Лекция 4. Сортировка подсчетом и применение встроенных сортировок

Стандартная библиотека шаблонов STL: встроенные сортировки и их применение. Использование собственных компараторов в алгоритмах сортировки. Стабильные сортировки. Быстрая сортировка. k-я порядковая статистика. Сортировка подсчетом. Пузырьковая сортировка, сортировка выбором, сортировка вставками. Сложность алгоритмов сортировки. Примеры задач, решаемых с использованием алгоритмов сортировки.

Общее представление

Алгоритмическая задача сортировки формулируется следующим образом: дан массив из **n** чисел. Необходимо упорядочить элементы массива так, чтобы они располагались в массиве по возрастанию (убыванию). Существует достаточно много алгоритмов сортировки, которые имеют различную сложность работы – от квадратичной за O(N^2), до сложности $O(Nlog\ N)$. Разумеется, для успешного выступления на олимпиадах надо понимать логику работы большинства алгоритмов сортировки и уметь пользоваться стандартными алгоритмами библиотеки STL.

Алгоритмы сортировки библиотеки STL

Сортировки контейнеров могут быть выполнены при помощи стандартных алгоритмов сортировки библиотеки STL. Алгоритмы сортировки требуют подключения заголовка <algorithm>. Приведем перечень алгоритмов сортировки и алгоритмов partition, понимание логики работы которых будет необходимо при изучении быстрой сортировки.

Алгоритм	**Описание алгоритма**
sort(it1, it2, Pred)	Упорядочивает элементы в указанном диапазоне итераторов в порядке возрастания или согласно указанному критерию упорядочивания, заданному бинарным предикатом Pred (компаратором). Сложность алгоритма в среднем – $O(NlogN)$.
stable_sort(it1, it2, Pred)	Упорядочивает элементы при помощи стабильной сортировки, при которой сохраняется относительный порядок сортируемых элементов. Сложность алгоритма больше, чем сложность алгоритма sort(), в лучшем случае она равна $O(NlogN)$, в худшем – $O(N(logN)^2)$. Работает медленнее, чем sort().
partition (it1, it2, Pred)	Разделяет элементы диапазона на два непересекающихся подмножества, при этом элементы, удовлетворяющие унарному предикату, предшествуют тем, которые не удовлетворяют ему. Возвращает итератор на начало второго подмножества. Сложность алгоритма разбиения - $O(N)$.
stable_partition (it1, it2, Pred)	Разделяет элементы диапазона на два непересекающихся множества, при этом элементы, удовлетворяющие унарному предикату, предшествуют тем, которые не удовлетворяют ему, сохраняя относительный порядок элементов массива. Сложность алгоритма разбиения $O(NlogN)$.

Наглядное представление о результате работы каждого из указанных алгоритмов дают нижеприведенные рисунки, на которых a[16]=(5, 1, 9, 2, 0, 5, 7, 3, 4, 5, 8, 5, 5, 5, 10, 6) – исходный массив элементов.

Исходный массив a[16]															
5	1	9	2	0	5	7	3	4	5	8	5	5	5	10	6
Массив после применения алгоритма sort(a, a+16)															
0	1	2	3	4	5	5	5	5	5	5	6	7	8	9	10

Массив после применения алгоритма partition (a,a+16,pr6), где pr6 – унарный предикат.

```
bool pr6(int val1){ return val1>6;}
```

10	8	9	7	0	6	2	3	4	5	1	5	5	5	5	5
Массив после применения алгоритма stable_partition (a,a+16,pr6)															
9	7	8	10	5	1	2	0	5	3	4	5	5	5	5	6

Стабильная сортировка

Часто применяются методы сортировки элементов с несколькими ключами. Например, если задан закодированный список учеников класса с указанием количества решенных ими задач и требуется упорядочить этот список по убыванию количества решенных задач, то нестабильные алгоритмы сортировки могут существенно перемешать исходные данные. Рассмотрим результаты сортировки записей алгоритмом sort() и алгоритмом стабильной сортировки stable_sort(), приведенные ниже. Записи сортируются по ключу - количеству решенных задач. Как видно из примера, стабильная сортировка сохраняет относительный порядок элементов в исходном массиве. Заметим, что увидеть отличие между результатами работы стандартных алгоритмов сортировки библиотеки STL: сортировки sort() и стабильной сортировки stable_sort() можно только на больших массивах исходных данных.

