分治法

分而治之

Divide and conquer

Youtube频道：**hwdong**

博客： **hwdong-net**.github.io

分治法

分而治之

Divide and conquer

Youtube频道：**hwdong**

博客: **hwdong-net**.github.io

分治法的思想

- 把一个复杂的(大)问题分成两个或更多的子问题，求解子问题，将子问题的解合并为问题的解。
- 分-治-合：

  -将问题分解成子问题

  -解决子问题

  -合并子问题解为原问题的解

分治递归

子问题和原问题性质一样

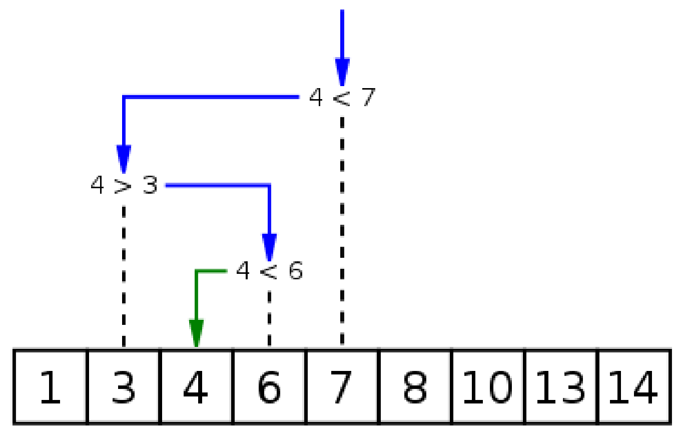
YouTube频道: hwdong

分治递归

- 如果子问题和原问题相同，只是规模不同，则可以对子问题重复上述过程，直到子问题可以直接求解。
- 分治递归通过将问题分解为两个或多个相同性质的子问题来解决问题，直到这些问题变得简单到可以直接解决为止。

分治递归

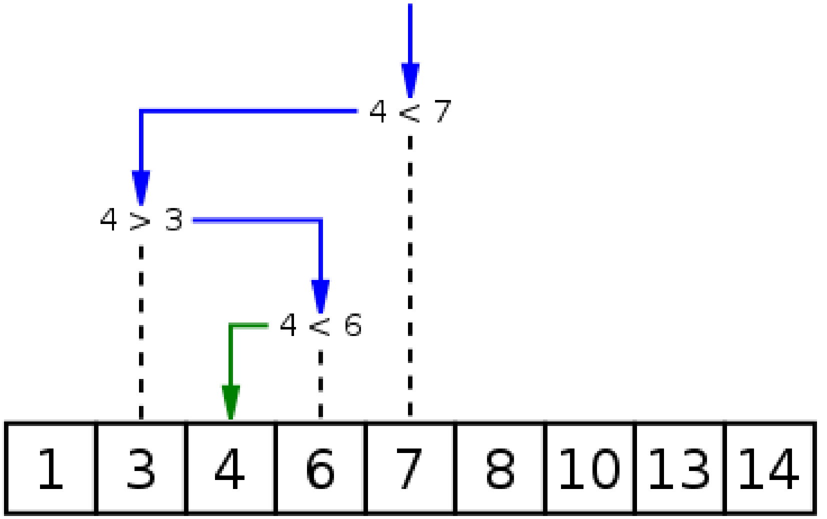
- 分治递归既适用于具有固有递归结构的问题，如树的遍历（搜索）。也适用于通过递归求解过程来解决的问题，如汉诺塔、二分查找、归并排序、快速排序、最近点对、Strassen矩阵乘法等。



分治递归算法模板

```
function divideAndConquer(problem)
  if problem is small enough：          //问题足够小
    return solve(problem)
  subproblems = divide(problem)         //分解成子问题
  subresults = []
  for each subproblem in subproblems //解决每个子问题
      subresults.append(divideAndConquer(subproblem))
  return combine(subresults)             // 合并子问题解
```

二分查找




```
binary_search(A,L,R,key)
    if L>R: return -1
    m = (L+R)/2
    if A[m]==key:
        return m
    else if A[m]>key:
      return binary_search(A,L,m-1,key)
    else:
      return binary_search(A,m+1,R,key)
```

时间复杂度

$$T(n) = \begin{cases} O(1) & n = 1 \\ T(\lfloor n/2 \rfloor) + 1 & n > 1 \end{cases}$$

$\mathrm{T(n)} = \mathrm{O}(\log_2 \mathrm{n})$ 或者 $T(n) = \Theta(nlog_2 n)$

汉诺塔

```
Hanoi(n,A,B,C)
    if n==1:        move(n, A,C)
    Hanoi(n-1,A,C,B)
    move(n, A,C)
    Hanoi(n-1,B,A,C)
```

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

$T(1) = 1$

$\Rightarrow$ $T(n) = 2^n-1$

归并排序

Youtube频道： **hwdong**

博客： **hwdong-net**.github.io

归并排序

- 将序列从中间分成2个子序列

38	27	43	3	9	82	10

38	27	43	3

9	82	10

归并排序

- 将序列从中间分成2个子序列
- 对2个子序列归并排序

38	27	43	3	9	82	10

38	27	43	3

9	82	10

⋮ ⋮

3	27	38	43

9	10	82

```
merge_sort(A,L,R)
  if (L == R) return;
  m = (L+R)/2
  merge_sort(A, L, m);
  merge_sort(A, m+1,R);
```

归并排序

- 将序列从中间分成2个子序列
- 对2个子序列归并排序
- 合并2个有序序列为一个有序序列

38	27	43	3	9	82	10

38	27	43	3

9	82	10

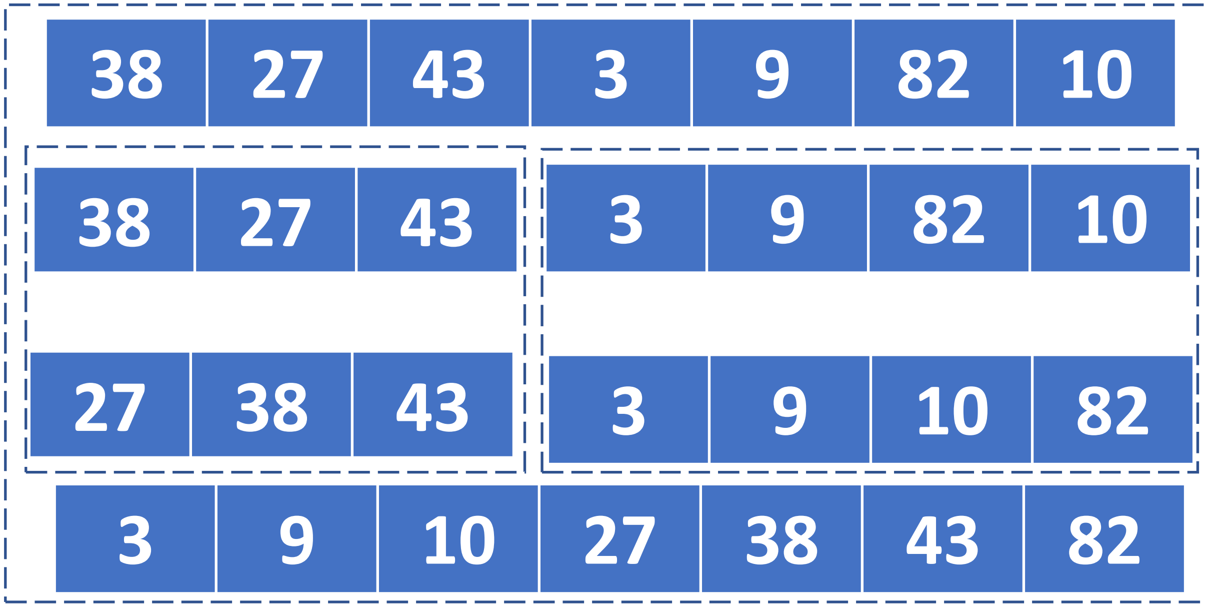
⋮ ⋮

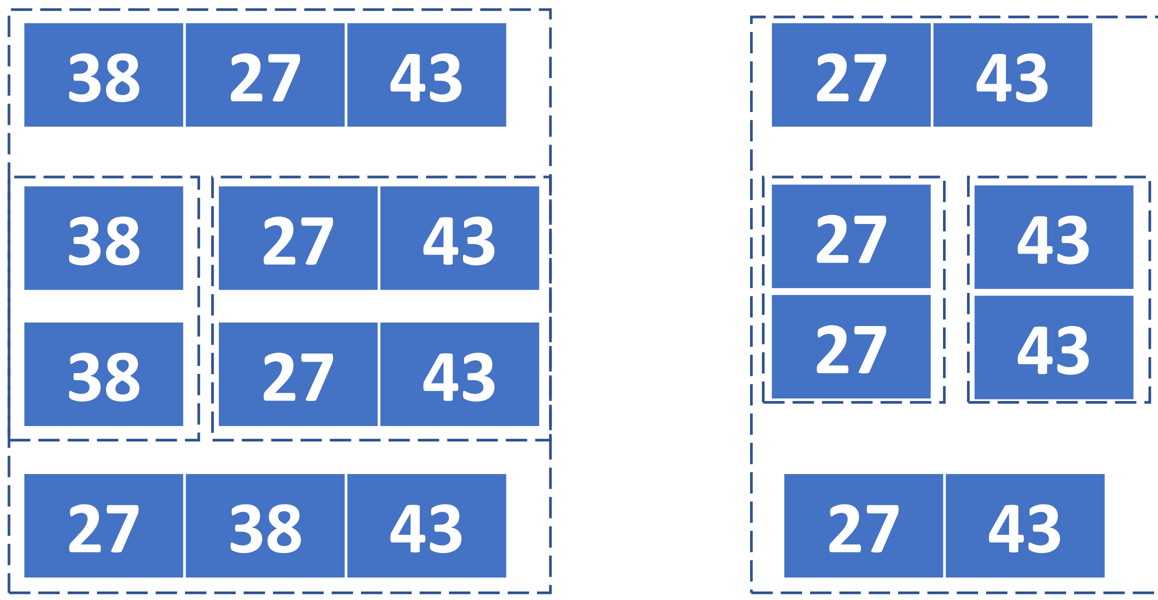
3	27	38	43

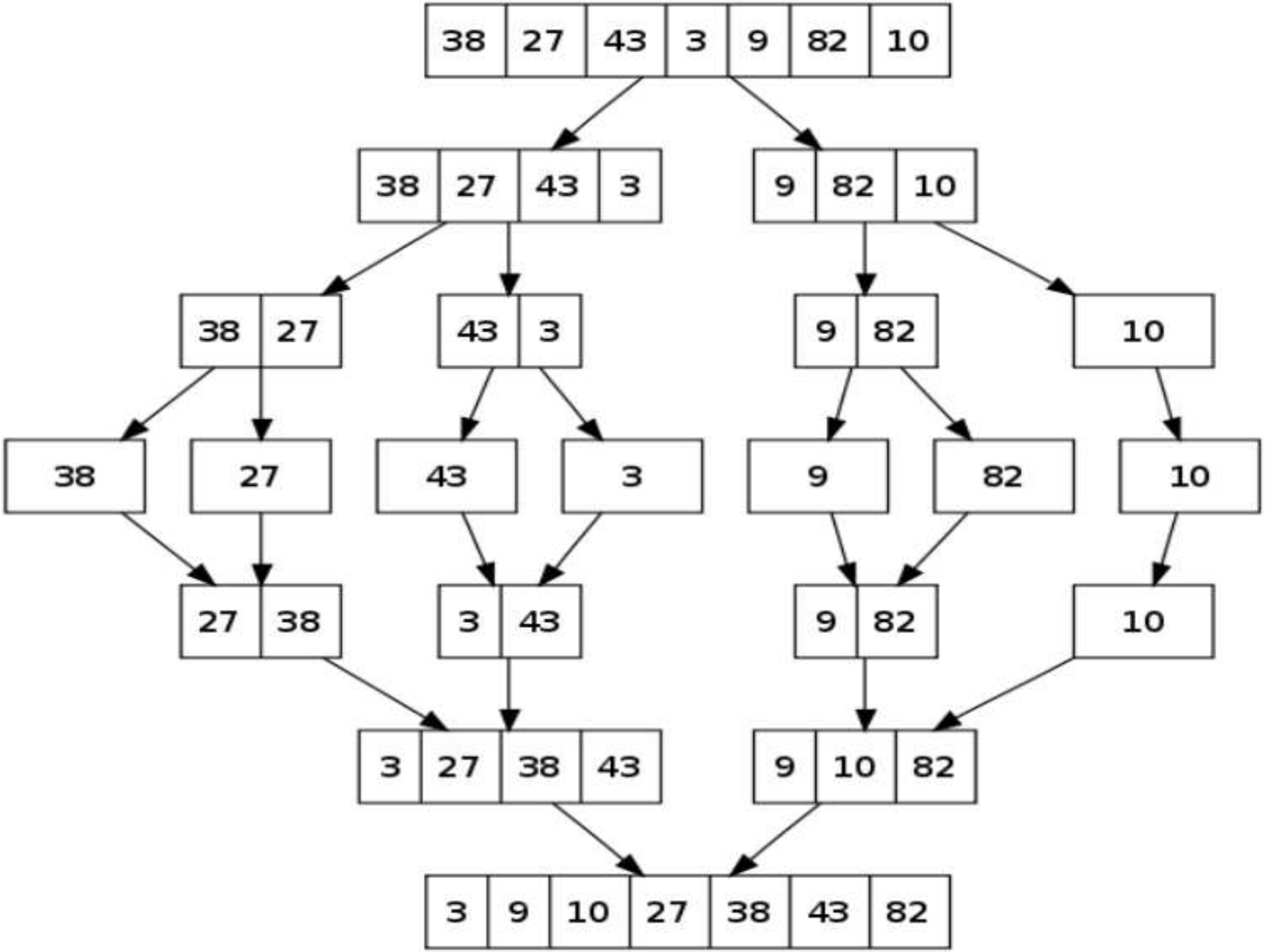
9	10	82

3	9	10	27	38	43	82

```
merge_sort(A,L,R)
    if (L == R) return;
    m = (L+R)/2
    merge_sort(A, L, m);
    merge_sort(A, m+1,R);
    merge(A,L,m,R)
```


38 27 43 3 9 82 10
38 27 43
3 9 82 10
27 38 43
3 9 10 82
3 9 10 27 38 43 82


38 27 43
38
27 43
38
27 43
27 38 43
27 43
27
43
27
43
27 43

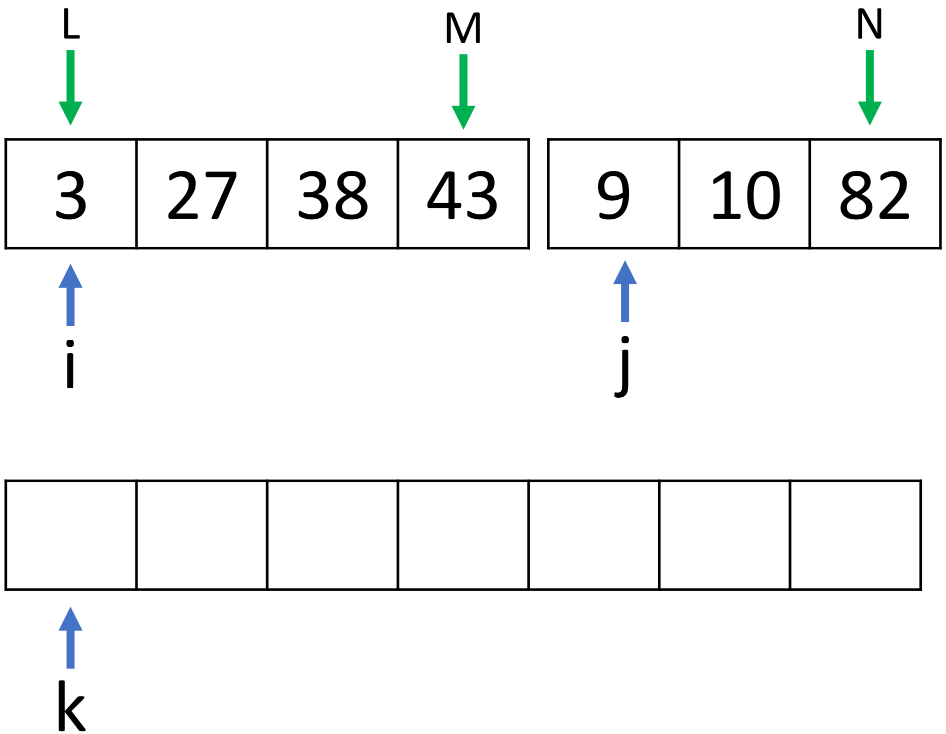

38 27 43 3 9 82 10
38 27 43 3
9 82 10
38 27
43 3
9 82
10
38
27
43
3
9
82
10
27 38
3 43
9 82
10
3 27 38 43
9 10 82
3 9 10 27 38 43 82

merge:合并2个有序序列

3	27	38	43		9	10	82

3	9	10	27	38	43	82

merge:合并2个有序序列



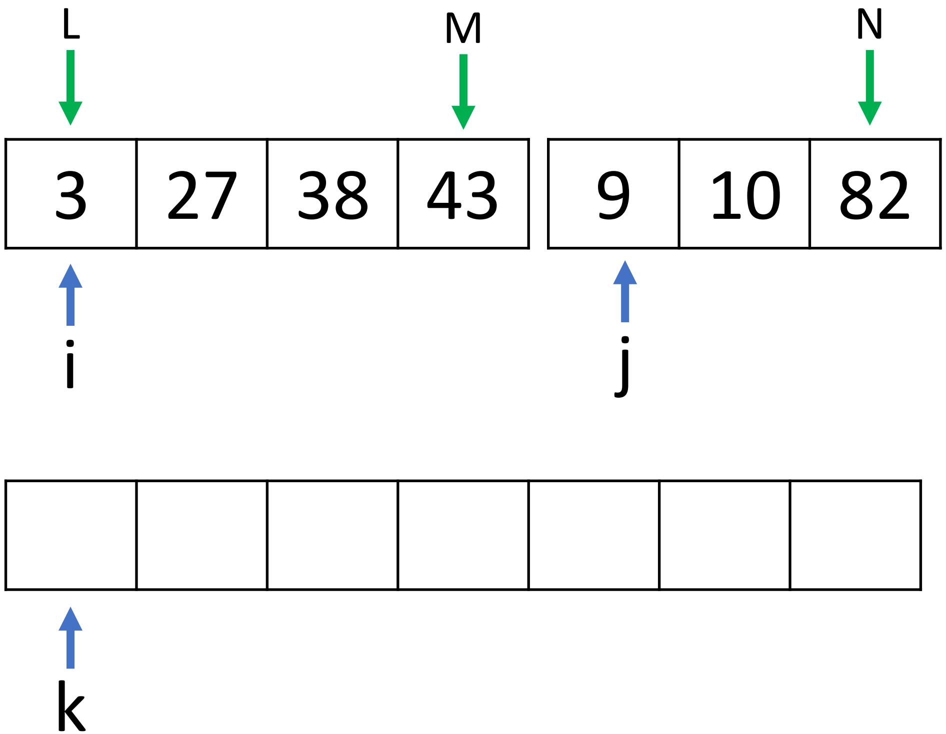
Merge(A, L ,M, N)

i=L, j=M+1

k= 0

p = new T[N-L+1]

merge:合并2个有序序列



Merge(A, L ,M, N)

    i=L, j=M+1

    k= 0

    p = new T[N-L+1]

    while i<=M&&j<=N:

        if A_i<A_j:

            p[k] = A[i]

