**Конспект: Алгоритмы сортировки в языке C++**

Сортировка – это процесс упорядочивания элементов в некотором наборе данных. Одним из способов сортировки являются алгоритмы сортировки, которые позволяют автоматически упорядочить элементы по заданному критерию. В языке C++ доступно несколько известных алгоритмов сортировки, которые мы рассмотрим.

**1. Сортировка пузырьком**:

Алгоритм сортировки пузырьком является одним из самых простых алгоритмов сортировки. Он работает путем многократного прохода по массиву и сравнения пар соседних элементов. Если элементы находятся в неправильном порядке, они меняются местами. Этот процесс продолжается до тех пор, пока весь массив не будет отсортирован. Хотя алгоритм пузырьковой сортировки прост в реализации, он не является эффективным для больших массивов из-за своей квадратичной временной сложности.

Как работает сортировка пузырьком:

Пусть у нас есть массив из 5 чисел, который нам надо отсортировать

int arr[] = {3, 5, 1, 356, 0};

Тогда мы берём первый элемент и сравниваем его со вторым: 3 < 5, значит, мы их не трогаем. Переходим к следующей паре: 5 > 1, значит, их надо поменять местами. Получаем [3, 1, 5, 356, 0]. Смотрим дальше: 5 < 356 - не трогаем. Потом 356 < 0 - меняем местами, получаем [3, 1, 5, 0, 356].

Очевидно, что это ещё не конец. Начинаем сначала: 3 > 1, меняем местами, получаем [1, 3, 5, 0, 356]. Идём дальше: 3 < 5, оставляем как есть. Следующая пара: 5 > 0, меняем местами, получаем [1, 3, 0, 5, 356]. Дальше нам смотреть смысла нет: самое большое число уже и так стоит в самом конце.

Третий заход. 1 < 3 - прекрасно, 3 > 0 - плохо. Меняем местами, получаем [1, 0, 3, 5, 356]. Сравнивать пары (3, 5) и (5, 356) не нужно: 5 и 356 оказались точно на своих местах.

Четвёртый заход: меняем 0 и 1 местами, получаем отсортированный массив [0, 1, 3, 5, 356].

#include <iostream>

using namespace std;

void bubbleSort(int list[], int listLength)

{

while (listLength--)

{

bool swapped = false;

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

{

if (list[i] > list[i + 1])

{

swap(list[i], list[i + 1]);

swapped = true;

}

}

if (swapped == false)

break;

}

}

int main()

{

int list[5] = { 3,19,8,0,48 };

cout << "Input array ..." << endl;

for (int i = 0; i < 5; i++)

cout << list[i] << '\t';

cout << endl;

bubbleSort(list, 5);

cout << "Sorted array ..." << endl;

for (int i = 0; i < 5; i++)

cout << list[i] << '\t';

cout << endl;

}

**2. Сортировка выбором**:

Алгоритм сортировки выбором работает по принципу выбора наименьшего (или наибольшего) элемента из неотсортированной части массива и помещения его в начало (или конец) отсортированной части. Этот процесс повторяется до полной сортировки всех элементов. Хотя алгоритм сортировки выбором имеет линейную временную сложность, он также не является оптимальным для больших массивов.

Возьмём тот же массив и отсортируем его:

int arr[] = {3, 5, 1, 356, 0};

Проход 1: ищем наименьший элемент среди всех. Сравниваем 3 и 5: 3 меньше, пока отмечаем этот элемент, как наименьший. Дальше идём, сравниваем 3 и 1, отмечаем 1, как наименьший. Дальше 1 и 356, наименьший по-прежнему 1. Сравниваем 1 и 0, видим, что 0 наименьший. Значит, меняем самый первый элемент и 0 местами. Получаем [0, 5, 1, 356, 3].

Проход 2. Сравниваем 5 и 1, запоминаем 1. Проходимся до конца, понимаем, что 1 сейчас - наименьший элемент. Меняем 1 и 5 местами. Получаем [0, 1, 5, 356, 3].

Проход 3. Обнаруживаем, что 3 наименьший элемент. Меняем 3 и 5 местами, получаем [0, 1, 3, 5, 356].

Проход 4. Проверяем оставшуюся пару. У нас здесь всё хорошо, массив отсортирован.

