Лекция 5.
Алгоритмы сортировки

СОДЕРЖАНИЕ

1 Характеристики алгоритмов сортировки 3

2 Алгоритмы сортировки 5

2.1 Сортировка вставками 5

2.2 Сортировка выбором 6

2.3 Сортировка пузырьком 6

2.4 Сортировка Шелла 7

2.5 Сортировка слиянием 8

2.6 Пирамидальная сортировка 9

2.7 Быстрая сортировка 10

3 Библиографический список 12

1 ХАРАКТЕРИСТИКИ АЛГОРИТМОВ СОРТИРОВКИ

Время - основной параметр, характеризующий быстродействие алгоритма. Называется также вычислительной сложностью.

Память - ряд алгоритмов требует выделения дополнительной памяти под временное хранение данных. При оценке не учитывается место, которое занимает исходный массив и независящие от входной последовательности затраты, например, на хранение кода программы. Алгоритмы сортировки, не потребляющие дополнительной памяти, относят к *сортировкам на месте*.

Устойчивость (*stability*) - устойчивая сортировка не меняет взаимного расположения элементов с одинаковыми ключами[3].

Количество элементарных операций, затраченных алгоритмом для решения конкретного экземпляра задачи, зависит не только от размера входных данных, но и от самих данных. Например, количество операций алгоритма сортировки вставками значительно меньше в случае, если входные данные уже отсортированы. Чтобы избежать подобных трудностей, рассматривают понятие временной сложности алгоритма *в худшем случае*.

Временная сложность алгоритма (в худшем случае) — это функция размера входных данных, равная максимальному количеству элементарных операций, проделываемых алгоритмом для решения экземпляра задачи указанного размера.

Аналогично понятию *временной сложности в худшем случае* определяется понятие *временная сложность алгоритма в наилучшем случае*. Также рассматривают понятие *среднее время работы алгоритма*, то есть математическое ожидание времени работы алгоритма.

По аналогии с временной сложностью, определяют *пространственную сложность алгоритма*, только здесь говорят не о количестве элементарных операций, а об объёме используемой памяти.

Несмотря на то, что функция временной сложности алгоритма в некоторых случаях может быть определена точно, в большинстве случаев искать точное её значение бессмысленно. Дело в том, что во-первых, точное значение временной сложности зависит от определения *элементарных операций* (например, сложность можно измерять в количестве арифметических операций), а во-вторых, при увеличении размера входных

данных вклад постоянных множителей и слагаемых низших порядков, фигурирующих в выражении для точного времени работы, становится крайне незначительным.

Рассмотрение входных данных большого размера и оценка порядка проста времени работы алгоритма приводят к понятию **асимптотической сложности** алгоритма. При этом алгоритм с меньшей асимптотической сложностью является более эффективным для всех входных данных, за исключением лишь, возможно, данных малого размера. Для записи асимптотической сложности алгоритмов используется обозначение функции ограничения сверху (с точностью до постоянного множества) O-большое. Например, алгоритм имеет сложность O(f(N)), если при увеличении размерности входных данных N, время выполнения алгоритма возрастает с той же скоростью, что и функция f(N).

Ниже приведены функции, которые чаще всего встречаются при вычислении сложности алгоритма. Чем выше в этом списке находится функция, тем быстрее будет выполняться алгоритм с такой оценкой.

1. O(1), константная сложность;
2. lg(lg(N));
3. lg(N), логарифмическая сложность;
4. O(N), линейная сложность;
5. N*lg(N);
6. C^N, C>1;
7. N!.

2 АЛГОРИТМЫ СОРТИРОВКИ

В данной части лекции рассматриваются широко известные алгоритмы сортировки, приведено их краткое описание, вычислительная сложность, анализ использования памяти, устойчивость и другие особенности реализации. Каждый метод снабжён примером на языке C, написанным для упорядочивания целочисленного массива по возрастанию.

2.1 Сортировка вставками

На каждом шаге алгоритма выбирается один из входящих элементов и вставляется на нужную позицию в уже отсортированной части массива, алгоритм выполняется до тех пор, пока набор входных данных не будет исчерпан. Метод выбора очередного элемента из исходного массива произволен. Обычно (и с целью получения устойчивого алгоритма сортировки), элементы вставляются по порядку их появления во входном массиве. Приведенный ниже алгоритм использует именно эту стратегию выбора.

```
1     void Swap (int *array, size_t first, size_t second) {
2          int dummy = array[first];
3          array[first] = array[second];
4          array[second] = dummy;
5     }
6
7     void InsertionSort (int *array, size_t len) {
8          for (size_t i = 1; i < len; ++i) {
9               size_t j = i;
10              while (j > 0 && array[j] < array[j-1]) {
11                   Swap(array, j, j - 1);
12                   --j;
13              }
14         }
15    }
```

Листинг 1 – Алгоритм сортировки вставками

На строке 1 дано объявление функции `Swap`, которая выполняет перестановку двух элементов `first` и `second` в массиве `array`. Далее следует функция `InsertionSort`, реализующая сортировку вставками.

