Съдържание

1. Сложност на алгоритми

2. Рекурентни зависимости

3. Амортизационен анализ

4. Някои примери

Пресмятане на сложност на реализация на алгоритми

Означения и дефиниции

- Масова задача (task) с входни данни (с ограничения на данните)
- Конкретна задача (instance)
- Алгоритъм за решаване на масовата задача
- Програмна реализация на алгоритъма

Количество ресурси (**време**, памет и пр.) за решаване на задачата
Сложност – computational complexity

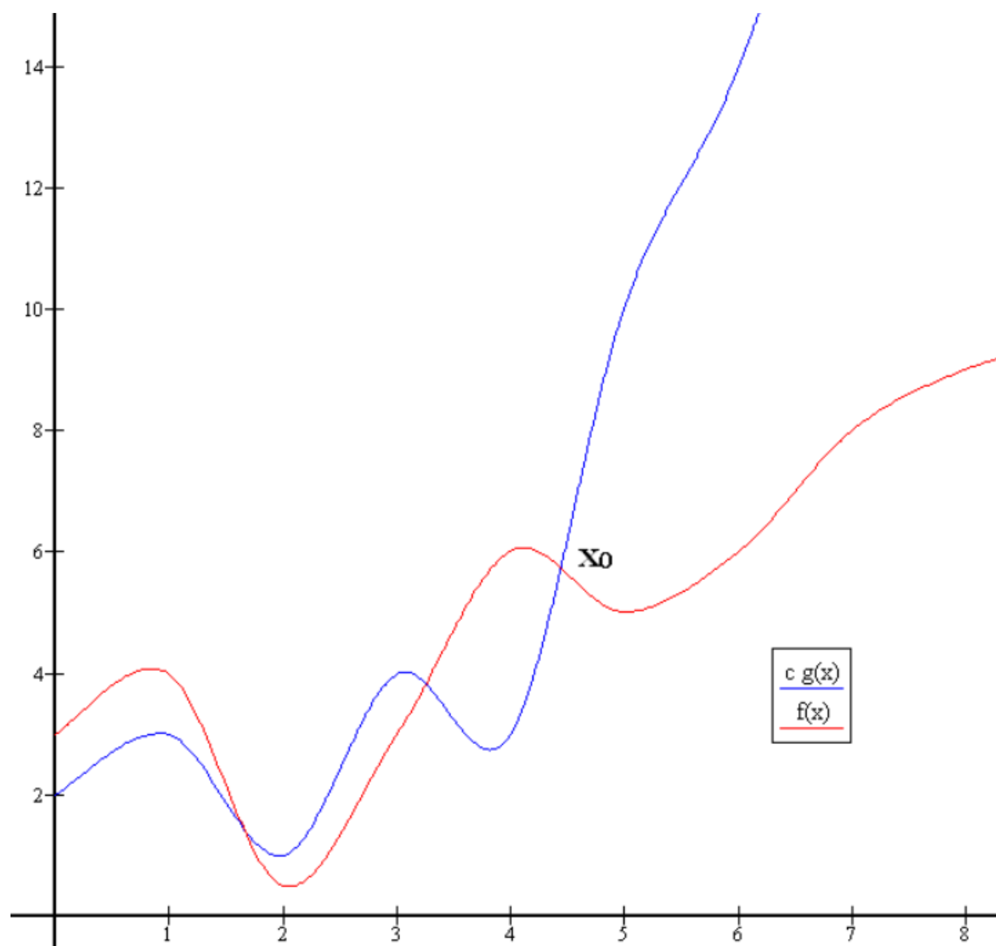
Най-лош случай и средно статистически случай.

Най-често се изследва ресурса време и говорим за бързина или скорост на пресмятанията

Оценяване на ресурса време: време за работа (например в милисекунди) в зависимост от дължината на входа (измерена с n)

Означения

$O(\log n)$, $O(n)$, $O(n \log n)$, $O(n^2)$, ..., $O(2^n)$



g(x) – синя
f(x) – червена

f(x) ∈ O(g(x)), ако съществува $C > 0$ (например $C = 1$) и съществува x_0 (например $x_0=5$) така че $f(x) \le Cg(x)$ за всяко $x \ge x_0$

Интуитивно f(x) расте по-бавно от g(x) при x отиващо в безкрайност

g(x) ограничава (асимптотично) отгоре стойностите на f(x)

Използват се логаритми
Двоичен логаритъм от n е степента k, за която $2^k = n$. Примери

$2^0=1$, log 1 =0
$2^1=2$, log 2 =1
$2^2=4$, log 4 =2
$2^3=8$, log 8 =3
…..
$2^{10}=1024$, log 1024 =10 или приблизително $\log 10^3 \approx 10$
$2^{20}=1\ 048\ 576$, log 1 048 576 =10 или прибл. $\log 10^6 \approx 20$

Когато n расте, log n расте значително по-бавно

log n има стойност, която показва колко пъти последователно трябва да разделим n на 2, за да получим 0.

Ще използваме формула за сума на аритметична прогресия:
$1+2+3+\ldots+n = (n+1)n/2$, или по-общо
$a+(a+d)+(a+2d)+(a+3d)+\ldots+(a+nd)$ и като означим $b=a+nd$,
сумата е $(a+b)(n+1)/2$

Формула за сума на геометрична прогресия:
$1+2+2^2+2^3+\ldots+2^{k-1}+2^k = 2^{k+1} - 1$

