# ЛЕКЦИЯ 15.

***Вопросы***

**1. Оценка алгоритмов**
**2. Алгоритмы поиска**
**2.1. Линейный поиск**
**2.2. Бинарный поиск**
**3. Преобразование массивов**

## 10.1. *Оценка алгоритмов*

Для большинства задач существует много различных алгоритмов. Какой из них выбрать для решения конкретной задачи? Чаще всего интересен порядок роста необходимых для решения задачи ***времени*** и ***емкости памяти*** при увеличении входных данных. Свяжем с каждой задачей некоторое число , называемое ее размером, которое выражало бы меру количества входных данных.

Например, размером задачи умножения матриц, может быть наибольший размер матриц сомножителей. Размером задачи сортировки массива – количество чисел.

***Пространственная (или емкостная) сложность*** измеряется количеством памяти, требуемой для выполнения алгоритма .

***Временная сложность*** алгоритма определяется временем, необходимым для его выполнения.

Лучший способ сравнения эффективностей алгоритмов состоит в сопоставлении их порядков сложности. Этот метод применим как к временной, так и пространственной сложности.

Если алгоритм обрабатывает входные данные размера n за время $cn^2$, где c – некоторая константа, то временная сложность этого алгоритма есть $O(n^2)$ (читается “порядка $n^2$”).

Точнее, говорят, что неотрицательная функция f(n) есть O(g(n)), если существует такая константа c, что для для всех $n$ из некоторой окрестности точки $n_0$ имеет место неравенство $f(n)<=c*g(n)$.

Обозначают

$f(n) = O(g(n))$

И функция f(n) есть o(g(n)), если для любого c>0 существует такая окрестность точки $n_0$, что для для всех $n$ из этой окрестности имеет место неравенство $f(n)<c*g(n)$.

Обозначают

$f(n) = o(g(n))$

В формулах вместо знака = может использоваться знак принадлежности:
Существуют три важных правила для определения сложности.

1. O(k*f)=O(f)
2. O(f*g)=O(f)*O(g) или O(f/g)=O(f)/O(g)
3. O(f+g) равна доминанте O(f) и O(g).

Здесь k обозначает константу, а f и g - функции.

Первое правило декларирует, что постоянные множители не имеют значения для определения порядка сложности.

1,5*N=O(N)

Из второго правила следует, что порядок сложности произведения двух функций равен произведению их сложностей.

O((17*N)*N) = O(17*N)*O(N) = O(N)*O(N)=O(N*N) = $O(N^2)$

Из третьего правила следует, что порядок сложности суммы функций определяется как порядок доминанты первого и второго слагаемых, т.е. выбирается наибольший порядок.

$O(N^5+N^2)=O(N^5)$

**O(1)**

Большинство операций в программе выполняются только раз или только несколько раз. Это алгоритм константной сложности. Любой алгоритм, всегда требующий независимо от размера данных одного и того же времени, имеет константную сложность.

**O(N)**

Время работы программы линейно обычно когда каждый элемент входных данных требуется обработать лишь линейное число раз.

**$O(N^2)$, $O(N^3)$, $O(N^a)$**

Полиномиальная сложность. $O(N^2)$-квадратичная сложность, $O(N^3)$- кубическая сложность

**O(Log(N))**

Когда время работы программы логарифмическое, программа начинает работать намного медленнее с увеличением N. Такое время работы встречается обычно в программах, которые делят большую проблему в маленькие и решают их по отдельности.

**O(N*log( N))**

Такое время работы имеют те алгоритмы, которые делят большую проблему в маленькие, а затем, решив их, соединяют их решения.

**$O(2^N)$**

Экспоненциальная сложность. Такие алгоритмы чаще всего возникают в результате подхода именуемого метод грубой силы.