Исходный массива	После сортировки sort()	После сортировки stable_sort()
Val0 6, Val1 6, Val2 4	Val16 4, Val30 4, Val2 4	Val2 4, Val5 4, Val6 4
Val3 6, Val4 6, Val5 4	Val29 4, Val27 4, Val5 4	Val8 4, Val16 4, Val19 4
Val6 4, Val7 6, Val8 4	Val6 4, Val24 4, Val8 4	Val21 4, Val22 4, Val24 4
Val9 5, Val10 6, Val11 6	Val19 4, Val22 4, Val21 4	Val27 4, Val29 4, Val30 4
Val12 5, Val13 6, Val14 5	Val31 4, Val12 5, Val14 5	Val31 4, Val9 5, Val12 5
Val15 5, Val16 4, Val17 6	Val15 5, Val32 5, Val18 5	Val14 5, Val15 5, Val18 5
Val18 5, Val19 4, Val20 6	Val23 5, Val25 5, Val28 5	Val23 5, Val25 5, Val28 5
Val21 4, Val22 4, Val23 5	Val9 5, Val10 6, Val17 6	Val32 5, Val0 6, Val1 6
Val24 4, Val25 5, Val26 6	Val7 6, Val13 6, Val26 6	Val3 6, Val4 6, Val7 6
Val27 4, Val28 5, Val29 4	Val4 6, Val20 6, Val3 6	Val10 6, Val11 6, Val13 6
Val30 4, Val31 4, Val32 5	Val1 6, Val11 6, Val0 6	Val17 6, Val20 6, Val26 6

Использование собственных предикатов для алгоритмов сортировки

Функция sort() может принимать в качестве третьего параметра функцию – критерий сортировки (компаратор). Приведем программу, в которой в векторе хранятся элементы структуры list_fam. Структура позволяет описать элементы, состоящие из двух объектов – фамилии и оценки ученика.

```
struct list_fam{
          string fam;
          int value;
     };
```

Сравнение двух элементов, имеющих тип описанной структуры, должно производиться по какому-либо критерию сортировки. Например, при сравнении элементов только по баллу ученика и упорядочивании элементов по убыванию, можно использовать следующий компаратор.

```
bool comp (list_fam a, list_fam b){
          return a.value>b.value;
     }
```

Алгоритм сортировки в этом случае будет вызываться с указанием компаратора.

```
stable_sort(rating.begin(),rating.end(),comp);
```

В случае необходимости сортировки по убыванию, можно либо написать собственный компаратор, либо выполнить сортировку, а потом воспользоваться методом reverse(), который расположит элементы контейнера в обратном порядке, либо передать сортировке итераторы rbegin(), rend().

Алгоритм partition

Алгоритм partition() позволяет разбить множество на два подмножества по критерию, описанному в унарном предикате. В примере ниже предикат возвращает значение true, если элемент не кратен 3. После применения алгоритма partition(), вначале массива будут расположены не кратные 3 числа, затем кратные 3 числа. Таким образом, массив **a[]** будет разбит на два подмножества – одно из них состоит из чисел {1, 2, 8, 4, 5, 7}, другое из чисел {6, 3, 9}. Алгоритм partition() возвращает итератор на первый элемент второго подмножества.

```
int a[]={1, 2, 3, 4, 5, 6, 7, 8, 9};
bool comp (int l) { return (l%3); }
int main()
{
     int *i;
     i = partition (a, a+9, comp);
     for (int *j=a; j!=a+9;++j)
          cout<<(*j)<<" ";
     cout<<endl;
     for (int *j=a; j!=i;++j)
          cout<<(*j)<<" ";
```

Сложность работы алгоритма O(N), где N – количество элементов в массиве.