$1+1/2+1/2^2+1/2^3+\ldots+1/2^{k-1}+1/2^k = 2 - 1/2^k$

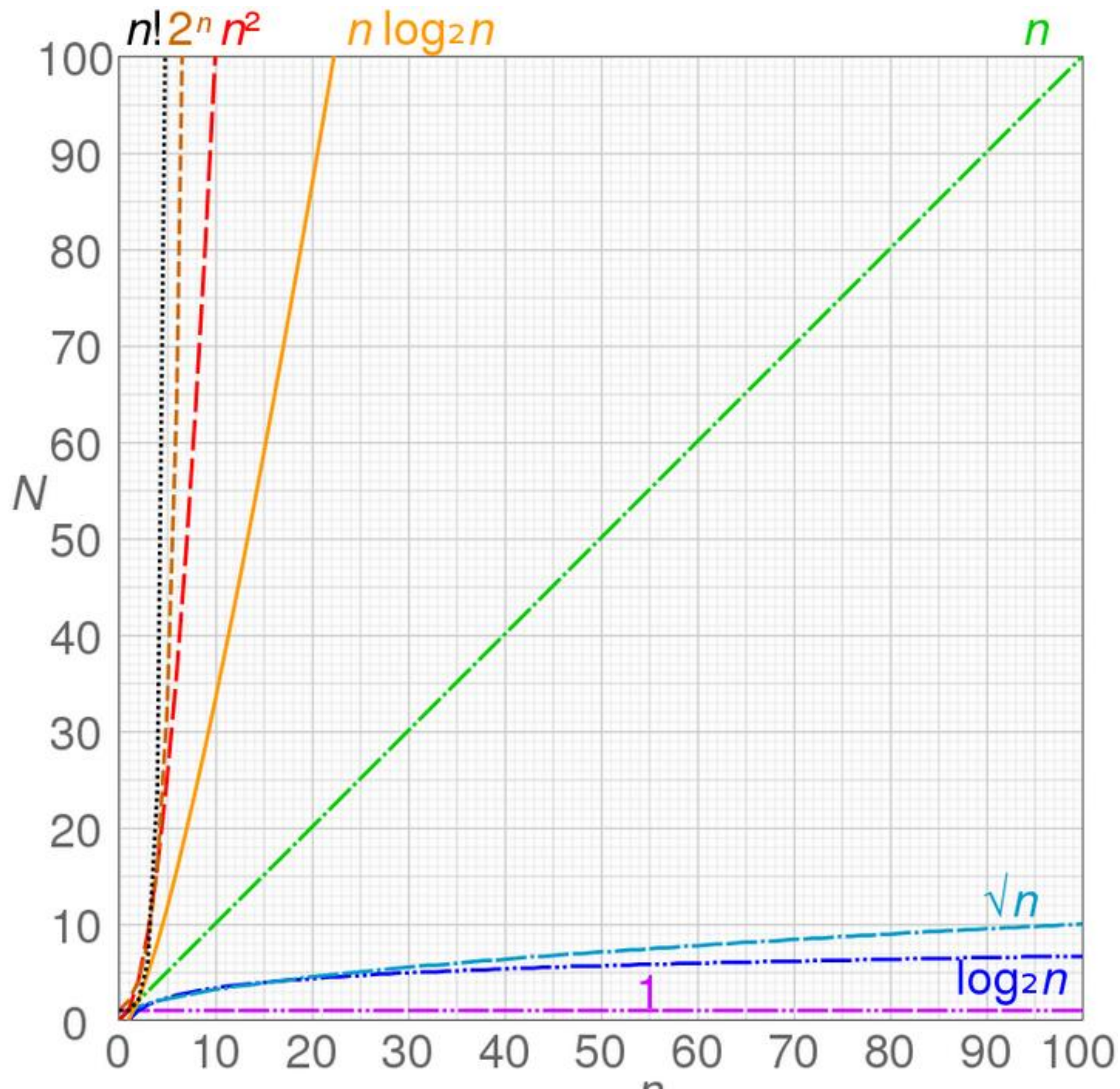
Ако T(n) е времето за работа на някакъв алгоритъм за решаване на дадена задача при големина n на входа и знаем например, че
$T(n) = O(n^2)$, това означава че времето за работа е (асимптотично) равно или по-малко от квадратично време по n.

В този пример също е вярно, че $T(n)=O(n^3)$ и затова на практика се интересуваме от най-бавно растящата функция (например от някой от видовете $\log n$, n, $n \log n$, n^2, ...,n^k,.., 2^n), която може да поставим в O().

Например ако P(n) е полином от степен k:
$P(n) = O(n^k)$ и $P(x) = O(n^s)$ за $s > k$, но не винаги е вярно за $s<k$.

Така търсим изразяване на P(n) чрез най-бавно растящата функция от вида n^s.

Пример: $P(n)=n^3+n^2$. Тогава $P(n) \leq 2n^3 \in O(n^3)$



Наредба по скорост на растеж:

n!, 2^n, n^2, n log n, n, √n, log n.

Записваме $n^2 \in$ O(n), или $n^2 = O(n)$.

$n^k = O(n^p)$ за k>p

Още означения и интуиция

Big-Oh: f(n) is O(g(n)) if f(n) is asymptotically **less than or equal** to g(n)

Big-Omega: f(n) is Ω(g(n)) if f(n) is asymptotically **greater than or equal** to g(n)

Big-Theta: f(n) is Θ(g(n)) if f(n) is asymptotically **equal** to g(n)

Little-oh: f(n) is o(g(n)) if f(n) is asymptotically **strictly less** than g(n)

Little-omega: f(n) is ω(g(n)) if is asymptotically **strictly greater** than g(n)

Измерване на време в програмата

```
#include<iostream>
#include<ctime>
using namespace std;

int main()
{
 clock_t t0 = clock();
 // example run:
 for(long long int i=0;i<1e9;i++);
 cout <<  ((double) (clock() - t0))/CLOCKS_PER_SEC << endl;
 system("pause");
}
```

Seconds	Equivalent
1	1 second
10	10 seconds
10^2	1.7 minutes
10^3	17 minutes
10^4	2.8 hours
10^5	1.1 days
10^6	1.6 weeks
10^7	3.8 months
10^8	3.1 years
10^9	3.1 decades
10^{10}	3.1 centuries
. . .	forever
10^{17}	age of universe

Run time in nanoseconds -->		$1.3\ N^3$	$10\ N^2$	$47\ N \log_2 N$	$48\ N$
Time to solve a problem of size	1000	1.3 seconds	10 msec	0.4 msec	0.048 msec
	10,000	22 minutes	1 second	6 msec	0.48 msec
	100,000	15 days	1.7 minutes	78 msec	4.8 msec
	million	41 years	2.8 hours	0.94 seconds	48 msec
	10 million	41 millennia	1.7 weeks	11 seconds	0.48 seconds
Max size problem solved in one	second	920	10,000	1 million	21 million
	minute	3,600	77,000	49 million	1.3 billion
	hour	14,000	600,000	2.4 trillion	76 trillion
	day	41,000	2.9 million	50 trillion	1,800 trillion
N multiplied by 10, time multiplied by		1,000	100	10+	10

Оценяване времето за работа на алгоритъм

Случай на вложени цикли - очевидно

Как да оценим при рекурсивна функция?