Программист должен уметь проводить анализ алгоритмов и определять их сложность. Временная сложность алгоритма может быть посчитана исходя из анализа его управляющих структур.
Алгоритмы без циклов и рекурсивных вызовов имеют константную сложность. Если нет рекурсии и циклов, все управляющие структуры могут быть сведены к структурам константной сложности. Следовательно, и весь алгоритм также характеризуется константной сложностью.

Например, рассмотрим алгоритм обработки элементов массива.

```
for (i=0;i<N;i++)
{
...
}
```

Сложность этого алгоритма O(N), т.к. тело цикла выполняется N раз, и сложность тела цикла равна O(1).

Если один цикл вложен в другой и оба цикла зависят от размера одной и той же переменной, то вся конструкция характеризуется квадратичной сложностью.

```
 for (i=0;i<N;i++)
     for (j=0;j<N;j++)
     {
    …
     }
```

Сложность этой программы $O(N^2)$.

Сложность ниже приведенного алгоритма $O(N^3)$.
O(H)=O(1)+O(1)+O(1)=O(1);
O(I)=O(N)*(O(F)+O(J))=O(N)*O(доминанты условия)=O(N);
$O(G)=O(N)*(O(C)+O(I)+O(K))=O(N)*(O(1)+O(N)+O(1))=O(N^2)$;
$O(E)=O(N)*(O(B)+O(G)+O(L))=O(N)* O(N^2)= O(N^3)$;
$O(D)=O(A)+O(E)=O(1)+ O(N^3)= O(N^3)$

```
      Writeln('Введите ограничение');
   A  Readln(N);
      small:=1;

      While small < N do
        begin
      B   next:=small;
          While next <= N do
           begin
      C     last:=next;
            While last<=N do
             begin
               if (last<=small*2) and (next<=small*2) and
               (last*last=small*small+next*next) then
D  E  G  I  F    begin
               H   writeln(small);
                   writeln(next);
                   writeln(last);
                 end;
            J  inc(last);
             end;
      K     inc(next);
           end;
      L   inc(small);
        end;
```

## 10.2. ***Алгоритмы поиска***

Будем рассматривать на примере поиска в массиве.

### 10.2.1. ***Линейный поиск***

Линейный, последовательный поиск — нахождения заданного элемента в некотором массиве. Поиск значения осуществляется простым сравнением очередного значения и искомого, если значения совпадают, то поиск считается завершённым.

Если у массива длина = N, найти решение можно за N шагов.

Сложность алгоритма - $O(n)$. В связи с малой эффективностью линейный поиск используют только если N не большое.

***Алгоритм:***

1. Определяем начало интервала поиска и выбираем элемент для поиска.
2. Сравниваем образец с выбранным элементом. Если элемент совпадает с образцом, то определяем его индекс и переходим к шагу 4.
3. Если элемент не совпадает с образцом, то выбираем следующий, если он есть и переходим к шагу 2.
4. Если индекс определен, то элемент найден, иначе - искомого элемента в

массиве нет.

```
int i=0;
int x; // образец
int n; // размерность массива
int a[100];
while (i<n)&&(a[i]!=x))
      i=i+1;
if (i==n) printf("no");
else printf("%d  ",i);
```

Можно упростить логическое выражение, которое состоит из двух членов.

Уберем условие (i<n), но при этом необходимо гарантировать, что совпадение произойдет всегда. Для этого достаточно в конец массива поместить дополнительный элемент со значением x. Такой вспомога-тельный элемент называется “барьером”.

Теперь массив будет описан так:

```
int a[101];
a[n]=x;
while (a[i]!=x) i=i+1;
int LineSearch (int *a, int n, int x)
{
  for (int i=0;i<n; i++)
    if (a[i]==x)
       return i;
  return -1; //элемент не найден
}
```

### 10.2.2. *Бинарный поиск*

***Или метод дихотомии или метод половинного деления*.**

Как обычно, за скорость взимается плата: массив должен быть упорядочен. Сам по себе этап предварительного упорядочения, или *сортировки*, обходится недешево, во всяком случае - дороже однократного линейного поиска.

