natural numbers using the following constraints:

* If n is less than 1, return 0 (indicating an invalid input).
* For n greater than or equal to 1, calculate the sum.

**Example**:

Input: n = 5

Output: 15 (Since 1+2+3+4+5=15)

### [Medium] Remove Adjacent Duplicates

**Problem**:

Given a string s, remove all its adjacent duplicate characters recursively, until there are no adjacent duplicate characters left. If the resultant string becomes empty, return an empty string.

**Examples**:

* Input: s = "geeksforgeek". Output: "gksforgk" geksforg
* Input: s = "abccbccba". Output: ""
* Input: s = "abcd". Output: "abcd"

**Constraints**:

1<= s.size() <= 105

### [Medium] Power Of Reverse Numbers

**Problem**:

Given a numbern, find the value of n raised to the power of its own reverse.

Note:The result will always fit into a 32-bit signed integer.

**Examples:**

|  |  |  |
| --- | --- | --- |
| **Input** | **Output** | **Explanation** |
| n = 2 | 4 | The reverse of 2 is 2, and 22 = 4. |
| n = 10 | 10 | The reverse of 10 is 1 (leading zero is discarded), and 101 = 10. |
| n = 3 | 27 | The reverse of 3 is 3, and 33 = 27. |

**Constraints**:

1 <= n <= 10

## Refs

<https://www.youtube.com/watch?v=ngCos392W4w&ab_channel=Reducible>

<https://www.youtube.com/watch?v=0nKIr3kAt-k&ab_channel=GinaSprint> (from 4:18 to 12:35)

<https://www.youtube.com/watch?v=B3U6LExgevE&ab_channel=BytebyByte>

# Backtracking Technique

## What Is Backtracking?

Backtracking is an algorithmic technique that involves **finding a solution incrementally by trying different options/choices**. If one doesn't work, it backtracks and tries another until it finds the right one.

Backtracking vs Brute Force:

They share many similarities. They both try to search for all elements to find the right one. They both have bad time complexity.

But backtrack is smarter. While brute force simply exhaustively checks all possibilities, backtracking builds solutions incrementally as it can detect early when an option does not work and backtracks to save performance.

Backtracking vs Recursion:

Not even the same. While recursion is a programming technique, backtracking is an algorithm technique which mostly uses **recursive depth-first search** to solve problem.

### Key Concepts

* **Candidate**: A candidate is a potential choice or option that can be added to the current solution.
* **Search space**: The search space includes all possible combinations of candidates.
* **Decision point**: A decision point is a specific step in the algorithm where a candidate is chosen and added to the partial solution.
* **Decision space**: The decision space is the set of all possible candidates at each decision point.
* **Solution**: The solution is a valid and complete configuration that satisfies all problem constraints.
* **Partial solution**: A partial solution is an intermediate or incomplete configuration being constructed during the backtracking process.
* **Optimal solution**: In optimization problems, the optimal solution is the best possible solution.
* **Dead end**: A dead end occurs when a partial solution cannot be extended without violating constraints. In other words, when a *dead end* is reached, the algorithm backtracks to the previous *decision point* and explores a different path.

### Usecases

* Backtracking is commonly used in situations where you **need to explore all possibilities** to solve a problem, like searching for a path in a maze.

## How Backtracking Works

### Steps

|  |  |
| --- | --- |
| backtracking | **IS**: The Initial State where the recursion call starts to find a valid solution.  **C**: A checkpoint for recursive calls.  **TN**: A terminal node where no further recursive calls can be made, these nodes act as base case of recursion and we determine whether the current solution is valid or not at this state.  The back arrows: A backtracking in actions, where we revert the changes made by some checkpoint.  At each checkpoint, our program makes some decisions and move to other checkpoints untill it reaches a terminal node, after determining whether a solution is valid or not, the program starts to revert back to the checkpoints and try to explore other paths.  For example, TN1 … TN5 are the terminal nodes where the solution is not acceptable, while TN6 is the state where we found a valid solution. |

### Psuedocode

*void* ***FIND\_SOLUTIONS****( parameters):*

*if (valid* ***solution****):*

*store the* ***solution***

*return*

*for (all* ***choices****):*

*if (valid* ***choice****):*

***APPLY*** *(****choice****)*

***FIND\_SOLUTIONS*** *(parameters)*

***BACKTRACK*** *(remove* ***choice****)*

### Time Complexity

Exponential: O(KN) – Often exponential in the worst case.

Factorial: O(N!)

### Space Complexity

Depends on recursion depth.

## Pros and Cons

### Pros

### Cons

## Tips

## Examples

### [Easy] Permutations of a String

**Problem**:

Given a string **s**, which may contain duplicate characters. Generate and return an array of all unique permutations of the string.

Examples:

|  |  |  |
| --- | --- | --- |
| **Input** | **Output** | **Explanation** |
| "ABC" | "ABC", "ACB", "BAC", "BCA", "CAB", "CBA" | ABC has 6 unique permutations. |
| "ABSG" | "ABGS", "ABSG", "AGBS", "AGSB", "ASBG", "ASGB", "BAGS", "BASG", "BGAS", "BGSA", "BSAG", "BSGA", "GABS", "GASB", "GBAS", "GBSA", "GSAB", "GSBA", "SABG", "SAGB", "SBAG", "SBGA", "SGAB", "SGBA" | ABSG has 24 unique permutations. |
| "AAA" | "AAA" | No other unique permutations. |

**Constraints**:

* 1 <= s.size() <= 9
* **s** contains only uppercase English alphabets

**Idea**:

|  |  |
| --- | --- |
|  | The idea is to use backtracking to generate all possible permutations of given **string s**.  First **initialize** an array of string **output[]** to store all the **permutations.** Start from the **0th** index and for each **index i**, *swap* the value **s[i]** with all the elements in its right i.e. from **i+1 to n-1**, and *recur* to the **index i+1**. If the i**ndex i** is equal to **n**, store the resultant string in **output[]**, else keep operating similarly for all other indices. Thereafter, *swap* back the values to original values to initiate *backtracking*. |

**Implementation:**

|  |  |
| --- | --- |
| #include <iostream>  #include <vector>  #include <algorithm>  using namespace std;    void permute(string str, int index, vector<string>& output, string depth) {      printf("%sCall permute(%s, %d)\n", depth.c\_str(), str.c\_str(), index);      if (index == str.length()) {          output.emplace\_back(str);          printf("%sBase case. Add %s\n", depth.c\_str(), str.c\_str());          return;      }        for (int i = index; i < str.length(); i++) {          if (index != i) {              swap(str[index], str[i]);              printf("%sSwap %c and %c at index %d and i %d\n", depth.c\_str(), str[index], str[i], index, i);          } else {              printf("%sDo not swap %c and %c at index %d and i %d\n", depth.c\_str(), str[index], str[i], index, i);          }          permute(str, index + 1, output, depth + "  ");          printf("%sReturn call to permute(%s, %d) at i %d\n", depth.c\_str(), str.c\_str(), index, i);      }  }    int main() {      string str = "ABC";      vector<string> output = {};      permute(str, 0, output, "");        for (const auto& str : output) {          cout << str << " ";      }      cout << endl;      return 0;  } | Call permute(ABC, 0)  Do not swap A and A at index 0 and i 0  Call permute(ABC, 1)  Do not swap B and B at index 1 and i 1  Call permute(ABC, 2)  Do not swap C and C at index 2 and i 2  Call permute(ABC, 3)  Base case. Add ABC  Return call to permute(ABC, 2) at i 2  Return call to permute(ABC, 1) at i 1  Swap C and B at index 1 and i 2  Call permute(ACB, 2)  Do not swap B and B at index 2 and i 2  Call permute(ACB, 3)  Base case. Add ACB  Return call to permute(ACB, 2) at i 2  Return call to permute(ACB, 1) at i 2  Return call to permute(ABC, 0) at i 0  Swap B and A at index 0 and i 1  Call permute(BAC, 1)  Do not swap A and A at index 1 and i 1  Call permute(BAC, 2)  Do not swap C and C at index 2 and i 2  Call permute(BAC, 3)  Base case. Add BAC  Return call to permute(BAC, 2) at i 2  Return call to permute(BAC, 1) at i 1  Swap C and A at index 1 and i 2  Call permute(BCA, 2)  Do not swap A and A at index 2 and i 2  Call permute(BCA, 3)  Base case. Add BCA  Return call to permute(BCA, 2) at i 2  Return call to permute(BCA, 1) at i 2  Return call to permute(BAC, 0) at i 1  Swap C and B at index 0 and i 2  Call permute(CAB, 1)  Do not swap A and A at index 1 and i 1  Call permute(CAB, 2)  Do not swap B and B at index 2 and i 2  Call permute(CAB, 3)  Base case. Add CAB  Return call to permute(CAB, 2) at i 2  Return call to permute(CAB, 1) at i 1  Swap B and A at index 1 and i 2  Call permute(CBA, 2)  Do not swap A and A at index 2 and i 2  Call permute(CBA, 3)  Base case. Add CBA  Return call to permute(CBA, 2) at i 2  Return call to permute(CBA, 1) at i 2  Return call to permute(CAB, 0) at i 2  **ABC ACB BAC BCA CAB CBA** |

**Time complexity: O(n!)**

**Space complexity: O(n)**

**Tip**

* STL has function next\_permutation(), using it helps our above code even simpler: <https://www.codeguru.com/cplusplus/permutations-in-c/>

## Practices

# Dynamic Programming (DB) Technique

## What Is DP?

Dynamic Programming is an algorithmic technique that solves complex problems by breaking them down into simpler subproblems. By **solving each subproblem only once and storing the results** (also known as "memoization"), it avoids redundant computations.

Dynamic Programming vs Backtracking:

At first glance, DP and backtracking seems very similar as they both break a problem down into simpler subproblems (thus, mostly use recursion) in a more optimized way than brute force. The difference comes from how they optimize. While backtracking incrementally builds solutions by early abandoning invalid solutions, DP stores the results of subproblems to avoid redundant computations. DP approach can be iterative or recursive with memoization.

### Key Concepts

NA

### Usecases

DP is an optimization technique used when solving problems that consists of the following characteristics:

* **Overlapping subproblems**: The same subproblems are solved repeatedly in different parts of the problem.

Example:

* Problem: Computing the Fibonacci series.
* Solution: To compute the Fibonacci number at index n, we need to compute the Fibonacci numbers at indices n-1 and n-2. This means that the subproblem of computing the Fibonacci number at index n-1 is used twice in the solution to the larger problem of computing the Fibonacci number at index n.
* **Optimal substructures**: Combining the optimal results of subproblems to achieve the optimal result of the bigger problem.

Example:

* Problem: Finding the minimum cost path in a weighted graph from a source node to a destination node.
* Solution: Break the problem down into smaller subproblems:
  + Find the minimum cost path from the source node to each intermediate node.
  + Find the minimum cost path from each intermediate node to the destination node.

## How DP Works

## Examples

### [Easy] Fibonacci Sequence

**Brute force approach:**

To find the **nth** Fibonacci number using a brute force approach, you would simply add the **(n-1)th**and**(n-2)th** Fibonacci numbers. This would work, but it would be inefficient for large values of **n**, as it would require calculating all the previous Fibonacci numbers.