Логаритмично време за работа O(log n)

```
int binarySearch(int size, string a[], string key)
{
  int left = 0; int right = size - 1;
  while (left <= right)
  {
    int mid = (left + right) / 2;  int cmp = key.compare(a[mid]);
    if (cmp < 0) right = mid - 1;
    else if (cmp > 0) left = mid + 1;
    else return mid;
  }
  return -1;
}
```

Рекурентни уравнения (Recurrences)

Намиране времето за работа на рекурсивна функция
Пример за tail recursion

```
f(data, n)
{ \\ data is size n
  tailData = preProcess(data, n); \\ generate tailData with
                                  \\ size n-1
  f(tailData, n-1);               \\ process tailData
  postProcess(data, tailData, n); \\ put things together
}
```

$$f(n) = p(n) + f(n-1) + g(n)$$

Разделяй и владей

```
f(data, n)
{
 // split data into k chunks
 dataChunks = preProcess(data, n); // dataChunk[] is an array
 for (i=0; i < k; i++)
  {  // process each chunk
     f(dataChunk[i], n/k);
  }
 postProcess(data, dataChunks, n);
}
```

$$f(n) = p(n) + k\, f(n/k) + q(n)$$

Решаване на рекурентни зависимости:

Пример 1:

$T(n) = T(n-1) + n$

Метод на телескопа:

$T(n)=T(n-1)+n = T(n-2)+(n-1)+n = T(n-3)+(n-2)+(n-1)+n = \ldots=$
$= T(1)+2+3+\ldots+(n-2)+(n-1)+n \leq$
$\leq C + n(n-1)/2$

$O(n^2)$

Пример 2.

(Произход: двоично търсене)

$T(n) = T(n/2)+1$, за $n>1$ при $T(1)=0$

За улеснение приемаме, че $n=2^k$, т.е $k = \log n$

$T(2^k) = T(2^{k-1}) +1 = T(2^{k-2}) + 1 +1 = \ldots = T(2^0) + k = 0+k=k$

Решение: $T(n) = O(\log n)$

Пример 3.

$T(n) = T(n/2)+n$, за $n>1$ при $T(1)=0$

За улеснение приемаме, че $n=2^k$, т.е. $k = \log n$

По метода на телескопа се получава
$T(n)=n + n/2 + n/4 + n/8 + \ldots + 4 + 2 + 1$.

По формула за геометрична прогресия:

резултатът е $2n – 1$

$O(n)$

Пример 4.

$T(n) = 2T(n/2)+n$, за $n>1$ при $T(1)=0$

За улеснение приемаме, че $n=2^k$, т.е. $k = \log n$

Решение: $T(2^k) = 2T(2^{k-1}) + 2^k$

$T(2^k)/ 2^k = T(2^{k-1})/ 2^{k-1} + 1 = T(2^{k-2})/ 2^{k-2} + 1 + 1 =...=$
$= T(2^0)/2^0 + k = k$;

$T(n) = n \log n$

Уравнението се появява при merge sort

Пример 5.

$T(n) = 2T(n/2)+1$, за $n>1$, при $T(1)=0$

За улеснение: $n=2^k$, i.e. $k = \log n$

$T(2^k) = 2\,T(2^{k-1})+1 = 2^2\,T(2^{k-2})+1+1=\ldots$
$= 2^k\,T(2^0) + k = 2^k\,T(1) + k = n+\log n.$

Решение: $T(n) = O(n)$.

Изразяване на редицата на Фибоначи:

$s[0]=0$, $s[1]=1$, $s[n] = s[n-1]+s[n-2]$ за $n > 1$

търсим решение във вида $s[n]= p^n$

Получаваме уравнение $p^n = p^{n-1}+ p^{n-2}$

$p^2 = p+1$
$p_1 = (1 + \mathrm{sqrt}(5))/2$ или $p_2=(1 - \mathrm{sqrt}(5))/2$
$p_1 = 1.618033...$ или $p_2 = -0.618033...$

Как да удовлетворим началните условия?

Търсим a и b така, че

$s[n] = a*p_1^n + b*p_2^n$

Понеже s[n]=s[n-1]+s[n-2]

да намерим a и b, като заместим n=0 и n=1:

s[0]=0= a+b
$s[1]=1=a*p_1+b*p_2$

Решение:
a = 1/(p1-p2) = 1/sqrt(5) = 0.44721399..
b = 1/(p2-p1) = −1sqrt(5) = 0.44721399..

$s[n] = 0.4472... * (1.618...)^n - 0.4472... * (0.618...)^n$

Phi=1.618... (Златно сечение)

$(0.618...)^n$ клони към нула

Пресмятаме:

(Phi^0)/√5 = 0.4472135954999579 rounds to Fib(0) =0

(Phi^1)/√5 = 0.7236067977499789 rounds to Fib(1) =1

(Phi^2)/√5 = 1.1708203932499368 rounds to Fib(2) =1

(Phi^3)/√5 = 1.8944271909999157 rounds to Fib(3) =2

(Phi^4)/√5 = 3.0652475842498528 rounds to Fib(4) =3

(Phi^5)/√5 = 4.959674775249769 rounds to Fib(5) =5

(Phi^10)/√5 = 55.00363612324741 rounds to Fib(10) =55

(Phi^20)/√5 = 6765.000029563931 rounds to Fib(20) =6765

Merge sort

Разделяме масива на две половини

Рекурсивно сортираме всяка половина

Смесваме двете сортирани половини в общ сортирам масив

Масив a[i], .., a[j]

1. ако i=j масивът е сортиран и се връщаме в рекурсията

2. пресмятаме среден елемент $k = (i+j)/2$

3. рекурсивно обработваме (сортираме) a[i], ..., a[k]

4. рекурсивно обработваме (сортираме) a[k+1], ..., a[j]

5. смесваме (merge) сортираните масиви от т.3 и т.4

a[i], ..., a[k] и a[k+1], ..., a[j]

Top-down mergesort

```
void merge(int a[], int beg, int mid, int end)
{
  int n = end - beg + 1;
  int b[n];
  int i1 = beg;
  int i2 = mid + 1;
  int j = 0;

  while ((i1 <= mid) && (i2 <= end))
  { if (a[i1] < a[i2])
     {b[j]=a[i1]; i1++;}
    else {b[j]=a[i2]; i2++;}
    j++;
  }
```

```
   while(i1 <= mid)
    {b[j]=a[i1]; i1++; j++;}
   while(i2 <= end)
    {b[j]=a[i2]; i2++; j++;}
   for(j=0; j<n; j++) a[beg+j] = b[j];
}
```

Пример за работата на merge:

```
int a[]={5,9,10,12,17,8,11,20,32};
merge(a,0,4,8);
```

(5,9,10,12,17)(8,11,20,32) $\Rightarrow$ (5)
(9,10,12,17)(8,11,20,32) $\Rightarrow$ (5,8)
(9,10,12,17)(11,20,32) $\Rightarrow$ (5,8,9)
(10,12,17)(11,20,32) $\Rightarrow$ (5,8,9,10)
(12,17)(11,20,32) $\Rightarrow$ (5,8,9,10,11)
(12,17)(20,32) $\Rightarrow$ (5,8,9,10,11,12)
(17)(20,32) $\Rightarrow$ (5,8,9,10,11,12,17)
()(20,32) $\Rightarrow$ (5,8,9,10,11,12,17,20)
()(32) $\Rightarrow$ (5,8,9,10,11,12,17,20,32)
()() $\Rightarrow$ end

Merge Sort: цялата програма

```
void merge_sort(int a[],int beg, int end)
{ if (beg == end) return;
  int mid = (beg + end) / 2;
  merge_sort(a, beg, mid);
  merge_sort(a, mid + 1, end);
  merge(a, beg, mid, end);
}
```

Анализ на merge sort:

Памет:

- Входен масив = N.
- Спомагателен масив = N.
- Локални променливи: константи
- Стек на рекурсията: $\log_2 N$.
- Общо = $2N + O(\log N)$.