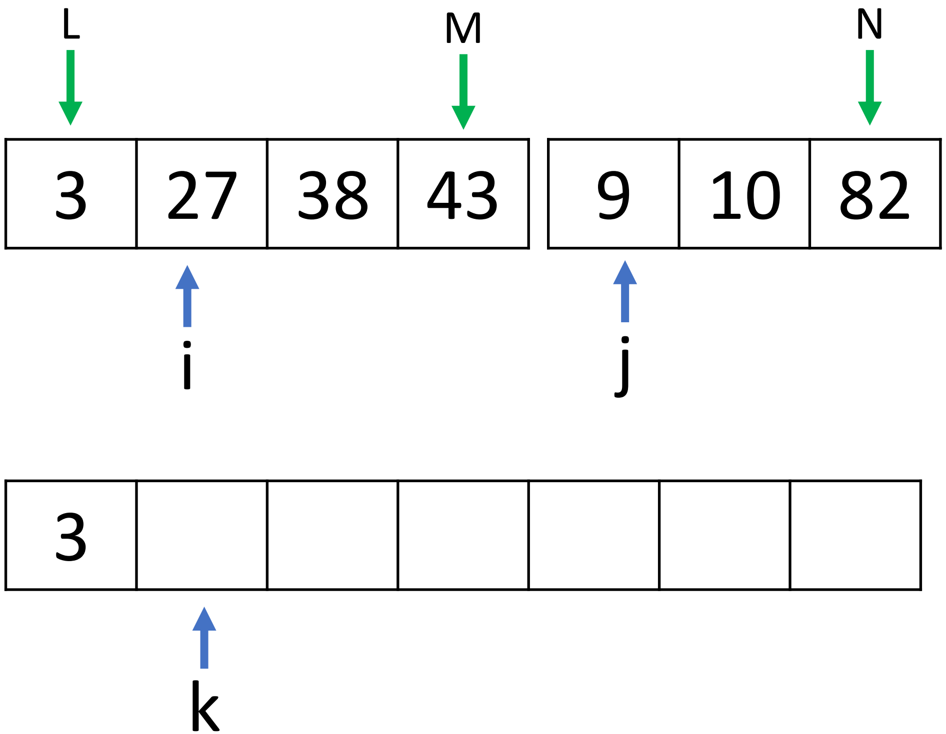
            k+=1 ; i+=1;

        else

            p[k] = A[j]

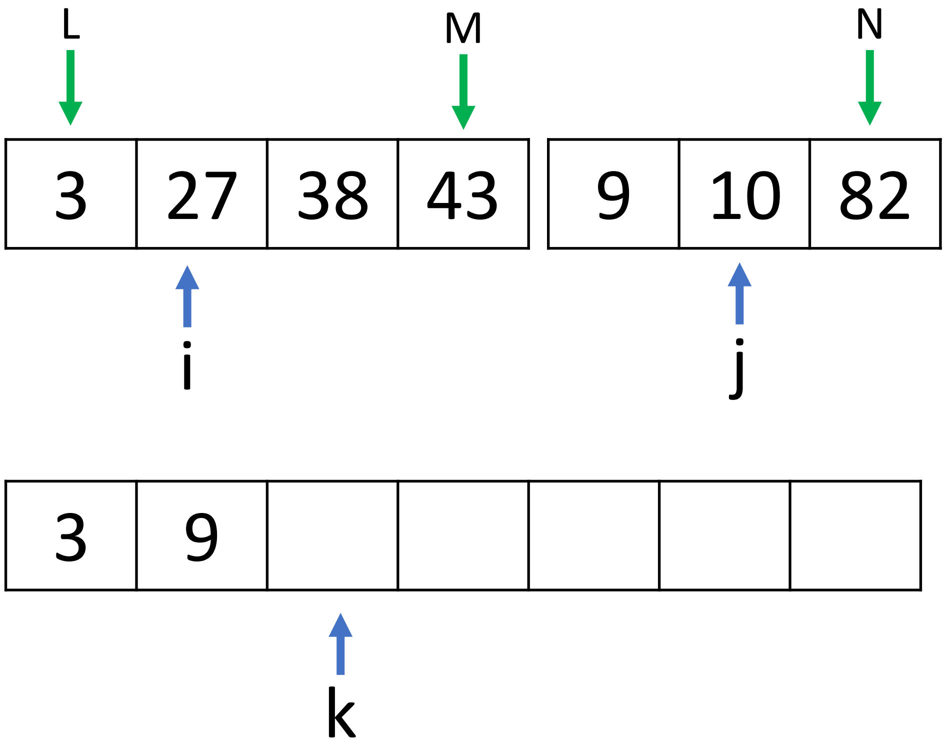
            k+=1 ; j+=1;

merge:合并2个有序序列



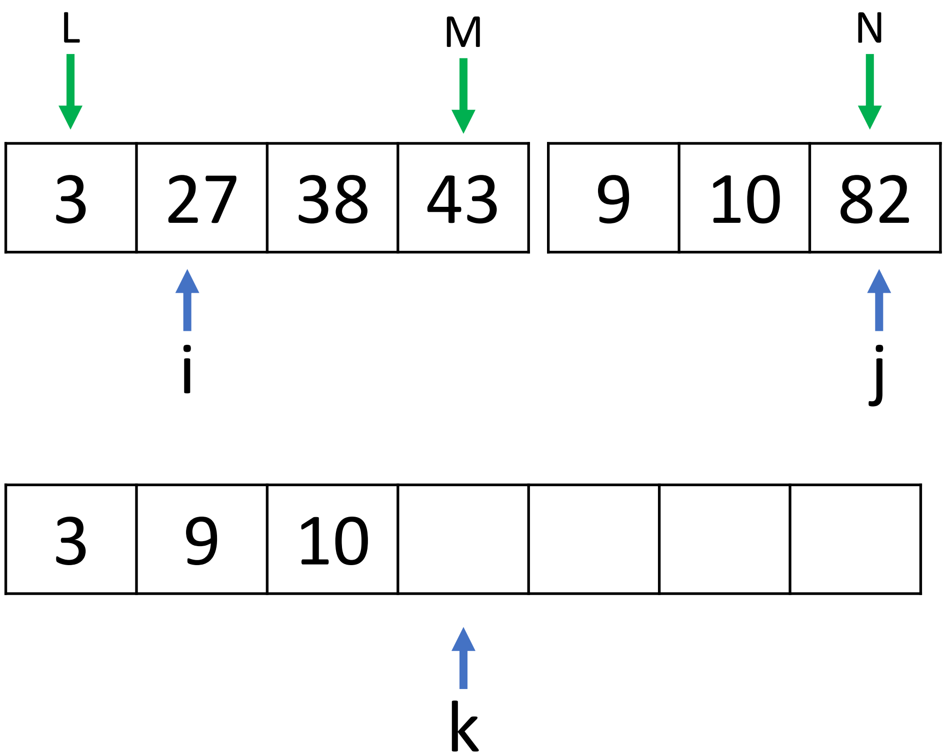

```
Merge(A, L ,M, N)
    i=L,  j=M+1
    k= 0
    p = new T[N-L+1]
    while i<=M&&j<=N:
        if A_i<A_j:
            p[k] = A[i]
            k+=1 ; i+=1;
        else
            p[k] = A[j]
            k+=1 ; j+=1;
```

merge:合并2个有序序列




```
Merge(A, L ,M, N)
    i=L,  j=M+1
    k= 0
    p = new T[N-L+1]
    while i<=M&&j<=N:
        if Ai<Aj:
            p[k] = A[i]
            k+=1 ; i+=1;
        else
            p[k] = A[j]
            k+=1 ; j+=1;
```

merge:合并2个有序序列



Merge(A, L ,M, N)

    i=L, j=M+1

    k= 0

    p = new T[N-L+1]

    while i<=M&&j<=N:

        if A_i<A_j:

            p[k] = A[i]

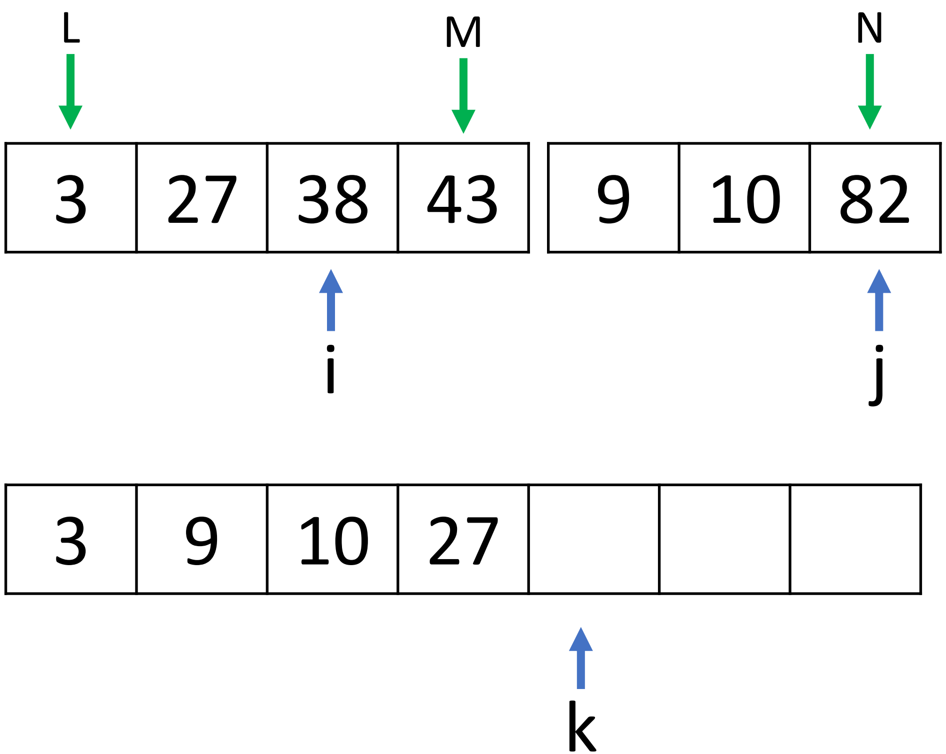
            k+=1 ; i+=1;

        else

            p[k] = A[j]

            k+=1 ; j+=1;

merge:合并2个有序序列



Merge(A, L ,M, N)

i=L, j=M+1

k= 0

p = new T[N-L+1]

while i<=M&&j<=N:

if A_i<A_j:

p[k] = A[i]

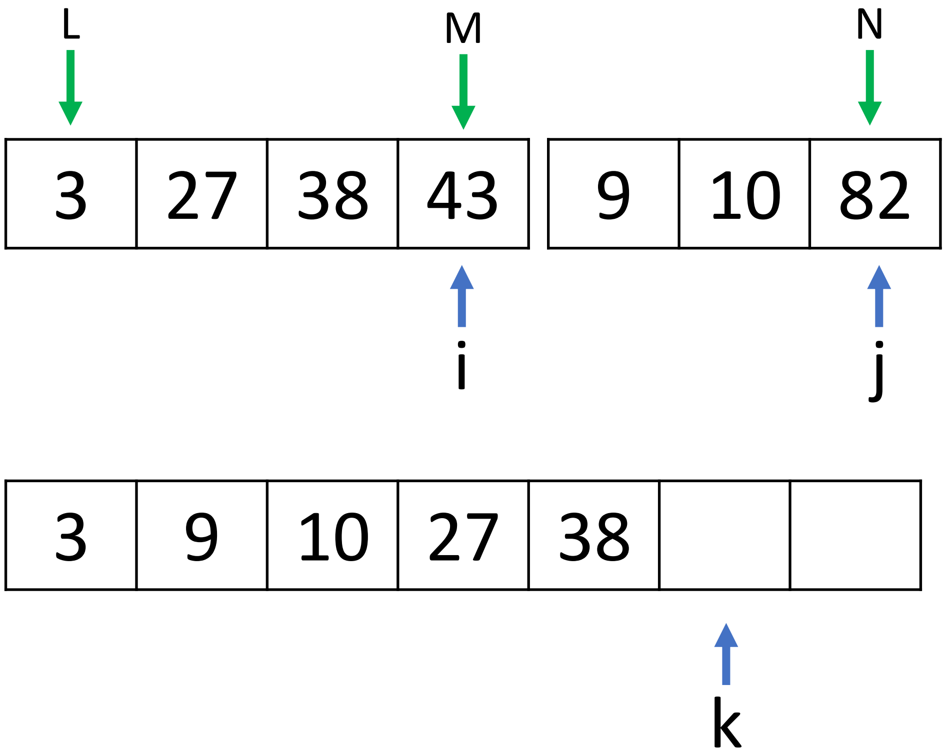
k+=1 ; i+=1;

else

p[k] = A[j]

k+=1 ; j+=1;

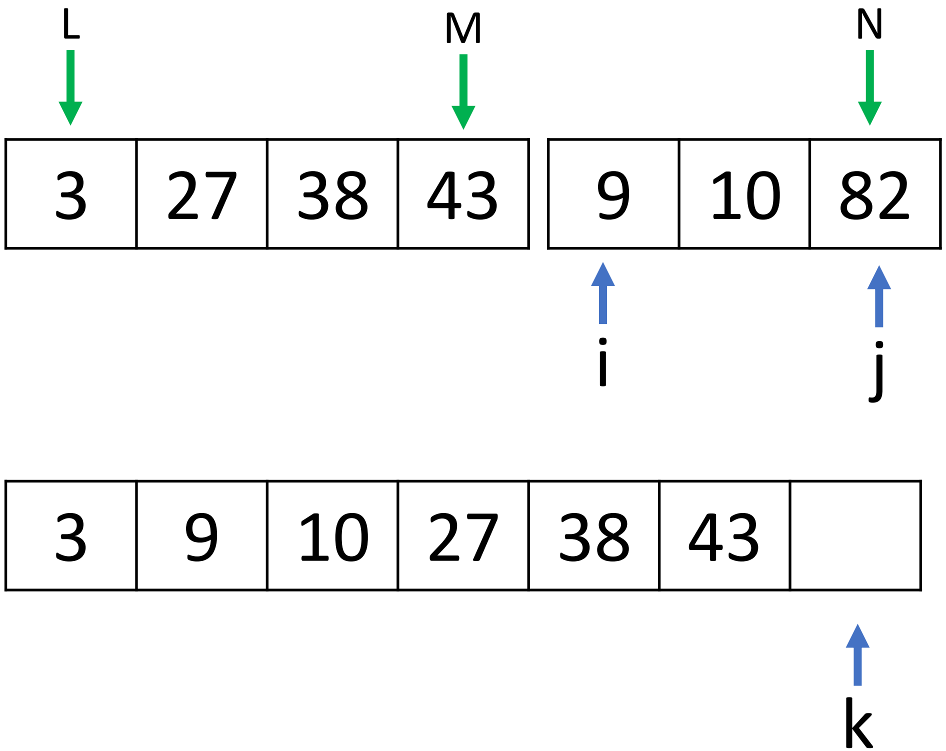
merge:合并2个有序序列



Merge(A, L ,M, N)

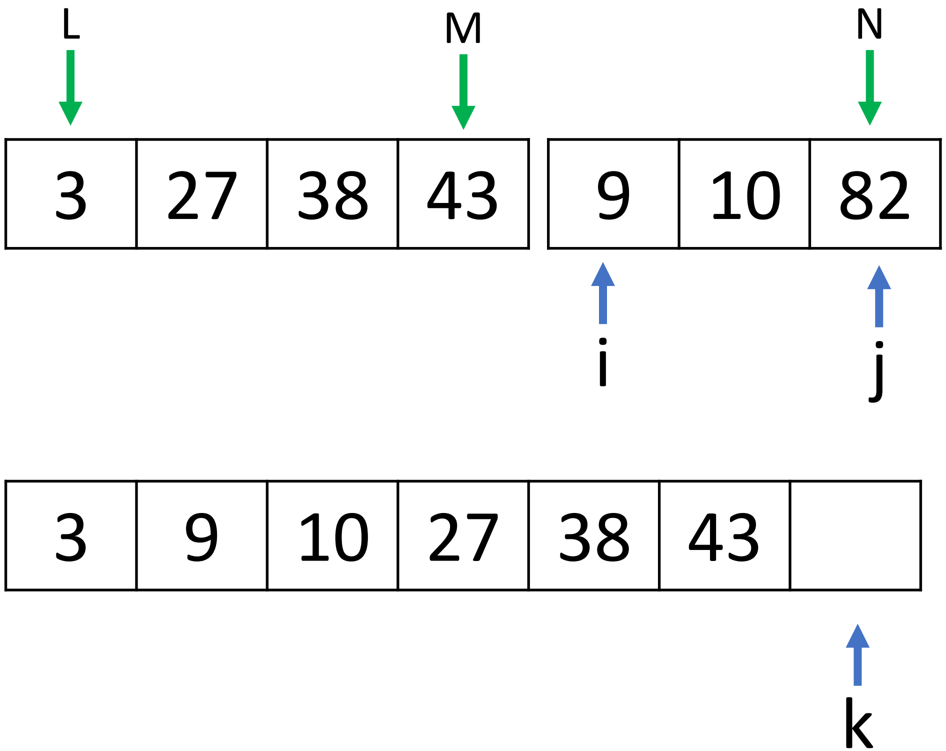
    i=L, j=M+1
    k= 0
    p = new T[N-L+1]
    while i<=M&&j<=N:
        if $\mathbf{A_i<A_j}$:
            p[k] = A[i]
            k+=1 ; i+=1;
        else
            p[k] = A[j]
            k+=1 ; j+=1;