**Dynamic programming approach:**

|  |  |
| --- | --- |
| Fibonacci series using DP:   * Subproblems: F(0), F(1), F(2), F(3), ... * Sore solutions: Create a table to store the values of F(n) as they are calculated. * Build up solutions: For F(n), look up F(n-1) and F(n-2) in the table and add them. * Avoid redundancy:The table ensures that each subproblem (e.g., F(2)) is solved only once.   By using DP, we can efficiently calculate the Fibonacci sequence without having to recompute subproblems. | Same colors denote overlapping subproblems. |

**Implementation**:

|  |  |
| --- | --- |
| #include <iostream>  #include <vector>  using namespace std;    int fibo(int n) {      vector<int> sequence = {};      sequence.emplace\_back(0);      sequence.emplace\_back(1);        int result = 0;      for (int idx = 2; idx <= n; idx++) {          sequence.emplace\_back(sequence[idx - 1] + sequence[idx - 2]);          result = sequence[idx];      }      return result;  }    int main() {      int n = 6;      int fibo\_num = fibo(n);      printf("%dth number in Fibo sequence: %d\n", n, fibo\_num);      return 0;  } | 6th number in Fibo sequence: 8 |

**Time complexity**: O(n)

**Space complexity**: O(n)

## Practices

# Divide & Conquer Technique

## What Is Divide & Conquer?

Divice and Conquer is an algorithmic technique that break the problem into smaller non-overlapping problems, solving them individually and then merging them to find solution to the original problem.

## How Divide & Conquer Works?

### Steps

Divide & Conquer algorithm can be divided into three steps:

1. **Divide**:

* **Recursively break down** the original problem into smaller subproblems.
* Each subproblem should represent a part of the overall problem.
* The goal is to divide the problem **until no further division is possible**.

1. **Conquer**:

* Solve each of the smaller subproblems individually.
* If a subproblem is small enough (often referred to as the "base case"), we **solve it directly and independently** without further recursion. This allows for concurrent execution of subproblems for better efficiency gains.
* The **subproblems can be solved by applying the same Divide & Conquer technique recursively, or by applying a different algorithm technique**.
* The goal is to find solutions for these subproblems independently.

1. **Merge**:

* Combine the subproblems to get the final solution of the whole problem.
* Once the smaller subproblems are solved, we **recursively combine** their solutions to get the solution of larger problem.
* This step should be straightforward, as the solutions to the subproblems should be designed to fit together seamlessly.
* The goal is to formulate a solution for the original problem by merging the results from the subproblems.

### Time Complexity

T(n) = aT(n/b) + f(n), where

n = size of input

a = number of subproblems in the recursion

n/b = size of each subproblem. All subproblems are assumed to have the same size.

f(n) = cost of the work done outside the recursive call, which includes the cost of dividing the problem and cost of merging the solutions

### Space Complexity

## Examples

### [Easy] Finding Maximum Element in Array

**Problem**:

Given an array arr = [3, 1, 4, 1, 5, 9, 2, 6, 5, 3]. Find the maximum element in the array.

**Idea**:

Divide the array into two equal-sized subarrays. Then find the maximum of those two individual halves by recursively dividing them into two smaller halves. This is done till we reach subarrays of size 1. Finally, return the maximum element by returning the maximum in each subarray.

**Implementation:**

|  |  |
| --- | --- |
| #include <iostream>  #include <algorithm>  using namespace std;    int findMax(int arr[], int startIdx, int endIdx, string depth) {  if (startIdx == endIdx) {  return arr[startIdx];  }    int midIdx = (endIdx + startIdx)/2;  int leftMax = findMax(arr, startIdx, midIdx, depth + " ");  int rightMax = findMax(arr, midIdx + 1, endIdx, depth + " ");    return std::max(leftMax, rightMax);  }    int main() {  int arr[] = {3, 1, 4, 1, 5, 9, 2, 6, 5, 3};  int size = sizeof(arr)/sizeof(arr[0]);  int startIdx = 0;  int endIdx = size - 1;  int max = findMax(arr, startIdx, endIdx, "");  cout << "Max: " << max << endl;  return 0;  } | Max: 9 |

**Time complexity**: O(n) (linear time due to visiting each element once).

**Space complexity**: O(logn) (due to the depth of the recursion stack).

### [Easy] Merge Sort

Check [this session](#_Merge_Sort_-).

### [Easy] Quick Sort

Check [this session](#_[Easy]_Quick_Sort).

## Practices

# Greedy Technique

## What Is Greedy?

Greedy is an algorithmic technique which makes locally optimal choices at each step with the hope of finding a global optimum solution. It operates on the principle of "taking the best option now" without considering the long-term consequences. This is the "greedy" part.

### Key Concepts

N/A

### Pros and Cons

#### Pros

* Usually **simple** and easy to implement.
* **Efficient** in time complexity, often providing quick solutions, as they don't reconsider previous choices, as they make decisions based on current information without looking ahead.

#### Cons

* May not find the best possible solution, as they focus on local optimizations and may miss better solutions that require considering a broader context.  
  So, they're **not applicable** to problems where the greedy choice does not lead to an optimal solution.

### Usecases

Most of the problems where greedy algorithms work follow these two properties:

1. **Greedy choice property**: This property states that choosing the best possible option at each step will lead to the best overall solution. If this is not true, a greedy approach may not work.
2. **Optimal substructure**: This means that you can break the problem down into smaller parts, and solving these smaller parts by making greedy choices helps solve the overall problem.

Dynamic Programming vs Greedy:

DP also works when a problem has Optimal substructure but it also requires Overlapping subproblems.

## How Greedy Works

### Steps

1. Start with the initial state of the problem. This is the **starting point** from where you begin making choices.
2. Evaluate **all possible choices** you can make from the current state. Consider all the options available at that specific moment.
3. Choose the option that **seems** **best at that moment**, regardless of future consequences.
4. Move to the new state based on your chosen option. This becomes your **new starting point** for the next iteration.
5. **Repeat** **steps 2-4** until you reach the goal state or no further progress is possible. Keep making the best local choices until you reach the end of the problem or get stuck.

### Time Complexity

### Space Complexity

## Examples

### [Easy] Minimum Number of Coin

**Problem**: Given a set of coins with values [1, 2, 5, 10]. Give minimum number of coin to change for 39.

**Steps**:

1. Start with the **largest coin** value that is less than or equal to the amount to be changed. In this case, the largest coin less than or equal to 39 is 10.
2. **Subtract** the largest coin value from the amount to be changed, and add the coin to the solution.

In this case, subtracting 10 from 39 gives 29, and we add one 10-coin to the solution.

1. Repeat steps 1 and 2 until the amount to be changed becomes 0.

The final chosen coin list will be [10, 10, 10, 5, 2, 2]. So, the minimum number of coins to change for 39 is 6.

**When Greedy fails**:

Given a set of coins with values [18, 1, 10]. Give minimum number of coin to change for 20. If we apply Greedy, then the final chosen coin list will be [18, 1, 1]. So, the minimum number of coins is 3. But in fact, the chosen coin list should be [10, 10], meaning the minimum number of coins should be 2.

**Implementation**:

|  |  |
| --- | --- |
| #include <iostream>  #include <algorithm>  using namespace std;    int findCoinCount(int arr[], int size, int change) {      int count = 0;      int curChange = change;      for (int i = 0; i < size; i++) {          if (curChange == 0) {              break;          } else if (curChange < 0) {              cout << "FAIL" << endl;              break;          }          int curMax = arr[i];            while (curChange >= curMax) {              count++;              curChange -= curMax;          }      }      return count;  }    int main() {      int arr[] = {1, 2, 5, 10};      int change = 39;      int size = sizeof(arr)/sizeof(arr[0]);      std::sort(arr, arr + size, std::greater<int>());      int count = findCoinCount(arr, size, change);      cout << "Max coin for change " << change << " is " << count << endl;      return 0;  } | Max coin for change 39 is 6 |

**Time complexity**: O(n) + O(nlogn)

**Space complexity**:

## Practices

# Graph Technique

## What Is Graph Technique?

Graph algorithm technique solves problem by manipulating and analyzing graph data structure.

## How Graph Technique Works

Check section *Graph* in *Tutorials\Data Structures - Algorithms\Data Structure Tutorial.docx.*

## Examples

## Practices

# Two-Pointers Technique

## What Is Two-Pointers Technique?

Two-Pointers is an algorithmic technique that is used to solve problems **related to arrays**. It involves **using two pointers (or indices), started from the corners (either left, right, or medium) of the array, to traverse it** in a way that allows for efficient searching, sorting, or processing.

## Examples

### [Easy] Find Pair of a Given Sum from a Sorted Array

**Problem**:

Given a sorted array arr (sorted in ascending order) and a target, find if there exists any pair of elements (arr[i], arr[j]) such that their sum is equal to the target.

**Examples**:

1. Input: arr[] = {10, 20, 35, 50}, target =70

Output: Yes

Explanation: There is a pair (20, 50) with given target.

1. Input: arr[] = {10, 20, 30}, target =70

Output : No

Explanation: There is no pair with sum 70.

**Steps**:

* Initialize: left = 0, right = n – 1
* Run a loop while left < right, do the following inside the loop:
  + Compute current sum, sum = arr[left] + arr[right]
  + If the sum equals the target, we’ve found the pair.
  + If the sum is less than the target, move the left pointer to the right to increase the sum.
  + If the sum is greater than the target, move the right pointer to the left to decrease the sum.

**Time complexity**: O(n) as the loops runs at most n times. We either increase left, or decrease right or stop the loop.

**Space complexity**: O(1)

**Implementation**:

|  |  |
| --- | --- |
| #include <iostream>  #include <utility>  using namespace std;    std::pair<int, int> findPair(int arr[], int size, int sum) {      std::pair<int, int> pair{-1, -1};      int leftIdx = 0;      int rightIdx = size - 1;      int tmpSum = 0;      while (leftIdx < rightIdx) {          tmpSum = arr[leftIdx] + arr[rightIdx];          if (tmpSum < sum) {              leftIdx++;          } else if (tmpSum > sum) {              rightIdx--;          } else {              pair.first = arr[leftIdx];              pair.second = arr[rightIdx];              break;          }      }      return pair;  }    int main() {      int arr[] = {10, 20, 23, 30, 35, 40, 50};      int sum = 75;      auto pair = findPair(arr, sizeof(arr)/sizeof(arr[0]), sum);      cout << pair.first << ", " << pair.second << endl;      return 0;  } | 35, 40 |

### [Medium] Trapping Rain Water

**Problem**: Given an array of n non-negative integers arr[] representing an elevation bar chart, where the width of each bar is 1, compute how much water it can trap after rain.

**Examples**:

1. Input: arr[] = [2, 1, 0, 3, 2, 1, 4, 1, 2]

Output: 7

Explanation: We trap (2 – 1) + (2 – 0) + (2 - 1) + (3 - 2) + (3 - 1) = 7

1. Input: arr[] = [3, 0, 2, 0, 4]

Output: 7

Explanation: We trap 0 + 3 + 1 + 3 + 0 = 7.