Приведённый выше алгоритм обладает следующими характеристиками:

- Высокая вычислительная сложность, необходимо сделать $O(N^2)$ сравнений и $O(N^2)$ перестановок. Если данные уже отсортированы, то необходимо сделать всего O(N) операций;

- Не требует использования дополнительной памяти (O(1));
- Устойчив;
- Прост в реализации.

2.2 Сортировка выбором

Идея алгоритма заключается в нахождении на каждой итерации минимального элемента и перемещение его на нужную позицию в уже отсортированном массиве. Ниже приведена функция `SelectionSort` реализующая данный алгоритм.

```
1       void SelectionSort (int *array, size_t len) {
2            for (size_t i = 0; i < len - 1; ++i) {
3                 size_t min = i;
4                 for (size_t j = i + 1; j < len; ++j) {
5                      if (array[j] < array[min])
6                           min = j;
7                 }
8                 Swap(array, i, min);
9            }
10      }
```

Листинг 2 – Алгоритм сортировки выбором

Сортировка вставками в данной реализации обладает следующими свойствами:

- Высокая вычислительная сложность, необходимо сделать $O(N^2)$ сравнений и O(N) перестановок
- Не требует использования дополнительной памяти (O(1));
- Неустойчива (имеются устойчивые модификации);
- Простота в реализации.

2.3 Сортировка пузырьком

Алгоритм состоит из повторяющихся проходов по сортируемому массиву. За каждый проход элементы последовательно сравниваются попарно и, если порядок в паре неверный, выполняется обмен элементов. Проходы по массиву повторяются N-1 раз или до тех пор, пока на очередном проходе не окажется, что обмены больше не нужны, что означает — массив отсортирован. Алгоритм имеет такое название из-за того, что на каждой итерации наименьший элемент (самый лёгкий) «всплывает» до своей позиции. Ниже дан пример реализации этого алгоритма.

```
1       void BubbleSort (int *array, size_t len) {
```

```
2           for (size_t i = 0; i < len - 1; ++i) {
3                bool swapped = false;
4                for (size_t j = len - 1; j > i; --j) {
5                     if (array[j] < array[j-1]) {
6                          Swap(array, j, j - 1);
7                          swapped = true;
8                     }
9                }
10               if (!swapped) break;
11          }
12     }
```

Листинг 3 – Алгоритм сортировки пузырьком

Сортировка пузырьком обладает следующими свойствами:

- Высокая вычислительная сложность, необходимо сделать $O(N^2)$ сравнений и $O(N^2)$ перестановок. Если данные уже отсортированы, то необходимо сделать всего O(N) операций;
- Не требует использования дополнительной памяти (O(1));
- Устойчива;
- Простота в реализации.

2.4 Сортировка Шелла

Сортировка Шелла является усовершенствованным вариантом сортировки вставками. Идея данного метода состоит в сравнении элементов, стоящих не только рядом, но и на определённом расстоянии друг от друга. Иными словами — это сортировка вставками с предварительными «грубыми» проходами. Имеется большое количество модификаций, отличающихся выбором шага расстояния для каждой итерации. Ниже приведена реализация с шагом, рассчитываемым по формуле: 3^i+1

```
1      void ShellSort (int *array, size_t len) {
2           size_t gap = 1;
3           while (gap < len) gap = 3*gap + 1;
4
5           while (gap > 0) {
6                gap = gap / 3;
7
8                for (size_t i = gap; i < len; ++i) {
9                     int dummy = array[i];
10                    size_t j = i;
11                    while (j >= gap && array[j - gap] > dummy) {
12                         array[j] = array[j - gap];
```

```
13                          j -= gap;
14                      }
15                      array[j] = dummy;
16                  }
17              }
18          }
```

Листинг 4 – Алгоритм сортировки Шелла

Алгоритм Шелла обладает следующими свойствами:

- Вычислительная сложность в среднем равна $O(N^{3/2})$). Если данные уже отсортированы, то необходимо сделать всего O(N*lgN) операций;
- Не требует использования дополнительной памяти (O(1));
- Неустойчив.