merge:合并2个有序序列



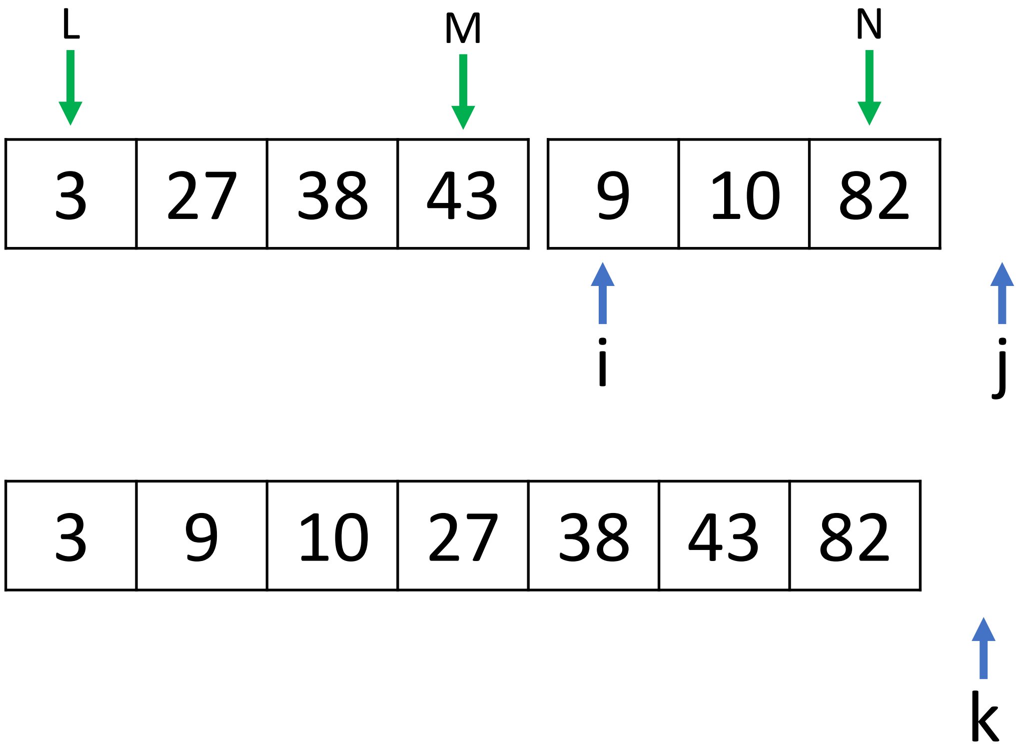

```
Merge(A, L ,M, N)
    i=L,  j=M+1
    k= 0
    p = new T[N-L+1]
    while i<=M&&j<=N:
        if A_i<A_j:
            p[k] = A[i]
            k+=1 ; i+=1;
        else
            p[k] = A[j]
            k+=1 ; j+=1;
```

merge:合并2个有序序列




```
Merge(A, L ,M, N)
    i=L,  j=M+1
    k= 0
    p = new T[N-L+1]
    while i<=M&&j<=N:
        …
    while(i <= M)
        p[k++] = A[i++];
    while(j <= N)
        p[k++] = A[j++];
```

merge:合并2个有序序列




```
Merge(A, L ,M, N)
    i=L,  j=M+1
    k= 0
    p = new T[N-L+1]
    while i<=M&&j<=N:
        ...
    while(i <= M)
        p[k++] = A[i++];
    while(j <= N)
        p[k++] = A[j++];
```

merge的事件复杂度：$T(n) = O(n)$

时间复杂度

```
merge_sort(A,L,R)
  if (L == R) return;
  m = (L+R)/2
  merge_sort(A, L, m);
  merge_sort(A, m+1,R);
  merge(A,L,m,R)
```

$\boxed{n\log_2 n}$

$$T(n) = \begin{cases} 0 & n = 1 \\ T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + n & n > 1 \end{cases}$$

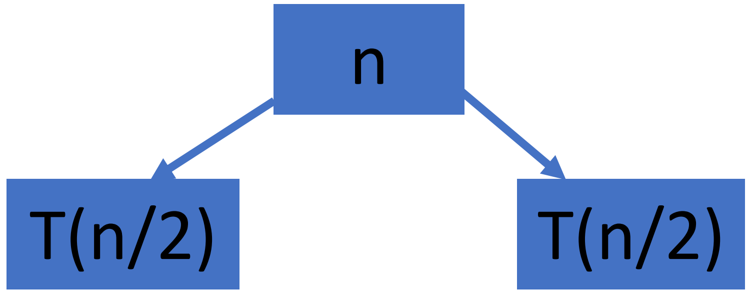
假设n = 2^k

$$T(n) = \begin{cases} 0 & n = 1 \\ 2T(n/2) + n & n > 1 \end{cases}$$

T(n)

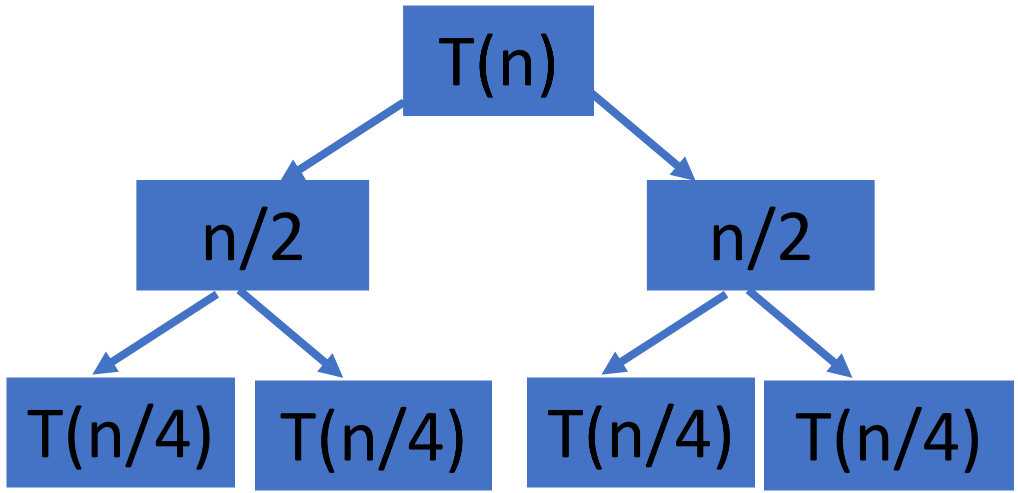
假设n = 2^k

$$T(n) = \begin{cases} 0 & n = 1 \\ 2T(n/2) + n & n > 1 \end{cases}$$



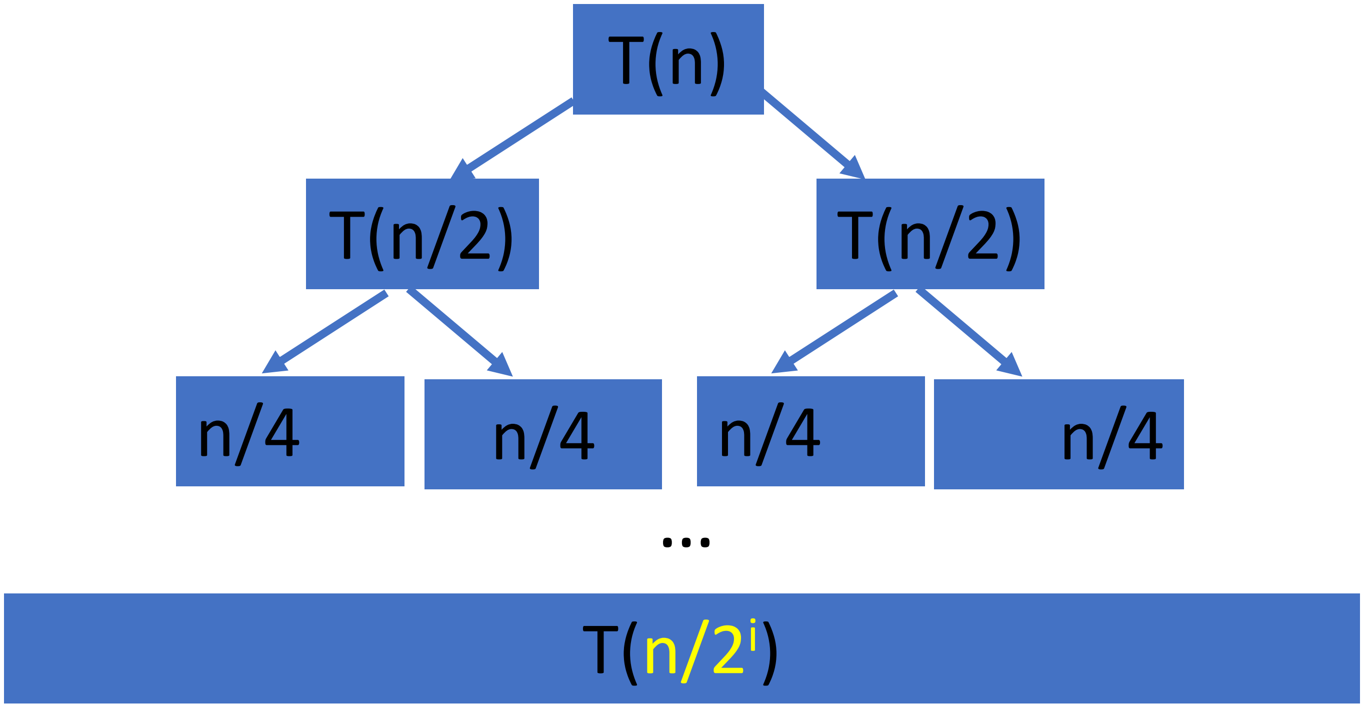
假设n = 2^k

$$T(n) = \begin{cases} 0 & n = 1 \\ 2T(n/2) + n & n > 1 \end{cases}$$

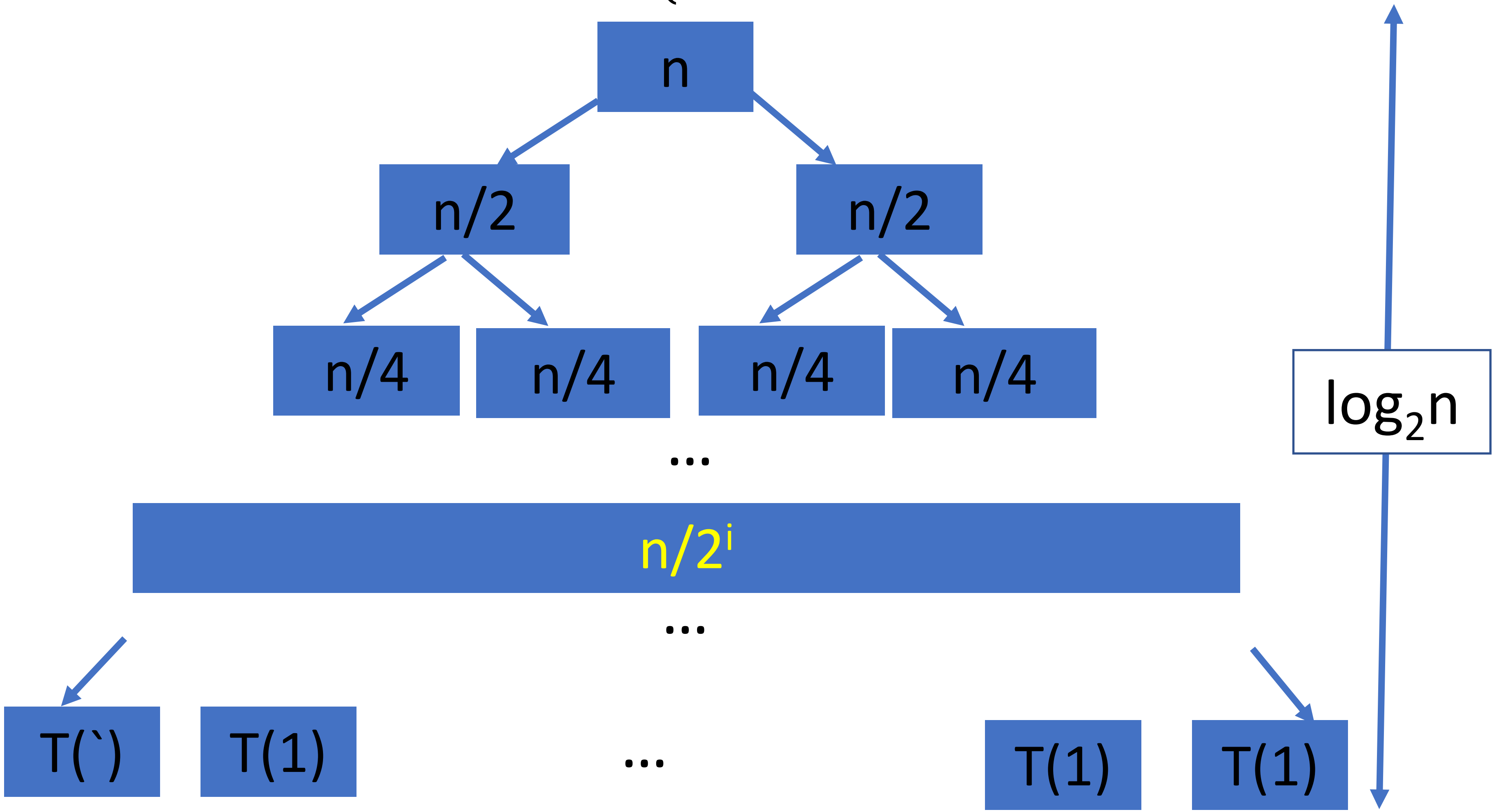


假设$n = 2^k$

$$T(n) = \begin{cases} 0 & n = 1 \\ 2T(n/2) + n & n > 1 \end{cases}$$



假设n = 2^k $$T(n) = \begin{cases} 0 & n = 1 \\ 2T(n/2) + n & n > 1 \end{cases}$$



- 应用主定理

$$T(n) = \begin{cases} 0 & n = 1 \\ T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + n & n > 1 \end{cases}$$

T(n) = 2T(n/2)+n

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + O(n^d)$$

1. $a = b^d$. T(n) = $O(n^d \log n)$

2. $a < b^d$. T(n) = $O(n^d)$

3. $a > b^d$. T(n) = $O(n^{\log_b a})$

归纳假设： $T(n) \le n\lceil \log_2 n \rceil$

Base case: $n = 1$.

令 $n_1 = \lfloor n/2 \rfloor$ $n_2 = \lceil n/2 \rceil$ 有 $n = n_1 + n_2$

$$
\begin{aligned}
T(n) &\le T(n_1) + T(n_2) + n \\
&\le n_1 \lceil \log_2 n_1 \rceil + n_2 \lceil \log_2 n_2 \rceil + n \\
&\le n_1 \lceil \log_2 n_2 \rceil + n_2 \lceil \log_2 n_2 \rceil + n \\
&= n \lceil \log_2 n_2 \rceil + n \\
&\le n (\lceil \log_2 n \rceil - 1) + n \\
&= n \lceil \log_2 n \rceil
\end{aligned}
$$

$$
\begin{aligned}
n_2 &= \lceil n/2 \rceil \\
&\le \left\lceil 2^{\lceil \log_2 n \rceil} / 2 \right\rceil \\
&= 2^{\lceil \log_2 n \rceil} / 2
\end{aligned}
$$

$$
\log_2 n_2 \le \lceil \log_2 n \rceil - 1
$$

an integer

根据定义证明

$$T(n)=\begin{cases} 1 & n=1 \\ T(\lfloor n/2 \rfloor)+T(\lceil n/2 \rceil)+n & n>1 \end{cases}$$

求证: T(n) = O(n$\log_2$n)

证明:

假设存在正常数C,使得小于n的任何正整数k,

$T(k) \le ck \log k$

要证$T(n) \le cn \log n$

$$\mathrm{T}(n) \le \mathrm{C}\left\lfloor \frac{n}{2} \right\rfloor \log_2 \left\lfloor \frac{n}{2} \right\rfloor + \mathrm{C}\left\lceil \frac{n}{2} \right\rceil \log_2 \left\lceil \frac{n}{2} \right\rceil + n$$

$$\le C\frac{n}{2}\log_2\frac{n}{2} + C\frac{n+1}{2}\log_2\frac{n+1}{2} + n$$

$$\le C\frac{n}{2}(\log_2 \mathrm{n} - \log_2 2) + C\frac{n+1}{2}(\log_2 n) + n$$

$$= C\frac{n}{2}\log_2 \mathrm{n} - C\frac{n}{2}\log_2 2 + C\frac{n}{2}\log_2 n + \frac{C}{2}\log_2 n + \mathrm{n}$$

$$= Cn\log_2 \mathrm{n} \text{ - } Cn + \frac{C}{2}\log_2 n + n$$

$$\le Cn\log_2 \mathrm{n} \text{ - } Cn + \frac{C}{2}\log_2 n + n$$

$$\mathrm{T}(n) \le Cn\log_2 \mathrm{n} - C_2 n + \frac{C}{2}\log_2 n + n$$

当$-Cn+\frac{C}{2}\log_2 n + n \le 0$时，$\mathrm{T}(n) \le Cn\log_2 \mathrm{n}$

$$C(n-\frac{\log_2 n}{2}) \ge n$$

$$C \ge \frac{n}{n-\frac{\log_2 n}{2}} = 1 + \frac{\frac{\log_2 n}{2}}{n-\frac{\log_2 n}{2}} \qquad C \ge 2，\ \mathrm{T}(n) \le Cn\log_2 n$$

$$\because n \ge 2\frac{\log_2 n}{2} = \log_2 n，\therefore \frac{\frac{\log_2 n}{2}}{n-\frac{\log_2 n}{2}} \le 1$$

逆序数

Youtube频道：**hwdong**

博客：**hwdong-net**.github.io

逆序数

- 对于数组中的2个元素，可以定义它们的正序和逆序。
- 如果规定当i<j，且$a_i<a_j$为**正序**。那么i<j，且$a_j<a_i$就称为**逆序**。
- 数组中如果任何2个元素都是正序的，这个数组称为有序数组。如：

  (1,3,5,6,9)
- 对于有序数组，其逆序对的数目为0

逆序数

- 对于非有序数组，其逆序对的数目（简称**逆序数**）不为0。

  (5,6,1,3,9)
- 逆序对有: (5,1),(5,3),(6,1),(6,3),
- 逆序数越大，有序性越差。

逆序数的应用

- 音乐网站将你对音乐的喜爱评分和其他人的评分计算逆序数，来判断你们的口味的相似性。
- 计算你对音乐评分和他人对音乐评分的逆序数

Songs

	A	B	C	D	E
Me	1	2	3	4	5
You	1	3	4	2	5

<u>Inversions</u>

3-2, 4-2

逆序数的应用

- Voting theory.
- Collaborative filtering.
- Measuring the "sortedness" of an array.
- Sensitivity analysis of Google's ranking function.
- Rank aggregation for meta-searching on the Web.
- Nonparametric statistics (e.g., Kendall's Tau distance).