Merge sort Анализ на времето за работа

Означаваме T(N) = брой на сравненията на mergesort при вход с размер N.

Рекурентна зависимост:

$$T(N) \le \begin{cases} 0 & \text{if } N = 1 \\ \underbrace{T(\lceil N/2 \rceil)}_{\text{solve left half}} + \underbrace{T(\lfloor N/2 \rfloor)}_{\text{solve right half}} + \underbrace{N}_{\text{merging}} & \text{otherwise} \end{cases}$$

Решение на рекурентната зависимост: $T(N) = O(N \log_2 N)$.

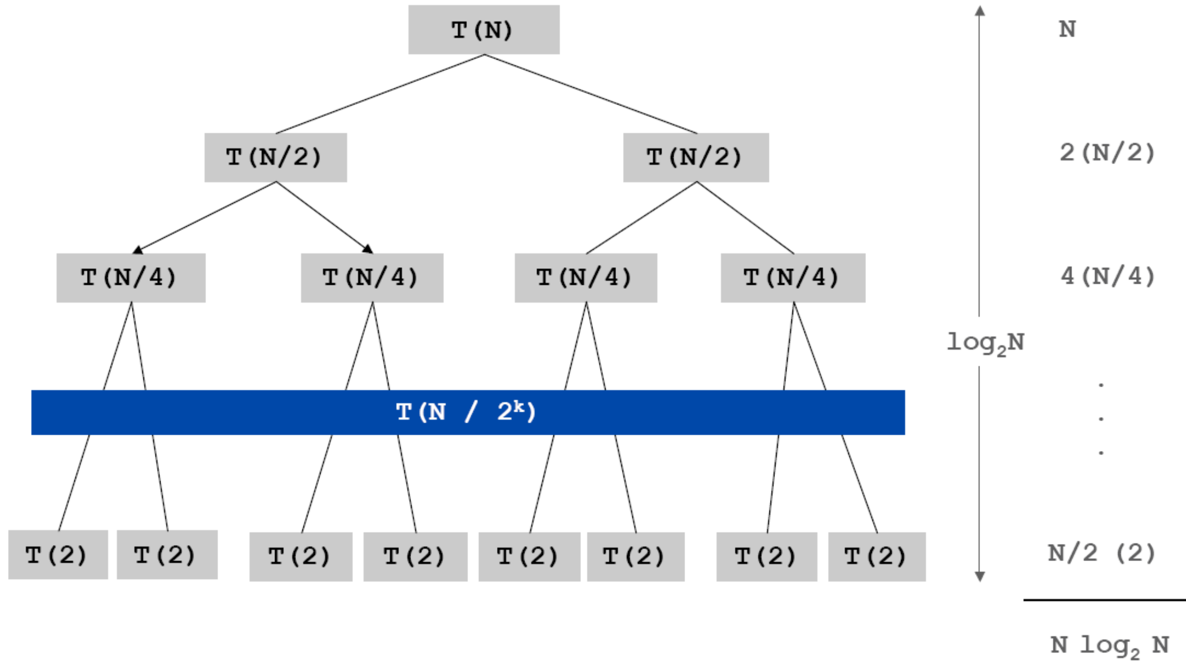
Доказателство с индукция:

- Базисен случай: $n = 1$.
- Хипотеза: $T(n) = n \log_2 n$.
- Да покажем, че $T(2n) = 2n \log_2 (2n)$.

$$
\begin{aligned}
T(2n) &= 2T(n) + 2n \\
&= 2n \log_2 n + 2n \\
&= 2n\left(\log_2(2n) - 1\right) + 2n \\
&= 2n \log_2(2n)
\end{aligned}
$$

Илюстрация чрез дърво на рекурсията:

$$T(N) = \begin{cases} 0 & \text{if } N = 1 \\ \underbrace{2T(N/2)}_{\text{sorting both halves}} + \underbrace{N}_{\text{merging}} & \text{otherwise} \end{cases}$$



Илюстрация за времето за работа
Стандартен (home) лаптоп с 10^8 операции сравняване за секунда
Мощен (super) компютър с 10^{12} операции сравняване за секунда

Insertion Sort (N^2)

computer	thousand	million	billion
home	instant	2.8 hours	317 years
super	instant	1 second	1.6 weeks

Mergesort (N log N)

thousand	million	billion
instant	1 sec	18 min
instant	instant	instant

Интересно приложение на модификация на алгоритъма merge sort за намиране броя на инверсиите в пермутация.

(понякога трябва да научим как работи някой алгоритъм, който може наготово да го ползване от STL)

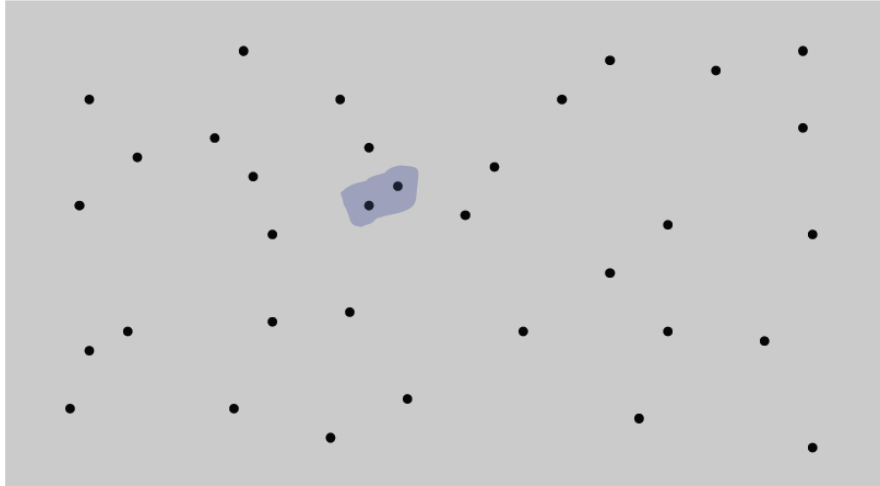
Ако a[1], a[2], …, a[n] е пермутация на 1,2,…,n, разглеждаме двойка индекси i,j такива, че $i < j$ и $a[i] > a[j]$. Всяка така двойка индекси наричаме инверсия.

