Алгоритмы поиска и сортировки

Лямин Андрей Владимирович

Задача сортировки
Задача поиска

- *Задачей сортировки* является преобразование исходной последовательности в последовательность, содержащую те же записи, но в порядке возрастания [или убывания] значений.

- В *задаче поиска* необходимо разработать алгоритм, позволяющий установить, есть ли заданное значение в списке, который отсортирован согласно некоторому правилу, позволяющему упорядочить его элементы. Если это значение в списке присутствует, поиск будет считаться успешным, в противном случае его считают завершившимся неудачей.

Сортировка включением

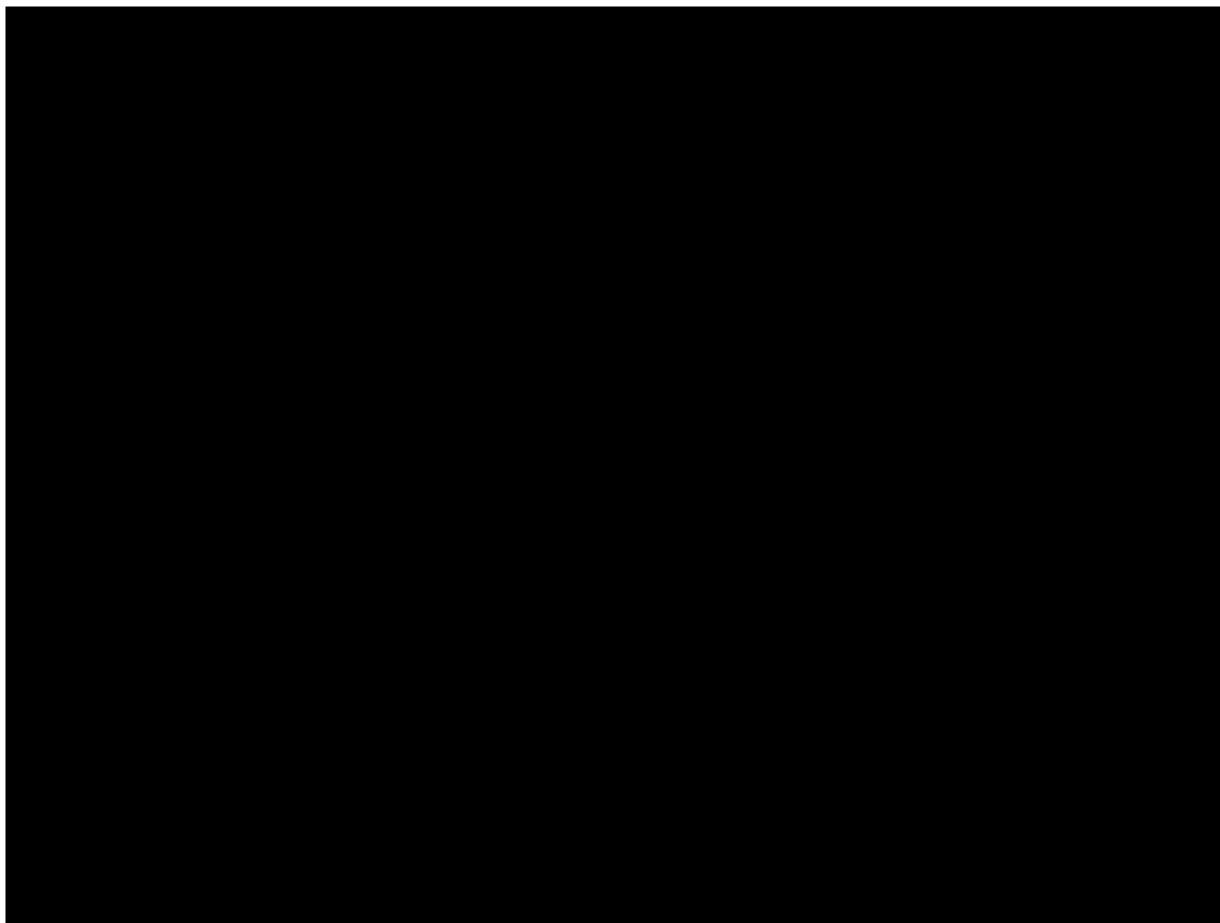
Одним из наиболее простых и естественных методов сортировки является сортировка с простыми включениями.