逆序数的应用

- 投票理论。
- 协作过滤。
- 测量数组的 "排序性"。
- Google排名函数的敏感性分析。
- 网络上元搜索的排名聚合。
- 非参数统计（如Kendall's Tau距离）。

逆序数的计算：蛮力法

```
counting_inversions(a)
    count = 0
    for i=1 to n:
        for j=i+1 to n:
            if a[j]<a[i] :
                count = count+1
    return count
```

$(\ldots\ a_i \ldots\ a_j \ldots)$

$T(n) = O(n^2)$

逆序数的计算：分治法

- 分： 原数组一分为二：A和B

1	5	4	8	10	2	6	9	3	7

1	5	4	8	10

2	6	9	3	7

逆序数的计算：分治法

- 分： 原数组一分为二： A和B
- 治： 对每个子数组递归计算逆序数

1	5	4	8	10	2	6	9	3	7

1	5	4	8	10

2	6	9	3	7

5-4

6-3 9-3 9-7

逆序数的计算：分治法

- 分： 原数组一分为二： A和B
- 治： 对每个子数组递归计算逆序数
- 合： 计算A和B之间的逆序数: (a, b) ,a∈A, b∈B

1	5	4	8	10	2	6	9	3	7

1	5	4	8	10

2	6	9	3	7

4–2 4–3 5–2 5–3 8–2 8–3 8–6 8–7 10–2 10–3 10–6 10–7 10–9

逆序数的计算：分治法

- 分： 原数组一分为二： A和B
- 治： 对每个子数组递归计算逆序数
- 合： 计算A和B之间的逆序数: (a,b) ,a∈A, b∈B
- 返回三个计数之和

1	5	4	8	10	2	6	9	3	7

1	5	4	8	10

2	6	9	3	7

1 + 3 + 13 = 17

逆序数的计算：如何组合2个子问题

- 如何计算A和B之间的逆序数: (a,b) ,a∊A, b∊B

1	5	4	8	10	2	6	9	3	7

4-2 4-3 5-2 5-3 8-2 8-3 8-6 8-7 10-2 10-3 10-6 10-7 10-9

逆序数的计算：如何组合2个子问题

- 如何2个子数组是有序的，则很快

逆序数的计算：如何组合2个子问题

- 如何2个子数组是有序的，则很快

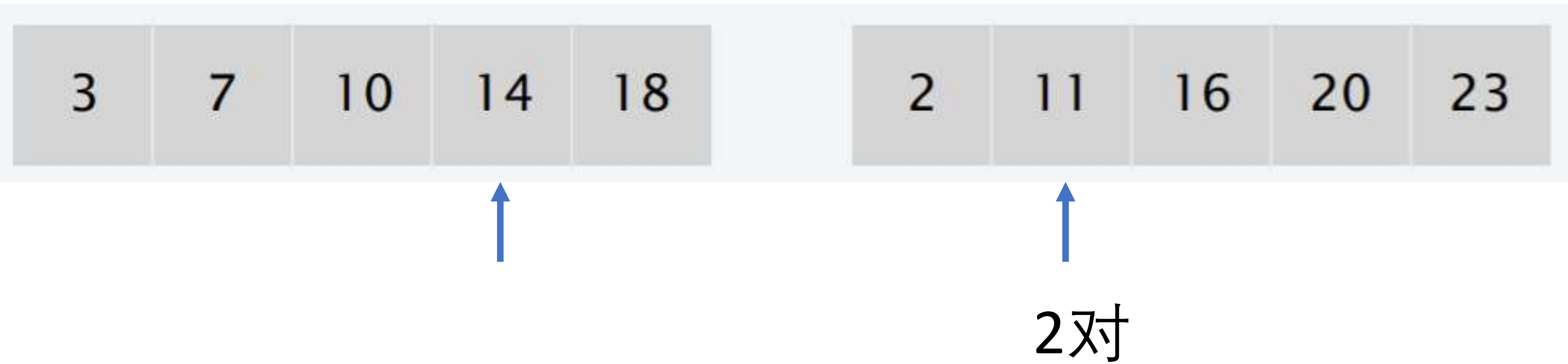
5对

逆序数的计算：如何组合2个子问题

- 如何2个子数组是有序的，则很快

逆序数的计算：如何组合2个子问题

- 如何2个子数组是有序的，则很快



逆序数的计算：如何组合2个子问题

- 如何2个子数组是有序的，则很快

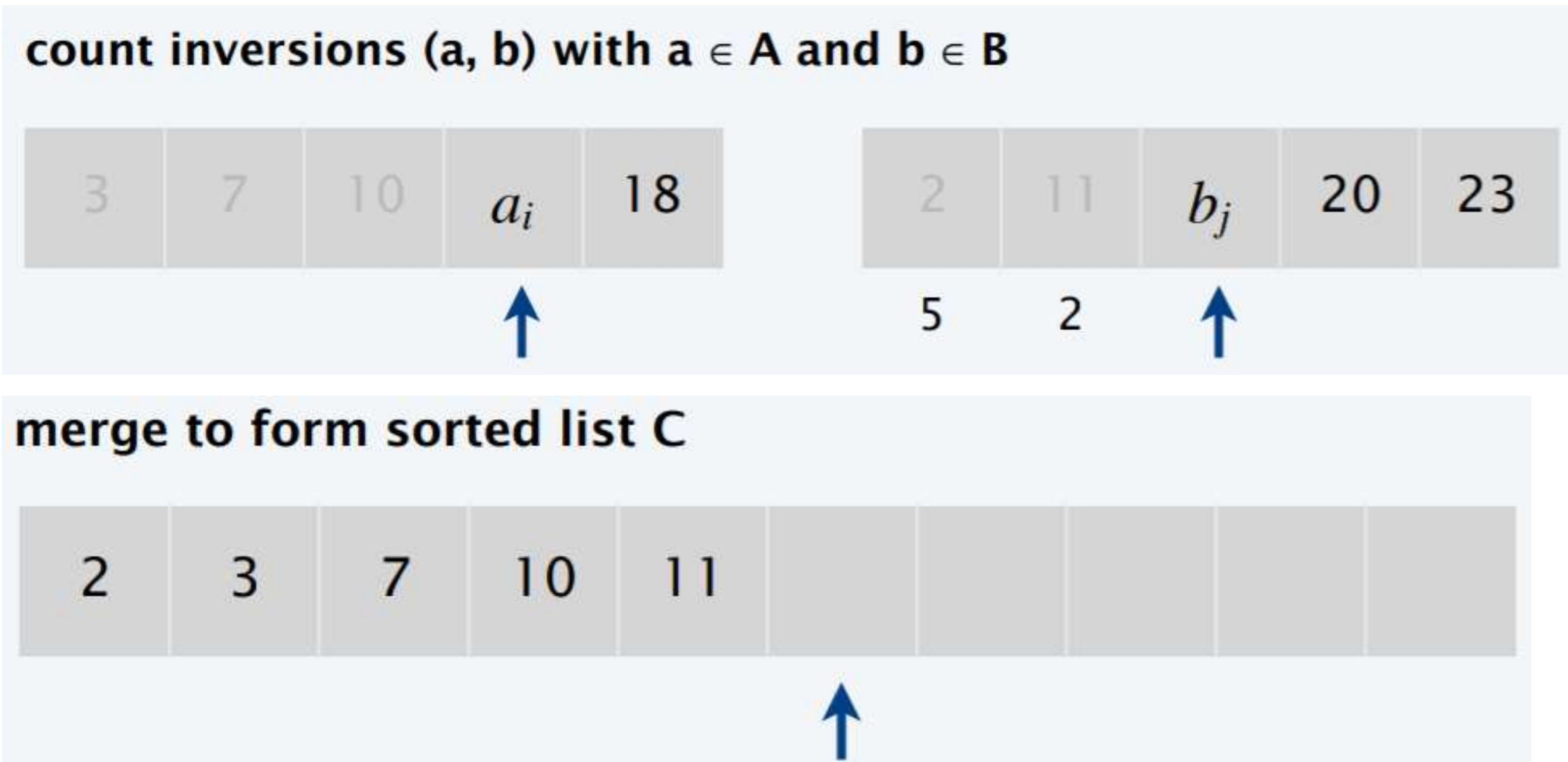
1对

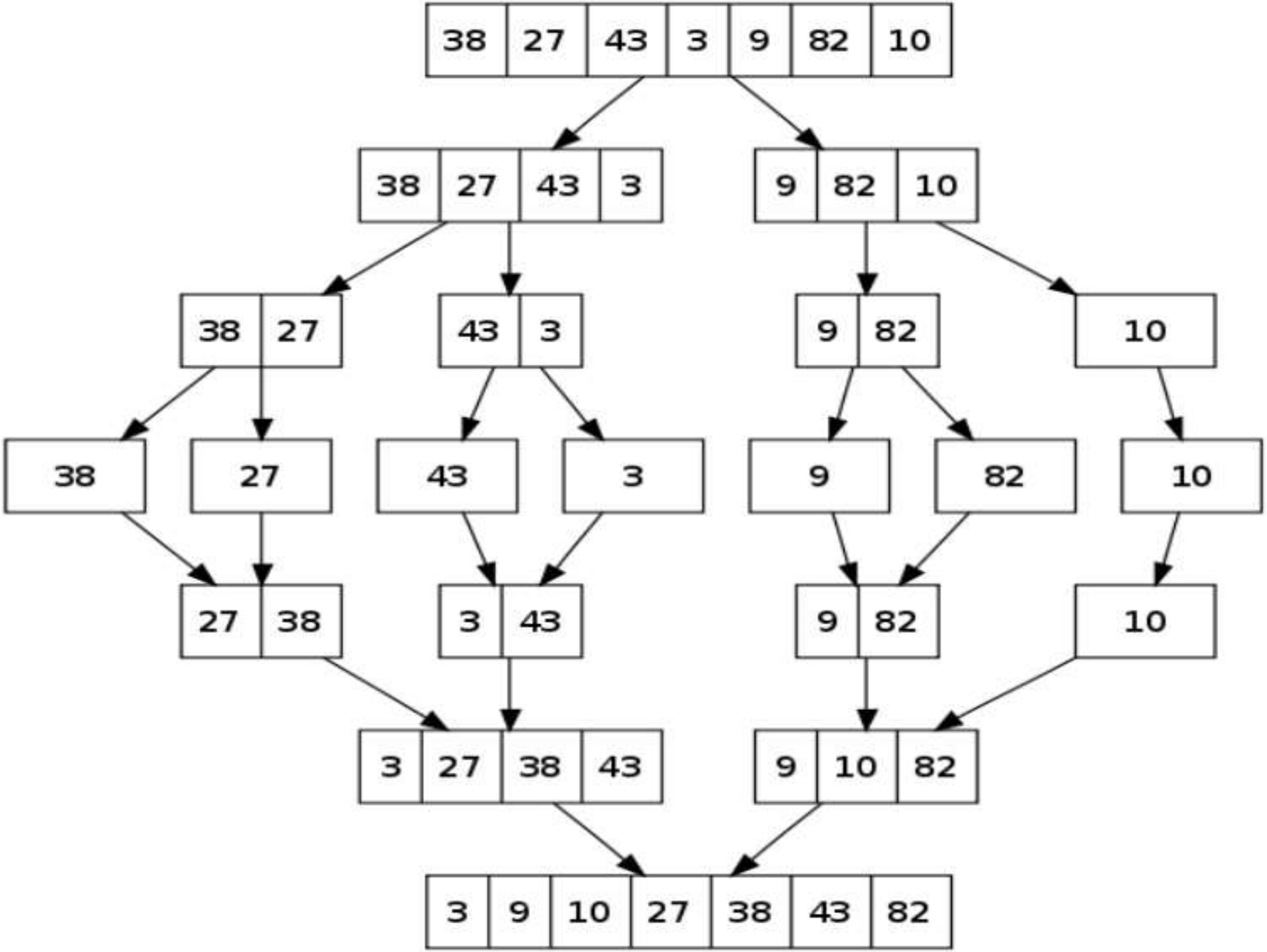
类似于2路归并，T(n) = O(n)

计算逆序数的同时，可以将这2个有序数组合并为一个更大的有序数组

逆序数的计算：如何组合2个子问题

- 归并排序的同时计算逆序数




38 27 43 3 9 82 10
38 27 43 3
9 82 10
38 27
43 3
9 82
10
38
27
43
3
9
82
10
27 38
3 43
9 82
10
3 27 38 43
9 10 82
3 9 10 27 38 43 82

```
sort_count(a, L, R ):
    if R==L:  return 0
    m = (L+R)/2
    c1 = sort_count(a, L, m )             ← T(n/2)
    c2 = sort_count(a, m+1,R )            ← T(n/2)
    c3 merge_count(a,L,m,R)               ← Θ(n)
     return c1+c2+c3
```

时间复杂度分析

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + \Theta(n) & \text{if } n > 1 \end{cases}$$

$T(n) = \Theta(n\log_2 n)$

快速排序

Youtube频道： **hwdong**

博客： **hwdong-net**.github.io

快速排序

- 任选一个基准元素将序列分为左右2部分，左边的不大于基准元素，右边的不小于基准元素

[70, 74,60,76, 83,72,55,65,79]

快速排序

- 任选一个基准元素将序列分为左右2部分，左边的不大于基准元素，右边的不小于基准元素

[70, 74,60,76, 83,72,55,65,79]

[65, 55,60] , 70, [83,72,76,74,79]

快速排序

- 任选一个基准元素将序列分为左右2部分，左边的不大于基准元素，右边的不小于基准元素
- 对左序列快速排序

[70, 74,60,76, 83,72,55,65,79]

[65, 55,60] , 70, [83,72,76,74,79]

⋮

[55, 60,65] , 70,

快速排序

- 任选一个基准元素将序列分为左右2部分，左边的不大于基准元素，右边的不小于基准元素
- 对左序列快速排序
- 对右序列快速排序

[70, 74,60,76, 83,72,55,65,79]

[65, 55,60] , 70, [83,72,76,74,79]

⋮ ⋮

[55, 60,65] , 70, [72,74,76,79,83]

快速排序

- 任选一个基准元素将序列分为左右2部分，左边的不大于基准元素，右边的不小于基准元素
- 对左序列快速排序
- 对右序列快速排序

[70, 74,60,76, 83,72,55,65,79]

[65, 55,60] , 70, [83,72,76,74,79]

⋮ ⋮

[55, 60,65] , 70, [72,74,76,79,83]

[70, 74,60,76, 83,72,55,65,79]

[65, 55,60] , 70, [83,72,76,74,79]

[60,55], 65,

[70, 74,60,76, 83,72,55,65,79]

[65, 55,60] , 70, [83,72,76,74,79]

[60,55], 65

[55],60, 65

[70, 74,60,76, 83,72,55,65,79]

[65, 55,60] , 70, [83,72,76,74,79]

[60,55], 65, 70, [83,72,76,74,79]

[55],60,65 [79,72,76,74], 83

[70, 74,60,76, 83,72,55,65,79]

[65, 55,60] , 70, [83,72,76,74,79]

[60,55], 65, 70, [83,72,76,74,79]

[55],60,65 [79,72,76,74], 83

[74,72,76],79

[70, 74,60,76, 83,72,55,65,79]

[65, 55,60] , 70, [83,72,76,74,79]

[60,55], 65, 70, [83,72,76,74,79]

[55],60,65 [79,72,76,74], 83

[74,72,76],79

[72],74,[76]

Qsort([70, 74,60,76, 83,72,55,65,79])

划分: [65, 55,60] , 70, [83,72,76,74,79]

Qsort([65, 55,60])

Qsort([83,72,76,74,79])

.

.

Qsort([70, 74,60,76, 83,72,55,65,79])

划分: [65, 55,60] , 70, [83,72,76,74,79]

Qsort([65, 55,60])

划分: [60, 55], 65, []

Qsort([60,55])

Qsort([])

Qsort([83,72,76,74,79])

.

.

Qsort([70, 74,60,76, 83,72,55,65,79])

划分: [65, 55,60] , 70, [83,72,76,74,79]

Qsort([65, 55,60])

划分: [60, 55], 65, []

Qsort([60,55])

划分: [55], 60, []

Qsort([55])

Qsort([])

Qsort([])

Qsort([83,72,76,74,79])

⋮

```
void QSort(T a[], int L, int H) {
    if(L < H){    //待排序数列长度大于1




    }
}
```

Qsort([70, 74,60,76, 83,72,55,65,79])

```
void QSort(T a[], int L, int H) {
    if(L < H){    //待排序数列长度大于1
        int pivotloc = Partition(a, L, H);



    }
}
```

Qsort([70, 74,60,76, 83,72,55,65,79])

划分: [65, 55,60] , 70, [83,72,76,74,79]

```
void QSort(T a[], int L, int H) {
    if(L < H){    //待排序数列长度大于1
        int pivotloc = Partition(a, L, H);
        //对左子序列进行快速排序
        QSort(a, L, pivotloc - 1);


    }
}
```

Qsort([70, 74,60,76, 83,72,55,65,79])

划分: [65, 55,60] , 70, [83,72,76,74,79]

Qsort([65, 55,60])

```
void QSort(T a[], int L, int H) {
   if(L < H){   //待排序数列长度大于1
      int pivotloc = Partition(a, L, H);
      //对左子序列进行快速排序
      QSort(a, L, pivotloc - 1);
        //对右子序列进行快速排序
      QSort(a, pivotloc + 1, H);
   }
}
```

Qsort([70, 74,60,76, 83,72,55,65,79])

划分: [65, 55,60] , 70, [83,72,76,74,79]

Qsort([65, 55,60])

Qsort([83,72,76,74,79])

一次划分：3-分割

给定一个数组A和一个基准元素p，将A分割成3部分：

- 小于p的元素在左序列L
- 等于p的元素在中间序列M
- 大于p的序列在右序列R

[70, 74,60,76, 83,72,55,65,79]

⇓

[65, 55,60] , 70, [83,72,76,74,79]

- 任选一个基准元素，将该元素放到临时存储里

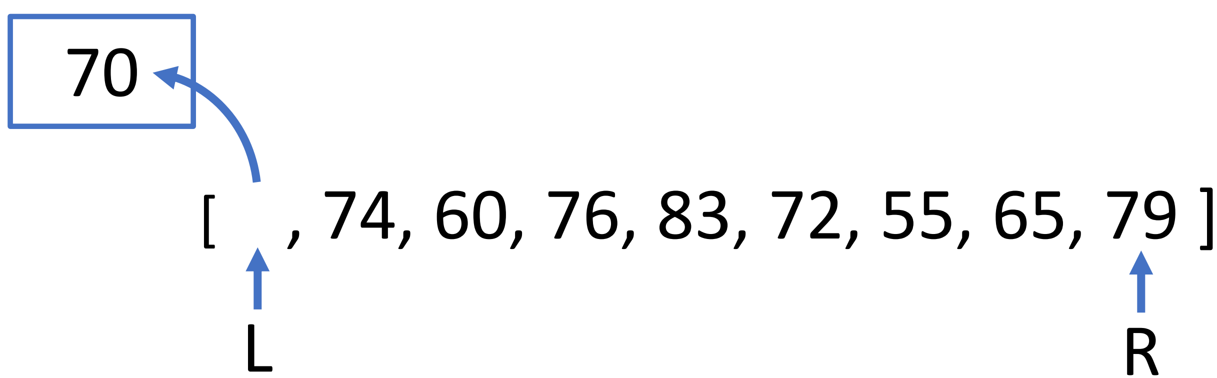
[**70** , 74, 60, 76, 83, 72, 55, 65, 79]