Наивният метод (груба сила) изисква време $O(n^2)$

Цялата модификация на merge sort е добавяне на глобален брояч `c` и вмъкване на един оператор за присвояване във функцията merge():

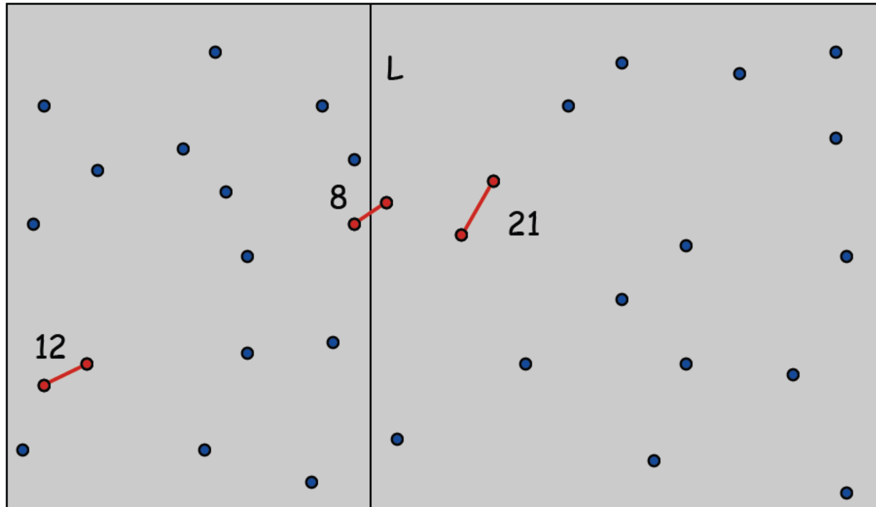
```
void merge(int a[], int beg, int mid, int end)
{
  int n = end - beg + 1;
  int b[n];
  int i1 = beg;
  int i2 = mid + 1;
  int j = 0;
  while ((i1 <= mid) && (i2 <= end))
  { if (a[i1] < a[i2]) {b[j]=a[i1]; i1++;}
    else {c += (mid-i1+1); b[j]=a[i2]; i2++;}
    j++;
  }
  while(i1 <= mid) {b[j]=a[i1]; i1++; j++;}
  while(i2 <= end) {b[j]=a[i2]; i2++; j++;}
  for(j=0; j<n; j++) a[beg+j] = b[j];
}
```

Задача за най-близки две точки



Алгоритъм

- Divide: Разделяме точките с вертикална линия на приблизително две равни части с по N/2 точки.
- Conquer: Намираме двойка най-близки точки във всяка част поотделно, прилагайки рекурсия
- Combine: Намираме двойка най-близки точки с по една точка в двете части.
- Return най-добрата от 3-те двойки



Нека най-малкото разстояние в двойките в частите е < δ. За да намерим двойка с точки в двете части:

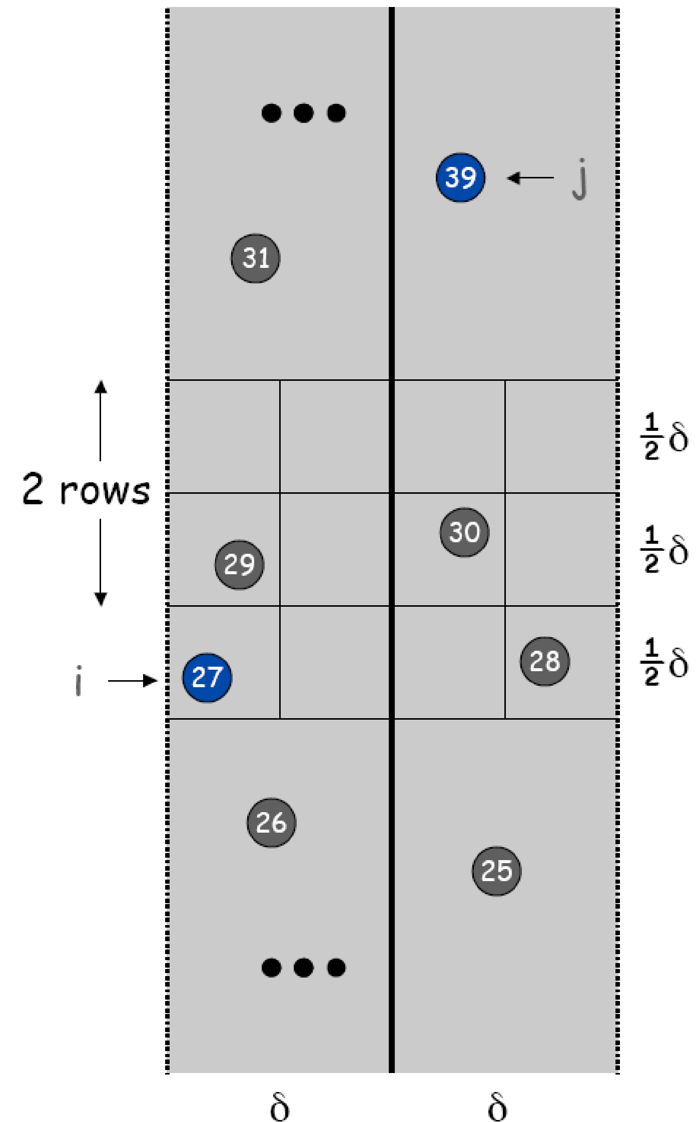
- трябва да разгледаме точките в ивица с ширина δ около правата L.
- сортираме точките в тази 2δ широка ивица по y координатите им.

- проверяваме за всяка точка в тази ивица само близките точки, с брой, който е константен спрямо N.

Обяснение:

Нека s_i е точка в 2δ-ивицата, която има i-та най-малка y-координата.

Ако $|i - j| \geq 12$, тогава разстоянието между s_i и s_j е поне δ, защото няма две точки в един и същ квадрат с размери δ/2 на δ/2 и две точки, намиращи се поне на 2 реда отдалеченост една от друга имат разстояние помежду си $\geq 2(\delta/2)$



Т.е. за всяка точка s_i в ивицата трябва да проверим не повече от 12 точки s_j, такива че $j=i+1, \ldots, i+12$.

По-прецизен анализ може да замени 12 с 6.