1. Input: arr[] = [4, 0, 3, 0, 1, 2]

Output: 6

Explanation: We trap 3 + 2 + 1 = 6.

1. Input: arr[] = [1, 2, 3, 4]

Output: 0

Explanation: We cannot trap water as there is no height bound on both sides

**Steps**:

|  |  |  |
| --- | --- | --- |
| Case 1  (2 bounds) | Arr[] = [3, 0, 2] | Water = 2 – 0 = 2  Idea: Calculate trapped water based on the lower bound, which is 2, regarding item is iterated by pointer *left* or *right*. |
| Case 2  (2 bounds) | Arr[] = [2, 0, 3] | Water = 2 – 0 = 2  Idea: Calculate trapped water based on the lower bound, which is 2, regarding item is iterated by pointer *left* or *right*. |
| Case 3  (0 bound) | Arr[] = [1, 2, 3] | Water = 0  Idea: If no upper bounds, no water |
| Case 4  (0 bound) | Arr[] = [2, 2, 2] | Water = 0  Idea: If no upper bounds, no water |
| Case 5  (3 bounds) | Arr[] = [2, 1, 3, 1, 4] | Water = (2 – 1) + (3 – 1)  Idea: Iterate pointer *left* only because upper bound of *left* (2, then 3) is smaller than upper bound of *right* (4).  So **the** **two pointers are not moved inward every iteration**. Instead, either one of them is moved depending on which side has the smaller upper bound. |
| Case 6  (n bounds) | Arr[] = [2, 1, 0, 3, 2, 1, 4, 1, 2] | Water = (2 – 1) + (2 – 0) + (2-1) + (3-2) + (3-1) = 7  Idea: Iterate pointer *left* and then iterate pointer *right*, and then iterate pointer *left.* |

**Time complexity**: O(n)

**Space complexity**: O(1)

**Implementation**:

|  |  |
| --- | --- |
| #include <iostream>  #include <vector>  using namespace std;    int findTrapperWater(std::vector<int> arr, int l, int r) {  int lmax = arr[l];  int rmax = arr[r];  int sum = 0;    while (l < r) {  if (lmax <= rmax) {  l += 1;  if (arr[l] > lmax) {  lmax = arr[l];  } else {  sum += min(lmax, rmax) - arr[l];  }  } else {  r -= 1;  if (arr[r] > rmax) {  rmax = arr[r];  } else {  sum += min(lmax, rmax) - arr[r];  }  }  }    return sum;  }    int main() {  std::vector<std::vector<int>> vecs = {  {2, 1, 0, 3, 2, 1, 4, 2, 3}, // output: 7  {3, 0, 2, 0, 4}, // output: 7  {4, 0, 3, 0, 1, 2}, // output: 6  {4, 0, 5, 3, 0, 1, 2}, // output: 7  {1, 2, 3} // output: 0  };    for (const auto& vec : vecs) {  int left = 0;  int right = vec.size() - 1;  int trappedWater = findTrapperWater(vec, left, right);  cout << trappedWater << endl;  }  return 0;  } |  |

## Practices

# Window Sliding Technique

## What Is Window Sliding Technique?

## How Window Sliding Technique Works?

### Steps

### Time Complexity

O(n)

### Space Complexity

O(1)

## Examples

### [Easy] Maximum Sum of a Subarray with K Elements

**Problem**

Given an *array* and an integer *k*, we need to calculate the maximum sum of a subarray having size exactly *k***.**

1. Input: arr[] = [100, 200, **300, 400**], k = 2

Output: 700

We get maximum sum by considering the subarray [300, 400]

1. Input: arr[] = [1, **4, 2, 10, 23**, 3, 1, 0, 20], k = 4

Output: 39

We get maximum sum by adding subarray [4, 2, 10, 23] of size 4.

1. Input: arr[] = [2, 3], k = 3

Output: Invalid

There is no subarray of size 3 as size of whole array is 2.

**Solution**

Consider an array *arr[] = [5, 2, -1, 0, 3]* and value of *k = 3*. The below representation shows how the window slides over the array.







So what we did is:

* Compute the sum of the first *k* elements out of n terms using a linear loop and store the sum in variable *window\_sum*.
* Then we will traverse linearly over the array till it reaches the end and simultaneously keep track of the maximum sum.
* To get the current sum of a block of *k* elements just subtract the first element from the previous block and add the last element of the current block.

Time complexity: O(n)

Space complexity: O(1)

**Snippet**

i = 0

for i to i+k:

curSum =+ arr[i]

maxSum = curSum

for i=1 to arr.size-k:

curSum = curSum – arr[i] + arr[i+k]

maxSum = max(maxSum, curSum)

**Implementation**

|  |  |
| --- | --- |
|  |  |

## Practices

# Pattern Search Algorithms

## What Is Pattern Search?

Pattern search is a collection of algorithms that involve **searching occurrences of a specific pattern (substring) within a larger text (string).** These algorithms are particularly useful in applications such as text processing, DNA sequencing, and data mining.

## Types

### Naive

This is the **simplest approach**, where you check for the pattern at every position in the text.

Time complexity: O(N\*M), where M is the length of the pattern and N is the length of the text.

Space complexity: O(1)

### Rabin-Karp

This compares string’s **hash values**, rather than the strings themselves. Instead of travelling through and compare every character, it only compares characters when it finds potential matches.

The steps are as follows:

1. Define a hash function:

* This step is a pain, but the most crucial step to determine success of the algorithm.
* We need to think of the hash function very carefully.

If the hash function is too simple, then it cannot handle collision case like "CAB" and "BAC" as both yield the same hash value if only considering the sum of ASCII values.

If the hash function is too complex, then it lead to implementation difficulty and calculation overhead.

Here we choose our hash function as:

*h(string) = Σ(text[i] x d m - r) mod q*

Where:

*i*: Index of the character in the pattern or the text. We'll use need this index to get the actual character from the string array.

*n*: Length of the text.

m: Length of the pattern or substring. They're actually the same.

*r*: Rank of the character in the pattern or substring. Basically the rank is *1*, *2*, … *m*.

*q*: Modulus. Should be a prime number as it will help in avoiding overflow issues and should be big as it will help in avoiding collision issues.

With this hash function, character values can be compared via *text[i]*, such as ASCII(A) != ASCII(B). Character orders can be compared via *r*, such as "CAB" != "BAC". Character numbers can be compared via *m*, such as len("ABC") != len("ABCD"). Also, each character's contribution is weighted by *d* m – r to ensure it's big enough before fitting it within the bounds by *mod*.

1. Calculate the hash value of the pattern using the above hash function.
2. Start iterating from the start of the string:
   * Calculate the hash value of the current substring having the same length as the pattern using the above hash function.
   * If the hash value of the current substring and the pattern are same, it means the pattern might be there. So, need to check if the substring is same as the pattern:
     + If they are same, store the starting index as a valid answer.
     + Otherwise, continue for the next substrings. We can re-calculate the hash value of the next substring by the same hash function as above, or by applying Window Sliding technique (remove the previous character from and include the next character to the old hash value) to reduce some overheads.

The hash function for Window Sliding is:

*h(string) = ((Σ(text[i] x d m - r) – text[i\_previous] x d m – 1) x d + text[i\_next]) mod q*

Formula derivation:

1. When you remove the contribution of the outgoing character, you need to subtract its contribution from the hash:

*Σ(text[i] x d m - r)* ***– text[i\_previous] x d m – 1***

2. Then you multiply the entire hash by *d* to shift the remaining characters to the left (this effectively increases their rank):

(*Σ(text[i] x d m - r) – text[i\_previous] x d m – 1)* ***x d***

3. Now add the contribution of the new character that has entered the window:

*(Σ(text[i] x d m - r) – text[i\_previous] x d m – 1) x d* ***+ text[i\_next]***

4. Finally, take the result modulo *q* to ensure it fits within the bounds:

*((Σ(text[i] x d m - r) – text[i\_previous] x d m – 1) x d + text[i\_next])* ***mod q***

Illustration:

The text is "AA**CAB**AC**CAB**" and the pattern is "**CAB**".

The number of characters in the text is n = 10. The number of characters in the pattern is m = 3.

The leftmost character has rank r = 1 and the rightmost one has rank r = 10.

The chosen modulus p = 13.

h(ptrn) = ((ptrn[0] x 102) + (ptrn[1] x 101) + (ptrn[2] x 100)) mod 13 // ASCII(C) = 67, ASCII(A) = 65, ASCII(B) = 66

= (67 x 102) + (65 x 101) + (66 x 100)) mod 13

= 6

h(AAC) = ((text[0] x 102) + (text[1] x 101) + (text[2] x 100)) mod 13

= (65 x 102) + (65 x 101) + (67 x 100)) mod 13

= 7217 mod 13

= 2

h(ACA) = ((text[1] x 102) + (text[2] x 101) + (text[3] x 100)) mod 13 // Way 1

= (65 x 102) + (67 x 101) + (65 x 100)) mod 13

= 7235 mod 13

= 7

h(ACA) = (h(AAC) – text[0] x 102) x 10+ text[3]) mod 13 // Way 2

= ((7217 - 65 x 102) x 10+ 65) mod 13

= 7235 mod 13

= 7

h(CAB) = ((text[2] x 102) + (text[3] x 101) + (text[4] x 100)) mod 13 // Way 1: Found potential match. Then with character-by-character comparion, confirm this is a match

= (67 x 102) + (65 x 101) + (66 x 100)) mod 13

= 7416 mod 13

= 6

h(CAB) = (h(ACA) – text[1] x 102) x 10+ text[4]) mod 13 // Way 2

= ((7235 - 65 x 102) x 10+ 66) mod 13

= 7416 mod 13

= 6

h(ABA) = 10

h(BAC) = 11 // Notice that h(CAB) != h(BAC), means we already handled case "CAB" and "BAC"

h(ACC) = 9

h(CCA) = 12

h(CAB) = 6 // Found another potential match. Then with character-by-character comparion, confirm this is a match

**Time complexity**: O(n+m), but can be O(nm) in worst case. The worst case occurs when all characters of pattern and text are the same as the hash values of all the substrings match with the hash value of pattern.

**Space complexity**: O(1), regardless of the size of the input text and pattern. This is because the algorithm only needs to store a few variables that are updated as the algorithm progresses through the text and pattern. Specifically, the algorithm needs to store the hash value of the pattern, the hash value of the current window in the text, and a few loop counters and temporary variables. Since the size of these variables is fixed, the space complexity is constant.

### KMP

KMP stands for Knuth-Morris-Pratt.

This algorithm compares character by character from left to right. Whenever a mismatch occurs, it uses a preprocessed table called **LPS Table** (Longest Prefix-Suffix) to skip unneccessary comparison while matching.

Time complexity: O(N + M), where M is the length of the pattern and N is the length of the text.