Быстрая сортировка QuickSort

Рассмотренные до этого алгоритмы сортировки Алгоритм быстрой сортировки QuickSort имеет в среднем сложность $O(Nlog\ N)$ и работает по принципу «разделяй и властвуй». Алгоритм QuickSort является составной частью стандартного алгоритма sort() библиотеки STL.

В общем виде работу алгоритма можно описать следующим образом: выбирается опорный элемент - partitioning element. Сортируемый массив переупорядочивается относительно опорного элемента, и разбивается на две части: левая часть состоит из элементов, не больших опорного, а правая часть состоит из элементов, не меньших опорного. Полная упорядоченность массива достигается разбиением его на части с последующим рекурсивным применением к ним этого же метода. Заметим, что быстрая сортировка не является стабильной сортировкой.

Приведем пример работы алгоритма быстрой сортировки на примере массива a[7]={7, 8, 1, 2, 4, 3, 6}.

Вначале работы алгоритма l=0; r=N-1; *Выбираем опорный элемент*. Существуют разные способы выбора опорного элемента p, например, p=a[m]=a[(l+r)/2] или p=a[l]. Теоретически можно выбирать случайный элемент между левой и правой границей, но функция генерации случайного числа работает медленнее, чем простые арифметические действия, поэтому выбор случайного числа может снизить производительность алгоритма. Компромиссным вариантом между выбором конкретного элемента последовательности и случайным его выбором, является способ выбора m=(x*l+y*r)/(x+y), где x и y - два произвольных числа.

После выбора опорного элемента *массив переупорядочивается* (происходит разбиение массива) таким образом, чтобы элементы не больше опорного находились бы вначале массива, а элементы, не меньшие опорного – в правой части массива. Это напоминает работу алгоритма *partition()*. На рисунке мы видим трассировку алгоритма – пока указатель i находится на элементе, меньшим, чем опорный, он передвигается вправо. Аналогично, пока указатель j находится на элементе, большим, чем опорный он передвигается влево. Указатели остановятся на «неправильно» расположенных элементах, в этом случае их надо поменять местами, и после этого сдвинуть указатели i и j на одну позицию вправо и влево соответственно. Процедура продолжается до тех пор,

пока выполняется условие i<=j. В конце работы процедуры переупорядочивания элементов указатели i и j поменяются местами.

i	j	p	0	1	2	3	4	5	6
0	6	p=a[3]=2	7↑i	8	1	(2)	4	3	6↑j
0	5	2	7↑i	8	1	(2)	4	3↑j	6
0	4	2	7↑i	8	1	(2)	4↑j	3	6
0	3	2	7↑i ⟷	8	1	(2) ↑j	4	3	6
1	2	2	(2)	8↑i ⟷	1↑j	7	4	3	6
2	1	2	(2)	1↑j	8↑i	7	4	3	6

Левая часть массива a[l, i-1] состоит из элементов не больших, чем опорный (на рисунке выделена серым цветом). Правая часть массива a[j+1, r] состоит из элементов не меньших, чем опорный.

Далее *вызывается рекурсивно процедура QuickSort* для левой и правой частей массива. Рекурсивный вызов осуществляется при условии, что l<j (для левой части) и r>i (для правой части).