Closest Pair Algorithm

```
Closest-Pair(p1, …, pn) {
   Compute separation line L such that half the points      O(N log N)
   are on one side and half on the other side.

   δ1 = Closest-Pair(left half)                              2T(N / 2)
   δ2 = Closest-Pair(right half)
   δ  = min(δ1, δ2)

   Delete all points further than δ from separation line L   O(N)

   Sort remaining points by y-coordinate.                    O(N log N)

   Scan points in y-order and compare distance between       O(N)
   each point and next 11 neighbors. If any of these
   distances is less than δ, update δ.

   return δ.
}
```

Анализ за Running time O(N log N)

$$T(N) \le 2T(N/2) + O(N \log N) \Rightarrow T(N) = O(N \log^2 N)$$

$N=2^k$

$T(N)=2T(2^{k-1}) + O(2^k k) = 2^2 T(2^{k-2}) + O(2^k k) + O(2^{k-1} (k-1)) =$
$= 2^3 T(2^{k-3}) + O(2^k k) + O(2^{k-1} (k-1)) + O(2^{k-2} (k-2)) = \ldots$
$= 2^k T(1) + O(2^k k) + O(2^{k-1} (k-1)) + O(2^{k-2} (k-2)) + \ldots + O(2^1) \le$
$\le 2^k T(1) + O(2^k k) + O(2^{k-1} (k)) + O(2^{k-2} (k)) + \ldots + O(2^1 k) =$
$= 2^k T(1) + k\, O(2^k k) = N + (\log N)\, O(N \log N) = O(N \log^2 N)$

```
#include<iostream>
#include<cmath>
#include<algorithm>
using namespace std;

struct Point{int x, y;};
Point P[] =
{{2,3}, {12,32}, {40,49}, {5,1}, {12,10},
{7,3}};
int n = 6;

int cmpX(Point a, Point b)
 {return (a.x < b.x);}
int cmpY(Point a, Point b)
 {return (a.y < b.y);}
```

```
float dist(Point p1, Point p2)
{ return sqrt( (p1.x - p2.x)*(p1.x - p2.x) +
              (p1.y - p2.y)*(p1.y - p2.y));
}
// Намиране на най-малкото разстояние между
// точки в ивица s[] с размер m.
// Точките в s[] са сортирани по y
// Имат горна граница d за минималното
разстояние

float mid_Zone(Point s[], int m, float d)
{
  float dmin = d;
  sort(s, s+m,cmpY);
```

```
   // Пресмята за всяка точки разстоянието й
до
   // другите докато то е по-малко от d
   // Вътрешния цикъл се изпълнява най-много
   //  6 пъти.
   // Сложност O(n)

   for (int i = 0; i < m; ++i)
    for (int j = i+1; j < m && (s[j].y -
s[i].y) < dmin; ++j)
        dmin = min(dmin, dist(s[i], s[j]));

   return dmin;
}
```

```
float compute(Point P[], int n)
{
   if(n<=3) // brute force
     {
       float r = 1.0e38;
        for (int i = 0; i < n; i++)
         for (int j = i+1; j < n; j++)
          float r=min(r,dist(P[i], P[j]));
       return r;
     }

   // среда:
   int c = n/2;
   Point midPoint = P[c];
```

```
  float dL = compute(P, c);
  float dR = compute(P+c, n-c);
  float d = min(dL, dR);

  // Построяване на ивицата s[] с точки
  // от двете страни на средната линия
  // не по-далечни от разстояние d
  Point s[n];
  int m = 0;
  for (int i = 0; i < n; i++)
      if (abs(P[i].x - midPoint.x) < d)
          {s[m] = P[i]; m++;}
    return min(d, mid_Zone(s, m, d) );
}
```

```
int main()
{
   sort(P,P+n,cmpX);
   float r=compute(P, n);
   cout << r<< endl;
}
```

Амортизационен анализ

Методи за пресмятане на сложност на алгоритми

(т.е. на времето за работа на съответните им програми):

- **За рекурсивно написани програми знаем метода с рекурентни уравнения и решаването им, например с т.н. телескопичен метод.**
- **За програми с вложени цикли класическият метод е да умножим броя операции от външния цикъл с броя операции от вътрешния цикъл и т.н. Това е метод, който понякога два твърде груба горна граница за сложността**

- **Амортизираният анализ е метод за анализиране на сложността на даден алгоритъм или анализиране на ресурси (особено време или памет), които са необходими за изпълнение.**
- **Мотивацията за амортизиран анализ е, че разглеждането на най-лошия случай за време на изпълнение може да бъде твърде песимистично. Вместо това, амортизираният анализ осреднява времената на изпълнение на операциите в последователност**
- **Амортизираният анализ е полезен инструмент, който допълва други техники като анализ на най-лошия и среден случай.**

- **Терминът е въведен от Robert Tarjan в 1985 г.**

При статични проблеми има вход (като масив от n обекта) целта е да се получи изход с някакво желано свойство (напр. същите обекти, но сортирани).

Да разгледаме задачи, в които има редица от операции и целта е да се анализира времето, необходимо за една операция.

Например, вместо да имаме набор от n елемента във входа, може да имаме редица от n заявки: вмъкване, търсене и премахване към някаква база данни и искаме тези операции да бъдат ефективни.

Амортизираната цена на *n* операции е общата цена на операциите, разделена на *n*.

Пример: Стек

Имплементиране на стек чрез (динамичен) масив A и променлива top, която сочи върха на стека, т.е. A[top] е следваща празна клетка.

push(x) се имплементира чрез A[top]=x; top++;

x=pop() се имплементира чрез top--; x=A[top];

Какво става, когато масивът е пълен и трябва да направим push(x)

Можем да алокираме нов по-голям масив, копирайте стария и след това да продължим.

Това е скъпа операция.

Да „амортизираме“ (разнесем) цената върху предишните евтини операции, които ни доведоха до този момент.

Дефинираме следния модел на разходите:

- Вмъкването на елемент в масива струва 1
- Премахването на елемент от масива струва 1
- Цената за преоразмеряване на масива е равна на броя на преместените елементи.
- Другите операции, като top++ и top-- са безплатни.

Идея: При преоразмеряване да увеличим размера с 1.

Не е добре идея: Ако нашите n операции се състоят от *n* пъти push, тогава като вземем предвид разходите за преоразмеряване на масива ще имаме общи разходи от поне

$1 + 2 + 3 + 4 + . . . + (n - 1) = n (n - 1) / 2.$

Това е амортизирана цена на операция в размер на $(n - 1) / 2$

Друга идея: Да удвояваме размера на масива, когато преоразмеряваме

Това е добра идея: При всяка последователност от *n* операции, общите разходи за преоразмеряване са $1 + 2 + 4 + 8 + . . . + 2^i$ за някое $2^i < n$ (ако всички операции са push, тогава 2^i ще бъде най-голямата степен на 2, която е по-малка от n).