For me, this algorithm is hard to understand. <https://www.geeksforgeeks.org/kmp-algorithm-for-pattern-searching/>

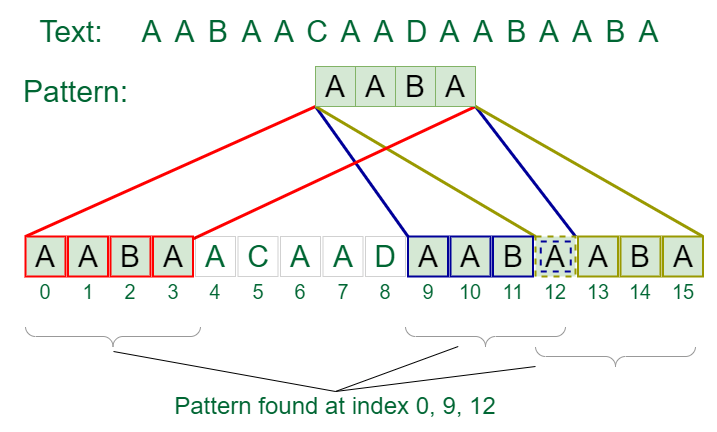
### Z

### Aho-Corasick

## Examples

### [Easy] Find Substrings from Text

**Problem**: Find all substring "AABA" from the text "AABAACAADAABAABAA" and return the start index of each substring.



**Implementation – Naive Pattern Search**:

|  |  |
| --- | --- |
| #include <iostream>  using namespace std;    void search(string text, string pattern) {  int textSize = text.size();  int patternSize = pattern.size();    for (int i = 0; i < textSize - patternSize; i++) {  for (int j = 0; j < patternSize; j++) {  if (text[i + j] != pattern[j]) {  break;  }  if (j == patternSize - 1) {  cout << "Found pattern at index: " << i << endl;  }  }  }  }    int main() {  string text = "AABAACAADAABAABAA";  string pattern = "AABA";    search(text, pattern);    return 0;  } | Found pattern at index: 0  Found pattern at index: 9  Found pattern at index: 12 |

**Implementation – Rabin Karp**

Code is not clean, but work with O(N + M)

|  |  |
| --- | --- |
| #include <iostream>  #include <utility>  #include <math.h>  using namespace std;    #define MODULUS 13    pair<int, int> calHashvalue(string substr, int substrSize, int textSize) {      int sum = 0;      for (int i = 0; i < substr.size(); i++) {          sum += substr[i] \* pow(textSize, substrSize - (i + 1));      }      int hashValue = sum % MODULUS;      cout << "Sum: " << sum << ". Hash value: " << hashValue << endl;        pair<int, int> p(sum, hashValue);      return p;  }    pair<int, int> slideWindow(int previousSum, char previousChar, char nextChar, int substrSize, int textSize) {      int sum = ((previousSum - (previousChar \* pow(textSize, substrSize - 1))) \* textSize) + nextChar;      int hashValue = sum % MODULUS;      cout << "Sum: " << sum << ". Hash value: " << hashValue << endl;        pair<int, int> p(sum, hashValue);      return p;  }    void search(string text, string pattern) {      int textSize = text.size();      int patternSize = pattern.size();        // Hash value of pattern      int patternHashValue = calHashvalue(pattern, patternSize, textSize).second;        // Hash value of first window      string firstWindow = "";      for (int i = 0; i < pattern.size(); i++) {          firstWindow += text[i];      }      pair<int, int> firstwindow = calHashvalue(firstWindow, patternSize, textSize);      int windowSum = firstwindow.first;      int windowHashValue = firstwindow.second;        // Check if the pattern and the first window are matched      if (windowHashValue == patternHashValue) {          bool found = true;          cout << "Found potential match" << endl;          for (int j = 0; j < pattern.size(); j++) {              if (text[j] != pattern[j]) {                  found = false;                  break;              }          }          if (found) {              cout << "Found match at index: 0" << endl;          }      }        // Check if the pattern and the other windows are matched      for (int i = 1; i < text.size() - pattern.size() + 1; i++) {          pair<int, int> p = slideWindow(windowSum, text[i-1], text[i+pattern.size()-1], patternSize, textSize);          windowSum = p.first;          windowHashValue = p.second;            if (windowHashValue == patternHashValue) {              bool found = true;              cout << "Found potential match" << endl;              for (int j = 0; j < pattern.size(); j++) {                  if (text[j+i] != pattern[j]) {                      found = false;                      break;                  }              }              if (found) {                  cout << "Found match at index: " << i << endl;              }          }      }  }    int main() {      string text = "AACABACCAB";      string pattern = "CAB";        search(text, pattern);        return 0;  } | Sum: 7416. Hash value: 6  Sum: 7217. Hash value: 2  Sum: 7235. Hash value: 7  Sum: 7416. Hash value: 6  Found potential match  **Found match at index: 2**  Sum: 7225. Hash value: 10  Sum: 7317. Hash value: 11  Sum: 7237. Hash value: 9  Sum: 7435. Hash value: 12  Sum: 7416. Hash value: 6  Found potential match  **Found match at index: 7** |

## Practices

# Searching Algorithms

## Linear Search – O(n)

**Problem**

Given an array arr[] of n elements, write a function to search a given element x in arr[].

**Examples**

Input: arr[] = {10, 20, 80, 30, 60, 50, 110, 100, 130, 170}; x = 110;

Output: 6 -> Element x is at index 6

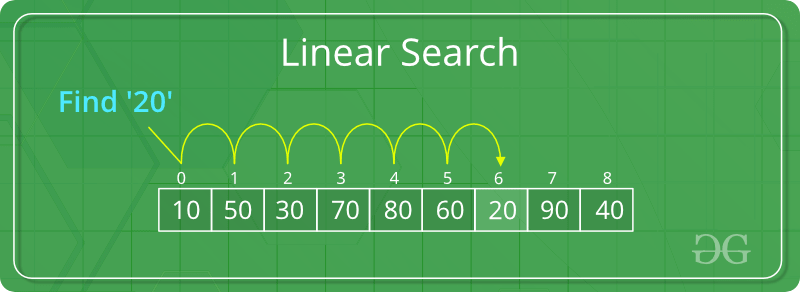
Input: arr[] = {10, 20, 80, 30, 60, 50, 110, 100, 130, 170}; x = 175;

Output: -1 -> Element x is not in arr[].

**Algorithm**

Start from the leftmost element of arr[] and one by one compare x with each element of arr[]:

1. If x matches with an element, return the index.
2. If x doesn’t match with any of elements, return -1.



**Time Complexity**:O(n)

**Code**

<https://www.geeksforgeeks.org/linear-search/>

## Binary Search – O(1) or O(logn)

**Problem**

Given a sorted array arr[] of n elements, write a function to search a given element x in arr[].

**Examples**

Input: arr[] = {10, 20, 30, 40, 50, 60}; x = 20;

Output: 1 -> Element x is at index 1

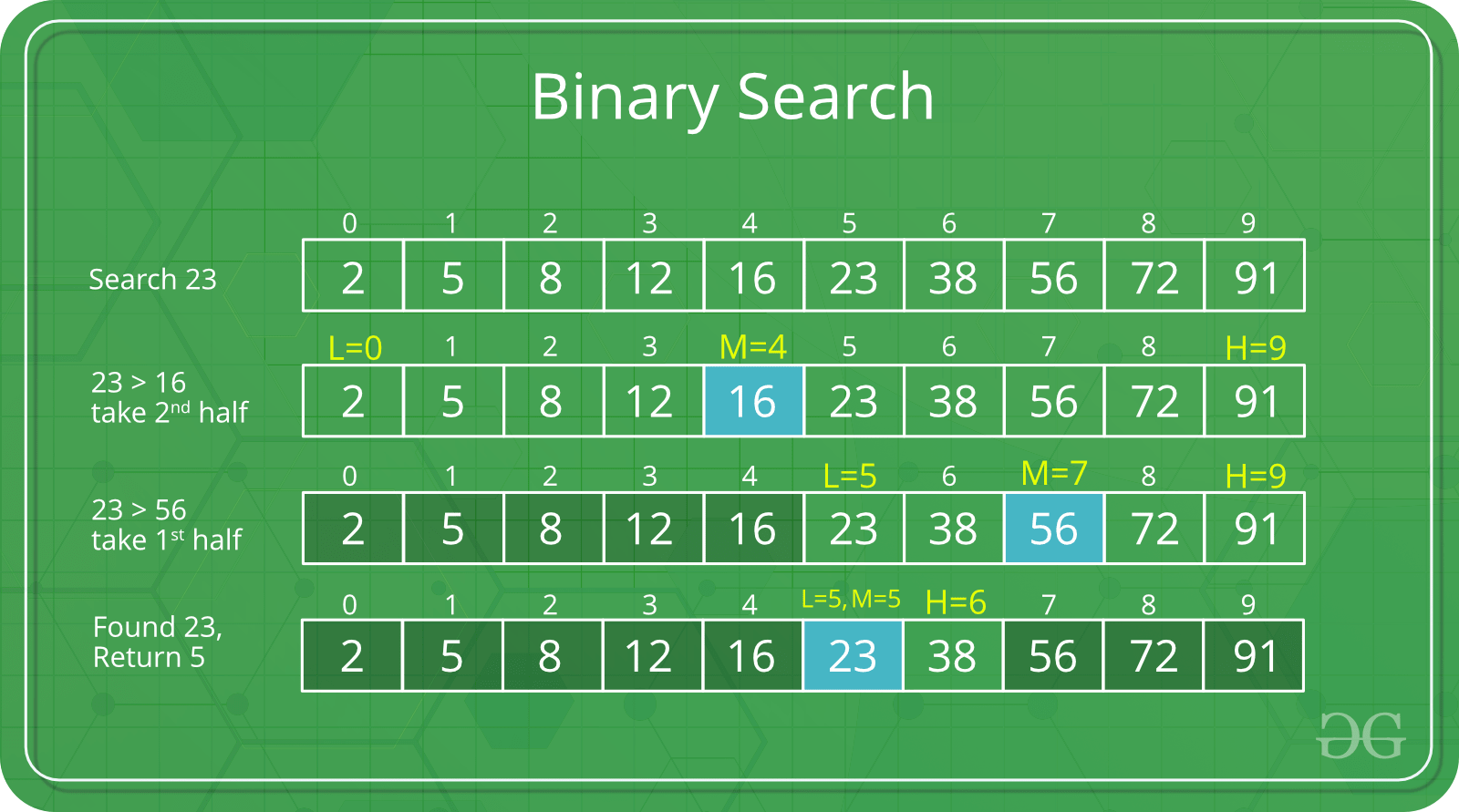
Input: arr[] = {10, 20, 30, 40, 50, 60}; x = 70;

Output: -1 -> Element x is not in arr[]

**Algorithm**

Compare x with the middle element in the array:

1. If x matches with middle element, return the mid index.
2. Else:
   1. If x is greater than the mid element, then x can only lie in right half subarray after the mid element. So we recur for right half.
   2. Else (x is smaller) recur for the left half.