- 任选一个基准元素，将该元素放到临时存储里

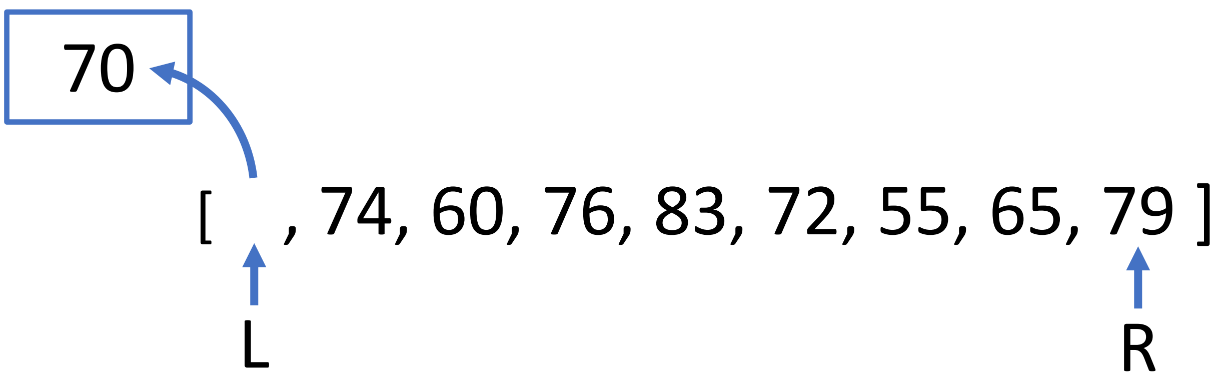
70

[, 74, 60, 76, 83, 72, 55, 65, 79]

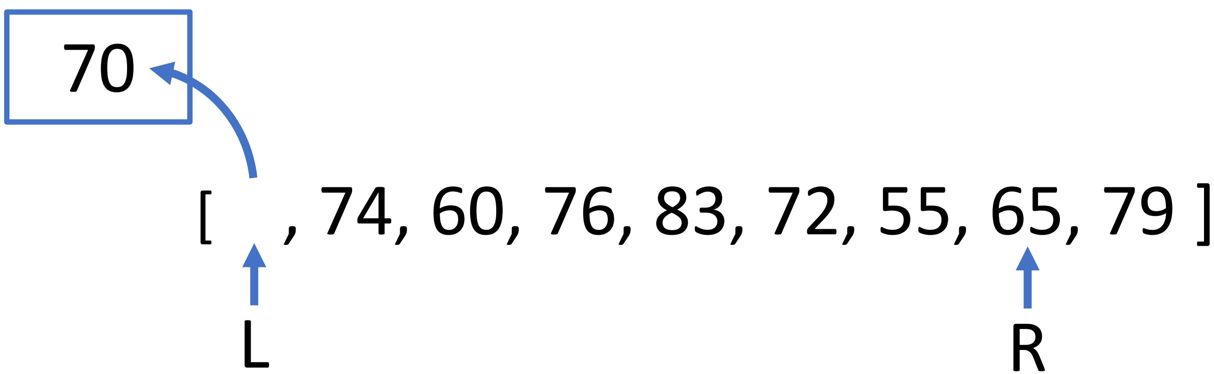
- 任选一个基准元素，将该元素放到临时存储里
- 两个指针(L、R)指向数组的开头和结束元素



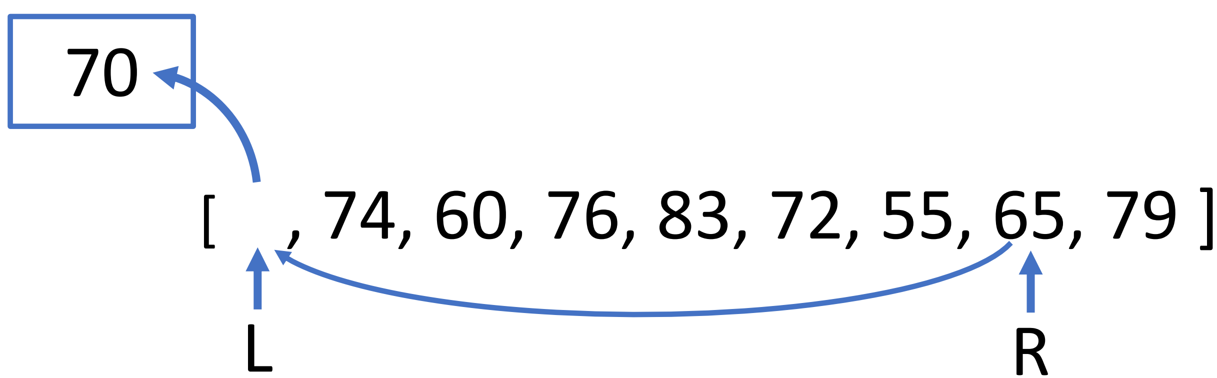
- 任选一个基准元素，将该元素放到临时存储里
- 两个指针(L、R)指向数组的开头和结束元素
- while L < R:
-     while L<R and A[R]>=t: R--;



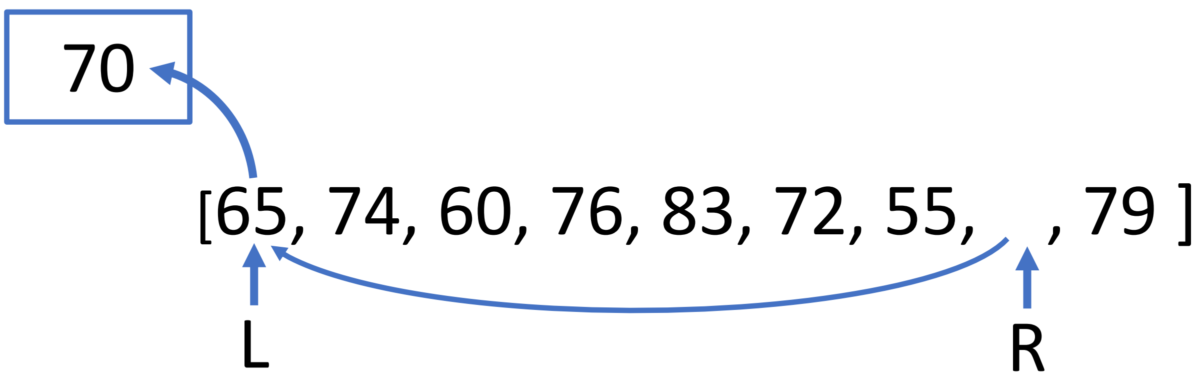
- 任选一个基准元素，将该元素放到临时存储里
- 两个指针(L、R)指向数组的开头和结束元素
- while L < R:
- while L<R and A[R]>=t: R--;



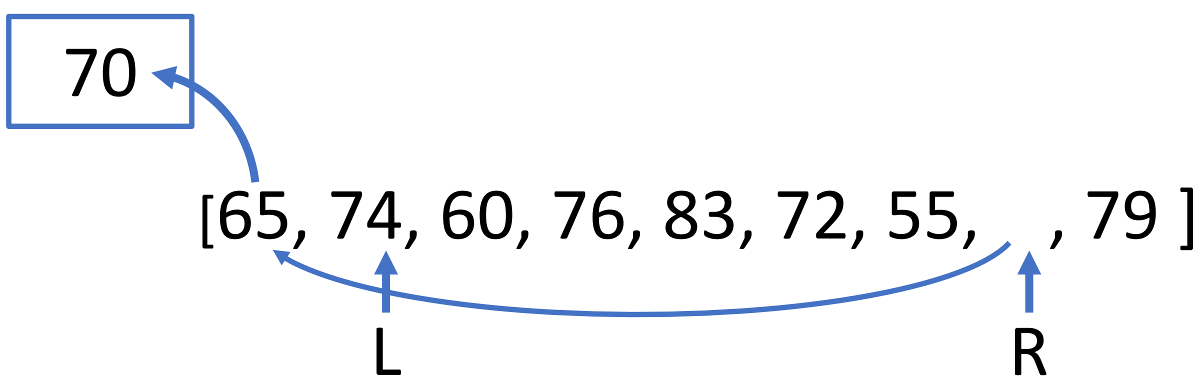
- 任选一个基准元素，将该元素放到临时存储里
- 两个指针(L、R)指向数组的开头和结束元素
- while L < R:
-     while L<R and A[R]>=t: R--;
-      A[L] = A[R];



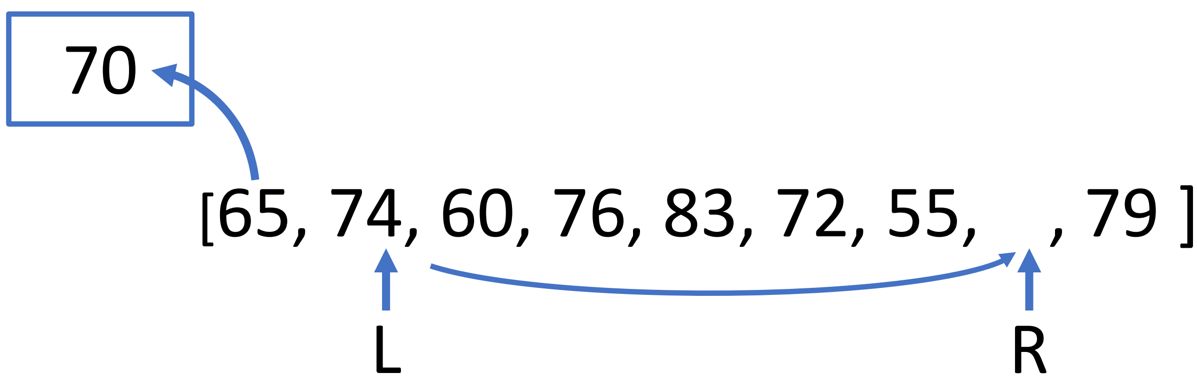
- 任选一个基准元素，将该元素放到临时存储里
- 两个指针(L、R)指向数组的开头和结束元素
- while L < R:
-     while L<R and A[R]>=t: R--;
-      A[L] = A[R];



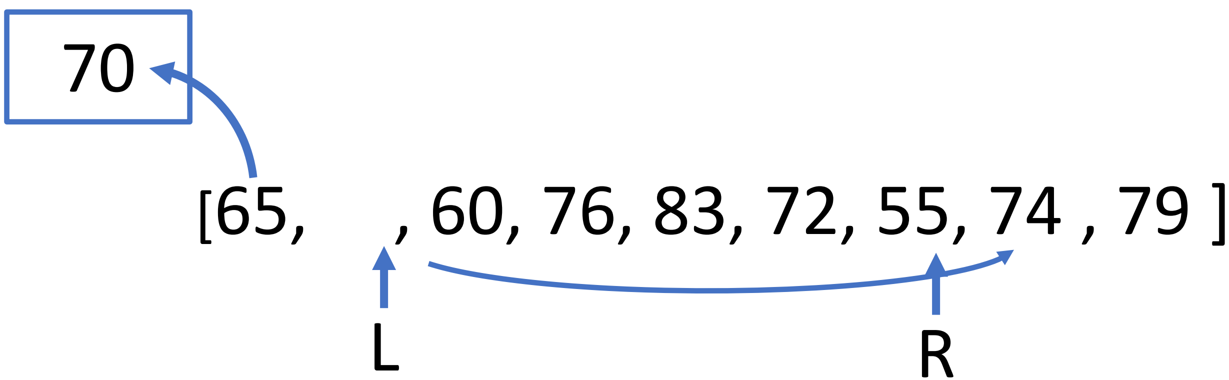
- 任选一个基准元素，将该元素放到临时存储里
- 两个指针(L、R)指向数组的开头和结束元素
- while L < R:
-     while L<R and A[R]>=t: R--;
-     A[L] = A[R];



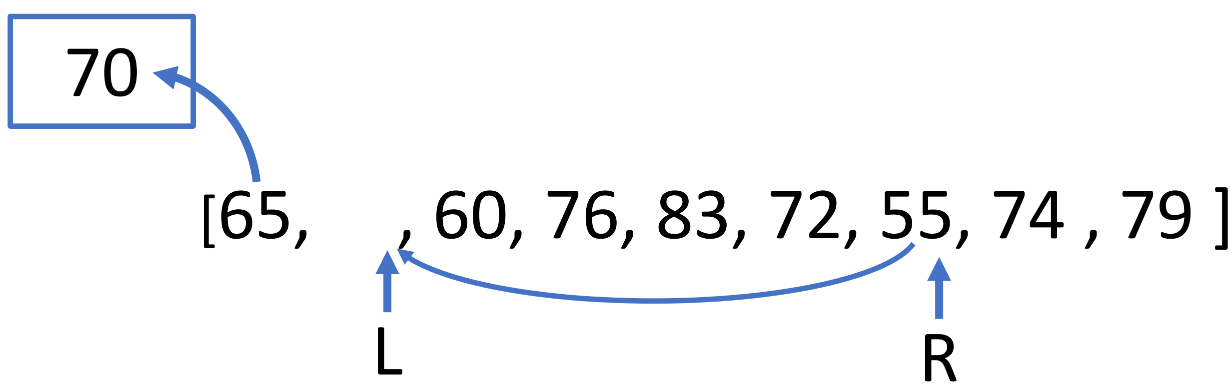
- 任选一个基准元素，将该元素放到临时存储里
- 两个指针(L、R)指向数组的开头和结束元素
- while L < R:
-     while L<R and A[R]>=t: R--;
-      A[L] = A[R];
-      while L<R and A[L]<=t: L++;
-      A[R] = A[L];



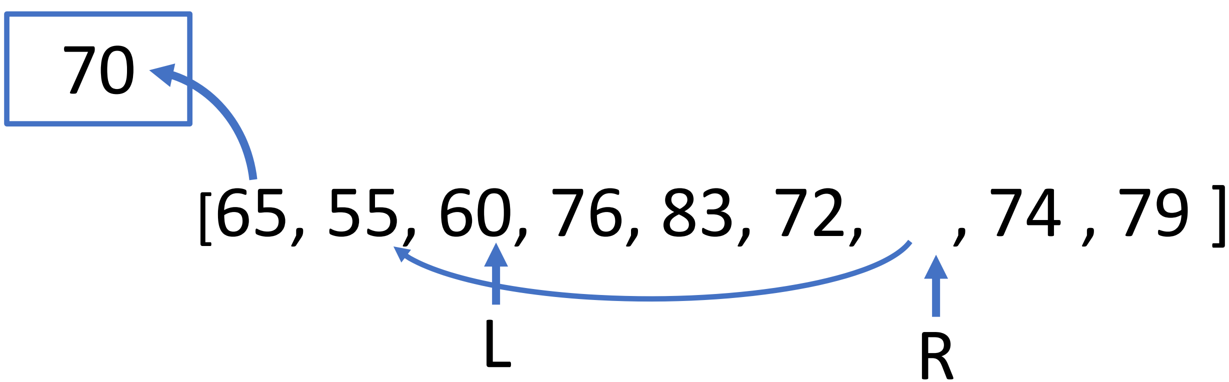
- 任选一个基准元素，将该元素放到临时存储里
- 两个指针(L、R)指向数组的开头和结束元素
- while L < R:
-     while L<R and A[R]>=t: R--;
-      A[L] = A[R];
-      while L<R and A[L]<=t: L++;
-      A[R] = A[L];



- 任选一个基准元素，将该元素放到临时存储里
- 两个指针(L、R)指向数组的开头和结束元素
- while L < R:
-     while L<R and A[R]>=t: R--;
-      A[L] = A[R];
-      while L<R and A[L]<=t: L++;
-      A[R] = A[L];



- 任选一个基准元素，将该元素放到临时存储里
- 两个指针(L、R)指向数组的开头和结束元素
- while L < R:
-     while L<R and A[R]>=t: R--;
-      A[L] = A[R];
-      while L<R and A[L]<=t: L++;
-      A[R] = A[L];



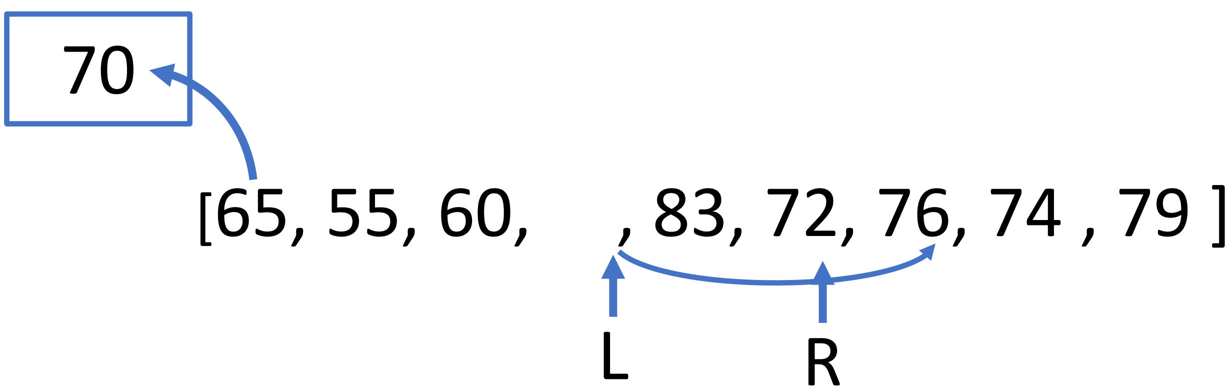
- 任选一个基准元素，将该元素放到临时存储里
- 两个指针(L、R)指向数组的开头和结束元素
- while L < R:
-     while L<R and A[R]>=t: R--;
-      A[L] = A[R];
-      while L<R and A[L]<=t: L++;
-      A[R] = A[L];

70

[65, 55, 60, 76, 83, 72,   , 74 , 79]

L   R

- 任选一个基准元素，将该元素放到临时存储里
- 两个指针(L、R)指向数组的开头和结束元素
- while L < R:
-     while L<R and A[R]>=t: R--;
-      A[L] = A[R];
-      while L<R and A[L]<=t: L++;
-      A[R] = A[L];