Тази сума е най-много равна на $2n - 1$.

Така получаваме обща цена $< 3n$ и

амортизираната цена за една операция е < 3

Друг поглед върху анализа на процеса на удвояване на масива в горния пример.

Нека всеки път, когато извършваме push операция, да плащаме 1 долар, за да я извършим, и поставяме 2 долара в касичка.

Така парите, с които олекваме са 3 долара.

Всеки път, когато трябва да удвоим масива, от размер L до

2L, плащаме за това с пари от касичката. Как да разберем, че ще има достатъчно пари (L долара) в касичката да се плати за това? Тъй като след последното преоразмеряване в масива има само L/2 елемента, трябват ни L/2 нови push операции, всяка допринасяйки по 2 долара.

Значи можем да платим за всичко, като изразходваме най-много по 3 долара за операция.

Казано по друг по начин, харчейки 3 долара на операция, успяхме да платим за всички операции плюс евентуално да останат още пари в касичката. Това означава, че нашата амортизирана цена е най-много 3 долара.

Този метод на „касичка“ често е много полезен за извършване на амортизиран анализ. Касичката се нарича още **потенциална функция**, тъй като подобно на потенциалната енергия във физиката, която можем да използвате по-късно.

Потенциалната функция е количеството пари в касичката. В горния случай потенциалът е 2 пъти броят на елементите в масива след средната точка.

При анализа е много важно да се докаже, че парите в касичката не стават отрицателни.

Пример. Двоичен брояч

Нека да трябва да имплементираме голям двоичен брояч в масив А. Всички записи започват от 0 и на всяка стъпка ще увеличаваме брояча с единица. Да приемем, че моделът на разходи е:

при всяко увеличаване на брояча, плащаме 1 долар за всеки бит, който трябва да обърнем (да си мислим, че обръщането на бит е обръщане на тежък камък, на който е написано „0“ на едната му страна и „1“ на другата.)

A[m]	A[m-1]	...	A[3]	A[2]	A[1]	A[0]	cost
0	0	...	0	0	0	0	
							$1
0	0	...	0	0	0	1	
							$2
0	0	...	0	0	1	0	
							$1
0	0	...	0	0	1	1	
							$3
0	0	...	0	1	0	0	
							$1
0	0	...	0	1	0	1	
							$2

В стандартното анализиране за „най-лош случай“ при стъпка от последователност от n стъпки, най-лошият случай на нарастване е O (log n), тъй като в най-лошия случай обръщаме log (n) + 1 бита.

Но каква е амортизираната цена на увеличение?

Отговорът е, че е най-много 2. Доказателство:

Колко често обръщаме A [0]? Отговор: всеки път.
Колко често обръщаме A [1]? Отговор: всеки втори път.
Колко често обръщаме A [2]? Отговор: на всеки 4-ти път и т.н.

Общите разходи за обръщане на A [0], са n, общите разходите за обръщане на A [1] са n/2, общите разходи за обръщане A [2] са n/4 и т.н.

Т.е. $n + n/2 + n/4 + ... < 2n$

Общите разходи за n стъпки са най-много 2n, т.е. по 2 долара на стъпка.

Пример. Двоичен брояч с цена 2^i за обръщане на A[i]

За последователност от n стъпки ($n = 2^k$), еднократно увеличение може да струва n, но амортизираната цена е само O (log n) на стъпка. Доказателство:

A [0] се обръща всеки път за по 1 долар (общо n долара).
A [1] се обръща всеки втори път за по 2 долара (общо n долара).
A [2] се обръща всеки 4-ти път за по 4 долара (общо n долара).
и т.н. ... $A[2^i]$ се обръща всеки 2^i -ти път за по 2^i долара (общо n долара) и накрая
$A[2^k]$ се обръща всеки 2^k -ти път (т.е веднъж) за по 2^k долара (общо n долара, защото $n = 2^k$).
Т.е. общата цена е $n \cdot k$, където $k=\log n$, което при n стъпки прави цена log n за стъпка.

Пример. Амортизационна структура от данни речник (dictionary)

Един от най-често срещаните класове структури от данни са „речниковите“ структури от данни (map от STL), които поддържат операции за бързо **вмъкване** и **търсене** в набор от елементи.

Знаем, че могат да се реализират с балансирано дърво, при което всяко вмъкване и всяко търсене за набор от n елемента струва O(log n).

Знаем, че сортираният масив е добър за търсене (с двоичното търсене за време O (log n)), но е лош за вмъкване, което става за време O(n)

Знаем, че свързаният списък е добър за вмъкване (става за време O(1)), но е лош за търсене, което става за време O(n).

Алгоритъм за речникова структура от данни, който се имплементира лесно и е ефективен като алгоритъма чрез балансирано дърво.

Алгоритъмът има $O(\log^2 n)$ време за търсене и $O(\log n)$ амортизирана цена за вмъкване.

Идея:

Програмираме колекция от масиви, в която i-тият масив има размер 2^i.

Всеки масив е или празен или изцяло пълен и всеки е сортиран.

Няма зависимост между елементите в различни масиви.

Въпросът кои масиви да са пълни и кои да са празни се решава чрез двоичното представяне на общия брои елементи, които съхраняваме.

Например, ако имаме 11 елемента (11 = 1 + 2 + 8), структурата може да изглежда така (с някакви случайни данни):

```
A0: [5]
A1: [4,8]
A2: []
A3: [2, 6, 9, 12, 13, 16, 20, 25]
```

За операцията **търсене**, извършваме двоично търсене във всеки непразен масив. В най-лошия случай това отнема следното време

$O(\log(n/2) + \log(n/4) + \log(n/8) + ... + 1) <=$

(броят на събираемите е $O(\log n)$)

$O(\log(n)*\log(n))=$

$O(\log^2 n)$

За операция **вмъкване**, използваме идеята на mergesort.

Например, за да вмъкнем числото 10, първо създаваме нов масив A с размер 1, който съдържа това число.

Гледаме дали A0 е празен. Ако е така, правим новия масив A да бъде A0. Ако A0 не е празен, обединяваме двата сортирани масива масив A и A0 и се получава нов сортиран масив, който означаваме отново с A (който в примера ще съдържа [5, 10])

Гледаме дали A1 е празен. Ако A1 е празен, правим A да е A1. Ако A1 не е празен обединяваме двата сортирани масива A и A1 и се получава нов сортиран масив, който означаваме отново с A.

И т.н.

От горния пример получаваме:

```
A0: empty
A1: empty
A2: [4, 5, 8, 10]
A3: [2, 6, 9, 12, 13, 16, 20, 25]
```

Модел за пресмятане на цената:

Създаването на първоначалния масив от размер 1 струва 1.