#include <iostream>

using namespace std;

int findSmallestPosition(int list[], int startPosition, int listLength)

{

int smallestPosition = startPosition;

for (int i = startPosition; i < listLength; i++)

{

if (list[i] < list[smallestPosition])

smallestPosition = i;

}

return smallestPosition;

}

void selectionSort(int list[], int listLength)

{

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

{

int smallestPosition = findSmallestPosition(list, i, listLength);

swap(list[i], list[smallestPosition]);

}

return;

}

int main()

{

int list[5] = { 12, 5, 48, 0, 4 };

cout << "Input array ..." << endl;

for (int i = 0; i < 5; i++)

cout << list[i] << '\t';

cout << endl;

selectionSort(list, 5);

cout << "Sorted array ..." << endl;

for (int i = 0; i < 5; i++)

cout << list[i] << '\t';

cout << endl;

}

**3. Сортировка вставками**:

Алгоритм сортировки вставками схож с процессом сортировки выбором. Он поочередно берет элементы из неотсортированной части массива и вставляет их в правильную позицию в отсортированной части. Это продолжается до полной сортировки всех элементов. Алгоритм сортировки вставками имеет квадратичную временную сложность, но он эффективен для небольших и уже частично отсортированных массивов.

#include <iostream>

using namespace std;

void insertionSort(int list[], int listLength)

{

for (int i = 1; i < listLength; i++)

{

int j = i - 1;

while (j >= 0 && list[j] > list[j + 1])

{

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

cout << "\ndid";

j--;

}

}

}

int main()

{

int list[8] = { 3,19,8,0,48,4,5,12 };

cout << "Input array ...\n";

for (int i = 0; i < 8; i++)

{

cout << list[i] << "\t";

}

insertionSort(list, 8);

cout << "\n\nSorted array ... \n";

for (int i = 0; i < 8; i++)

{

cout << list[i] << "\t";

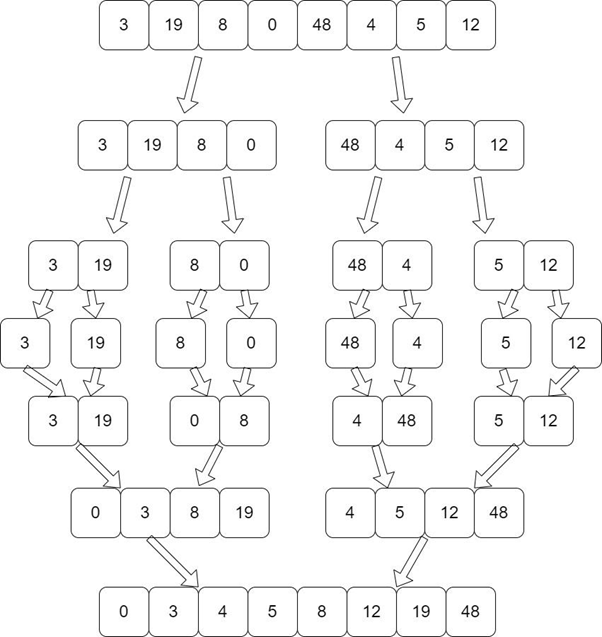
}

return 0;

}

**4. Сортировка слиянием**:

Алгоритм сортировки слиянием использует принцип "разделяй и властвуй". Он разделяет массив на две половины, рекурсивно сортирует каждую половину, а затем объединяет их в один отсортированный массив. Этот процесс продолжается до полной сортировки. Алгоритм сортировки слиянием имеет логарифмическую временную сложность и является стабильным. Он часто применяется в практике из-за своей эффективности на больших массивах.



#include <iostream>

using namespace std;

void merge(int list[], int start, int end, int mid);

void mergeSort(int list[], int start, int end)

{

int mid;

if (start < end) {

mid = (start + end) / 2;

mergeSort(list, start, mid);

mergeSort(list, mid + 1, end);

merge(list, start, end, mid);

}

}

void merge(int list[], int start, int end, int mid)

{

int mergedList[8];

int i, j, k;

i = start;

k = start;

j = mid + 1;

while (i <= mid && j <= end) {

if (list[i] < list[j]) {

mergedList[k] = list[i];

k++;

i++;

}

else {

mergedList[k] = list[j];

k++;

j++;

}

}

while (i <= mid) {

mergedList[k] = list[i];

k++;

i++;

}

while (j <= end) {

mergedList[k] = list[j];

k++;

j++;

}

for (i = start; i < k; i++) {

list[i] = mergedList[i];

}

}

int main()

{

int list[8] = { 3,19,8,0,48,4,5,12 };

cout << "Input array ...\n";

for (int i = 0; i < 8; i++)

{

cout << list[i] << "\t";

}

mergeSort(list, 0, 7);

cout << "\n\nSorted array ... \n";

for (int i = 0; i < 8; i++)