Рекурсивный вызов Quicksort(l=0,j=1) от левой части массива									
0	1	2	(2) ↑i ⟷	1↑j	8	7	4	3	6
0	1	2	1↑j	(2) ↑i	8	7	4	3	6
Рекурсивный вызов Quicksort(l=2,j=6) от правой части массива									
2	6	4	1	2	8↑	7	(4)	3	6↑
2	5	4	1	2	8↑i ⟷	7	4	3↑j	6
3	4	4	1	2	3	7↑i ⟷	(4) ↑j	8	6
4	3	4	1	2	3	(4) ↑j	7↑i	8	6
Рекурсивный вызов Quicksort(l=4,j=6)									
4	6	8	1	2	3	4	7↑i	(8)	6↑j
5	6	8	1	2	3	4	7	(8) ↑i	6↑j
6	5	8	1	2	3	4	7	6↑j ⟷	(8) ↑i
Рекурсивный вызов Quicksort(l=4,j=5)									
4	5	7	1	2	3	4	(7) ↑i	6↑j	8
5	4	7	1	2	3	4	6↑j ⟷	(7) ↑i	8

Приведем программную реализацию функции быстрой сортировки.

```
void quick_sort(int l, int r){
     int i=l, j=r, p=a[(l+r)/2],temp;
     while(i<=j){
          while(a[i]<p) i++;
          while(a[j]>p) j--;
          if(i<=j){
               temp=a[i];
               a[i]=a[j];
               a[j]=temp;
               i++; j--;
          }
     }
     if (l<j) quick_sort(l,j);// Левая часть
     if (i<r) quick_sort(i,r); //Правая часть
}
```

Оценка сложности быстрой сортировки

Алгоритм быстрой сортировки в среднем использует $O(Nlog\,N)$ сравнений и $O(Nlog\,N)$ присваиваний, $O(log\,N)$ дополнительной памяти для хранения стека рекурсивных вызовов.

Быстрая сортировка неэффективна на некоторых простых массивах данных, например, если она применяется для сортировки уже отсортированных массивов. В худшем случае алгоритм имеет сложность $O(N^2)$. Кроме того, рекурсивные вызовы являются затратными операциями, и для небольших массивов данных быстрая сортировка будет работать не очень эффективно. Поэтому, в случае небольших объемов данных (<K, где K – некоторое небольшое число, например, 32), быстрая сортировка также работает не очень хорошо. В случае небольшого количества данных используют нерекурсивные методы сортировки, например, сортировку вставками или выбором, что дает прирост производительности алгоритма сортировки.

Рассмотрим *худший случай работы алгоритма быстрой сортировки* на примере упорядоченного массива данных, в котором N элементов. При первом разбиении массива процедурой partition производится N сравнений. Рекурсивный вызов от левой части массива размером (N-1)/2 и в дальнейшем от правой части массива размером (N-1)/2 приводит в общей сложности к N сравнениям. Рассуждая далее, получим общее количество сравнений для упорядоченного файла, равное N+(N-1)+(N-2)+…+2+1=(N+1)*N/2, то есть порядка N^2.

В среднем быстрая сортировка выполняет $2NlogN$ сравнений. Проведем оценку количества сравнений, выполняемых во время быстрой сортировки N>=2 случайно распределенных различных элементов.

$$\mathrm{C}_N = N + 1 + \frac{1}{N}\sum_{1\le k\le N}(C_{k-1} + C_{N-k})$$

N+1 – количество сравнений опорного элемента на первом шаге алгоритма с *N-1* элементом и еще двумя, когда индексы пересекутся. Каждый *k*-й элемент может быть центральным с вероятностью *1/N*. После выбора *k-го* опорного элемента получаем частично упорядоченные части массива с размерами *k-1* и *N-k*. Заметим, что $\mathrm{C}_0 + \mathrm{C}_1 + \cdots \mathrm{C}_{N-1} = \mathrm{C}_{N-1}+\mathrm{C}_{N-2}+\ldots+\mathrm{C}_0$. Значит,

$$\mathrm{C}_N = N + 1 + \frac{2}{N}\sum_{1\le k\le N} C_{k-1}.$$

Умножим обе части сравнения на N и вычтем эту же формулу для *N-1*.

$$N\mathrm{C}_N - (N-1)\mathrm{C}_{N-1} = N(N+1) - (N-1)N + 2\mathrm{C}_{N-1}.$$

$$N\mathrm{C}_N = 2N + (N+1)\mathrm{C}_{N-1}.$$

Поделим обе части на $N(N+1)$ и получим соотношение $\mathrm{C}_N/(N+1) = 2/(N+1) + \mathrm{C}_{N-1}/N$.