Пусть имеется массив $M[0]$, $M[1]$, ..., $M[n\text{-}1]$.. Для каждого элемента массива, начиная со второго, производится сравнение с элементами с меньшим индексом [элемент $M[i]$ последовательно сравнивается с элементами $M[i\text{-}1]$, $M[i\text{-}2]$ ...] и до тех пор, пока для очередного элемента $M[j]$ выполняется соотношение $M[j] > M[i]$, $M[i]$ и $M[j]$ меняются местами. Если удается встретить такой элемент M[j], что $M[j] \leq M[i]$, или если достигнута нижняя граница массива, производится переход к обработке элемента $M[i+1]$ [пока не будет достигнута верхняя граница массива].

Пример

```
function Grade (Array[], LengthOfArray) {
if (LengthOfArray < 2) then return Array[];
else {
CurrentIndex:=1;
while (CurrentIndex < LengthOfArray) do {
CurrentElement:=Array[CurrentIndex]; i:=0;
if (CurrentElement < Array[CurrentIndex-i-1]) then Flag=1; else Flag=0;
while (Flag=1) do {
Array[CurrentIndex-i]:= Array[CurrentIndex-i-1];
i:=i+1; Flag=0;
if (i < CurrentIndex) then {
if (CurrentElement < Array[CurrentIndex-i-1]) then Flag=1;
}
}
Array[CurrentIndex-i]:= CurrentElement;
CurrentIndex:= CurrentIndex+1;
}
return Array[];
}
}
```