{

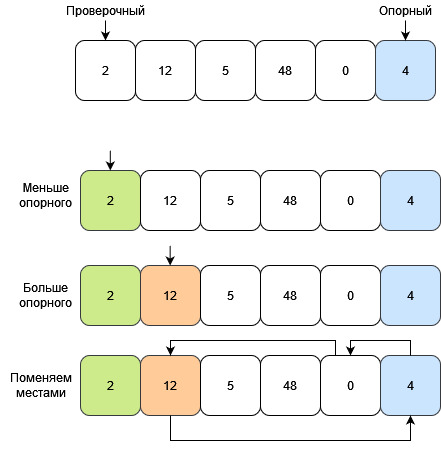
cout << list[i] << "\t";

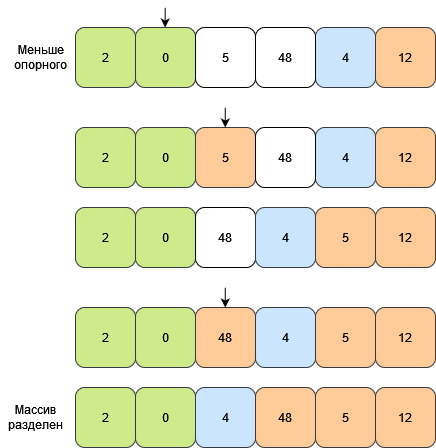
}

}

**5. Быстрая сортировка**:

Алгоритм быстрой сортировки также применяет принцип "разделяй и властвуй". Он выбирает опорный элемент из массива и разделяет массив на две части: элементы, меньшие опорного, и элементы, большие опорного. Затем он рекурсивно сортирует каждую из этих частей. Этот процесс продолжается до полной сортировки. Быстрая сортировка обладает средней временной сложностью O(n log n) и может быть очень эффективной, но в некоторых случаях может давать худшую временную сложность O(n^2).





Пример быстрой сортировки:

#include <iostream>

using namespace std;

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

int pivot = arr[high];

int i = low - 1;

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

if (arr[j] < pivot) {

i++;

// Меняем местами элементы

swap(arr[i], arr[j]);

}

}

// Меняем местами опорный элемент и элемент после всех меньших элементов

swap(arr[i+1], arr[high]);

return i + 1;

}

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

if (low < high) {

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

quickSort(arr, low, pi - 1);

quickSort(arr, pi + 1, high);

}

}

int main() {

int arr[] = { 5, 2, 8, 12, 3, 5, 5, 5, 4, 3, 2, 1, 3, 5, 6 };

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

quickSort(arr, 0, size - 1);

cout << "Отсортированный массив: ";

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

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

}

cout << endl;

return 0;

}

**Сортировка кучей (пирамидальная)**

// Реализация пирамидальной сортировки на C++

#include <iostream>

using namespace std;

// Процедура для преобразования в двоичную кучу поддерева с корневым узлом i, что является

// индексом в arr[]. n - размер кучи

void heapify(int arr[], int n, int i)

{

int largest = i;

// Инициализируем наибольший элемент как корень

int l = 2 \* i + 1; // левый = 2\*i + 1

int r = 2 \* i + 2; // правый = 2\*i + 2

// Если левый дочерний элемент больше корня

if (l < n && arr[l] > arr[largest])

largest = l;

// Если правый дочерний элемент больше, чем самый большой элемент на данный момент

if (r < n && arr[r] > arr[largest])

largest = r;

// Если самый большой элемент не корень

if (largest != i)

{

swap(arr[i], arr[largest]);

// Рекурсивно преобразуем в двоичную кучу затронутое поддерево

heapify(arr, n, largest);

}

}

// Основная функция, выполняющая пирамидальную сортировку

void heapSort(int arr[], int n)

{

// Построение кучи (перегруппируем массив)

for (int i = n / 2 - 1; i >= 0; i--)

heapify(arr, n, i);

// Один за другим извлекаем элементы из кучи

for (int i = n - 1; i >= 0; i--)

{

// Перемещаем текущий корень в конец

swap(arr[0], arr[i]);

// вызываем процедуру heapify на уменьшенной куче

heapify(arr, i, 0);

}

}

/\* Вспомогательная функция для вывода на экран массива размера n\*/

void printArray(int arr[], int n)

{

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

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

cout << "\n";

}

// Управляющая программа

int main()

{

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

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

heapSort(arr, n);

cout << "Sorted array is \n";

printArray(arr, n);

}

Примеры где используется быстрая сортировка:

1. Сортировка списков: В играх часто нужно упорядочить список врагов по удаленности от игрока или список объектов по их характеристикам. Алгоритм быстрой сортировки поможет быстро сделать это.