**Time Complexity**: O(1) if using iteration. Or O(logn) if using recursion.

**Code**

<https://www.geeksforgeeks.org/binary-search/>

## Ternary Search

## Jump Search – O(√n)

## Interpolation Search

## Exponential Search

## Fibonacci Search

## The Ubiquitous Binary Search

# Sorting Algorithms

Visualization of the most famous sorting algorithms: <https://www.toptal.com/developers/sorting-algorithms>

## Selection Sort – O(n2)

**Problem**

Given an array **arr[]** of **n** elements, write a function to sort this array in an ascending order.

**Algorithm**

Repeatedly find the minimum element (considering ascending order) from unsorted part. Each time, the found element from the unsorted array is picked and moved to the sorted subarray (one by one, after each other).

The algorithm maintains two subarrays in a given array.

1. The subarray which is already sorted.
2. The remaining subarray which is unsorted.

Example: arr[] = { 64 25 12 11 }

// Find the minimum element in { 64 25 12 11 } and place it at the beginning of { 64 25 12 11 }:

**11** 25 12 64

// Find the minimum element in { 25 12 64 } and place it at the beginning of { 25 12 64 }:

11 **12** 25 64

// Find the minimum element in { 25 64 } place it at the beginning of { 25 64 }:

11 12 **25** 64

**Time Complexity**: O(n2) as there are two nested loops.

**Implementation**:

#include <bits/stdc++.h>

using namespace std;

void selectionSort(int arr[], int n)

{

    int i, j, min\_idx;

    for (i = 0; i < n - 1; i++) {

        // Find the minimum element in unsorted array

        min\_idx = i;

        for (j = i + 1; j < n; j++) {

            if (arr[j] < arr[min\_idx])

                min\_idx = j;

        }

        // Swap the found minimum element with the first element

        if (min\_idx != i)

            swap(arr[min\_idx], arr[i]);

    }

}

void printArray(int arr[], int size)

{

    for (int i = 0; i < size; i++) {

        cout << arr[i] << " ";

    }

}

int main()

{

    int arr[] = { 64, 25, 12, 22, 11 };

    int n = sizeof(arr) / sizeof(arr[0]);

    selectionSort(arr, n);

    cout << "Sorted array: \n";

    printArray(arr, n);

    return 0;

}

## Bubble Sort – O(n2)

**Problem**

Given an array **arr[]** of **n** elements, write a function to sort this array in an ascending order.

**Algorithm**

Repeatedly swap adjacent elements in the array if they are in wrong order.

**Example**:

First pass:

{ **5 1** 4 2 8 } –> { **1 5** 4 2 8 }, compares the first two elements and swaps because 5 > 1.

{ 1 **5 4** 2 8 } –> { 1 **4 5** 2 8 }, swaps because 5 > 4

{ 1 4 **5 2** 8 } –> { 1 4 **2 5** 8 }, swaps because 5 > 2

{ 1 4 2 **5 8** } –> { 1 4 2 **5 8** }, because these elements are already in order, does not swap them.

Second pass:

{ **1 4** 2 5 8 } –> { **1 4** 2 5 8 }

{ 1 **4 2** 5 8 } –> { 1 **2 4** 5 8 }, swap because 4 > 2

{ 1 2 **4 5** 8 } –> { 1 2 **4 5** 8 }

{ 1 2 4 **5 8** } –> { 1 2 4 **5 8** }

Now, the array is already sorted, but our algorithm does not know if it is completed. It needs one more pass without any swap to know it is sorted.

Third pass:

{ 1 2 4 5 8 } –> { 1 2 4 5 8 }

{ 1 2 4 5 8 } –> { 1 2 4 5 8 }

{ 1 2 4 5 8 } –> { 1 2 4 5 8 }

{ 1 2 4 5 8 } –> { 1 2 4 5 8 }

**Time Complexity**: O(n2)

**Implementation:**

#include <bits/stdc++.h>

using namespace std;

void bubbleSort(int arr[], int n)

{

    int i, j;

    bool swapped;

    for (i = 0; i < n - 1; i++) {

        swapped = false;

        for (j = 0; j < n - i - 1; j++) {

            if (arr[j] > arr[j + 1]) {

                swap(arr[j], arr[j + 1]);

                swapped = true;

            }

        }

        if (swapped == false)

            break;

    }

}

void printArray(int arr[], int size)

{

    int i;

    for (i = 0; i < size; i++)

        cout << " " << arr[i];

}

int main()

{

    int arr[] = { 64, 34, 25, 12, 22, 11, 90 };

    int n = sizeof(arr) / sizeof(arr[0]);

    bubbleSort(arr, n);

    cout << "Sorted array: \n";

    printArray(arr, n);

    return 0;

}

## Merge Sort - O(nLogn)

**Problem**

Given an array **arr[]** of **n** elements, write a function to sort this array in an ascending order.

**Algorithm**

|  |  |
| --- | --- |
| Merge sort is a recursive algorithm based on Divide & Conquer technique.  It continuously splits the array in half until it cannot be further divided i.e., the array has only one element left (an array with one element is always sorted). Then the sorted subarrays are merged into one sorted array.  Example: arr[] = {38, 27, 43, 10}  Dividing step:  1st dividing: {38, 27, 43, 10} -> **{38, 27} and {43, 10}**  2nd dividing: {38, 27} -> **{38} and {27}**; {43, 10} -> **{43} and {10}**  Stop dividing because now no longer be divided more.  Merging step:  Sorted subarrays are merged together: {27, 38}, {10, 43}  Continuing mergeing until the sorted array is built from the smaller subarrays.: {10, 27, 38, 43} |  |

**Time complexity**: O(nlogn) – The array is divided into halves O(logn), and merging takes linear time O(n).

**Space complexity**: O(n) – Additional space is used for the temporary arrays during the merge process.

**Implementation:**

#include <iostream>

#include <vector>

#include <algorithm>

using namespace std;

void print(int arr[], int size) {

for (int i = 0; i < size; i++) {

printf("%d ", arr[i]);

}

printf("\n");

}

// This function resolves subproblems and merge them. It takes O(n)

// So the total time complexity for the whole program is O(n logn)

void sort(int arr[], int midIdx, int startIdx, int endIdx) { // O(n)

int n1 = midIdx - startIdx + 1;

int n2 = endIdx - midIdx;

// Create temp vectors

vector<int> L(n1), R(n2);

// Copy data to temp vectors L[] and R[]

for (int i = 0; i < n1; i++)

L[i] = arr[startIdx + i];

for (int j = 0; j < n2; j++)

R[j] = arr[midIdx + 1 + j];

int i = 0, j = 0;

int k = startIdx;

// Merge the temp vectors back into arr[startIdx..endIdx]

while (i < n1 && j < n2) {

if (L[i] <= R[j]) {

arr[k] = L[i];

i++;

}

else {

arr[k] = R[j];

j++;

}

k++;

}

// Copy the remaining elements of L[], if there are any

while (i < n1) {

arr[k] = L[i];

i++;

k++;

}

// Copy the remaining elements of R[], if there are any

while (j < n2) {

arr[k] = R[j];

j++;

k++;

}

}

void mergesort(int arr[], int startIdx, int endIdx, string depth) { // O(logn)

printf("%sCall sort(%d, %d)\n", depth.c\_str(), startIdx, endIdx);

if (startIdx == endIdx) {

return;

}

int midIdx = (endIdx + startIdx)/2;

mergesort(arr, startIdx, midIdx, depth + " ");

mergesort(arr, midIdx + 1, endIdx, depth + " ");

sort(arr, midIdx, startIdx, endIdx);

}

int main() {

int arr[] = { 12, 11, 13, 5, 6, 7 };

int size = sizeof(arr)/sizeof(arr[0]);

int startIdx = 0;

int endIdx = size - 1;

mergesort(arr, startIdx, endIdx, "");

print(arr, size);

return 0;

}

## Quick Sort - O(nLogn)

**Problem**

Given an array **arr[]** of **n** elements, write a function to sort this array in an ascending order.

**Algorithm**

|  |  |
| --- | --- |
| Quick sort is a recursive algorithm based on Divide & Conquer technique.It picks an element as pivot and partitions the array around that pivot. The pivot can be picked in different ways:   * Always pick first element as pivot. * Always pick last element as pivot (illustration below). * Pick a random element as pivot. * Pick median as pivot.   The key process in quick sort is partition: put all smaller elements (than the pivot) before the pivot (if ascending order), and put all greater elements (than the pivot) after the pivot. All this should be done in linear time. Then place the pivot at its correct place. | quicksort |

For example:

arr[] = {10, 80, 40, 90, 30, 50, 70}

Index: 0 1 2 3 4 5 6

start = 0, end = 6, pivot = arr[end] = 70

Index of smaller element: i = -1

Traverse elements from j = start to end-1

j = 0 : Because arr[j] <= pivot, do i++ and swap(arr[i], arr[j])

i = 0, arr[] = {10, 80, 40, 90, 30, 50, 70} // No change as i and j are same

j = 1 : Because arr[j] > pivot, do nothing

j = 2 : Because arr[j] <= pivot, do i++ and swap(arr[i], arr[j])

i = 1, arr[] = {10, **40**, **80**, 90, 30, 50, 70} // We swapped 80 and 40

j = 3 : Because arr[j] > pivot, do nothing

j = 4 : Because arr[j] <= pivot, do i++ and swap(arr[i], arr[j])

i = 2, arr[] = {10, 40, **30**, 90, **80**, 50, 70} // We swapped 80 and 30

j = 5 : Because arr[j] <= pivot, do i++ and swap arr[i] with arr[j]

i = 3, arr[] = {10, 40, 30, **50**, 80, **90**, 70} // We swapped 90 and 50

Loop ends because j = end-1.

We place pivot at correct position by swapping arr[i+1] and arr[end] (or pivot)

arr[] = {10, 40, 30, 50, **70**, 90, **80**} // We swapped 80 and 70

Now 70 is at its correct place. All elements smaller than 70 are before it and all elements greater than 70 are after it.

But we have not done sorting. For each sub-array {10, 40, 30, 50} and {90, 80}, we partition again (and might again and again …) until start > end.

The final result will be arr[] = {10, 30, 40, 50, 70, 80, 90}.

**Time Complexity**

Worst case: Θ(n2)

Best case: Θ(nLogn)

**Implementation:**

#include <bits/stdc++.h>

using namespace std;

int partition(int arr[], int low, int high)

{

    // Choose the rightmost element as pivot

    int pivot = arr[high];

    // Index of smaller element and indicates the right position of pivot found so far

    int i = low - 1;

    for (int j = low; j <= high - 1; j++) {

        // If current element is smaller than the pivot

        if (arr[j] < pivot) {

            i++;

            swap(arr[i], arr[j]);

        }

    }

    swap(arr[i + 1], pivot);

    return i + 1;

}

// low --> Starting index,

// high --> Ending index

void quickSort(int arr[], int low, int high)

{

    if (low < high) {

        // Partition the array and get the partitioning index

        int pi = partition(arr, low, high);

        // Recursively sort the sub-arrays

        quickSort(arr, low, pi - 1);

        quickSort(arr, pi + 1, high);

    }

}

void printArray(int arr[], int size)

{

    int i;

    for (i = 0; i < size; i++)

        cout << " " << arr[i];

}

int main()

{

    int arr[] = { 10, 7, 8, 9, 1, 5 };

    int n = sizeof(arr) / sizeof(arr[0]);

    quickSort(arr, 0, n - 1);

    cout << "Sorted array: " << endl;

    printArray(arr, n);

    return 0;

}

## Heap Sort - O(nLogn)

**Problem**

Given an array **arr[]** of **n** elements, write a function to sort this array in an ascending order.

