МИНОБРНАУКИ РОССИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«ВОРОНЕЖСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»

(ФГБОУ ВО «ВГУ»)

Факультет прикладной математики, информатики и механики

Сравнение и анализ алгоритмов: сортировка выбором и быстрая сортировка

01.03.02 Прикладная математика и информатика

Обучающийся Якимов А. С.

Руководитель Коток И. Д.

Воронеж 2023

Оглавление

Введение

Сортировка — это процесс упорядочивания элементов в массивах или структурах данных в определенном порядке с помощью алгоритма. Сортировки широко используются в программировании, поиске информации и анализе данных.

Важно отметить, что существует множество различных видов сортировок. Каждый тип сортировки имеет свои особенности, преимущества и недостатки, и подходит для определенных начальных условий.

Цель данной работы заключается в изучении алгоритмов сортировки выбором и быстрой сортировки, анализе их сложности и сравнении их эффективности в зависимости от различных факторов.

1. Теоретическая часть

1.1. Сортировка выбором

Сортировка выбором — это алгоритм сортировки, который работает путем нахождения наименьшего элемента в неотсортированной части массива и перемещения его в начало этой части. Затем наименьший элемент исключается из неотсортированной части, и процесс повторяется для оставшихся элементов до тех пор, пока все элементы не будут отсортированы.

Подробно алгоритм сортировки выглядит так:

1. Найдите наименьший элемент в массиве.

2. Поместите его в начало (или конец) отсортированной части массива.

Этот алгоритм имеет сложность O(n^2), что делает его неэффективным для больших массивов, но он прост в реализации и может быть полезен для небольших массивов или в качестве части более сложных алгоритмов сортировки.

1.2. Быстрая сортировка

Быстрая сортировка — это эффективный алгоритм сортировки, который основан на принципе "разделяй и властвуй". Он работает путем выбора опорного элемента из массива и разделения массива на две подгруппы: одну с элементами, меньшими или равными опорному, и другую с элементами, большими опорного. Затем рекурсивно применяется тот же процесс к каждой подгруппе до тех пор, пока массив не будет полностью отсортирован.

Алгоритм реализации быстрой сортировки:

1. Выбор опорного элемента и разделение массива на две части, в левую часть переносятся элементы меньше опорного, в правую часть больше.
2. Далее та же операция применяется для двух получившихся частей, пока длина массива, на который применяется эта операция, не достигнет 1.

Быстрая сортировка имеет среднюю сложность O(n log n), что делает ее одним из самых быстрых алгоритмов сортировки. Однако, в худшем случае, когда опорный элемент выбирается не оптимально, сложность может быть O(n^2). Тем не менее, с учетом оптимизаций и правильного выбора опорного элемента, быстрая сортировка обычно является предпочтительным выбором для сортировки больших массивов данных.

2. Сравнение алгоритмов

1.Сложность:

Сложность алгоритма сортировки выбором составляет O(n^2), что делает его менее эффективным, чем быстрая сортировка. Однако, сортировка выбором может быть полезна для небольших массивов или в качестве части более сложных алгоритмов сортировки.

Сложность алгоритма быстрой сортировки в средних значениях составляет O(n log n), что делает его одним из самых быстрых алгоритмов сортировки. Однако, в худшем случае, когда опорный элемент выбирается нерационально, сложность может быть O(n^2).

2. Устойчивость:

Сортировка выбором является устойчивой. Это означает, что порядок равных элементов в исходном массиве сохраняется после сортировки.

Алгоритм быстрой сортировки не является устойчивым. Это означает, что порядок равных элементов в исходном массиве может измениться после сортировки.

3. Затраты по памяти:

Алгоритм сортировки выбором не требует дополнительной памяти для хранения временных структур данных. Он выполняет сортировку непосредственно в исходном массиве. Поэтому затраты памяти для сортировки выбором ограничены только размером исходного массива.

Алгоритм быстрой сортировки требует дополнительной памяти для выполнения рекурсивных вызовов функции сортировки. Кроме того, быстрая сортировка может использовать дополнительную память для временного хранения элементов при перестановках.