Применив дальнейшие преобразования, получим $\frac{\mathrm{C}_N}{N+1} = \frac{2}{N+1} + \frac{2}{N} + \frac{\mathrm{C}_{N-2}}{N-1} = \cdots = \frac{\mathrm{C}_2}{3} + \sum_{3\le k\le N}\frac{2}{k+1}$.

Из курса математического анализа в дальнейшем вы узнаете, что приблизительное значение $\frac{\mathrm{C}_N}{N+1} \approx 2\sum_{1\le k<N}\frac{1}{k} \approx 2lnN$. То есть, сложность алгоритма быстрой сортировки в среднем равно $O(Nlog\,N)$.

Поиск k-й порядковой статистики

Поиск *k*-й порядковой статистики заключается в поиске *k*-го элемента в упорядоченном по неубыванию массиве *A*, состоящего из *N* элементов. Первая порядковая статистика (*k=1*)– это минимум массива, *N*-я порядковая статистика (*k=N*) – это максимум массива. Медиана – это порядковая статистика, где *k=N/2*. Приведем пример k-х порядковых статистик на примере конкретного массива.

Исходный массив a[16]															
5	1	9	2	0	5	7	3	4	5	8	5	5	5	10	6

При *k=1*, *k*-я порядковая статистика равна 0. При *k=3*, *k*-я порядковая статистика равна 2.

Конечно, чтобы найти порядковую статистику, сначала можно отсортировать массив (например, при помощи быстрой сортировки за *NlogN*), а затем вывести *k*-й элемент. Но оптимальный алгоритм поиска *k*- порядковой статистики работает за линейное в среднем время со сложностью *O(N)*.

Идея алгоритма схожа с алгоритмом быстрой сортировки. На вход алгоритма подаются *l* и *r* – левая и правая граница поиска *k* -ой порядковой статистики, число *k*. Перестановку элементов делаем так: находим первый «неправильный» элемент слева, затем первый «неправильный» элемент справа, и если не сошлись указатели обмениваем найденные значения. В зависимости от того, в какой части частично-упорядоченного массива находится индекс *k*, переходит в левую или правую часть.

```
int k_st(vector<int>a, int l, int r,int k)
{
     if (l>=r) return a[l];
     int i=l, j=r, p=a[(l+r)/2];
     while (i<j){
          while(a[i]<p) i++;
          while(a[j]>p) j--;
          if (i<j) {swap(a[i],a[j]); i++; j--;}
     }
    if (k>=l&&k<i)
          return k_st(a,l,i-1,k);
     else if (j<k&&k<=r)
          return k_st(a,j+1,r,k);
     else
          return p;
}
```

Рассмотрим работу функции на примере массива a[7]={8, 9, 9, 1, 7, 5, 5}. k=5.

8 9 9 1 7 5 5 l=0 r=6 p=1
1 9 9 8 7 5 5 l=1 r=6 p=8
1 5 5 7 8 9 9 l=4 r=6 p=9
1 5 5 7 8 9 9 l=4 r=5 p=8

k-я порядковая статистика равна 8. Заметим, что алгоритм нахождения k-й порядковой статистики ставит k-й элемент упорядоченного массива на свое место. Также с помощью этого алгоритма можно найти k наименьших чисел массива – они будут находиться в интервале [0, k], но не будут упорядочены.