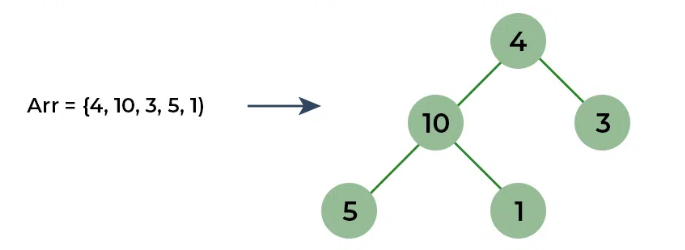
**Algorithm**

Steps:

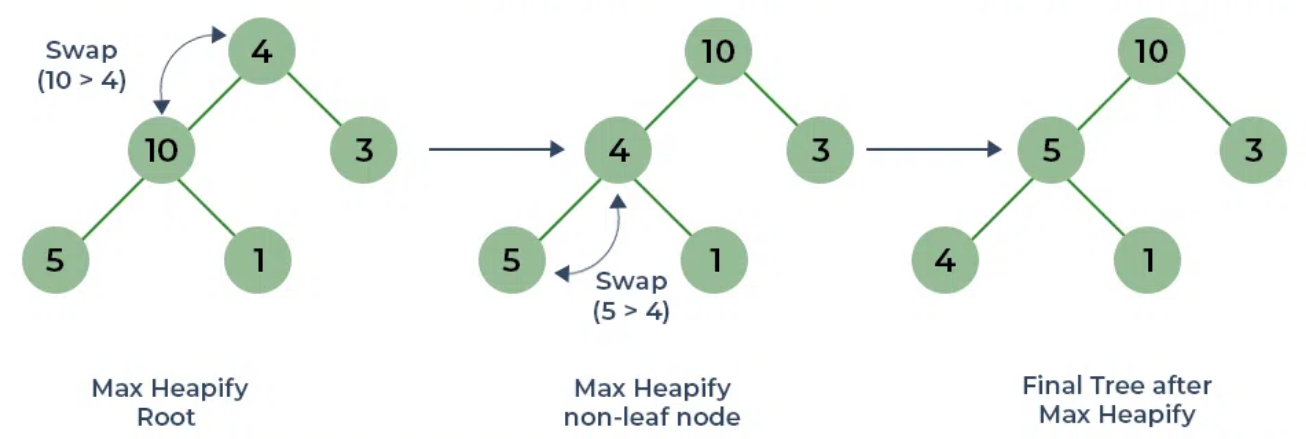
1. **Build heap**: Build a max heap from the input array.
   * Starting from the last non-leaf node, heapify each node in reverse order. Heapify is the process of adjusting the heap to maintain the max heap property.
   * Heapify compares the node with its children and swaps the node with the larger child if necessary. This process is recursively applied to the affected child until the max heap property is satisfied.
2. **Heapify and extract**: After building the max heap, the largest element (at the root) is in the correct position. We swap it with the last element of the heap (which is the last element of the array) and reduce the size of the heap by 1.
   * Swap the root with the last element of the heap.
   * Reduce the heap size by 1.
   * Heapify the root element to restore the max heap property.
3. **Repeat**: Repeat step 2 until the heap size is 1. This process moves the largest elements to the end of the array in ascending order.
4. **Sorted array**: The array is now sorted in ascending order.

Example:

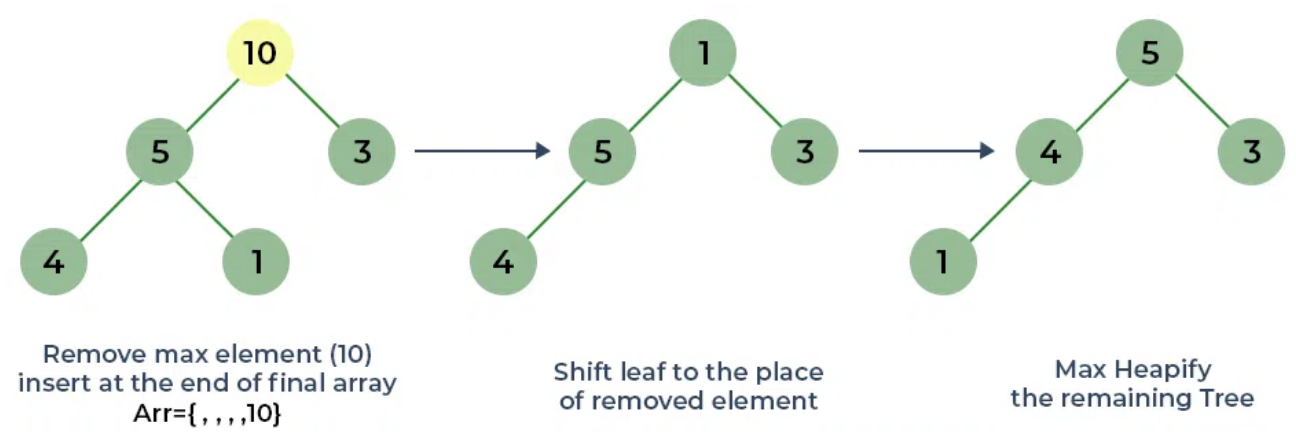
1. Initial array: {4, 10, 3, 5, 1}
2. Build a binary tree:



1. Build max heap:
   * Convert the array into a max heap: [10, 5, 3, 4, 1]



1. Heapify and extract:
   * Swap the root (10) with the last element (1): [**1**, 5, 3, 4, **10**]
   * Reduce the heap size by 1.
   * Heapify the root (1) to restore the max heap property: [5, 4, 3, 1, 10]



1. Heapify and extract:
   * Swap the root (5) with the last element (1): [1, 4, 3, 5, 10]
   * Reduce the heap size by 1.
   * Heapify the root (1) to restore the max heap property: [4, 1, 3, 5, 10]
2. Heapify and extract:
   * Swap the root (4) with the last element (3): [3, 1, 4, 5, 10]
   * Reduce the heap size by 1.
   * Heapify the root (3) to restore the max heap property: [1, 3, 4, 5, 10]
3. Heapify and extract:
   * Swap the root (1) with the last element (1): [1, 3, 4, 5, 10]
   * Reduce the heap size by 1.
   * Heapify the root (1) to restore the max heap property: [3, 1, 4, 5, 10]
4. Heapify and extract:
   * Swap the root (3) with the last element (1): [1, 1, 4, 5, 10]
   * Reduce the heap size by 1.
   * Heapify the root (1) to restore the max heap property: [1, 1, 4, 5, 10]
5. The heap size is now 1, and the array is sorted: [1, 1, 4, 5, 10]

The final sorted array is [1, 1, 4, 5, 10].

**Time Complexity:**

**Implementation:**

# Bitwise Algorithms

Bitwise algorithms refer to *bit manipulation* which performs **operations on individual bits or bit patterns** within computer data. These bitwise operations are AND, OR, XOR, NOT, bit shifting, etc. They're usually faster and use less memory than regular arithmetic operations because they work directly with the *binary representation* of data.

## Bitwise Operations

|  |  |  |  |
| --- | --- | --- | --- |
| **Operator** | **Notation** | **Description** | **Truth Table Example** |
| **AND** | & | Compares each bit of two numbers.  If both bits are 1, the resulting bit is set to 1; otherwise, it is set to 0. | |  |  |  | | --- | --- | --- | | **A** | **B** | **A & B** | | 0 | 0 | 0 | | 0 | 1 | 0 | | 1 | 0 | 0 | | 1 | 1 | 1 | |
| **OR** | | | Compares each bit of two numbers.  If at least one of the bits is 1, the resulting bit is set to 1; otherwise, it is set to 0. | |  |  |  | | --- | --- | --- | | **A** | **B** | **A | B** | | 0 | 0 | 0 | | 0 | 1 | 1 | | 1 | 0 | 1 | | 1 | 1 | 1 | |
| **XOR** | ^ | Compares each bit of two numbers.  If the bits are different, the resulting bit is set to 1; if they are the same, it is set to 0. | |  |  |  | | --- | --- | --- | | **A** | **B** | **A ^ B** | | 0 | 0 | 0 | | 0 | 1 | 1 | | 1 | 0 | 1 | | 1 | 1 | 0 | |
| **NOT** | ~ | Inverts the bits of a number.  Each 0 becomes 1 and each 1 becomes 0. | |  |  | | --- | --- | | **A** | **~A** | | 0 | 1 | | 1 | 0 | |
| **NAND** |  |  |  |
| **NOR** |  |  |  |
| **Left Shift** | << | Shifts all bits of the number to the left by a specified number of positions. |  |
| * **Logical shift**: New bits on the right are filled with 0.   This is equivalent to multiplying the number by 2n (where n is the number of shifted positions). | |  |  |  | | --- | --- | --- | | **A** | **A << 1** | **A << 2** | | 0000 | 0000 | 0000 | | 0001 | 0010 | 0100 | | 0010 | 0100 | 1000 | | 0011 | 0110 | 1100 | | 1111 | 1110 | 1100 | |
| * **Arithmetic shift**: Similar to logical shift (New bits on the right are filled with 0). | |  |  |  | | --- | --- | --- | | **A** | **A << 1** | **A << 2** | | 0000 | 0000 | 0000 | | 0001 | 0010 | 0100 | | 0010 | 0100 | 1000 | | 0011 | 0110 | 1100 | | 1111 | 1110 | 1100 | |
| * **Rotate shift**: Moves bits around in a circular fashion.   Bits shifted out from the left are added back on the right. | |  |  |  | | --- | --- | --- | | **A** | **A << 1** | **A << 2** | | 0101 | 1010 | 0101 | | 0111 | 1110 | 1101 | | 1111 | 1110 | 1111 | |
| **Right Shift** | >> | Shifts all bits of the number to the right by a specified number of positions. |  |
| * **Logical shift**: New bits on the left are filled with 0. | |  |  |  | | --- | --- | --- | | **A** | **A >> 1** | **A >> 2** | | 0000 | 0000 | 0000 | | 1000 | 0100 | 0010 | | 0100 | 0010 | 0001 | | 1100 | 0110 | 0011 | | 1111 | 0111 | 0011 | |
| * **Arithmetic shift**: Similar to logical shift but preserves the sign of a signed integer when shifting right.   If the sign bit is 0 (positive number), new bits are filled with 0.  If the sign bit is 1 (negative number), new bits are filled with 1.  Note: The sign bit is the most significant bit (MSB). | **Sign bit is 0:**   |  |  |  | | --- | --- | --- | | **A** | **A >> 1** | **A >> 2** | | 0001 | 0000 | 0000 | | 0010 | 0001 | 0000 | | 0110 | 0011 | 0001 |   **Sign bit is 1:**   |  |  |  | | --- | --- | --- | | **A** | **A >> 1** | **A >> 2** | | 1011 | 1101 | 1110 | | 1100 | 1110 | 1111 | | 1111 | 1111 | 1111 | |
| * **Rotate shift**: Moves bits around in a circular fashion.   Bits shifted out from the right are added back on the left. | |  |  |  | | --- | --- | --- | | **A** | **A >> 1** | **A >> 2** | | 0101 | 1010 | 0101 | | 0111 | 1011 | 1101 | | 1111 | 1111 | 1111 | |

