АНАЛИЗ НА АЛГОРИТМИ

Дефиниции:

- **T(N) = O(f(N)) ако съществуват c , N и за ∀ N >= No →**
  - **T(N) <= c f(N);**

цели: пренебрегване малки членове, облекчен анализ, оценка по горна граница.

пр: $N (N - 1) / 2 \to N^2 / 2$

или $N^2 = O(N^3)$

или $2a_0 N H_N + a_1 N + a_2 \to 2a_0 N H_N + O(N)$ (за големи N)

вътр.цикъл; външен цикъл

• $T(N) = \Omega(g(N))$ ако съществуват const c и n_0, такива че $T(N) >= cg(N)$ за $N>n_0$

* $T(N) = \Theta(h(N))$ ако и само ако $T(N) = O(h(N))$ и $T(N) = \Omega(h(N))$
казваме : "growth rate" на T(N) = growth rate h(N)

ефект от удвояване на големината на задача върху t:

1	- никакъв	2^N	- квадратичен
lgN	- лек	N^2	- с коефициент 4
N	- удвоява	N^3	- с коеф. 8
NlgN	- малко > от двойно		

пр: $(N + O(1))(N + O(\log N) + O(1)) =$

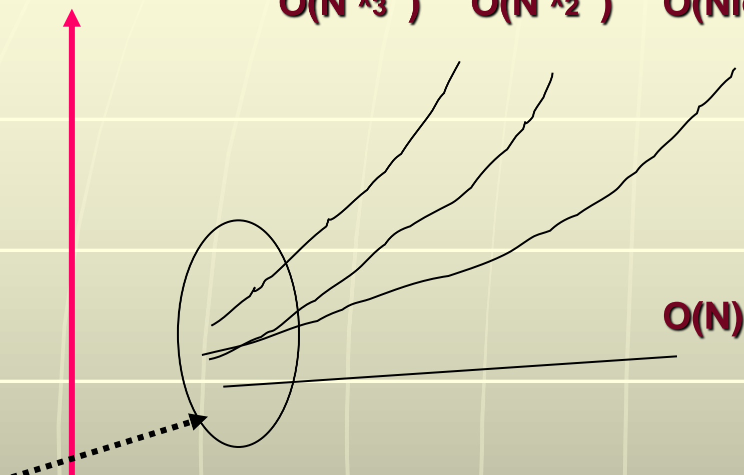
$= N^2 + O(N) + O(N\log(N)) + O(\log N) + O(N) + O(1) =$

$= N^2 + O(N\log N)$ приблизит. $= N^2$ за големи N.

- **правило 1**: ако $T1(N) = O(f(N))$ и $T2(N) = O(g(N))$, то
  a) $T1(N) + T2(N) = \max(O(f(N)), O(g(N)))$.
  b) $T1(N) * T2(N) = O(f(N) * g(N))$.
- **правило 2**: ако T(N) е полином от степен k, то $T(N) = \Theta(N^k)$
  напр: вместо $T(N) = O(2N^2) \rightarrow T(N) = \Theta(N^2)$
- **правило 3** : $\log^k N = O(N)$ за $\forall$ k (логаритмите растат бавно)
- **Правило 4**: без константи в изразите с O: вместо $T(N) = O(2N^2) \rightarrow$
  $\rightarrow T(N)= O(N^2)$

какво анализираме:
- t (зависи от комп., компилатор,…..)
- вх. поредица (средна, лоша, добра (best case))

пример:(анализът по-късно):от вх. поредица числа да се намери подредица с max сума (по A_k)



4 алгоритъма с огромни различия в оценките.

при малък брой ч-ла, различията са незначителни→усилието за добър алгор., зависи от диапазона на вх.ч-ла.

алгоритъм	1	2	3	4
време (сек.)	O(N ^3)	O(N^2)	O(NlogN)	O(N)
N=10	0,00103	0,00045	0,00066	0,00034
N=100	0,47015	0,01112	0,00486	0,00063
N= 1000	448,77	1,1233	0,0584	0,00333
N= 10 000	NA	111,13	0,6863	0,03042
N= 100 000	NA	NA	8,013	0,2867

- данните: Mark Weiss, Algorithm Analysis

начален пример за анализ:

```
int sum (int n)
{           int partialSum;
            partialSum = 0;                          1
            for( int I = 1; I <= n; i++)             2N +2
                    partialSum +=   i* i  * i;       4N      6N + 4  →O(N)
            return partialSum;                       1
}
```

$$\sum_{i=1}^{N} i^3$$

Общи правила в анализа на алгоритми

- 1. за цикли for: N пъти 't' тялото.
- 2. за вложени цикли: произведение от оценките

```
пр:  for (i= 1; i < n; i++)
        for (j = 0; j < N ; j++)                     O(N ^2  )
            k++;
```