Сортировка слиянием

Идея алгоритма заключается в разделении массива на два подмассива. Каждый подмассив сортируется (к ним рекурсивно применяется сортировка слиянием). Полученные упорядоченные подмассивы объединяются в один отсортированный массив. Рекурсивное разбиение задачи на меньшие происходит до тех пор, пока размер массива не достигнет единицы – в массиве будет находиться один элемент (массив из одного элемента упорядочен).

```
void Merge(vector<int>&tm,int l1,int m2,int r)
{
     int k1=l1, k2=m2+1;
     vector<int>temp;
```

```
     while (k1<=m2&&k2<=r){
          if (tm[k1]>tm[k2])
               temp.push_back(tm[k2++]);
          else
               temp.push_back(tm[k1++]);
     }
     while (k1<=m2) temp.push_back(tm[k1++]);
     while (k2<=r) temp.push_back(tm[k2++]);
     for (int i=l1;i<=r;++i) tm[i]=temp[i-l1];
}

void MergeSort(vector<int>&t,int l, int r)
{
     if(l>=r) return;
     int m=(l+r)/2;
     MergeSort(t,l,m);
     MergeSort(t,m+1,r);
     Merge(t,l,m,r);
}
```

Вызов сортировки

```
     vector<int>a(n);
     for (int i=0;i<n;++i)
     {
          a[i]=rand()%11;
          printf("%d",a[i]);
     }
     MergeSort(a,0,a.size()-1);
```

Алгоритм имеет сложность $O(Nlog\ N)$.

Сортировка подсчётом

Сортировка QuickSort использовала операции сравнения элементов. Сортировка подсчетом (Counting Sort) не использует операции сравнения, и применяется для массива **A** дискретных данных (например, целых чисел, символов), которые принимают значения из небольшого диапазона $a_i = [0, K-1]$. Для данного алгоритма применяют вспомогательный массив **C**, элементы которого $c[i] = count(A, i)$, где $count(A, i)$ - количество элементов в массиве A, равных значению ***i***. Размер массива C равен максимальному элементу массива A.

Приведем пример исходного массива A и вспомогательного массива C.

a[i]	2	5	3	0	2	3	0	3

i	0	1	2	3	4	5	6	7
c[i]	2	1	3	2	2	3	2	3

По массиву **C** легко получить упорядоченный массив **A**. Делая проход по массиву C, мы столько раз последовательно включаем в массив **A** элемент ***i***, сколько раз он встречался в **A,** а именно $c[i]$ раз.

```
for (int i=0;i<n;++i)
     c[a[i]]++;
int l=0;
for (int i=0;i<k;++i){
          for (int j=1;j<=c[i];++j){
               a[l]=i;
               l++;
          }
}
```

Массив использует *O(K)* дополнительной памяти и имеет сложность *O(N+K)*. Сортировка подсчетом может быть реализована так, чтобы она обладала свойством стабильности.

Известны и другие сортировки, не использующие операции сравнения, например поразрядная сортировка.

Поразрядная сортировка

Рассмотрим идею поразрядной сортировки массива чисел. Например, необходимо выполнить сортировку массива трехразрядных чисел a[5]=(523, 153, 088, 554, 235). k=3 – количество разрядов. Числа даны в десятичной системе счисления – m=10. Дополнительно потребуются 10 векторов, отвечающие за цифру обрабатываемого разряда. Записываем числа исходного массива в векторы, отвечающие за цифры: 0, 1, 2, …, 9 в младшем разряде.

523, 153 – числа в векторе, отвечающем за цифру 3 .

554 - число в векторе, отвечающем за цифру 3.

235 – число в векторе, отвечающем за цифру 5.

088 -число в векторе, отвечающем за цифру 8.

Помещаем числа из вспомогательных векторов в исходный вектор.

Получаем: 523, 153, 554, 235, 088.

Производим распределение чисел во вспомогательные векторы по второму разряду.

523

235

153

554

088

Помещаем числа из вспомогательных векторов в исходный вектор.