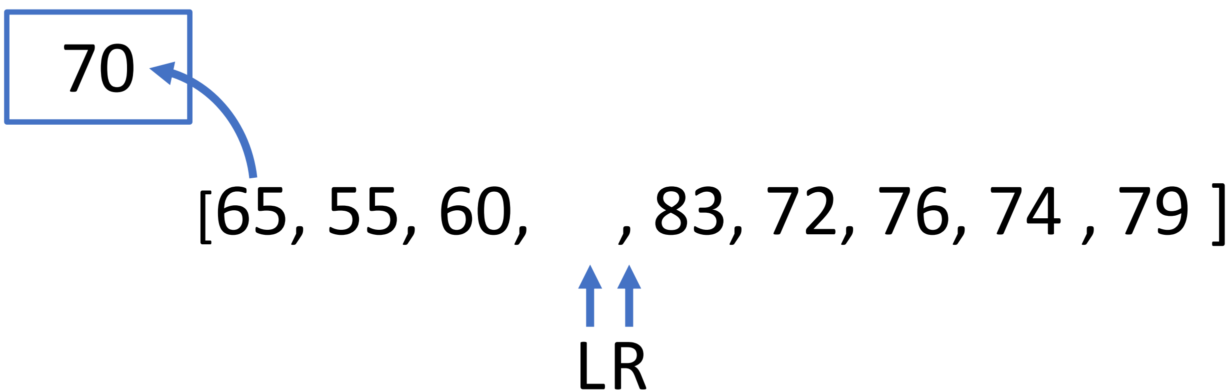
- 任选一个基准元素，将该元素放到临时存储里
- 两个指针(L、R)指向数组的开头和结束元素
- while L < R:
-     while L<R and A[R]>=t: R--;
-     A[L] = A[R];
-     while L<R and A[L]<=t: L++;
-     A[R] = A[L];

$\boxed{70}$

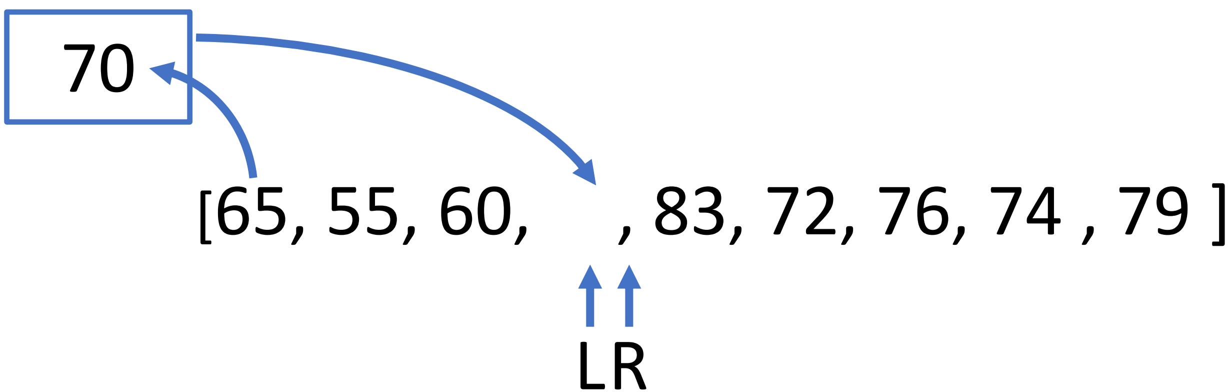
[65, 55, 60,   , 83, 72, 76, 74 , 79]

L   R

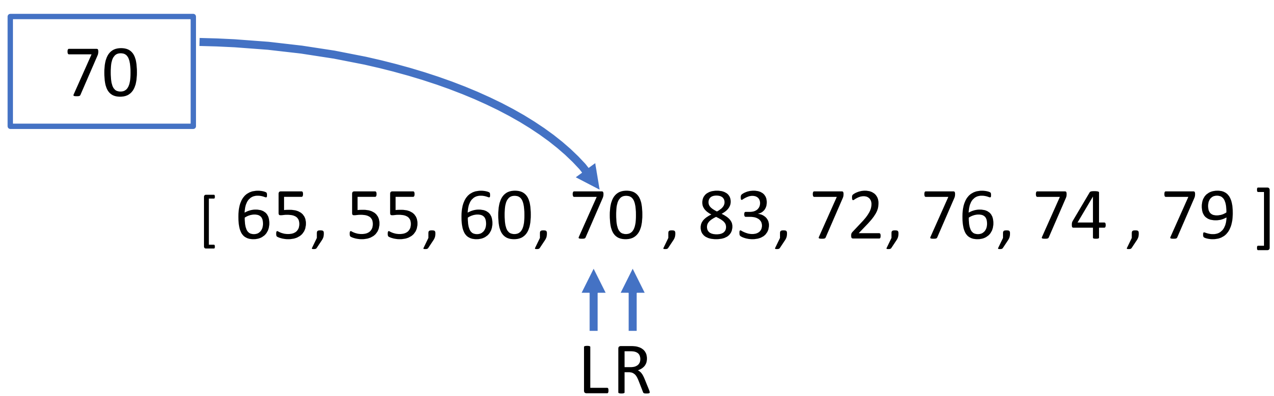
- 任选一个基准元素，将该元素放到临时存储里
- 两个指针(L、R)指向数组的开头和结束元素
- while L < R:
-     while L<R and A[R]>=t: R--;
-     A[L] = A[R];
-     while L<R and A[L]<=t: L++;
-     A[R] = A[L];



- 任选一个基准元素，将该元素放到临时存储里
- 两个指针(L、R)指向数组的开头和结束元素
- while L < R:
-     while L<R and A[R]>=t: R--;
-      A[L] = A[R];
-      while L<R and A[L]<=t: L++;
-      A[R] = A[L];



- 任选一个基准元素，将该元素放到临时存储里
- 两个指针(L、R)指向数组的开头和结束元素
- while L < R:
-     while L<R and A[R]>=t: R--;
-      A[L] = A[R];
-     while L<R and A[L]<=t: L++;
-      A[R] = A[L];

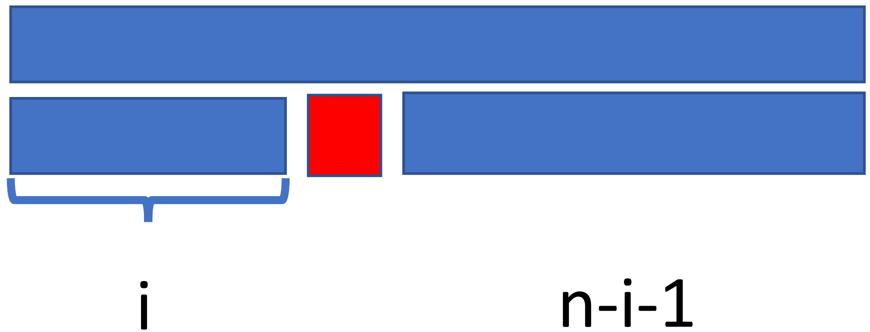


- 任选一个基准元素，将该元素放到临时存储里
- 两个指针(L、R)指向数组的开头和结束元素
- while L < R:
-     while L<R and A[R]>=t: R--;
-     A[L] = A[R];
-     while L<R and A[L]<=t: L++;
-     A[R] = A[L];

[65, 55, 60, 70 , 83, 72, 76, 74 , 79]

LR

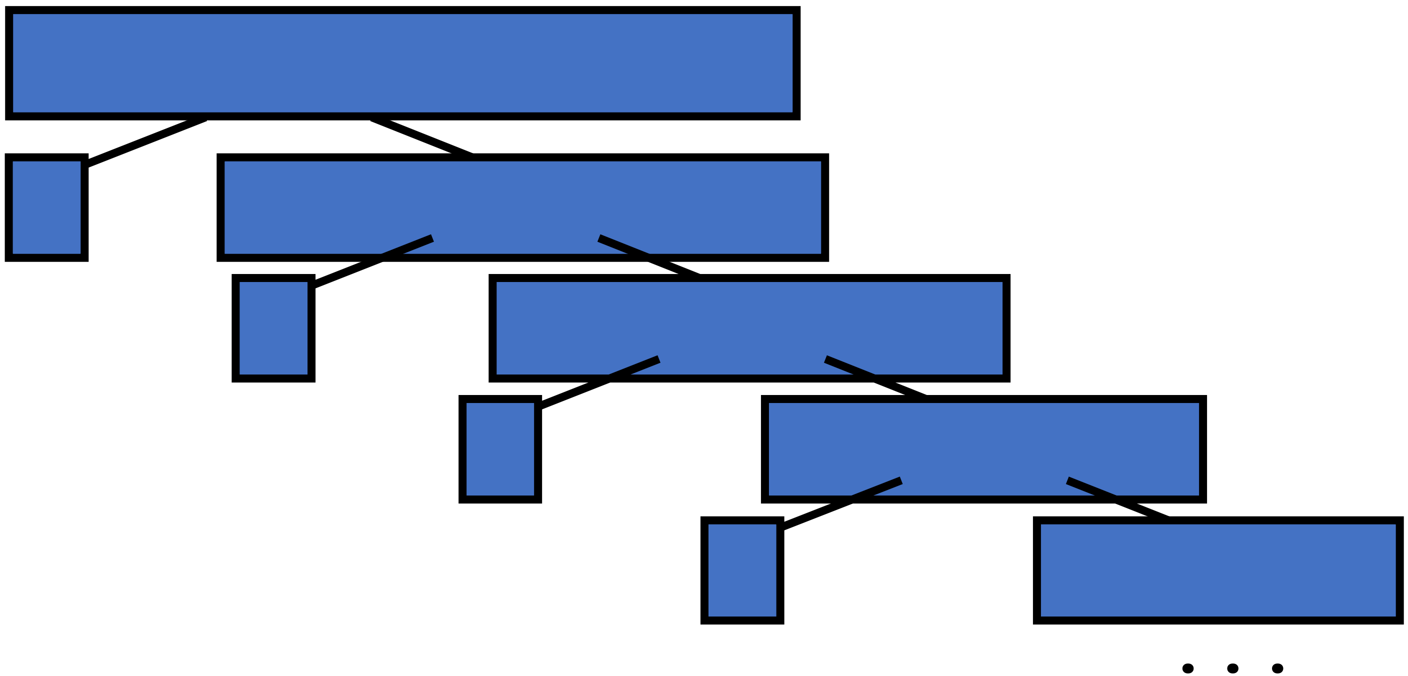
时间复杂度分析



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

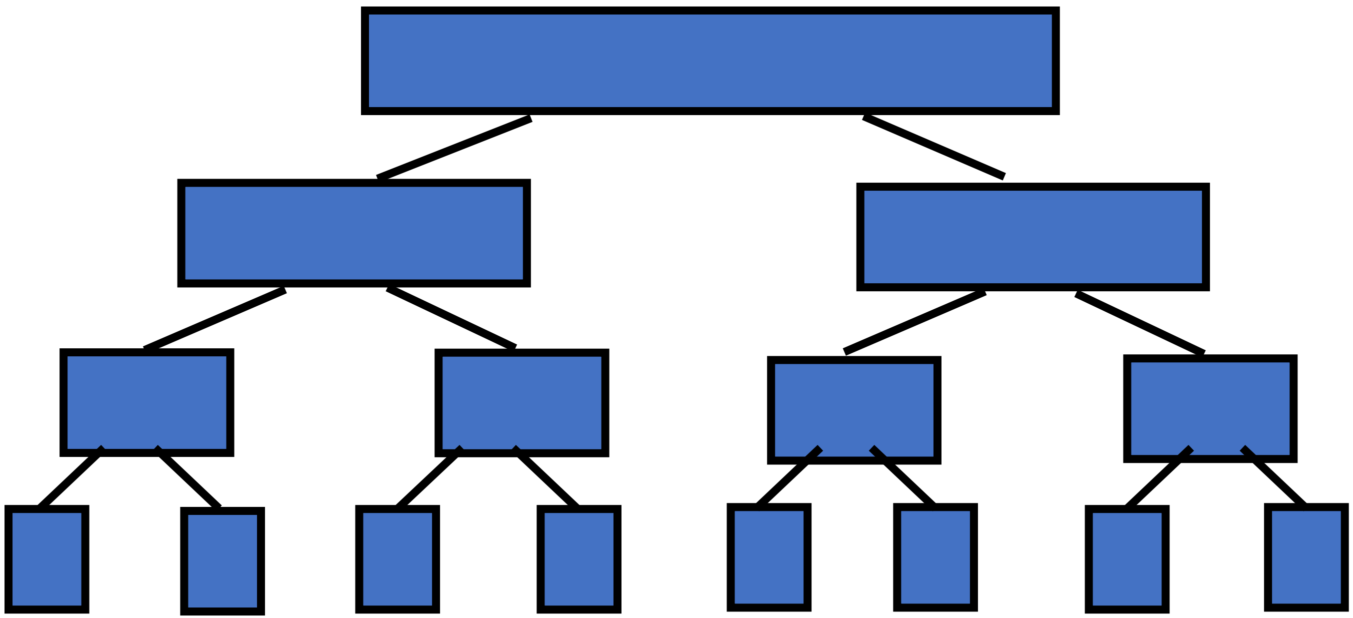
- 每次划分都只分出一个元素
- 需要n层递归

$O(n*n) = O(n^2)$



- 有序性好的原始数据不适合用快速排序

- 每次划分，分成相等的2部分
- 只要$\log_2 n$层。 $O(n*\log_2 n)$ $2^k>=n$



- 杂乱无章的原始数据适合用快速排序

随机快速排序

- 随机选择一个元素p作为基准元素
- 根据p对数组进行3-分割：L、M、R
- 递归的对L,R进行**随机快速排序**

```
randomized_quicksort(A):
    if(A的元素个数不超过1): return
    在A中均匀的随机选择一个元素p
    L,M,R ← partition(A,p)                ← Θ(n)
    randomized_quicksort(L)                ← T(i)
    randomized_quicksort(R)                ← T(n-i-1)
```

时间复杂度分析

$$T(n) = (n-1) + \frac{1}{n}\sum_{i=0}^{n-1}(T(i) + T(n-i-1)).$$

$$T(n) = (n-1) + \frac{2}{n}\sum_{i=1}^{n-1} T(i)$$

猜测(假设总是一分为二的话):

$T(n) = O(n\log_2 n)$

证明（数学归纳法）：

$$
\begin{aligned}
T(n) &\le (n-1) + \frac{2}{n}\sum_{i=1}^{n-1}(ci\ln i) \\
&\le (n-1) + \frac{2}{n}\int_1^n (cx\ln x)dx \\
&\le (n-1) + \frac{2}{n}\Big((c/2)n^2\ln n - cn^2/4 + c/4\Big) \\
&\le cn\ln n, \quad \text{for } c = 2.
\end{aligned}
$$

if $f(x)$ is an increasing function, then

$$
\sum_{i=1}^{n-1} f(i) \le \int_1^n f(x)dx,
$$

整数乘法

Youtube频道： **hwdong**

博客： **hwdong-net**.github.io

整数乘法

- 2个n位整数相乘，时间复杂度是$O(n^2)$

```
      4 2 6 5
      3 7 1 8
  -------------
    3 4 1 2 0
     4 2 6 5
 2 9 8 5 5
1 2 7 9 5
  -------------
1 5 8 5 7 2 7 0
```

整数乘法:分治法

- n位整数分解成n/2位的2个整数,4个n/2位整数乘

65*18 = 1170　　4 2*37 = 1554

4 2*18+37*65 = 756+2405 = 3161

$$
\begin{array}{r}
1170 \\
316100 \\
15540000 \\
\hline
15857270
\end{array}
$$

整数乘法:分治法

- n位整数分解成n/2位的2个整数,4个n/2位整数乘

65*18 = 1170　　4 2*37 = 1554

4 2*18+37*65 = 756+2405 = 3161

```
     1170
   316100
 15540000
---------
 15857270
```

整数乘法:分治法

- n位整数分解成n/2位的2个整数,4个n/2位整数乘

$$x = 10^{n/2}a + b$$

$$y = 10^{n/2}c + d$$

$$\begin{aligned} x \cdot y &= (10^{n/2}a + b) \cdot (10^{n/2}c + d) \\ &= 10^n a \cdot c + 10^{n/2}(a \cdot d + b \cdot c) + b \cdot d \end{aligned}$$

整数乘法:分治法

- n位整数分解成n/2位的2个整数,4个n/2位整数乘

Algorithm 1: Mult1(x, y)

Split x and y into $x = 10^{\frac{n}{2}} a + b$ and $y = 10^{\frac{n}{2}} c + d$

$z_1 =$ Mult1(a, c)

$z_2 =$ Mult1(a, d)

$z_3 =$ Mult1(b, c)

$z_4 =$ Mult1(b, d)

return $z_1 \cdot 10^n + 10^{\frac{n}{2}}(z_2 + z_3) + z_4$

$$x \cdot y = (10^{n/2}a + b) \cdot (10^{n/2}c + d)$$
$$= 10^n a \cdot c + 10^{n/2}(a \cdot d + b \cdot c) + b \cdot d$$

$$T(n) = 4T(n/2) + c_0 n$$

$$T(n) = 4T(n/2) + c_0 n$$
$$= 4(4T(n/2^2) + c_0 n/2) + c_0 n = 4^2 T(n/2^2) + 2c_0 n + c_0 n$$
$$= 4^k T(1) + 2^{k-1} c_0 n + \cdots + 2c_0 n + c_0 n$$
$$= 2^k \cdot 2^k + (2^k - 1)c_0 n = n^2 + c_0 n^2 - c_0 n = \Theta(n^2)$$

因为

$$a \cdot d + b \cdot c = (a+b)\cdot(c+d) - a\cdot c - b\cdot d$$

所以

$$\begin{aligned} x \cdot y &= (10^{n/2}a + b)\cdot(10^{n/2}c + d) \\ &= 10^{n} a\cdot c + 10^{n/2}(a\cdot d + b\cdot c) + b\cdot d \\ &= 10^{n}\boxed{a\cdot c} + 10^{n/2}(\boxed{(a+b)\cdot(c+d)} - \boxed{a\cdot c} - \boxed{b\cdot d}) + \boxed{b\cdot d} \end{aligned}$$

$$T(n) \le \underbrace{T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + T(1 + \lceil n/2 \rceil)}_{\text{递归}} + \underset{\text{加、减、移位}}{\Theta(n)}$$

$$T(n) = O(n^{log_2 3}) = O(n^{1.585})$$

整数乘法:分治法

- n位整数分解成n/2位的2个整数,3次n/2位整数乘

Algorithm 2: Karatsuba(x, y)