- 3. за последователни оператори: важи максималното

```
for ( I = 0; … )
        a[I] = 0;                                    O(N)
for ( I = 0; …………….)
        for ( j = k; ………….)                         O(N^2   )
            a[I] = a[j]  + I;
```

4. if ('тестов израз') S_1 else S_2 → $t \le t_{test} + \max(t_{S1}, t_{S2})$

5. при рекурсия:

А.

```
long fact( int i)
{ if ( n <= 1)  return 1;
        else
        return n * fact(n – 1) ; }
```

опашна → преминава в цикъл → O(N)

Б.

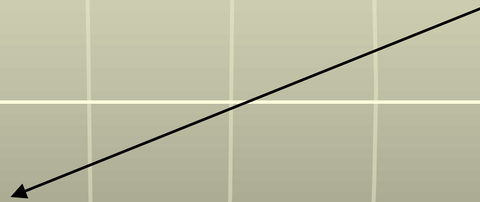
```
long  fib( int n)
{ if( n < = 1)  return 1;
else
return fib( n – 1)  + fib( n – 2);}
```

тежък анализ, бавна.

нека T(N) → fib(n)

$T(0) = T(1) = 1$

$T(N) = T(N-1) + T(N-2) + 2$



- ще докажем по индукция: $fib(N) < (5/3)^N$

$F_0, F_1 = 1; F_2 = 2; F_3 = 3$ → съществува k $F_k < (5/3)^k$.

ще докажем : $F_{k+1} < (5/3)^{(k+1)}$

$F_{k+1} = F_k + F_{k-1} < (5/3)^k + (5/3)^{(k-1)} < (3/5)(5/3)^{(k+1)} + (3/5)^2 * (5/3)^{(k+1)}$

$< ((3/5) + (9/25))(5/3)^{(k+1)} < (24/25)(5/3)^{(k+1)} < (5/3)^{(k+!)}$

експоненциална оценка – не е добре!

Алгоритми за откриване max сума на подредица

1. преглед на всички възможности

O(N^3)

```
int maxSubSum1(const vector<int>  &a)
{ int maxSum = 0;
for( int I = 0; I < a.size(); I++)
        for( int j = I; j < a.size(); j++)
        { int thisSum = 0;
         for( int k = I; k < j; k++)
                thisSum += a[k];
        if( thisSum > maxSum)
                maxSum =  thisSum; }
return maxSum;          }
```

O(1*N*N*N) = O(N ^3)

$$\sum_{i=0}^{N-1} \sum_{j=0}^{N-1} \sum_{k=i}^{N-1} *1$$

2. премахваме третия for:

```
int maxSubSum2( const vector <int> & a)
    { int maxSum = 0;
    for (int I = 0; I < a.size(); I++)
    { int thisSum = 0; //преместено тук

 for( int j = I; j < a.size(); j++)
    {           thisSum += a[j];
                if( thisSum > maxSum)

                maxSum = thisSum;           }}

 return maxSum; }
```

натрупваме сумата като сме я нулирали в началото на външния for.(От вар.1 знаем че:)

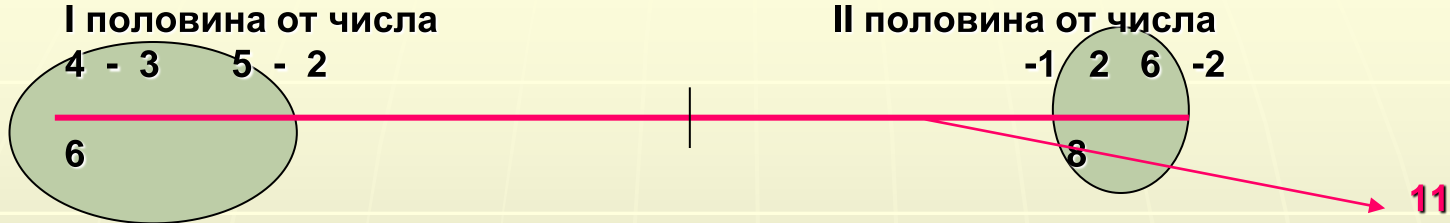
$$\sum_{k=I}^{j} A\kappa = Aj + \sum_{k=i}^{j-1} Ak$$

O(N^2)

3. divide& conquer стратегия:

цепим проблема на 2 части и т. н. рекурсивно. Съединяваме двете решения.