Ролик



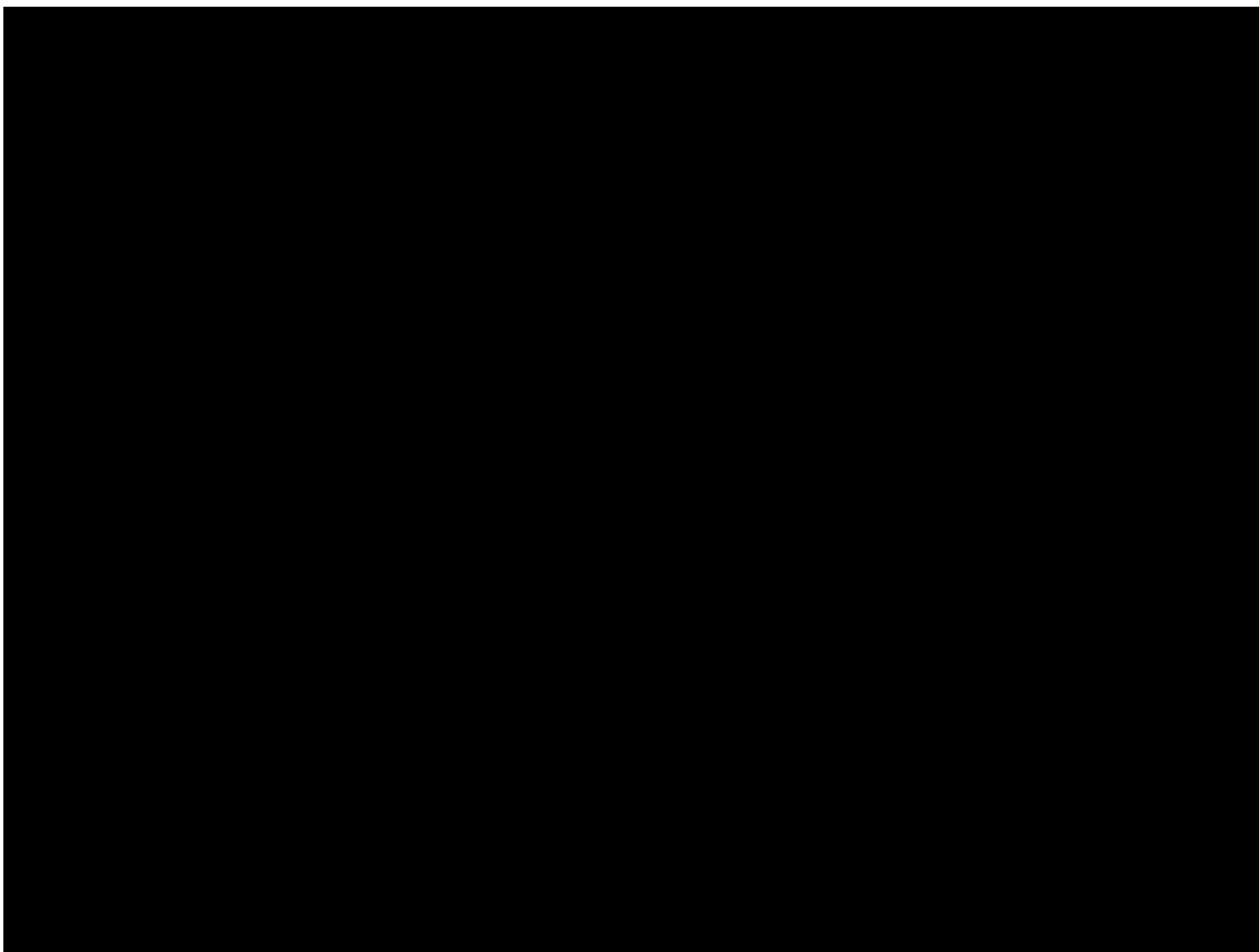
Обменная сортировка

Простая обменная сортировка [в просторечии называемая "методом пузырька"] для массива $M[0]$, $M[1]$, ..., $M[n\text{-}1]$ работает следующим образом. Начиная с конца массива сравниваются два соседних элемента [$M[n\text{-}1]$ и $M[n\text{-}2]$]. Если выполняется условие $M[n\text{-}2] > M[n\text{-}1]$, то значения элементов меняются местами. Процесс продолжается для $M[n\text{-}2]$ и $M[n\text{-}3]$ и т.д., пока не будет произведено сравнение $M[1]$ и $M[0]$. Понятно, что после этого на месте $M[0]$ окажется элемент массива с наименьшим значением. На втором шаге процесс повторяется, но последними сравниваются $M[2]$ и $M[1]$. И так далее. На последнем шаге будут сравниваться только текущие значения $M[n\text{-}1]$ и $M[n\text{-}2]$. Понятна аналогия с пузырьком, поскольку наименьшие элементы [самые "легкие"] постепенно "всплывают" к верхней границе массива.

Пример

```
function Grade (Array[], LengthOfArray) {
 if (LengthOfArray < 2) then return Array[];
 else {
LimitIndex:=1;
while (LimitIndex < LengthOfArray) do {
 CurrentIndex:= LengthOfArray-1;
 while (CurrentIndex > 0) do {
if (Array[CurrentIndex-1] > Array[CurrentIndex]) then {
 Temp:= Array[CurrentIndex-1];
 Array[CurrentIndex-1]:= Array[CurrentIndex];
 Array[CurrentIndex]:=Temp;
}
CurrentIndex:= CurrentIndex-1;
 }
 LimitIndex:= LimitIndex+1;
}
return Array[];
 }
}
```