***Алгоритм:***

1. Определяем середину интервала поиска.
2. Сравниваем образец с элементом, расположенным посередине. Если образец оказался больше, то областью дальнейшего поиска становится правая половина; в противном случае - левая половина интервала, но в любом случае индексный интервал уменьшается вдвое. (Если осуществить еще одну проверку, то можно установить и совпадение, после чего дальнейшие шаги не обязательны.)

3. Если остался интервал единичной длины, то переходим к заключительному шагу 4, в противном случае - к шагу 1.
4. Либо единственный элемент интервала совпадает с образцом, либо - искомого элемента в массиве нет.

```
int BinarySearch(int a[],
              int n, int x)
{
  int i, j, middle;
  i=0; j=n-1;
  while (i<=j)
  {
     middle=(i+j)/2;
     if (x==a[middle])
         return middle;
     else
     if (x>a[middle])
        i=middle+1;
     else
       j=middle-1;
  }
  return -1;
}
```

Оценим алгоритм бинарного поиска в массиве.

Первая итерация цикла имеет дело со всем массивом. Каждая последующая итерация делит пополам размер подмассива. Так, размерами массива для алгоритма являются

$n\ n/2^1\ n/2^2\ n/2^3\ n/2^4\ \dots\ n/2^m$

В конце концов будет такое целое m, что

$n/2^m<2$ или $n<2^{m+1}$

Так как m - это первое целое, для которого $n/2^m<2$, то должно быть верно

$n/2^{m-1}>=2$ или $2^m<=n$

Из этого следует, что

$2^m<=n<2^{m+1}$

Возьмем логарифм каждой части неравенства и получим

$m<=\log_2 n=x<m+1$

Значение m - это наибольшее целое, которое <=x.

Итак, $O(\log_2 n)$.

***Упр. Читать “Оценка программ”.***

## 10.3. Преобразования массивов

### Переворачивание элементов массива

```
void Revers(int *a, int n)
 // Переворачивание элементов  массива
{int k;
 for (int i=0;i<n/2;i++)
 {
   k=a[i];
   a[i]=a[n-i-1];
   a[n-i-1]=k;
 }
}


void ReversP(int *a, int n)
{
  int k;
  for (int i=0;i<n/2;i++)
  {
     k=*(a+i);
     *(a+i)=*(a+n-i-1);
     *(a+n-i-1)=k;
   }
}

void  ReversL(int *a, int n)
{
  for (int i=0;i<n/2;i++)
  {
    *(a+i)^=*(a+n-i-1);
    *(a+n-i-1)^=*(a+i);
    *(a+i)^=*(a+n-i-1);
  }
}
```

### Расширение и сжатие массивов

При обработке массивов можно вставлять и удалять элементы

```
void MoveRight(int * a,  int *n, int num)
{     //сдвигает все элементы на
      // одну позицию вправо до номера num
 for (int i=*n; i>=(num+1); i--)
```

```
   a[i]=a[i-1];
  (*n)++; //увелич реальный размер массива
//не путать с *n++ !!!!!
}

void MoveLeft(int *a, int *n, int num)
{//сдвигает все элементы на
 // одну позицию влево с номера num
 for (int i=num; i<*n-1; i++)
   a[i]=a[i+1];
 *n=*n-1;
}
```

***Дублирование четных в массиве***

```
 void InsertCh(int *a, int *n)
 {int i=0;
  while (i<*n)
  {
   if (a[i]%2==0)
   { MoveRight(a, n, i+1);
     //Сдвигаем и  добавляем элемент
     a[i+1]=a[i]; i++ ;
   }
   i++;
  }
}
```

***Удаление чисел равных item в массиве***

```
 void DeleteCh(int *a, int *n,int item)
 {
  int i=0;
  while (i<*n)
  {
   if (a[i]==item)
       MoveLeft(a, n, i);
       //Сдвигаем и  удаляем элемент
   else
       i++;   }
  }
```

**КОНЕЦ ЛЕКЦИИ**