Мах сума може да е на 3 места:




```
int maxSumRec( const vector< int> &a, int left, int right)
        { if( left == right)                              // базов случай
                    if( a[left]) > 0)                     return a[left];
                              else                        return 0;

int center = (left + right )  / 2;
int maxLeftSum = maxSumRec( a, left, center);                    // T(N/2)
int maxRightSum = maxSumRec( a, center + 1, right);              //T(N/2)

int maxLeftBorderSum = 0; leftBorderSum = 0;
for( int I = center; I > = left; I--)
        { leftBorderSum += a[I];
if(leftBorderSum > maxLeftBorderSum) maxLeftBorderSum = leftBorderSum }

 int maxRightBorderSum = 0; rightBorderSum = 0;
 for( int j = center + 1; j < right; j++)
 {                    rightBorderSum  +=a[j];
 if(rightBorderSum > maxRightBorderSum)  maxRightBorderSum = rightBorderSum; }
```

O(N)

return max3(maxLeftSum, maxRightSum, MaxLeftBorderSum + maxRightBorderSum);}

```
int maxSubSum3 (const vector<int> &a)
{ return maxSumRec( a,0, a.size() – 1);        }
```

анализ (както Фибоначи): нека T(N) за N числа; за N = 1 → T(1) = 1;
при N > 1 имаме 2 рекурсии. Всеки for → O(N)
Следователно

T(N) = 2 T(N/2) + O(N) → 2T(N/2) + N

ако N = 2^к T(N) = N * (k+1)
= NlogN+N = O(NlogN)

T(2) = 4 = 2 * 2
T(4) = 12 = 4 * 3
T(8) = 32 = 8 * 4
T(16) = 80 = 16 * 5

O(NlogN)

ако N != 2 ^к - анализът е по-тежък , но резултатът е същия.

- **4.линейно време**

```
int maxSubSum4 ( const vector <int> &a)
    {int maxSum = 0; thisSum = 0;
            for( int j = 0; j < a.size(); j++)
                    { thisSum += a[j];
                    if( thisSum > maxSum)
                            maxSum = thisSum;
                    else  if( thisSum < 0)        // отриц. ч-ло или  сума едва ли е
                    thisSum = 0;       }          // част от търсена подредица.
return maxSum;      }                             // можем да скочим към i+1
                                                  // и дори към j+1.
```

O(N)

а)всеки момент алгоритъмт дава решението до което е стигнал (on-line алгоритми)
б)не изисква много памет.

АНАЛИЗ С ЛОГАРИТМИЧНИ ОЦЕНКИ

1.оценки на divide&conquer

2. Ако log в оценката не е очевиден, то все пак може да се счита че алгоритъмт е O(logN) ако изисква const време (O(1)) за разделяне задачата на части (обикновено ½).

- Ако е необходимо const време за частичното решаване на проблема (напр. намаляване сложността с 1), то оценката е около O(N).

Някои примери с log оценка:

- ***двоично търсене:***

  търсим 'X' в сортирана редица A0, A1, …An-1.

Вариант: ***А. последователно***

Б. чрез разделяне

```
int low = 0, high = a.size() – 1;
while( low <= high)
    {       int mid = ( low + high) / 2;
            if( a[mid] < x)                 low = mid + 1;
            else if(  x < a[ mid])          high = mid – 1;
                  else  return mid;               // открито
}           return  NOT_FOUND; }
```

O(1) вътре във всеки цикъл. Циклите са <= $\lceil \log(N - 1) \rceil$ + 2.

следоват: O(log(N))

Напр. ако high – low = 128, то се цикли за 64, 32, 16, 8, 4, 2, 1, 0, -1)

Евклидов алгоритъм за НОД(M, N) M >= N

```
long gcd(long m, long n)
{ while ( n != 0)
{long rem = m % n;
 m = n; n = rem; }
return m;}
```

// доказва се че след 2 итерации остатъкът
// е поне наполовина смален. Следователно:

// оценка: **2 logN = O(log(N))**

- **Повдигане в степен**

1 вариант: N-1 послед. умножения;
2 вариант: с използване на рекурсия:

```
long pow(long x, int n)
    { if(  n == 0)           return 1;
      if( n == 1)            return x;              // излишна проверка.
    if (isEven(n))
                 return pow ( x*x, n/2);            // брой умножения при четно
    else
                 return pow( x*x, n-1/2) *x;} // при нечетно
```

При n=1 последния ред решава нещата правилно

Пример: $x^{62} = (x^{31})^2$;
$x^{31}=(x^{15})^2 *x$;
$x^{15}=(x^7)^2 *x$;
$x^7=(x^3)^2 *x$;
$x^3 = (x^2)*x$ *----общо 9 умножения.Max 2 умножения(при нечетно)намаляват двойно степента.*

Следователно умноженията са <= 2logN

Тогава оценката е: O(logN)