Ролик



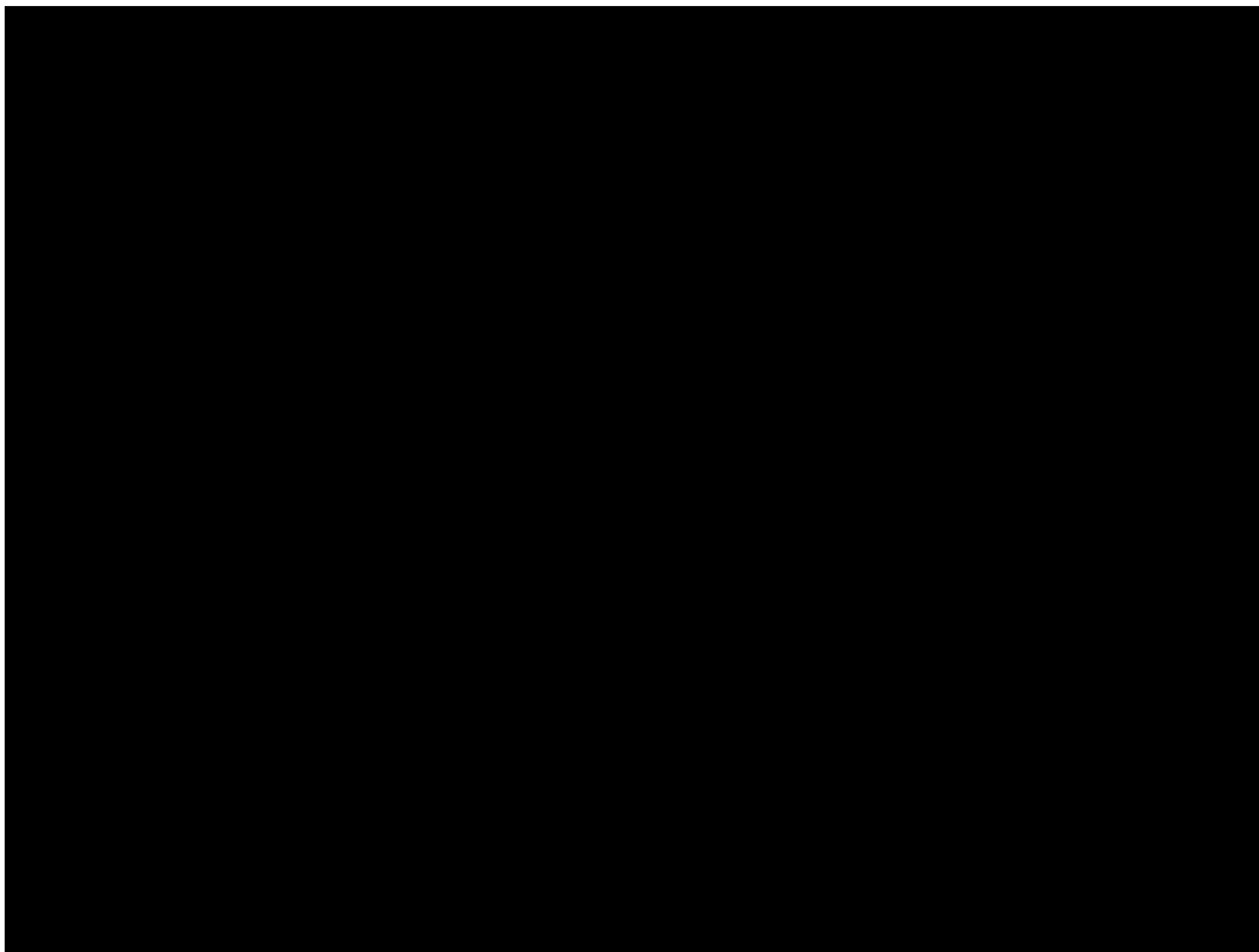
Сортировка выбором

При сортировке массива $M[0], M[1], \ldots, M[n\text{-}1]$ методом простого выбора среди всех элементов находится элемент с наименьшим значением $M[i]$, и $M[0]$ и $M[i]$ обмениваются значениями. Затем этот процесс повторяется для получаемых подмассивов $M[1], M[2], \ldots, M[n\text{-}1], \ldots M[j], M[j+1], \ldots, M[n\text{-}1]$ до тех пор, пока мы не дойдем до подмассива $M[n\text{-}1]$, содержащего к этому моменту наибольшее значение.

