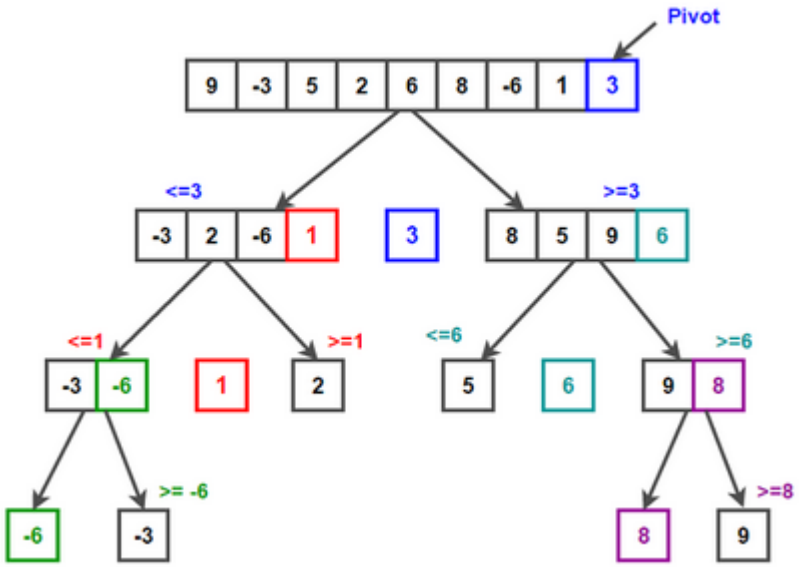
Содержание

quick sort

Quick Sort отличается быстротой выполнения и эффективностью использования памяти. Это один из самых популярных алгоритмов в мире программирования.

Основная идея базируется на парадигме «разделяй и властвуй». Алгоритм сначала выбирает опорный элемент (pivot). Затем делит массив на два подмассива: в первом элементы меньше или равны pivot, а во втором — больше. После этого каждый подмассив сортируется независимо. Затем процесс рекурсивно повторяется для обоих подмассивов, что позволяет эффективно упорядочить элементы.



Quick Sort показывает одну из лучших производительностей среди алгоритмов сортировки:

- В среднем случае временная сложность составляет O(n log n). Это объясняется тем, что массив делится на две равные части, а операции выполняются для каждой из них рекурсивно.
- В худшем случае сложность может возрасти до O(n2). Это происходит, когда массив разбивается крайне неравномерно, например, если элементы уже упорядочены.
- Пространственная сложность зависит от глубины рекурсии и составляет O(log n), так как хранение дополнительных данных минимально.

Где применять и как оптимизировать

Quick Sort подходит для множества задач — от упорядочивания массивов в простых приложениях до сложных системных решений, где нужно обеспечить эффективную сортировку на месте. Этот алгоритм часто используется в качестве встроенного метода сортировки в популярных библиотеках, например, в C++ и Python.

Чтобы улучшить производительность Quick Sort, можно использовать несколько оптимизаций:

- Выбор опорного элемента (pivot selection). Как уже упоминалось, использование median-of-three или случайного элемента уменьшает вероятность худшего сценария.
- Оптимизация хвостовых вызовов (Tail Call Optimization (TCO)). Оптимизация рекурсии

путем отказа от рекурсивного вызова для одной из частей массива (например, более короткой) позволяет избежать глубоких стеков вызовов и, как следствие, переполнения стека.
- Гибридные алгоритмы. На практике часто используется комбинация Quick Sort и других алгоритмов, таких как Insertion Sort, для небольших подмассивов. Эта техника снижает количество рекурсивных вызовов и уменьшает накладные расходы.
- Параллельная сортировка. Если необходимо работать с большими объемами данных, можно распределить вычисления на несколько ядер процессора или кластеров, реализовав параллельную версию Quick Sort.

Пример реализации

```
#include <iostream>
#include <vector>
#include <algorithm> // для std::copy, std::back_inserter

// Рекурсивная реализация quick sort, возвращающая новый вектор:
// - принимает const ссылку на входной вектор, чтобы не копировать сразу
// - возвращает отсортированный вектор
std::vector<int> quick_sort(const std::vector<int>& arr) {
  if (arr.size() <= 1) return arr; // базовый случай: пустой или единичный вектор уже отсортирован

  // Выбор опорного элемента — средний элемент вектора
  int pivot = arr[arr.size() / 2];

  // Разделяем элементы на три группы:
  // left  - элементы < pivot
  // middle - элементы == pivot (чтобы корректно обрабатывать дубликаты)
  // right - элементы > pivot
  std::vector<int> left, middle, right;
  left.reserve(arr.size()); // резервируем память заранее, чтобы уменьшить число выделений
  middle.reserve(arr.size());
  right.reserve(arr.size());

  for (int x : arr) {
    if (x < pivot) left.push_back(x);
    else if (x == pivot) middle.push_back(x);
    else right.push_back(x);
  }

  // Рекурсивно сортируем левую и правую подпоследовательности
  std::vector<int> sorted_left = quick_sort(left);
  std::vector<int> sorted_right = quick_sort(right);

  // Объединяем результаты: отсортленная левая часть + середина + отсортированная правая часть
  std::vector<int> result;
```

```
  result.reserve(sorted_left.size() + middle.size() + sorted_right.size());
// резервируем итоговый размер

  // Копируем элементы в итоговый вектор
  std::copy(sorted_left.begin(), sorted_left.end(),
std::back_inserter(result));
  std::copy(middle.begin(), middle.end(), std::back_inserter(result));
  std::copy(sorted_right.begin(), sorted_right.end(),
std::back_inserter(result));

  return result;
}

int main() {
  // Пример использования: исходный массив
  std::vector<int> array = {10, 7, 8, 9, 1, 5};
  // Получаем отсортированный массив (функция возвращает новый вектор)
  std::vector<int> sorted_array = quick_sort(array);

  // Вывод результата в консоль
  std::cout << "Отсортированный массив: ";
  for (size_t i = 0; i < sorted_array.size(); ++i) {
    if (i) std::cout << ", "; // выводим запятую перед элементом, начиная со
второго
    std::cout << sorted_array[i];
  }
  std::cout << std::endl;
  return 0;
}
```