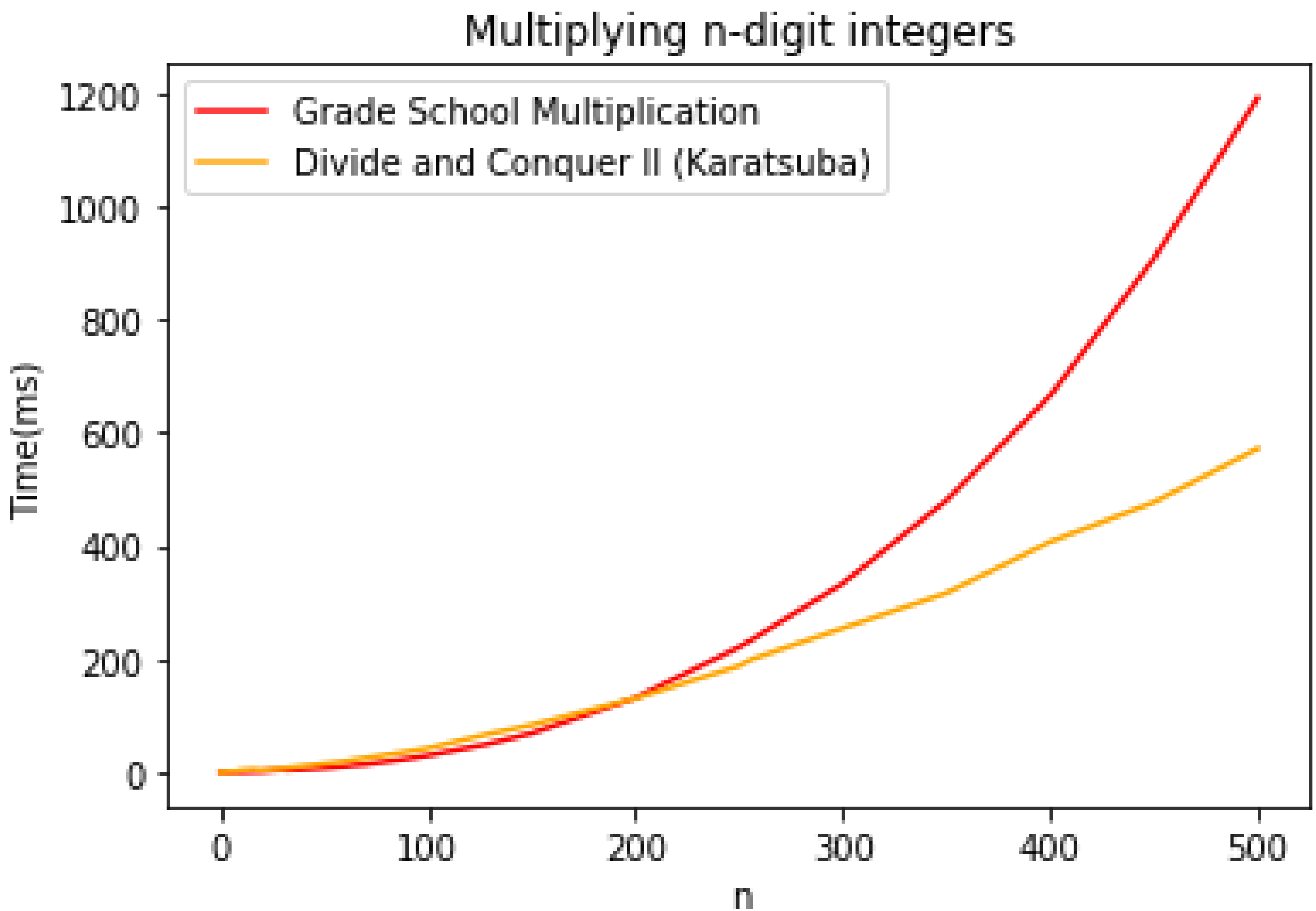
Split $x = 10^{\frac{n}{2}}a + b$ and $y = 10^{\frac{n}{2}}c + d$
$z_1 =$ Karatsuba$(a,\ c)$
$z_2 =$ Karatsuba$(b,\ d)$
$z_3 =$ Karatsuba$(a + b,\ c + d)$
$z_4 = z_3 - z_1 - z_2$
return $z_1 \cdot 10^n + z_4 \cdot 10^{\frac{n}{2}} + z_2$

- 提高效率的技巧：减少子问题数量

$$T(n) = 4T(n/2) + c_0 n$$

$$\Downarrow$$

$$T(n) \leq T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + T(1 + \lceil n/2 \rceil) + \Theta(n)$$



Multiplying n-digit integers
Grade School Multiplication
Divide and Conquer II (Karatsuba)
Time(ms)
0
200
400
600
800
1000
1200
0
100
200
300
400
500
n

```
karatsuba(X, Y):
        if  X < 10 and Y < 10:
                return X*Y;
        n= maximum(get_n(X), get_n(Y))
        n_2 = n/2
        p = power(10,n_2)    // equivalent to 10^n_2
        a = floor(X/p)
        b = X%p
        c = floor(Y/p)
        d = Y%p
```

```
ac = karatsuba(a,c)
bd = karatsuba(b,d)
e = karatsuba(a+c, b+d) - ac - bd

return power(10,2*n_2)*ac + power(10,n_2)*e + b
```

```cpp
#include <stream>
#include <cmath>
#include <algorithm>

int get_n(long num){
    int count = 0;
    while (num > 0)  {
        count++;
        num /= 10;
    }
    return count;
}
int n = std::max(get_n(X), get_n(Y));
```

```
long karatsuba(long X, long Y){
    if (X < 10 && Y < 10)       return X * Y;    // Base Case
    int n = fmax(get_n(X), get_n(Y));
    int n_2 = (int)ceil(n / 2.0);
    long p = (long)pow(10, n_2);
    long a = (long)floor(X / (double)p);
    long b = X % p;
    long c = (long)floor(Y / (double)p);
    long d = Y % p;
    long ac = karatsuba(a, c);
    long bd = karatsuba(b, d);
    long e = karatsuba(a + b, c + d) - ac - bd;
    return (long)(pow(10 * 1L, 2 * n_2) * ac + pow(10 * 1L, n_2) * e + bd);
}
```

```
#include <string>
#include <iostream>
using namespace std;
int multiplication(int X, int Y){
    string x = to_string(X);
    string y = to_string(Y);
    int result = 0;

    for (int i = 0; i < y.length(); i++)    {
        int carry = 0;               // 进位
        string inter_res = "";   // intermediate result

        for (int j = x.length() - 1; j >= 0; j--)       {
            int num = (y[i] - '0') * (x[j] - '0') + carry;
            if (num > 9 && j > 0)   {
                inter_res = to_string(num % 10) + inter_res;
                carry = num / 10;
            }
```

```
        else  {
            inter_res = to_string(num) + inter_res;
            carry = 0;
        }
    }

    result *= 10;
    result += stoi(inter_res);
  }
  return result;
}

int main(){
  cout << multiplication(12, 5);
}
```

矩阵乘法

Youtube频道： **hwdong**

博客： **hwdong-net**.github.io

矩阵乘法

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix}, \quad \mathbf{B} = \begin{pmatrix} b_{11} & b_{12} & \cdots & b_{1p} \\ b_{21} & b_{22} & \cdots & b_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{np} \end{pmatrix}$$

$$\mathbf{C} = \mathbf{AB} = \begin{pmatrix} a_{11}b_{11} + \cdots + a_{1n}b_{n1} & a_{11}b_{12} + \cdots + a_{1n}b_{n2} & \cdots & a_{11}b_{1p} + \cdots + a_{1n}b_{np} \\ a_{21}b_{11} + \cdots + a_{2n}b_{n1} & a_{21}b_{12} + \cdots + a_{2n}b_{n2} & \cdots & a_{21}b_{1p} + \cdots + a_{2n}b_{np} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1}b_{11} + \cdots + a_{mn}b_{n1} & a_{m1}b_{12} + \cdots + a_{mn}b_{n2} & \cdots & a_{m1}b_{1p} + \cdots + a_{mn}b_{np} \end{pmatrix}$$

$$c_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j} + \cdots + a_{in}b_{nj} = \sum_{k=1}^{n} a_{ik}b_{kj}$$

$$T(m,n,p) = \Theta\,(m*n*p)$$

矩阵乘法

- 如果A,B都是n*n的方阵，则时间复杂度为

$$T(n) = \Theta\ (n^3)$$

```
multiply_matrix( A, B, n):
  C← initMatrix(n)
  for i = 0 to n-1:
      for j = 0 to n-1:
          c[i][j] = 0
          for k = 1 to n:
              c[i][j]+= (c[i][k]*c[k][j])
  return C
```

矩阵乘法:分治法

1) 分：将n*n矩阵分解成4个n/2*n/2的子矩阵

$$A=\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix},\quad B=\begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix},\quad C=\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

矩阵乘法:分治法

2) 治：计算8次小矩阵的乘积

$$C_{11} = A_{11}B_{11} + A_{12}B_{21}$$
$$C_{12} = A_{11}B_{12} + A_{12}B_{22}$$
$$C_{21} = A_{21}B_{11} + A_{22}B_{21}$$
$$C_{22} = A_{21}B_{12} + A_{22}B_{22}$$

$$W(n) = 8W(n/2) + O(n^2) \Rightarrow W(n) = O(n^{\log_2 8}) = O(n^3)$$

3) 合：直接

矩阵乘法: Strassen算法(1969)

- 将小矩阵乘积的次数从8次减少为7次。

$$M_1 = (A_{11} + A_{22})(B_{11} + B_{22})$$
$$M_2 = (A_{21} + A_{22})B_{11}$$
$$M_3 = A_{11}(B_{12} - B_{22})$$
$$M_4 = A_{22}(B_{21} - B_{11})$$
$$M_5 = (A_{11} + A_{12})B_{22}$$
$$M_6 = (A_{21} - A_{11})(B_{11} + B_{12})$$
$$M_7 = (A_{12} - A_{22})(B_{21} + B_{22})$$

$$C_{11} = M_1 + M_4 - M_5 + M_7$$
$$C_{12} = M_3 + M_5$$
$$C_{21} = M_2 + M_4$$
$$C_{22} = M_1 - M_2 + M3 + M_6$$

10+8次加减

$$W(n) = 7W(n/2) + O(n^2) \implies W(n) = O(n^{\log_2 7}).$$

- 分：将A,B都分解成4个n/2*n/2的子矩阵，基于10次矩阵加减法得到14个小矩阵，转化为7个n/2规模的矩阵乘积问题。

$$
\begin{aligned}
M_1 &= (A_{11} + A_{22})(B_{11} + B_{22}) \\
M_2 &= (A_{21} + A_{22})B_{11} \\
M_3 &= A_{11}(B_{12} - B_{22}) \\
M_4 &= A_{22}(B_{21} - B_{11}) \\
M_5 &= (A_{11} + A_{12})B_{22} \\
M_6 &= (A_{21} - A_{11})(B_{11} + B_{12}) \\
M_7 &= (A_{12} - A_{22})(B_{21} + B_{22})
\end{aligned}
$$

- 治：递归计算14个小矩阵的两两乘积即M_1,…M_7.
- 合：根据M1,…,M7计算构成C的4个子矩阵C_1,C_2,C_3,C_4. 共需 8次加减法。

$$W(n) = 7W(n/2) + O(n^2) \implies W(n) = O(n^{\log_2 7}).$$

```
strassenMultiply(A, B, n):
    C← initMatrix(n)
    if n==1: C[0][0]  =A[0][0] *B [0][0]; return C;
    k← n/2
    A11← initMatrix(k)
    A12← initMatrix(k)
    A21← initMatrix(k)
    A22← initMatrix(k)
    B11← initMatrix(k)
    B12← initMatrix(k)
    B21← initMatrix(k)
    B22← initMatrix(k)
```

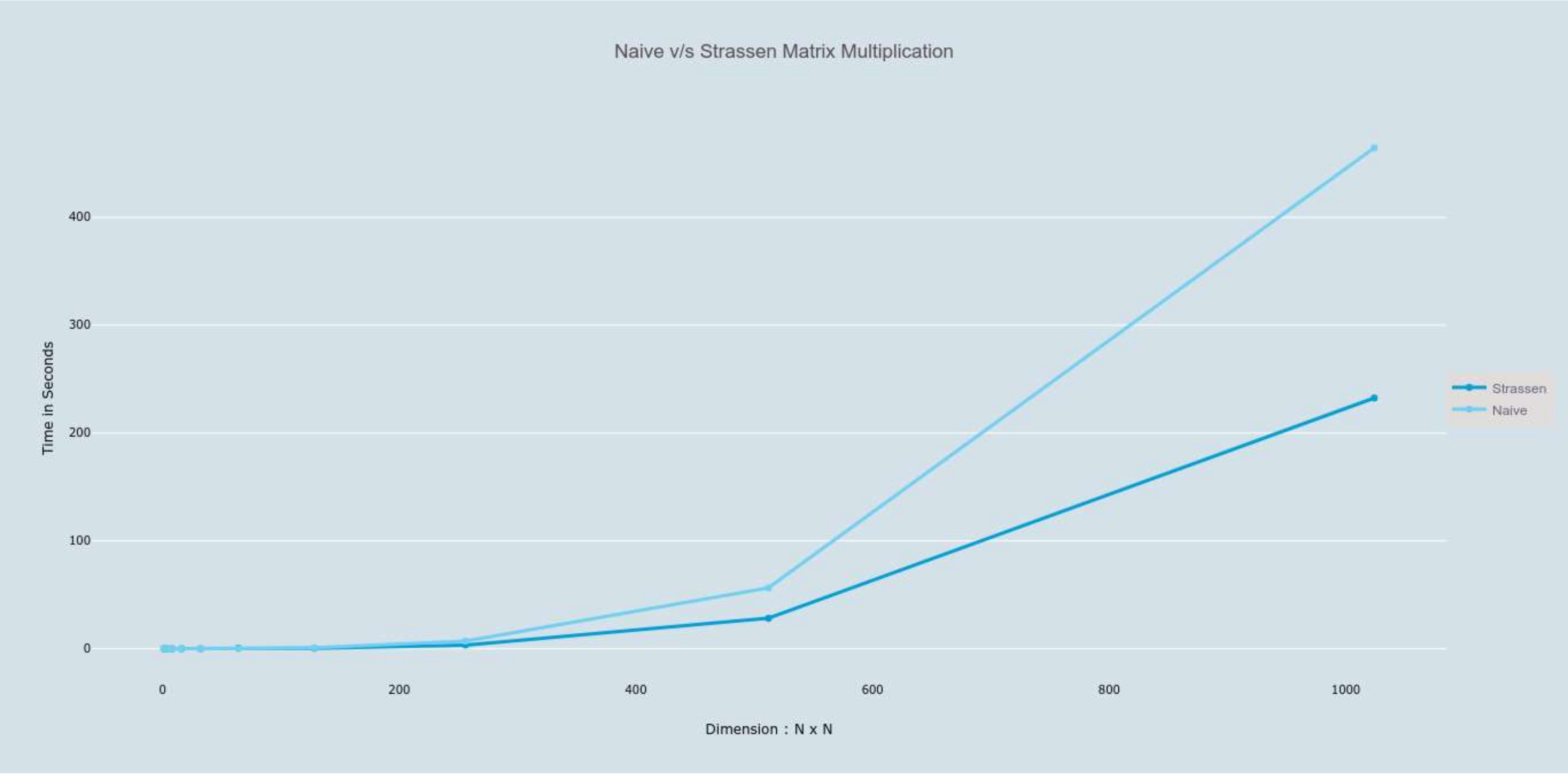
M1 ← strassenMultiply(A11, subtract(B12, B22, k), k);
M2 ← strassenMultiply(add(A11, A12, k), B22, k);
M3 ← strassenMultiply(add(A21, A22, k), B11, k);
M4 ← strassenMultiply(A22, subtract(B21, B11, k), k);
M5 ← strassenMultiply(add(A11, A22, k), add(B11, B22, k), k);
M6 ← strassenMultiply(subtract(A12, A22, k), add(B21, B22, k), k);
M7 ← strassenMultiply(subtract(A11, A21, k), add(B11, B12, k), k);

```
C11← subtract(add(add(M5, M4, k), M6, k), M2, k);
C12← add(M1, M2, k);
C21← add(M3, M4, k);
C22← subtract(subtract(add(M5, M1, k), M3, k), M7, k);

for (int i = 0; i < k; i++)
    for (int j = 0; j < k; j++) {
        C[i][j] = C11[i][j];
        C[i][j + k] = C12[i][j];
        C[k + i][j] = C21[i][j];
        C[k + i][k + j] = C22[i][j];
    }
```

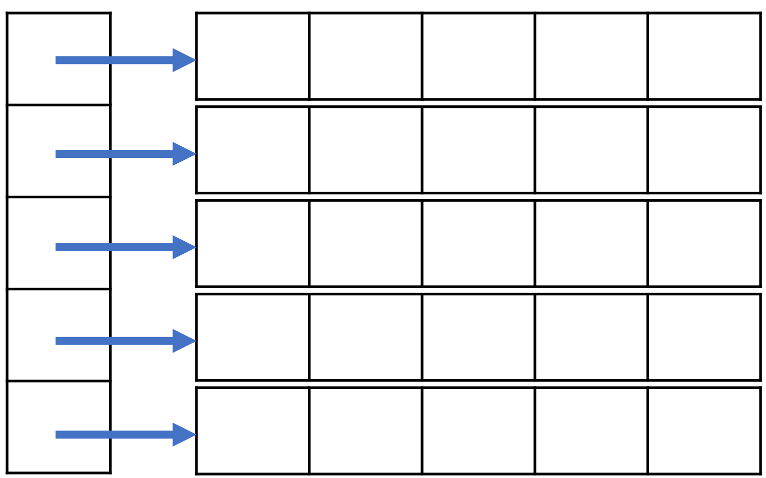
思考：矩阵的维n!=2^k?

- 填充0
- 513->1024?