Пример

```
function Grade (Array[], LengthOfArray) {
 if (LengthOfArray < 2) then return Array[];
 else {
FirstIndex:=1;
while (FirstIndex < LengthOfArray) do {
 i:=FirstIndex;
 MinValue:= Array[i-1];
 MinIndex:= i-1;
 while (i < LengthOfArray) do {
if (MinValue > Array[i]) then {
 MinValue:= Array[i];
 MinIndex:= i;
}
i:=i+1;
 }
 if (MinIndex <> (FirstIndex-1)) then {
Array[MinIndex]:= Array[FirstIndex-1];
Array[FirstIndex-1]:= MinValue;
 }
 FirstIndex:=FirstIndex+1;
}
```

Ролик



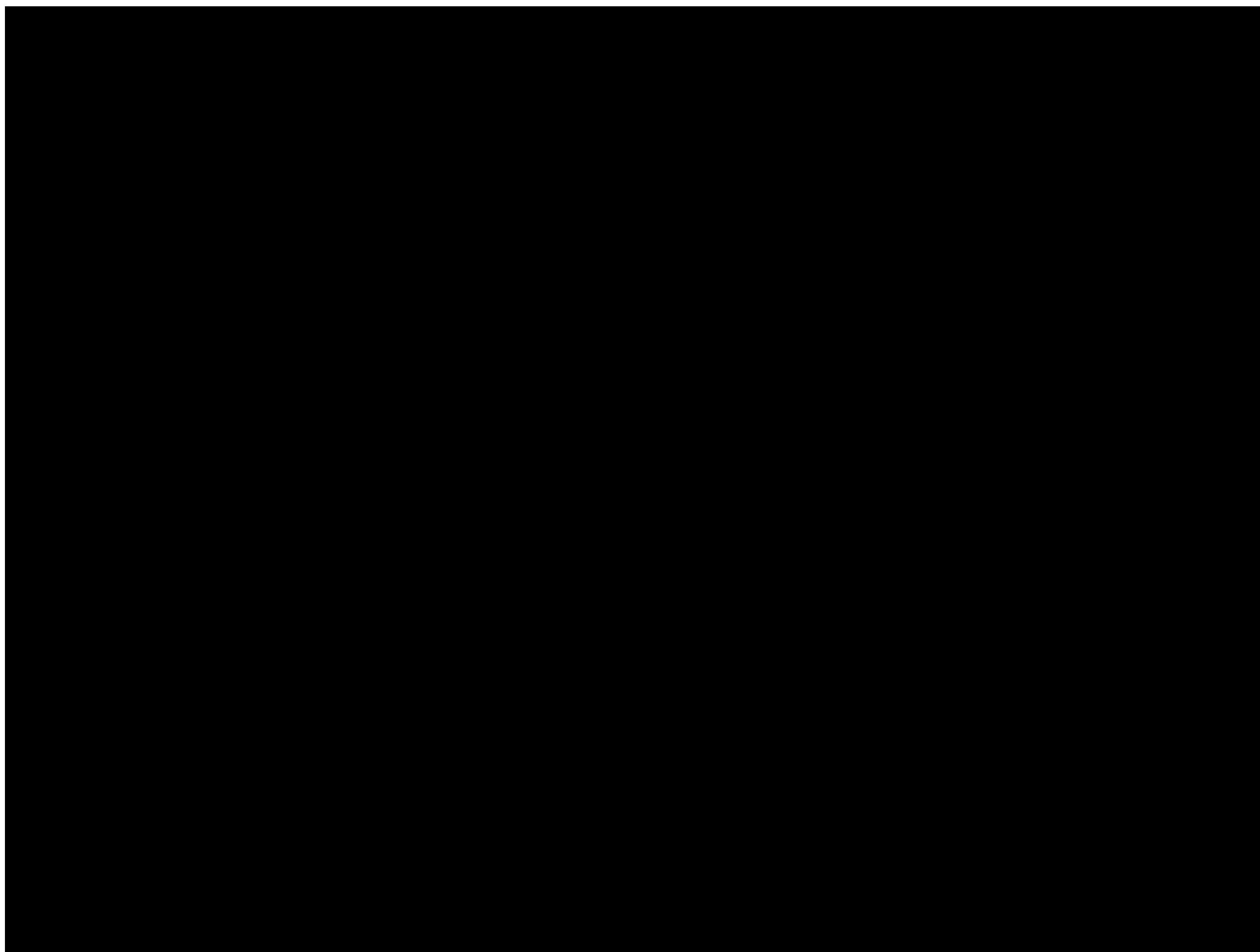
Последовательный поиск

Решение задачи поиска можно получить, воспользовавшись *алгоритмом последовательного поиска*. Псевдокод данного алгоритма представлен ниже. Алгоритм возвращает индекс элемента массива *Array*[], значение которого равно *Value*, когда такой элемент существует, или **Null**, когда такого элемента в массиве нет. Предполагается, что массив имеет *LengthOfArray* элементов.