Получаем: 523, 235, 153, 554, 088.

Производим распределение чисел во вспомогательные векторы по старшему разряду.

088

153

235

523

554

Помещаем числа из вспомогательных векторов в исходный вектор.

Получаем: 088, 153, 235, 523, 554. Получили отсортированный массив. Сложность сортировки O(kn+km). Используется дополнительная память O(n+m).

Квадратичные сортировки. Справочная информация

В качестве справочной информации, приведем программные реализации некоторых квадратичных сортировок, то есть сортировок, которые имеют сложность - $O(N^2)$. Приведенные ниже сортировки используют только сравнения и перестановки соседних элементов. Пузырьковая сортировка, сортировка вставками и выборочная сортировка не используют дополнительной памяти и являются стабильными.

Название сортировки	**Пример реализации**

Пузырьковая сортировка
BubbleSort
Сложность - $O(N^2)$.
	`for (int i=0;i<n-1;++i){`
	`  for (int j=0;j<n-i-1;++j){`
	`  if(a[j+1]<a[j])`
	`  swap (a[j],a[j+1]);`
	`  }`
	`}`

Сортировка вставками
InsertSort
Сложность в среднем $O(N^2)$, в лучшем случае $O(N)$.
Может сортировать элементы в процессе их ввода
	`  int n,x,j; int a[100];`
	`  cin>>n;`
	`  cin>>x;`
	`  a[0]=x;`
	`for (int i=1; i<n;++i){`
	`  cin>>x;`
	`  for (j=i-1; j>=0&&a[j] > x; j--)`
	`  a[j+1]=a[j];`
	`  a[j+1]=x;`

Выборочная сортировка
SelectSort
Сложность - $O(N^2)$
	`for(int i =0 ; i<n; ++i){`
	`  min=a[i]; min_i=i;`
	`  for(int j = i+1; j<n; ++j)`
	`  if(a[j]<min){`
	`  min = a[j]; min_i = j;`
	`  }`
	`  swap(a[min_i],a[i]);`
	`  }`

Для файлов небольших размеров сортировка вставками и сортировка выбором работают примерно в два раза быстрее пузырьковой сортировки, однако время работы любого из этих алгоритмов квадратично зависит от размера файла. Поэтому ни один из этих методов не нужно использовать для сортировки больших случайно упорядоченных файлов.

Задание

1. Составьте тест для выявления отличий результатов работы стабильной сортировки и стандартной сортировки sort() библиотеки STL. Тест, например, может содержать записи с двумя ключами типа string и int.
2. Проведите трассировку алгоритма сортировки quicksort для упорядоченного массива: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10. Оцените количество операций сравнения, выполненных алгоритмом на данном массиве.
3. Приведите примеры массивов из 10 элементов, для которых быстрая сортировка выполняет такое же количество сравнений, что и в худшем случае упорядоченности исходного массива.
4. Реализуйте нерекурсивный вариант нахождения k-й порядковой статистики массива.

5. При помощи бинарного дерева поиска, алгоритм построения которого был изложен в предыдущей лекции, реализовать функцию вывода всех чисел в диапазоне от *x* до *y* в возрастающем порядке.

Литература

Курс сайта www.stepic.org «Алгоритмы: теория и практика. Методы». Александр Куликов

Ворожцов А.В., Винокуров Н.А. Лекции «Алгоритмы: построение, анализ и реализация на языке программирования Си». – М.: Издательство МФТИ, 2007.

Курс сайта www.stepic.org «Введение в теоретическую информатику». Александр Шень.

Курс «Олимпиадные задачи по программированию». М.С. Густокашин. Электронный ресурс. http://prog.mathbaby.ru/dokuwiki/lib/exe/fetch.php?media=2015-9m:9m-binsearch-lection.pdf

Седжвик Р. Алгоритмы на C++. – М.: ООО «И.Д. Вильямс», 2014 .