## Examples

### [Easy] Understand Bitwise Operations

|  |  |
| --- | --- |
| #include <iostream>  #include <bitset>  using namespace std;    // Size of the short is 16 bits (or 2 bytes) regardless of whether you are on a 32-bit or 64-bit OS.  #define BIT\_COUNT\_FOR\_A\_SHORT 16    unsigned short AND(unsigned short value1, unsigned short value2) {  return value1 & value2;  }    unsigned short OR(unsigned short value1, unsigned short value2) {  return value1 | value2;  }    unsigned short XOR(unsigned short value1, unsigned short value2) {  return value1 ^ value2;  }    unsigned short NOT(unsigned short value) {  return ~value;  }    unsigned short LogicalLeftShift(unsigned short value, unsigned short count) {  return value << count;  }    unsigned short LogicalRightShift(unsigned short value, unsigned short count) {  return value >> count;  }    unsigned short RotateLeftShift(unsigned short value, unsigned short count) {  return (value << count) | (value >> (sizeof(value) \* BIT\_COUNT\_FOR\_A\_SHORT - count));  }    unsigned short RotateRightShift(unsigned short value, unsigned short count) {  return (value >> count) | (value << (sizeof(value) \* BIT\_COUNT\_FOR\_A\_SHORT - count));  }    short ArithmeticLeftShift(short value, short count) {  return value << count;  }    short ArithmeticRightShift(short value, short count) {  return value >> count;  }    int main() {  std::bitset<BIT\_COUNT\_FOR\_A\_SHORT> binary;    binary = AND(0b0101, 0b0011);  std::cout << "AND: 0b" << binary << std::endl;    binary = OR(0b0101, 0b0011);  std::cout << "OR: 0b" << binary << std::endl;    binary = XOR(0b0101, 0b0011);  std::cout << "XOR: 0b" << binary << std::endl;    binary = NOT(0b0101);  std::cout << "NOT: 0b" << binary << std::endl;    // Logical shift //  binary = LogicalLeftShift(0b0101, 1);  std::cout << "LogicalLeftShift: 0b" << binary << std::endl;    binary = LogicalLeftShift(0b0111, 1);  std::cout << "LogicalLeftShift: 0b" << binary << std::endl;    binary = LogicalLeftShift(0b1111, 1);  std::cout << "LogicalLeftShift: 0b" << binary << std::endl;    binary = LogicalRightShift(0b0101, 1);  std::cout << "LogicalRightShift: 0b" << binary << std::endl;    binary = LogicalRightShift(0b0111, 1);  std::cout << "LogicalRightShift: 0b" << binary << std::endl;    binary = LogicalRightShift(0b1111, 1);  std::cout << "LogicalRightShift: 0b" << binary << std::endl;      // Rotate shift //  binary = RotateLeftShift(0b0101, 1);  std::cout << "RotateLeftShift: 0b" << binary << std::endl;    binary = RotateLeftShift(0b0111, 1);  std::cout << "RotateLeftShift: 0b" << binary << std::endl;    binary = RotateLeftShift(0b1111, 1);  std::cout << "RotateLeftShift: 0b" << binary << std::endl;    binary = RotateRightShift(0b0101, 1);  std::cout << "RotateRightShift: 0b" << binary << std::endl;    binary = RotateRightShift(0b0111, 1);  std::cout << "RotateRightShift: 0b" << binary << std::endl;    binary = RotateRightShift(0b1111, 1);  std::cout << "RotateRightShift: 0b" << binary << std::endl;      // Arithmetic shift //  binary = ArithmeticLeftShift(0b0001, 1);  std::cout << "ArithmeticLeftShift: 0b" << binary << std::endl;    binary = ArithmeticLeftShift(0b0010, 1);  std::cout << "ArithmeticLeftShift: 0b" << binary << std::endl;    binary = ArithmeticLeftShift(0b0011, 1);  std::cout << "ArithmeticLeftShift: 0b" << binary << std::endl;    binary = ArithmeticLeftShift(0b1111, 1);  std::cout << "ArithmeticLeftShift: 0b" << binary << std::endl;    binary = ArithmeticRightShift(0b0001, 1);  std::cout << "ArithmeticRightShift: 0b" << binary << std::endl;    binary = ArithmeticRightShift(0b0010, 1);  std::cout << "ArithmeticRightShift: 0b" << binary << std::endl;    binary = ArithmeticRightShift(-11, 1); // 0b1011  std::cout << "ArithmeticRightShift: 0b" << binary << std::endl;    binary = ArithmeticRightShift(-12, 1); // 0b1100  std::cout << "ArithmeticRightShift: 0b" << binary << std::endl;    return 0;  } | AND: 0b0000000000000001  OR: 0b0000000000000111  XOR: 0b0000000000000110  NOT: 0b1111111111111010  LogicalLeftShift: 0b0000000000001010  LogicalLeftShift: 0b0000000000001110  LogicalLeftShift: 0b0000000000011110  LogicalRightShift: 0b0000000000000010  LogicalRightShift: 0b0000000000000011  LogicalRightShift: 0b0000000000000111  RotateLeftShift: 0b0000000000001010  RotateLeftShift: 0b0000000000001110  RotateLeftShift: 0b0000000000011110  RotateRightShift: 0b0000000000000010  RotateRightShift: 0b0000000000000011  RotateRightShift: 0b0000000000000111  ArithmeticLeftShift: 0b0000000000000010  ArithmeticLeftShift: 0b0000000000000100  ArithmeticLeftShift: 0b0000000000000110  ArithmeticLeftShift: 0b0000000000011110  ArithmeticRightShift: 0b0000000000000000  ArithmeticRightShift: 0b0000000000000001  ArithmeticRightShift: 0b1111111111111010  ArithmeticRightShift: 0b1111111111111010 |

### [Easy] Set Bit

Problem: Set a bit at nth position in the number to 1. Position index is started from 0.

Example:

* Input: n = 0b110001.
* Output: After setting bit 2, n becomes 0b110011.

Solution:

* Left shift 1 to n position.
* Use the OR operator to set the bit at that position.

Implementation:

|  |  |
| --- | --- |
| #include <iostream>  using namespace std;    void set(int& num, int pos) {  num |= (1 << pos);  }    int main() {  int num = 4, pos = 1;  set(num, pos);  cout << num << endl;  return 0;  } | 6 |

### [Easy] Clear Bit

Problem: Clear a bit at nth position in the number to 0. Position index is started from 0.

Example:

* Input: n = 0b1011.
* Output: After setting bit 2, n becomes 0b1001.

Solution:

* Left shift 1 to n position.
* Use the NOT operator to unset this shift.
* Use the AND operator to set the bit at that position.

Implementation:

|  |  |
| --- | --- |
| #include <iostream>  using namespace std;    void set(int& num, int pos) {  num &= (~(1 << pos));  }    int main() {  int num = 7, pos = 1;  set(num, pos);  cout << num << endl;  return 0;  } | 5 |

### [Easy] Toggle Bit

Problem: Toggle a bit at nth position. Toggling means to turn bit ON (1) if it was OFF (0) and to turn OFF (0) if it was ON (1) previously. Position index is started from 0.

Example:

* Input: n = 0b1011.
* Output: After toggling bit 2, n becomes 0b1001.

Solution:

* Left shift 1 to n position.
* Use the XOR operator to toggle the bit.

Implementation:

|  |  |
| --- | --- |
| #include <iostream>  using namespace std;  void toggle(int& num, int pos) {  num ^= (1 << pos);  }  int main() {  int num = 4, pos = 1;  toggle(num, pos);  cout << num << endl;  return 0;  } | 5 |

### [Easy] Check if a Number Is Odd

Problem: Check if a number is an odd number.

Solution: AND the number with 1. If the result is 1, then it’s an odd number.

Implemenation:

|  |  |
| --- | --- |
| #include <iostream>  using namespace std;    int main() {  int num = 0b00101;  if (num & 1 == 1) {  cout << "Odd number" << endl;  return 0;  }  return 0;  } | Odd number |

## Practices

### [Easy] Check If a Number Is a Power of 2

Solution: If a number N is a power of 2, then the bitwise AND of N and N-1 will be 0. But this will not work if N is 0. So just check these two conditions, if any of these two conditions is true.

Implementation:

|  |  |
| --- | --- |
| #include <iostream>  using namespace std;    bool isPowerOfTwo(int x) {  // First x in the below expression is  // for the case when x is 0  return x && (!(x & (x - 1)));  } |  |

### [Easy] Count Set Bits

Problem: Count total number of 1's in the binary representation of an integer.

Solution: Go through all the bits of given number and check whether it is set or not by performing AND operation with 1.

Implementation:

|  |  |
| --- | --- |
| int countBits(int n) {  int count = 0;  while (n) {  count += n & 1;  n >>= 1;  }  return count;  } |  |

### [Easy] Find Rightmost Set Bit

Problem: Find the position of the rightmost set bit in a number. The set bit has value of 1.

Example:

* Input: n = 0b1000100.
* Output: The rightmost set bit is at position 3 (note: Position index is started from 0).

Solution 1:

1. Create a temporay number with the same value as the input number.
2. Set each bit of the temporay number, started from bit 1.

For example:

1st time: 0b1000100 🡪 0b100010**1**

2nd time: 0b1000100 🡪 0b10001**1**0

3rd time: 0b1000100 🡪 0b1000**1**00

1. XOR the temporay number with the input number. If the result is 0, then stop because we successfully found the rightmost set bit.

Solution 2:

1. Check if the result is an odd number.
2. If not, right shift the number by 1 unit. And check again. We’ll stop when finding the odd number.

Implementation:

|  |  |
| --- | --- |
| #include <iostream>  using namespace std;    #define BIT\_COUNT\_FOR\_A\_INT 64    int main() {  int num = 0b00100;  int tmp = 0;  for (int i = 0; i < BIT\_COUNT\_FOR\_A\_INT; i++) {  tmp = num | (1 << i);  if ((tmp ^ num) == 0) { // IMPORTANT: Writing "if (tmp ^ num == 0)" is wrong.  cout << "Found rightmost bit at position " << i + 1 << endl;  return i + 1;  }  }  return -1;  } | Found rightmost bit at position 3 |
| #include <iostream>  using namespace std;    #define BIT\_COUNT\_FOR\_A\_INT   64    int main() {      int num = 0b00100;      for (int i = 0; i < BIT\_COUNT\_FOR\_A\_INT; i++) {          if (num & 1 == 1) {              cout << "Found rightmost bit at position " << i + 1 << endl;              return i + 1;          }          num = num >> 1;      }      return -1;  } | Found rightmost bit at position 3 |

### [Medium] Swap Two Numbers (No Temporary Variable)

Solution:

1. a ^= b
2. b ^= a;
3. a ^= b

Example:

Input: a = 0b1100, b = 0b0011

Steps:

1. a = a XOR b = 0b1111
2. b = b XOR a = 0b1100
3. a = a XOR b = 0b0011

Output: a = 0x0011, b = 0b1100

Implementation:

|  |  |
| --- | --- |
| #include <iostream>  using namespace std;    int main() {  int a = 1;  int b = 2;    a = a ^ b;  b ^= a;  a ^= b;  cout << "a = " << a << ", b = " << b << endl;    return 0;  } | a = 2, b = 1 |

# Page Replacement Algorithms

In OS using paging for memory management, page replacement algorithms help **decide which page needed to be replaced** when there is a new page request, but there is not enough space in the main memory to allocate the new page.