2.5 Сортировка слиянием

Данный алгоритм сортировки эффективен для структур с последовательным доступом. Идея алгоритма состоит в поэлементном слиянии двух отсортированных последовательностей. Эти последовательности могут получаться путём рекурсивного вызова. Такой подход реализован в примере.

```
1       void MergeSort_recursion (int *array, size_t len, int * tmp) {
2            if (len < 2) return;
3
4            size_t middle = len / 2;
5            MergeSort_recursion(array, middle, tmp);
6            MergeSort_recursion(array + middle, len - middle, tmp);
7            memcpy(tmp, array, middle * sizeof(array[0]));
8
9            size_t i = 0, j = middle, k = 0;
10           while (i < middle && j < len)
11                array[k++] = (array[j] < tmp[i])?array[j++]:tmp[i++];
12           while (i < middle)
13                array[k++] = tmp [i++];
14      }
15
16      void MergeSort(int *array, size_t len) {
17           const size_t half_len = (len + 1) / 2;
18           int *tmp = new int[half_len];
19           MergeSort_recursion(array, len, tmp);
20           delete[] tmp;
21      }
```

Листинг 5 – Алгоритм сортировки слиянием

Сортировка слиянием обладает следующими свойствами:

- Вычислительная сложность в среднем равна O(N*lgN);
- Требует использования дополнительной памяти для слияния последовательностей (O(N));
- Устойчива.

2.6 Пирамидальная сортировка

Этот алгоритм сортировки использует специальную структуру данных – сортирующее дерево. Глубина листов такого дерева отличается не больше чем на 1, а значение в любой вершине не меньше значения потомков. На первом этапе пирамидальная сортировка создаёт такое дерево, а на втором последовательно удаляет из него корень и заново перестраивает

```
1   void HeapSort_sift (int *array, size_t root, size_t len) {
2        while (root * 2 + 1 < len) {
3             size_t child_l = 2 * root + 1;
4             size_t child_r = 2 * root + 2;
5             size_t child   = child_l;
6             if (child_r < len && array[child_l] < array[child_r])
7                  child = child_r;
8             if (array[root] < array[child]) {
9                  Swap(array, child, root);
10                 root = child;
11            }
12            else return;
13       }
14  }
15
16  void HeapSort(int *array, size_t len) {
17      for (size_t i = len/2 - 1; i != (size_t)-1; --i) {
18          HeapSort_sift(array, i, len);
19      }
20
21      for (size_t i = len - 1; i > 0; --i) {
22          Swap(array, 0, i);
23          HeapSort_sift(array, 0, i);
24      }
25  }
```

Листинг 6 – Алгоритм пирамидальной сортировки

Пирамидальная сортировка обладает следующими свойствами:

- Вычислительная сложность в среднем равна O(N*lgN);

- Не требует использования дополнительной памяти (O(1));
- Неустойчива;
- Сложна в реализации.

2.7 Быстрая сортировка

Общая идея алгоритма состоит в разделении входного набора данных относительно опорного элемента, выбираемого случайным образом. Все элементы, которые меньше опорного перемещаются в нижнюю часть массива, большие остаются на своих местах. В результате выполнения этих действий весь массива должен разделиться на три группы, следующие друг за другом: меньше опорного, равная опорному и больше опорного. Для «больших» и «меньших» отрезков рекурсивно выполняется та же последовательность действий. Для реализации данного алгоритма дополнительно приведена функция `RandRange`, дающая псевдослучайные числа в требуемом диапазоне.

```
1   size_t RandRange(size_t start, size_t end) {
2       return rand() % (end - start) + start;
3   }
4
5   void QuickSort(int *array, size_t len) {
6       Swap(array, 0, RandRange(1, len));
7
8       size_t pivot = 0;
9       for (size_t i = 1; i < len; ++i)
10          if (array[i] < array[0]) Swap(array, ++pivot, i);
11      Swap(array, 0, pivot);
12
13      if (pivot > 1)
14          QuickSort(array, pivot);
15      if ((pivot + 2) < len)
16          QuickSort(array + pivot + 1, len - pivot - 1);
17  }
```

Листинг 7 – Алгоритм быстрой сортировки

Быстрая сортировка обладает следующими свойствами:

- Вычислительная сложность в среднем равна O(N*lg(N)), в худших случая, при большом количестве неуникальных ключей может доходить до $O(N^2)$
- Требует использования дополнительной памяти (в среднем O(lgN), в худшем O(N));

- Неустойчива;

3 БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Роберт Седжвик, Алгоритмы на C++. Фундаментальные алгоритмы и структуры данных / пер. с англ. (Algorithms in C++) — М.: Вильямс. — 2011 г. — 1056 с.
2. Наглядная демонстрация алгоритмов сортировки. — 11 апреля 2011 г. — Режим доступа: http://habrahabr.ru/post/117200/, свободный.