Пример

```
function Search(Value, Array[], LengthOfArray) {
 if (LengthOfArray < 1) then return Null;
 else {
CurrentIndex:=0;
while (Value > Array[CurrentIndex] and CurrentIndex < LengthOfArray) do
 CurrentIndex:= CurrentIndex+1;
if (Array[CurrentIndex]=Value) then return CurrentIndex; else return Null;
 }
}
```

Ролик



Двоичный поиск

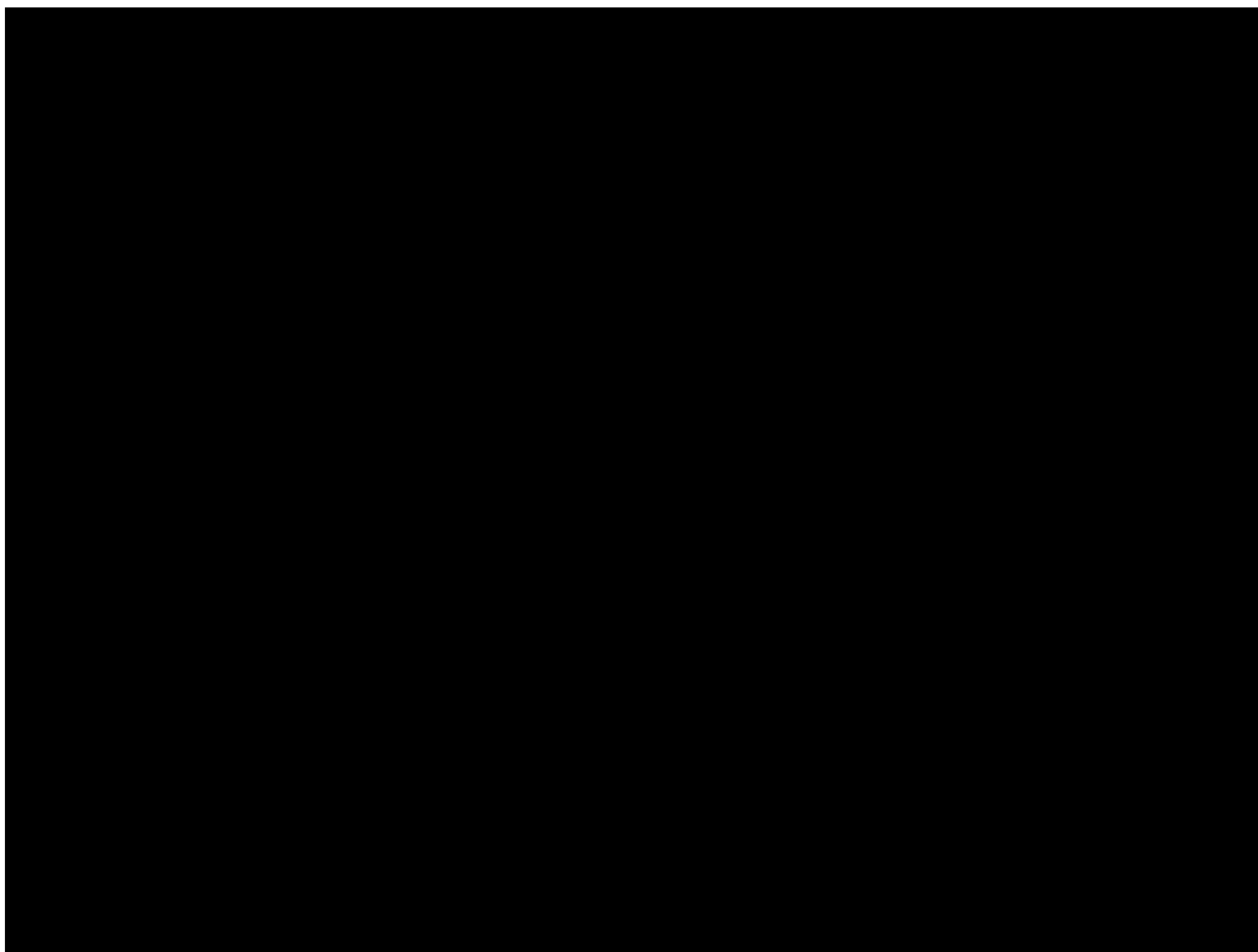
Алгоритм двоичного поиска предполагает на каждом шаге деления массива на две части и отбрасывание той его части, элементы которой заведомо имеют значения либо меньше, либо больше искомого. Именно это повторяющееся деление на два послужило причиной того, что данный алгоритм был назван двоичным поиском. Псевдокод алгоритма приведен ниже. Входными параметрами алгоритма являются: *Value* - искомое значение; *Array*[] - массив элементов; *FirstIndex* - индекс начала фрагмента массива; *LastIndex* - индекс конца фрагмента массива.

