Сортировка пузырьком

Подробно пузырьку, больший элемент массива поднимается "вверх".

Описание алгоритма

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. При каждом проходе алгоритма по внутреннему циклу, очередной наибольший элемент массива ставится на своё место в конце массива рядом с предыдущим «наибольшим элементом», а наименьший элемент перемещается на одну позицию к началу массива («всплывает» до нужной позиции как пузырёк в воде, отсюда и название алгоритма).

Реализация

```
void bubble(int* a, int n)
{
  for (int i=n-1; i>=0; i--)
  {
    for (int j=0; j<i; j++)
    {
      if (a[j] > a[j+1])
      {
        int tmp = a[j];
        a[j] = a[j+1];
        a[j+1] = tmp;
      }
    }
```

```
  }
}
```

Анализ алгоритма

Теоретическая оценка

Лучший	Средний	Худший	Вспомогательная сложность*
O(n)	O(n^2)	O(n^2)	O(1)

Вывод

Метод пузырька оказывается крайне неэффективным на любом входном наборе данных. Единственный плюс алгоритма - простота его исполнения.

Сортировка вставками

Выбираем и вставляем элемент в нужную позицию.

Описание алгоритма

На каждом шаге алгоритма мы выбираем один из элементов входных данных и вставляем его на нужную позицию в уже отсортированном списке, до тех пор, пока набор входных данных не будет исчерпан. Метод выбора очередного элемента из исходного массива произволен; может использоваться практически любой алгоритм выбора. 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.

Реализация

```
void insert_sort(int *a, int n)
{
    int i, j, value;

    for(i = 1; i < n; i++)
    {
        value = a[i];
        for (j = i - 1; j >= 0 && a[j] > value; j--)
        {
            a[j + 1] = a[j];
        }
        a[j + 1] = value;
    }
}
```

Анализ алгоритма

Теоретическая оценка

Лучший	Средний	Худший	Вспомогательная сложность*
O(n)	O(n^2)	O(n^2)	O(1)

Вывод

Метод оказывается эффективным только на маленьких массивах (не более 10), на массивах больших размеров, в силу асимптотической сложности n^2, алгоритм оказывается крайне не эффективным.

Сортировка выбором

Упорядочиваем постепенно массив, заполняя первую позицию неупорядоченной части минимальным элементом из неупорядоченной части.

Описание алгоритма

Шаги алгоритма: находим номер минимального значения в текущем списке, производим обмен этого значения со значением первой неотсортированной позиции (обмен не нужен, если минимальный элемент уже находится на данной позиции), теперь сортируем хвост списка, исключив из рассмотрения уже отсортированные элементы. Для реализации устойчивости алгоритма необходимо в пункте 2 минимальный элемент непосредственно вставлять в первую неотсортированную позицию, не меняя порядок остальных элементов.

Реализация

```
void select_sort(int *a, int length)
{
    for (int i = 0; i < length - 1; i++)
    {
            int min_i = i;
            /* находим индекс минимального элемента */
            for (int j = i + 1; j < length; j++)
            {
               if (a[j] < a[min_i])
               {
                       min_i = j;
               }
            }
            /* меняем значения местами */
            int temp = array[i];
            array[i] = array[min_i];
            array[min_i] = temp;
    }
```

```
}
```

Анализ алгоритма

Теоретическая оценка

Лучший	Средний	Худший	Вспомогательная сложность*
O(n^2)	O(n^2)	O(n^2)	O(1)

Вывод

Метод оказывается эффективным только на маленьких массивах (не более 10), на массивах больших размеров, в силу асимптотической сложности n^2, алгоритм оказывается крайне не эффективным.

Сортировка слиянием

Упорядочиваем списки (или другие структуры данных, доступ к элементам которых можно получать только последовательно, например — потоки) в определённом порядке.

Описание алгоритма

Для решения задачи сортировки эти три этапа выглядят так:

1) Сортируемый массив разбивается на две части примерно одинакового размера;

2) Каждая из получившихся частей сортируется отдельно, например — тем же самым алгоритмом;

3) Два упорядоченных массива половинного размера соединяются в один.

1.1. - 2.1. Рекурсивное разбиение задачи на меньшие происходит до тех пор, пока размер массива не достигнет единицы (любой массив длины 1 можно считать упорядоченным).

3.1. Соединение двух упорядоченных массивов в один. Основную идею слияния двух отсортированных массивов можно объяснить на следующем примере. Пусть мы имеем два подмассива. Пусть также, элементы подмассивов в каждом из этих подмассивов отсортированы по возрастанию. Тогда:

3.2. Слияние двух подмассивов в третий результирующий массив. На каждом шаге мы берём меньший из двух первых элементов подмассивов и записываем его в результирующий массив. Счетчики номеров элементов результирующего массива и подмассива из которого был взят элемент увеличиваем на 1.3. "Прицепление" остатка. Когда один из подмассивов закончился, мы добавляем все оставшиеся элементы второго подмассива в результирующий массив.

Реализация

```
void merge(int *a, int low, int mid, int high)
{
        int *b = (int*)malloc((high+1-low)*sizeof(int));
        int h,i,j,k;
        h=low;
        i=0;
        j=mid+1;
        while((h<=mid)&&(j<=high))
        {
                if(a[h]<=a[j])
                {
                        b[i]=a[h];
                        h++;
                }
                else
                {
```

```
                b[i]=a[j];
                j++;
            }
            i++;
        }
        if(h>mid)
        {
            for(k=j;k<=high;k++)
            {
                b[i]=a[k];
                i++;
            }
        }
        else
        {
            for(k=h;k<=mid;k++)
            {
                b[i]=a[k];
                i++;
            }
        }
        for(k=0;k<=high-low;k++)
        {
            a[k+low]=b[k];
        }
        free(b);
}
void merge_sort(int *a, int low, int high)
{
        int mid;
        if(low<high)
        {
            mid=(low+high)/2;
            merge_sort(a, low, mid);
            merge_sort(a, mid+1,high);
            merge(a, low, mid, high);
```

```
        }
}
```

Анализ алгоритма

Теоретическая оценка

Лучший	Средний	Худший	Вспомогательная сложность*
O(n log(n))	O(n log(n))	O(n log(n))	O(n)

Вывод

Метод оказывается крайне эффективным, однако требуется выделение дополнительной памяти.