Обединяването в сортиран масив на два сортирани масива с размер m струва 2 m.

В примера операцията вмъкване на елемента 10 има цена

1 + 2 + 4.

Твърдение: Гореописаната речникова структура от данни има амортизирана цена O (log n) за операция вмъкване.

Доказателство: Моделът на цените, дефиниран по-горе, е същият като за двоичния брояч с цена 2^i за позиция i.

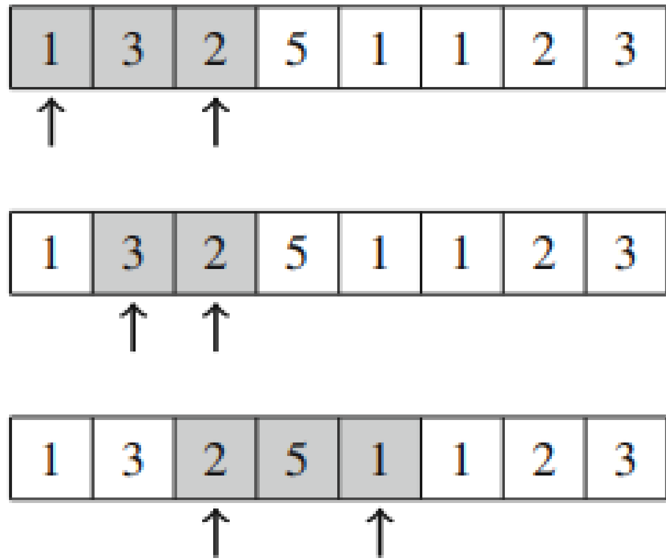
Метод на двата указателя (Two Pointers Method)

Два указателя се движат само в една посока по елементите на масив. Това гарантира, че алгоритъмът работи ефективно.

Пример за задача:

Даден е масив от n положителни цели числа и целева сума x. Трябва да намерим подмасив (от последователни елементи), чиято сума от елементи е равна на x, или да кажем, че няма такъв подмасив.

Пример за x=8:



Левият и десният указатели определят подмасив. Когато неговата сума е равна на x, програмата завършва.

В началото двата указателя започват от първата позиция. Пробваме да местим десният указател и го местим, ако сумата е $\leq$ x. Ако това не е възможно местим левия указател на една позиция.

Общото време за работа е O(n).

```
const int n=7; int a[n]={5,2,5,6,1,1};
int x=8;

int main()
{
  int s=a[0]; int L = 0;
  for (int R = 1; R <= n; R++)
  {
    while (s > x && L < R - 1) {s -= a[L]; L++;}
    if (s == x)
     {cout << "Found between indexes" << L << " " << R-1; return 0;}
    if (R < n) s += a[R];
  }
  cout << "Not found";
}
```

Задача 2SUM: Даден е масив от n цели числа и целева сума x. Намерете два елемента на масива, така че тяхната сума да е x.

Решение: O(n log n). Сортираме масива в нарастващ ред. След това, итерираме през масива, като използваме два указателя. Левият указател започва от първия индекс и се придвижва с една стъпка надясно на всеки ход. Десният указател започва от последния индекс и винаги се движи наляво.

```
while (L < R)
{if (A[L] + A[R] == x) then  cout << „found”;
 else if (A[L] + A[R] < x) then L++
 else R--;
}
```

Пример за x=12

1	4	5	6	7	9	9	10

1	4	5	6	7	9	9	10

1	4	5	6	7	9	9	10

Обща задача kSUM. Да се намерят k елементи, така че тяхната сума да е x. Оказва се, че можем да решим 3SUM проблем за O (n^2) време чрез използване на идеята на горния алгоритъм 2SUM.

Задача: За всеки елемент a[i] в масив, да се намери най-близкото отляво по-малко число от a[i]

Пример:

за всеки елемент на масива {1, 3, 0, 2, 5}

решението е {_, 1, _, 0, 2}

Наивното решение е за $O(n^2)$

Решение за O(n)

```
stack<int> S;
for (int i=0; i<n; i++)
{
   // Keep removing top element from S while the top
   // element is greater than or equal to a[i]
   while (!S.empty() && S.top() >= a[i]) S.pop();

   // If all elements in S were greater than a[i]
   if (S.empty())  cout << "_, ";       else  cout << S.top() << ", ";
  //Else print the nearest smaller element

   S.push(a[i]);
}
```

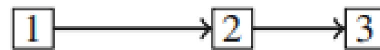
step 1

step 2

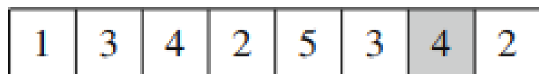
step 3

step 4

step 5



step 6



step 7

step 8

Задача: Минимум в плъзгащ се прозорец с дължина k по масив с дължина n

Пример:

{2,1,4,5,3,4,1,2}

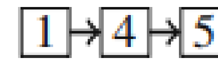
min{2,1,4,5}=1

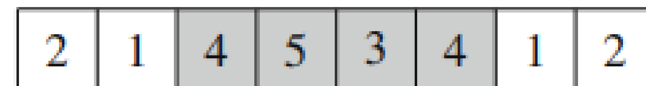
min{1,4,5,3}=1

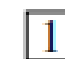
min{4,5,3,4}=3

min{5,3,4,1}=1

min{3,4,1,2}=1







Идея:

Създаваме Deque Q с капацитет k, което съхранява само полезни елементи от текущия прозорец от k елементи. Елементът е полезен, ако е в текущия прозорец и е по-малък от всички други елементи от дясната му страна в текущия прозорец. Обработваме всички елементи на масива един по един и поддържаме Q да съдържа полезни елементи от текущия прозорец и тези полезни елементи се поддържат в **сортиран ред**. Елементът отпред-front (изобразен в ляво) на Q е най-малкият, а елементът отзад-back (изобразен в дясно) в Q е най-големия от текущия прозорец

```
int a[] = {2,1,4,5,3,4,1,2}; int n = 8; int k = 4;
int main()
{ deque<int> Q(k);
  for (int i = 0; i < k; ++i)
  { while ((!Q.empty()) && a[i] <= a[Q.back()])
      Q.pop_back();
   Q.push_back(i);
  }
  for (int i=k; i < n; ++i)
  { cout << a[Q.front()] << " ";
    while ((!Q.empty()) && Q.front() <= i - k)
      Q.pop_front();
    while ((!Q.empty()) && a[i] <= a[Q.back()])
      Q.pop_back();
    Q.push_back(i);
  }
  cout << a[Q.front()];
}
```