2. Определение порядка отрисовки: Если в игре используется 3D-графика, то объекты должны быть нарисованы в правильном порядке, чтобы они не перекрывали друг друга. Алгоритм быстрой сортировки поможет правильно расставить объекты на экране.

3. Проверка столкновений: В играх важно знать, сталкиваются ли объекты друг с другом. Алгоритм быстрой сортировки поможет быстро проверить, сталкиваются ли объекты на основе их координат.

4. Управление действиями: В некоторых играх нужно определить, какое действие выполнить первым. Алгоритм быстрой сортировки поможет упорядочить действия по приоритету или времени.

Это простые примеры, где алгоритм быстрой сортировки может быть полезен в играх. Он поможет сделать игру более эффективной и интересной.

**6. Алгоритмическая сложность**

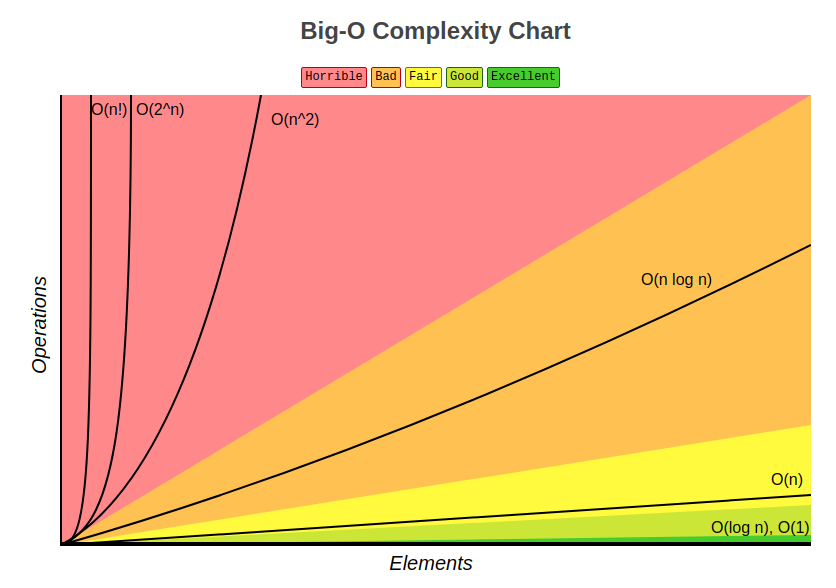
Сложность алгоритма в программировании - это показатель, который позволяет оценить, насколько быстро или медленно алгоритм будет работать в зависимости от размера входных данных.

* Константная сложность (O(1)): Время выполнения не зависит от размера данных.
* Линейная сложность (O(n)): Время выполнения пропорционально размеру данных.
* Квадратичная сложность (O(n^2)): Время выполнения увеличивается квадратично с ростом данных.
* Логарифмическая сложность (O(log n)): Время выполнения уменьшается с ростом данных.
* Экспоненциальная сложность (O(2^n)): Время выполнения экспоненциально растет с увеличением данных.

Алгоритмическая сложность может быть выражена в виде количества операций, времени выполнения или использования ресурсов (таких как память или процессорное время). Чем больше операций или ресурсов требуется для выполнения алгоритма, тем более сложным он считается.

Давай рассмотрим пример сортировки чисел. Пусть у нас есть 10 чисел в неотсортированном порядке, и мы используем алгоритм **сортировки пузырьком**. Если мы должны сравнить каждую пару чисел и выполнить обмен в случае необходимости, то алгоритм **будет занимать больше времени и операций, чем** если бы мы использовали более эффективный алгоритм, например, **быструю сортировку**.

1024 элемента в контейнере *в среднем* будут сортироваться за:  
 1048576 (= 1024\*1024) операций **пузырьковой сортировкой**  
 10240 (= 1024 \* log21024 = 1024 \* 10) операций **быстрой сортировкой**



Алгоритмическая сложность важна, потому что позволяет нам выбирать наиболее эффективные алгоритмы для решения задач. Мы можем измерять и сравнивать сложности разных алгоритмов и выбирать тот, который требует меньше ресурсов и выполняется быстрее.

**Заключение**:

Алгоритмы сортировки в языке C++ представляют собой мощные инструменты для упорядочивания данных. Они позволяют нам эффективно сортировать массивы и другие структуры данных. При выборе алгоритма сортировки важно учитывать особенности задачи и требования к производительности. Изучение этих алгоритмов позволяет развивать навыки программирования и понимание основных принципов алгоритмического мышления.