Whenever a new page is referred and not present in memory, *page fault* occurs and OS replaces one of the existing pages with the new page.

Different page replacement algorithms suggest different ways to decide which page to replace. The target for all algorithms is to reduce number of page faults.

For details about paging technique, check Personal\Tutorials\Embedded Systems\Embedded System Tutorial.docx

## FIFO (First In First Out)

This is the simplest page replacement algorithm. In this algorithm, OS keeps track of all pages in the memory in a queue, the **oldest page is in the front of the queue, which will be selected for replaced**.

For example: Consider the page references 1, 3, 0, 3, 5, 6, 3 with 3 page frames:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 1 | 3 | 0 | 3 | 5 | 6 | 3 |
|  |  | 0 | 0 | 0 | 0 | 3 |
|  | 3 | 3 | 3 | 3 | 6 | 6 |
| 1 | 1 | 1 | 1 | 5 | 5 | 5 |
| Miss | Miss | Miss | Hit | Miss | Miss | Miss |

**Explanation:**

* Initially all slots are empty.
* When 1, 3, 0 came, they are allocated to the empty slots —> 3 Page Faults.
* When 3 comes, it is already in memory —> 0 Page Faults.
* When 5 comes, it is not available in  memory —> replaces the oldest page 1 —> 1 Page Fault.
* When 6 comes, it is not available in memory —> replaces the oldest page 3 —> 1 Page Fault.
* Finally, when 3 come, it is not available —> replaces the oldest page slot 0 —> 1 Page Fault.

**Algorithm:**

Input:

* Memory capacity (number of pages the memory can hold): N
* A string representing pages to be referred (stored in a list): L[i]

Output:

* Find number of page faults: PF
* Find the final queue which holding all pages after all operations: Q

Steps:

I - Start traversing the pages in a loop (the looping times = the number of requested pages).

1. If current page is present in list L, do nothing:

2. Else:

If list L has less pages than N:

a) Insert new page into list L.

b) Increase page fault PF by 1.

Else:

a) Remove the first page from queue Q as it was the first to be entered in the memory

b) Insert new page into list L.

d) Increase page fault PF by 1.

II - Return page fault PF.

III - Return queue Q.

Implementation (C++):

Personal\Tutorials\Data Structures - Algorithms\Code\PageReplacementAlgorithms

## LRU (Least Recently Used)

In this algorithm, **page which is least recently used will be replaced**.

For example, consider the page reference 7, 0, 1, 2, 0, 3, 0, 4, 2 with 4 page frames:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 7 | 0 | 1 | 2 | 0 | 3 | 0 | 4 | 2 |
|  |  |  | 2 | 2 | 2 | 2 | 2 | 2 |
|  |  | 1 | 1 | 1 | 1 | 1 | 4 | 4 |
|  | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 7 | 7 | 7 | 7 | 7 | 3 | 3 | 3 | 3 |
| Miss | Miss | Miss | Miss | Hit | Miss | Hit | Miss | Hit |

**Explanation**:

* Initially all slots are empty
* When 7 0 1 2 are allocated to the empty slots —> 4 Page Faults.
* 0 is already their —> 0 Page Fault.
* When 3 came, it will take the place of 7 because it is least recently used —> 1 Page Fault.
* 0 is already in memory —> 0 Page Fault.
* 4 will takes place of 1 —> 1 Page Fault.
* 2 is already in memory —> 0 Page Fault.

**Algorithm:**

Input:

* Memory capacity (number of pages the memory can hold): N
* A string representing pages to be referred (stored in a list): L[i]

Output:

* Find number of page faults: PF
* Find the final list which holding all pages after all operations: Q

Steps:

I - Start traversing the pages in a loop (the looping times = the number of requested pages).

1. If current page is present in list L:

a) Increase page frequency by 1

2. Else:

If list L has less pages than N:

a) Insert new page into list L.

b) Increase page frequency by 1

c) Increase page fault PF by 1.

Else:

a) Find the page which is least recently used.

b) Remove this page.

c) Insert new page into list L.

d) Increase page fault PF by 1.

II - Return page fault PF.

III - Return list L.

Implementation (C++):

Personal\Tutorials\Data Structures - Algorithms\Code\PageReplacementAlgorithms

## LFU (Least Frequently Used)

In this algorithm, **page which is least recently used will be replaced**.

For example, consider the page reference 1, 2, 3, 4, 2, 1, 5 with 3 page frames:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 2 | 1 | 5 |
|  |  | 3 | 3 | 3 | 1 | 1 |
|  | 2 | 2 | 2 | 2 | 2 | 2 |
| 1 | 1 | 1 | 4 | 4 | 4 | 5 |
| Miss | Miss | Miss | Miss | Hit | Miss | Miss |

**Explanation**:

* Initially all slots are empty
* When 1 2 3 are allocated to the empty slots —> 3 Page Faults.
* When 4 cames —> replace page 1 --> 1 Page Fault.
* 2 is already in memory —> 0 Page Fault.
* When 1 cames —> replace page 3 (because compared to page 2, page 3 is the least frequently used) --> 1 Page Fault.
* When 5 cames —> replace page 4 (because compared to page 1 and 2, page 4 is the least frequently used) --> 1 Page Fault.

**Algorithm:**

Input:

* Memory capacity (number of pages the memory can hold): N
* A string representing pages to be referred (stored in a list): L[i]

Output:

* Find number of page faults: PF
* Find the final list which holding all pages after all operations: Q

Steps:

I - Start traversing the pages in a loop (the looping times = the number of requested pages).

1. If current page is present in list L:

a) Increase page frequency by 1

2. Else:

If list L has less pages than N:

a) Insert new page into list L.

b) Increase page frequency by 1.

c) Increase page fault PF by 1.

Else:

a) Find the page which is least frequently used.

b) Remove this page.

c) Insert new page into list L.

d) Increase page fault PF by 1.

II - Return page fault PF.

III - Return list L.

Implementation (C++):

Personal\Tutorials\Data Structures - Algorithms\Code\PageReplacementAlgorithms

## OPT (Optimal Page Replacement)

In this algorithm, pages are replaced which would **not be used for the longest duration of time in the future**. This is the best page replacement algorithm as it gives the least number of page faults.

# Others

## Checking Balancing of Brackets

### Problem-1: Discuss how stacks can be used for checking balancing of brackets.

**Solution**

Stacks can be used to check whether the given expression has balanced brackets. This algorithm is very useful in compilers. Each time the parser reads one character at a time. If the character is an opening delimiter such as (, {, or [ - then it is written to the stack. When a closing delimiter is encountered like ), }, or ] - the stack is popped.

The opening and closing delimiters are then compared. If they match, the parsing of the string continues. If they do not match, the parser indicates that there is an error on the line.

A linear-time O(n) algorithm based on stack can be given as:

1. Create a stack.
2. while (end of input is not reached) {
3. If the character read is not a bracket to be balanced, ignore it.
4. Else if the character is an opening bracket like (, [, {, push it onto the stack.
5. Else if it is a closing symbol like ), ], }
6. If the stack is empty, report an error. Otherwise pop the stack.
   * 1. If the bracket popped is not the corresponding opening bracket, report an error.

}

1. At end of input, if the stack is not empty report an error.

Time Complexity: O(n). Space Complexity: O(n) [for stack]. Since we are scanning the input only once (using one loop).

# C++ Standard Built-In Functions

## Std::sort

you can use std::sort() from the <algorithm> header to sort an array. Here’s how to use it with a simple example:

**Example 1: Sorting a Simple Array**

cpp

Copy

#include <iostream>

#include <algorithm> // for std::sort

int main() {

int arr[] = {5, 2, 8, 1, 3};

int size = sizeof(arr) / sizeof(arr[0]);

// Sort the array in ascending order

std::sort(arr, arr + size);

// Output the sorted array

std::cout << "Sorted array: ";

for (int i = 0; i < size; ++i) {

std::cout << arr[i] << " ";

}

std::cout << std::endl;

return 0;

}

**Breakdown of the Code**

1. **Include Headers**: You need to include <algorithm> for std::sort and <iostream> for input/output.
2. **Array Declaration**: Define the array you want to sort.
3. **Calculate Size**: Determine the size of the array using sizeof(arr) / sizeof(arr[0]).
4. **Sorting**: Call std::sort(arr, arr + size):
   * The first argument is the beginning of the array.
   * The second argument is one past the last element of the array.
5. **Output**: Use a loop to print the sorted array.

**Example 2: Sorting in Descending Order**

If you want to sort the array in descending order, you can pass a custom comparator to std::sort():

cpp

Copy

#include <iostream>

#include <algorithm> // for std::sort

int main() {

int arr[] = {5, 2, 8, 1, 3};

int size = sizeof(arr) / sizeof(arr[0]);

// Sort the array in descending order

std::sort(arr, arr + size, std::greater<int>());

// Output the sorted array

std::cout << "Sorted array (descending): ";

for (int i = 0; i < size; ++i) {

std::cout << arr[i] << " ";

}

std::cout << std::endl;

return 0;

}

**Example 3: Sorting with a Custom Comparator**

You can also define your own comparator function:

cpp

Copy

#include <iostream>

#include <algorithm> // for std::sort

bool customCompare(int a, int b) {

return a < b; // Ascending order

}

int main() {

int arr[] = {5, 2, 8, 1, 3};

int size = sizeof(arr) / sizeof(arr[0]);

// Sort the array using a custom comparator

std::sort(arr, arr + size, customCompare);

// Output the sorted array

std::cout << "Sorted array: ";

for (int i = 0; i < size; ++i) {

std::cout << arr[i] << " ";

}

std::cout << std::endl;

return 0;

}

**Summary**

* Use std::sort() to sort arrays.
* You can sort in ascending or descending order by using a comparator.
* std::sort() works with both raw arrays and STL containers like std::vector.

This makes std::sort() a versatile tool for sorting in C++.

## Swap

<https://www.geeksforgeeks.org/swap-in-cpp/>

## Max

[std::max in C++ - GeeksforGeeks](https://www.geeksforgeeks.org/stdmax-in-cpp/)

Min

## std::bitset

https://www.geeksforgeeks.org/stdstable\_partition-in-c/

# Supported Tools

<https://pythontutor.com/>