Пример

```
function Search(Value, Array[], FirstIndex, LastIndex) {
 MiddleIndex:= FirstIndex+(LastIndex-FirstIndex +1) div 2;
 if (Value=Array[MiddleIndex]) then return MiddleIndex;
 if (Value < Array[MiddleIndex]) then {
if (FirstIndex=MiddleIndex) then return Null;
Index:=Search(Value, Array[],FirstIndex, MiddleIndex-1);
if (Index <> Null) then return Index; else return Null;
 }
 if (Value > Array[MiddleIndex]) then {
if (MiddleIndex=LastIndex) then return Null;
Index:=Search(Value, Array[],MiddleIndex+1, LastIndex);
if (Index <> Null) then return Index; else return Null;
 }
}
```

В приведенном выше псевдокоде последовательность символов **div** обозначает операцию целочисленного деления.

Ролик



Эффективность алгоритмов

- Эффективность алгоритма принято оценивать количеством элементарных операций, например сравнений, которые необходимо выполнить для решения задачи, а также количеством памяти, которая требуется для выполнения алгоритма.
- Анализ включает изучение ситуаций, в которых алгоритм демонстрирует свои наилучшие свойства, ситуаций, когда его эффективность минимальна, а также оценку его средней производительности.

Эффективность алгоритмов

Название алгоритма	Минимальное число сравнений	Среднее число сравнений	Максимальное число сравнений
Сортировка включением	n-1	$(n^2 + n - 2)/4$	$(n^2 - n)/2$
Сортировка выбором	$(n^2 - n)/2$	$(n^2 - n)/2$	$(n^2 - n)/2$
Обменная сортировка	$(n^2 - n)/2$	$(n^2 - n)/2$	$(n^2 - n)/2$
Последовательный поиск	1	$(n+1)/2$	n
Двоичный поиск	1	$(\log_2(n)+1)/2$	$\log_2(n)$