4. Эффективность:

Алгоритм сортировки выбором не является самым эффективным методом сортировки. Его сложность всегда составляет O(n^2), где n - размер массива. Это означает, что время выполнения алгоритма растет квадратично с увеличением размера входных данных.

Быстрая сортировка обычно считается одним из самых эффективных алгоритмов сортировки. Его средняя сложность составляет O(n log n), где n - размер массива. Это означает, что время выполнения алгоритма растет пропорционально логарифму от размера входных данных.

Если выбирать между быстрой сортировкой и сортировкой выбором, то быстрая сортировка обычно является более предпочтительным выбором. Она имеет среднюю сложность O(n log n), что делает ее эффективной для большинства случаев. Сортировка выбором, в свою очередь, имеет сложность O(n^2), что делает ее менее эффективной, особенно для больших объемов данных.

Быстрая сортировка также обладает другими преимуществами, возможность оптимизации с помощью выбора опорного элемента и других техник.

Однако, стоит отметить, что выбор алгоритма сортировки также зависит от конкретных требований и особенностей задачи. В некоторых случаях, когда размер данных невелик или когда требуется устойчивая сортировка, сортировка выбором может быть приемлемым выбором. В целом, быстрая сортировка обычно является более эффективным и широко используемым алгоритмом сортировки.

3. Реализация сортировок

Итак, рассмотрим реализацию алгоритмов сортировок на языке C++. Для удобства был создан класс Sort.

Итак, у класса имеется массив чисел, заполняющийся случайными элементами с помощью функции RandomArr .У класса также есть функция вывода и два метода сортировок (SelSort, FastSort).

Рассмотрим метод сортировки выбором(приложение 1).

Теперь рассмотрим метод быстрой сортировки(приложение 2).

4. Тестирование сортировок

Проведем тестирование сортировок – проверим алгоритмы массивами – размерами 100 и 10000 элементов. Результаты запишу в таблицу.

Таблица 1. Результаты тестов сортировок, в секундах

Метод сортировки / Случай	SelSort, сортировка выбором	FastSort, быстрая сортировка
100 элементов, стандартная сортировка	0.0000152	0.0000085
100 элементов, сортировка в обратном порядке	0.0000151	0.0000076
10000 элементов, стандартная сортировка	0.1054934	0.0018265
10000 элементов, сортировка в обратном порядке	0.104559	0.0018779

Из таблицы видно, что быстрая сортировка выигрывает по времени как на малых, так и на больших размерах массивов, однако в первом случае разницей можно пренебречь, если необходим метод, обладающий свойством устойчивости.

Заключение

Исследовав оба метода сортировки, можно сделать следующие выводы:

1. Алгоритм сортировки выбором не является оптимальным способом сортировки и всегда имеет временную сложность O(n^2). Из-за этого он неэффективен для сортировки больших объемов данных и не рекомендуется в таких случаях.
2. Быстрая сортировка обычно считается одним из наиболее эффективных алгоритмов сортировки. Ее средняя сложность составляет O(n log n). Обычно и в большинстве практических сценариев быстрая сортировка обеспечивает высокую производительность и является предпочтительным выбором для сортировки данных.
3. Метод быстрой сортировки превосходит метод сортировки выбором, но важно отметить, что выбор алгоритма сортировки также зависит от конкретных требований и особенностей задачи. В некоторых ситуациях, при небольшом объеме данных и необходимости устойчивой сортировки, подходящим вариантом может стать использование сортировки выбором.

Приложения

Приложение 1. SelSort:

```
void Sort::SelSort(int m, int n){
   int i=m,j,minIndex;
   for (i; i<n; i++) {
      minIndex = i;
      for (j=i+1; j<n+1; j++){
         if (arr[j]<arr[minIndex]){
            minIndex=j;
         }
      }
      std::swap(arr[minIndex], arr[i]);
   }
}
```

Приложение 2. FastSort:

```
void Sort::FastSort(int low, int high) {
    if (low<high){
        int pivot=Partition(low, high);
        FastSort(low, pivot-1);
        FastSort(pivot+1, high);
    }
}

int Sort::Partition(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++;
            std::swap(arr[i], arr[j]);
        }
    }
    std::swap(arr[i+1], arr[high]);
    return i+1;
}
```