Naive v/s Strassen Matrix Multiplication
Time in Seconds
400
300
200
100
0
0
200
400
600
800
1000
Dimension : N x N
Strassen
Naive

空间换时间

```
int** initializeMatrix(int n) {
    int** p = new int* [n];
    for (int i = 0; i < n; i++)
        p[i] = new int[n];
    return p;
}
```



```
int** multiply(int** A, int** B, int n) {
    int** C = initializeMatrix(n);
    zero(C, n);
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            for (int k = 0; k < n; k++)
                C[i][j] += A[i][k] * B[k][j];
    return C;
}
```

```
int** strassenMultiply(int** A, int** B, int n) {
    if (n == 1) {
        int** C = initializeMatrix(1);
        C[0][0] = A[0][0] * B[0][0];
        return C;
    }
```

```
int** C = initializeMatrix(n);
int k = n / 2;

int** A11 = initializeMatrix(k);
int** A12 = initializeMatrix(k);
int** A21 = initializeMatrix(k);
int** A22 = initializeMatrix(k);
int** B11 = initializeMatrix(k);
int** B12 = initializeMatrix(k);
int** B21 = initializeMatrix(k);
int** B22 = initializeMatrix(k);
```

```
for (int i = 0; i < k; i++)
    for (int j = 0; j < k; j++) {
        A11[i][j] = A[i][j];
        A12[i][j] = A[i][k + j];
        A21[i][j] = A[k + i][j];
        A22[i][j] = A[k + i][k + j];
        B11[i][j] = B[i][j];
        B12[i][j] = B[i][k + j];
        B21[i][j] = B[k + i][j];
        B22[i][j] = B[k + i][k + j];
    }
```

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}$$

```
int** P1 = strassenMultiply(A11, subtract(B12, B22, k), k);
int** P2 = strassenMultiply(add(A11, A12, k), B22, k);
int** P3 = strassenMultiply(add(A21, A22, k), B11, k);
int** P4 = strassenMultiply(A22, subtract(B21, B11, k), k);
int** P5 = strassenMultiply(add(A11, A22, k), add(B11, B22, k), k);
int** P6 = strassenMultiply(subtract(A12, A22, k), add(B21, B22, k), k);
int** P7 = strassenMultiply(subtract(A11, A21, k), add(B11, B12, k), k);
```

```
int** C11 = subtract(add(add(P5, P4, k), P6, k), P2, k);
int** C12 = add(P1, P2, k);
int** C21 = add(P3, P4, k);
int** C22 = subtract(subtract(add(P5, P1, k), P3, k), P7, k);

for (int i = 0; i < k; i++)
    for (int j = 0; j < k; j++) {
        C[i][j] = C11[i][j];
        C[i][j + k] = C12[i][j];
        C[k + i][j] = C21[i][j];
        C[k + i][k + j] = C22[i][j];
    }
```

选择问题

Youtube频道：**hwdong**

博客：**hwdong-net**.github.io

选择最小

[1, 23, 12, 9, 30, 2, 50]

```
MIN(A, ):
    min = ∞
    for i=0, ..., n-1:
        if A[i] < min:
            min = A[i]
    return min
```

O(1)

O(n)

Time $\Theta(n)$

选择第2小

[1, 23, 12, 9, 30, 2, 50]

先选最小，再在剩下的选择最小

```
SELECT2(A):
    min_ind = -1
    for i=0, .., n-1:
        if A[i] < A[min_ind]:
            min_ind = i
    Swap(A[0],A[min_ind])
    ... //在A[1…n-1]选最小
    return min2
```

Time: $\Theta(2n\text{-}1) = \Theta(n)$

选择第2小

[1, 23, 12, 9, 30, 2, 50]

```
SELECT2(A):
    min2 = ∞
    min = ∞
    for i=0, .., n-1:
        if A[i] < min:
            min2 = min
            min = A[i]
        else if A[i] < min2:
            min2 = A[i]
    return min2
```

Time： O(2n)

选择中位数

[1, 23, 12, 9, 30, 2, 50]

- n/2次选最小： $\Theta(n+(n-1)+\ldots n/2) = \Theta(n^2)$

选择第k小?

$[1, 23, 12, 9, 30, 2, 50] \quad k = 4$

- K次选最小
- $T(n) = \Theta(n+(n-1)+...+(n-k+1)) = \Theta(\,(2n-k+1)k/2\,)$

  $=O(kn)$
- 如果$k=n/2$，则$T(n) = \Theta(n^2)$

选择第k小?

[1, 23, 12, 9, 30, 2, 50] k = 4

- 排序（如归并排序） T(n) = O(nlogn)
- 第k个 O(kn) vs O(nlogn)
- 能否更好?

选择第k小?

[1, 23, 12, 9, 30, 2, 50]   k = 4

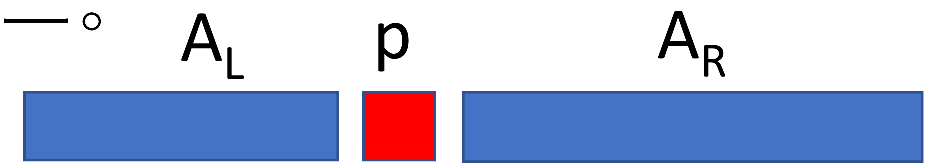
- 建堆 $O(n/2\log_2 n)$
- k次输出、调整， $klong_2 n$

$$T(n) = O(n/2\log_2 n + k\log_2 n)$$

- 时间复杂度： $O(n\log_2 n)$
- 能否达到O(n)?

选择第k小？分治法

- 采用快速排序的思想，用一个基准元素将序列一分为二。



if len(A_L)==k-1 : return p

else if len(A_L)<k-1:

    select(A_R, k-len(A_L)-1)

else:

    select(A_L, k)

$k = 4$

[12, 23, 1, 9, 30, 2, 50]

[2, 9, 1] 12 [30, 23, 50]

if len(A_L)==k-1 : return p
else if len(A_L)<k-1:
    select(A_R, k-len(A_L)-1)
else:
    select(A_L, k)

k = 3

[12, 23, 1, 9, 30, 2, 50]

[2, 9, 1] 12 [30, 23, 50]

if $\mathrm{len}(A_L) == k-1$: return p

else if $\mathrm{len}(A_L) < k-1$:

    select(A_R, k-len(A_L)-1)

else:

    select(A_L, k)

[2, 9, 1]

k = 3

[1] 2 [9]

```
if len(A_L)==k-1 : return p
else if len(A_L)<k-1:
    select(A_R, k-len(A_L)-1)
else:
    select(A_L, k)
```

```
[9]
```

k = 1

基情况： len(A)<=1

k = 5

[12, 23, 1, 9, 30, 2, 50]

[2, 9, 1] 12 [30, 23, 50]

if len(A_L)==k-1 : return p

else if len(A_L)<k-1:

select(A_R, k-len(A_L)-1)

else:

select(A_L, k)

```
Select(A, k):
    if len(A) <= 50:
        A = MergeSort(A)
        return A[k-1]
    L, pivot, R = Partition(A)

    if len(L) == k-1:
        return A[pivot]
    else if len(L) > k-1:
        return Select(L, k)
    else if len(L) < k-1:
        return Select(R, k – len(L) – 1)
```

时间复杂度分析

一次划分

$$T(n) = \begin{cases} T(\mathbf{len(L)}) + O(n) & \mathbf{len(L)} > k - 1 \\ T(\mathbf{len(R)}) + O(n) & \mathbf{len(L)} < k - 1 \\ O(n) & \mathbf{len(L)} = k - 1 \end{cases}$$

如果len(L) = len(R) = n/2

$$T(n) = T\left(\frac{n}{2}\right) + O(n \quad)$$

- 即a=1,b=2,d=1，从而：
- $T(n) \leq O\left(n^d\right) = O(n)$

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + O\left(n^d\right)$$

$$T(n) = \begin{cases} \mathrm{O}\left(n^d \log(n)\right) & \text{if } a = b^d \\ \mathrm{O}\left(n^d\right) & \text{if } a < b^d \\ \mathrm{O}\left(n^{\log_b(a)}\right) & \text{if } a > b^d \end{cases}$$

如果len(L) = 1, len(R) = n-1

$$T(n) = T(n-1) + O(n \quad)$$

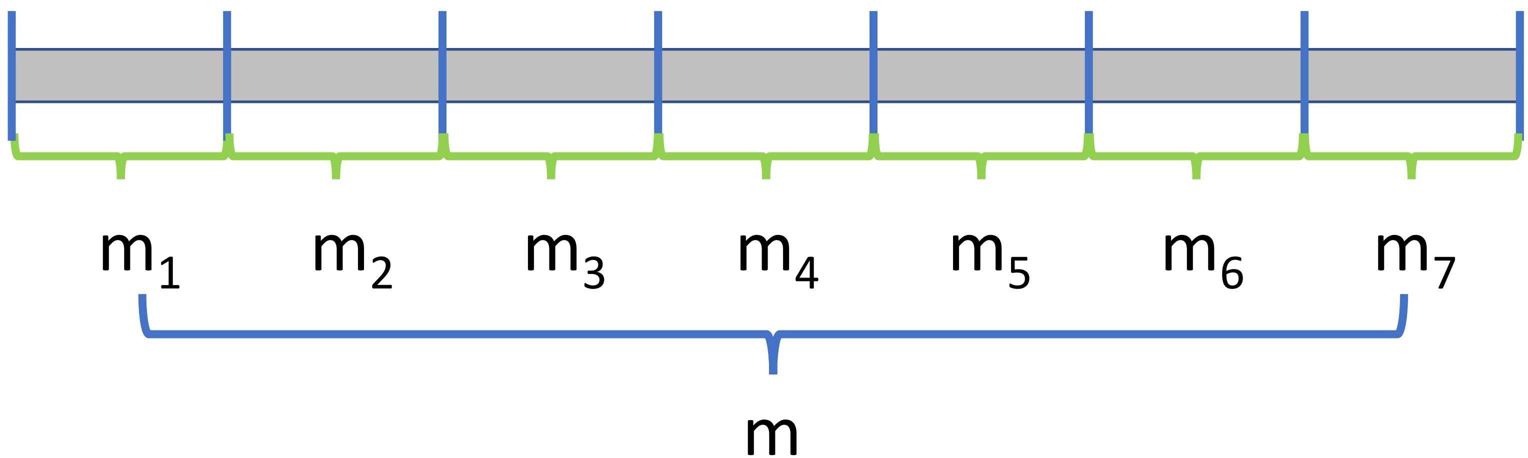
- $T(n) = \Theta(n^2)$

- 效率取决于是否均衡分割，即取决于基准元素的选取
- 随机选择基准元素，仍不可避免O(n^2)
- 如果选择中位数作为基准，就能均衡分割成相等长度的2个子序列

- 如何找中位数？是一个“鸡生蛋、蛋生鸡”的问题？
- 准确的中位数找不到，可以找接近中位数的。

median-of-medians算法确定第k小元素

- 将序列5个一组，分成g=⌈ *n/5* ⌉组
- 对每组排序，确定每组的中值m_j.
- 使用median-of-medians算法(Select(C,g/2))递归地确定中值的中值m 。
- 用m为基准元素对原序列3-分割



Select(A,k)

将n个元素分成$\lceil n/5 \rceil$组,

确定每组的中值m_j（排序找中值）

用Select寻找中值中的中值m

用m将数组划分为3部分：A_L, p, A_R

if len(A_L)==k-1: return p

else if (A_L)<k-1: return Select(A_R, k-len(A_L)-1)

else: return Select(A_L, k)

时间复杂度

- 一共有$g=\lceil n/5 \rceil$组, 有$\lceil g/2 \rceil-1$组的中位数m_i比m小。
- 每组里至少3个元素比m小（中值及小于它的2个数）。
- 考虑到有一组可能不足5个元素。可不考虑这个组。
- 因此，比m小的元素至少有： $3(\lceil g/2 \rceil-2)$
- 比m大的元素至多有：

  $n - 3(\lceil g/2 \rceil-2) = n-3n/10+6 = 7n/10+6$

时间复杂度

- 比m大的元素至多有:

  $$n - 3(\lceil g/2 \rceil - 2) = n - 3n/10 + 6 = 7n/10 + 6$$

- 同理，比m小的元素也至多有7n/10+6。
- 因此:

  选择中值的中值　　3-分割

  $$T(n) \le T(\lceil n/5 \rceil) + T(7n/10+6) + 11*n/5 + n$$

  子问题　　每组的中值

- 数学归纳法可证明:

  $$T(n) \le cn$$

应用

- 可以用median-of-median算法为快速排序选择基准元素，从而保证快速排序的最坏时间复杂度为O(nlogn)。

C++代码实现：

```
template <typename T>
T findmean(T a[],int n=5){
   sort(a, a+n);
   return a[n/2];
}

template <typename T>
int partition(T a[], int l, int r, T x) { ...}

template <typename T>
T select_k(T a[], int l, int r,int k) { ...}
```

```
template <typename T>
T select_k(T a[], int l, int r,int k) {
    if(k<=0 | |k>r-l+1) throw “k error”;
    if(r-l<50) {   sort(a+l, a+r); return a[k-1];   }
    int n = r-l+1;
    T medians[(n+4)/5];  //所有分组的中值
    int g = n/5;
    for (auto i=0; i<g; i++)
        medians[i] = findMedian(arr+l+i*5, 5);
     if (n%5!=0)  //最后一个分组
        medians[g] = findMedian(arr+l+ g*5, n%5);
     g = (n+4)/5;
     int m = (g== 1)? medians[0]: select_k(medians, 0, g-1, g/2);
     int pos = partition(a, l, r, m);
     if (pos-l == k-1)           return a[pos];
     else if (pos-l > k-1)
         return select_k(a, l, pos-1, k);
      else return select_k(a, pos+1, r, k-pos+l-1);
}
```

最大子段和

Maximum Subarray

Youtube频道： **hwdong**

博客: **hwdong-net**.github.io

最大和数组(最大子段和)

- 最大和子数组(最大子段和)问题是寻找一个具有最大和的连续子数组。
- 如一维子数组A[1..n]，寻找索引i和j (1≤i≤j≤n)，使得：

$$\sum_{x=i}^{j} A[x]$$

- 最大

最大子段和

- 给定n个数(可以是负数)的序列，求该序列连续的子段和的最大值。

  输入： nums = [-2,1,-3,4,-1,2,1,-5,4]
  输出： 6
  解释： 连续子数组 [4,-1,2,1] 的和最大， 为 6 。

最大子段和

- 最大子阵列问题是由Ulf Grenander在1977年提出的，是对数字化图像中的图案进行最大似然估计的一个简化模型。
- 最大子阵列问题出现在许多领域，如基因组序列分析和计算机视觉。
- 基因组序列分析采用最大子段和算法来识别蛋白质序列中的重要生物片段。这些问题包括保守片段、富含GC的区域、串联重复、低复杂度过滤器、DNA结合域和高电荷区域。
- 在计算机视觉中，最大子段和算法被用于位图图像上，以检测图像中最亮的区域。

1	2	3	4	5	6	7	8	9
-2	1	-3	4	-1	2	1	-5	4

line	i	x[i]	cur	cu+x[i]	best	
2					0	
3			0			
4	1	-2		-2		
5			0			c
6					0	b
4	2	1		1		
5			1			CCC
6					1	BBB
4	3	-3		-2		
5			0			c
6					1	BBB
4	4	4		4		
5			4			CCC
6					4	BBB
4	5	-1		3		
5			3			CCCCCCC
6					4	BBB
4	6	2		5		
5			5			CCCCCCCCCCC
6					5	BBBBBBBBBBB
4	7	1		6		
5			6			CCCCCCCCCCCCCCC
6					6	BBBBBBBBBBBBBBB
4	8	-5		1		
5			1			CCCCCCCCCCCCCCCCCCC
6					6	BBBBBBBBBBBBBBB
4	9	4		5		
5			5			CCCCCCCCCCCCCCCCCCCCCCC
6					6	BBBBBBBBBBBBBBB
4						
7						BBBBBBBBBBBBBBB

- 在计算机视觉中，最大子段和算法被用于位图图像上，以检测图像中最亮的区域。

最大子段和：蛮力法

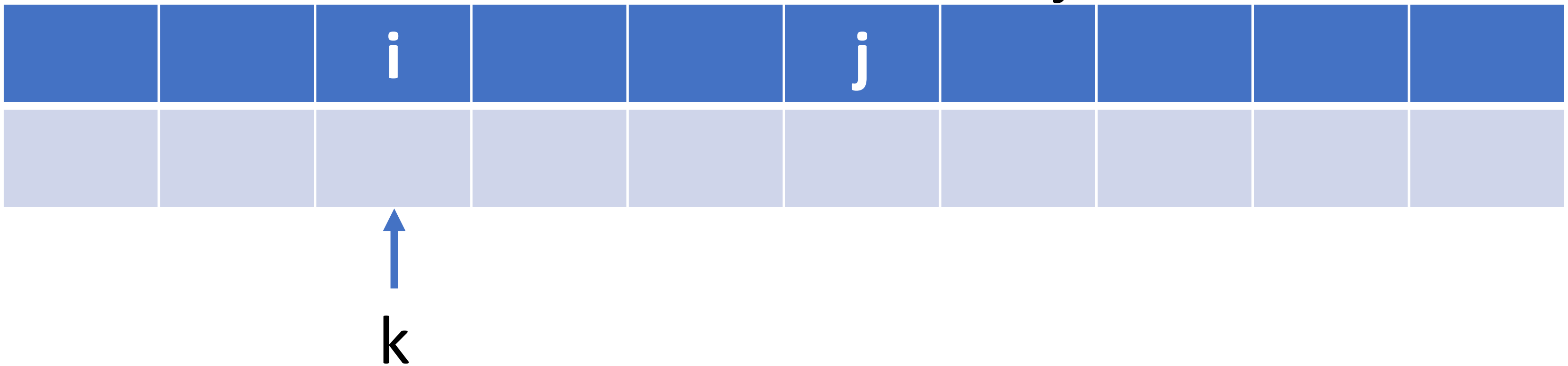
- 对所有的区间[i,j]，计算子段和$a_i+\ldots+a_j$。

```
max_sum=-∞
for i=1 to n:
  for j= i to n:
    sum = 0
    for k=i to j:
      sum += a[k];
    if (sum > max_sum)
      max_sum = sum;
  }
```

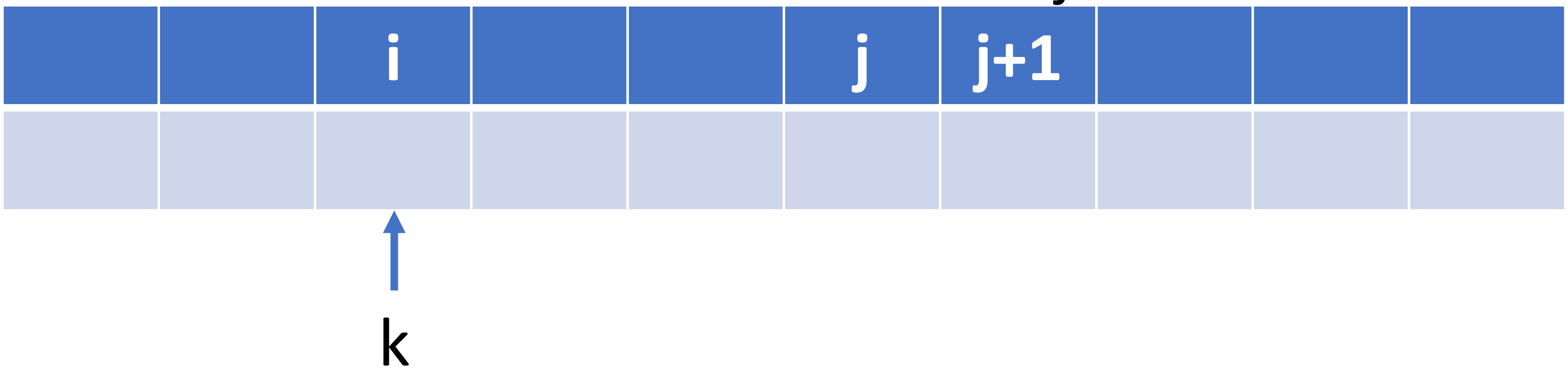
$$T(n) = (1+2+\ldots+n)+(1+2+\ldots+n-1)+\ldots+(1+2)+1 = \Theta(n^3)$$

	1	2	3	4	5
1	**1**	**2**	**3**	**4**	**5**
2		1	2	3	4
3			1	2	3
4				1	2
5					1

$$\text{sum} = a_i + \ldots + a_j$$



$$sum = a_i + \ldots + a_j$$



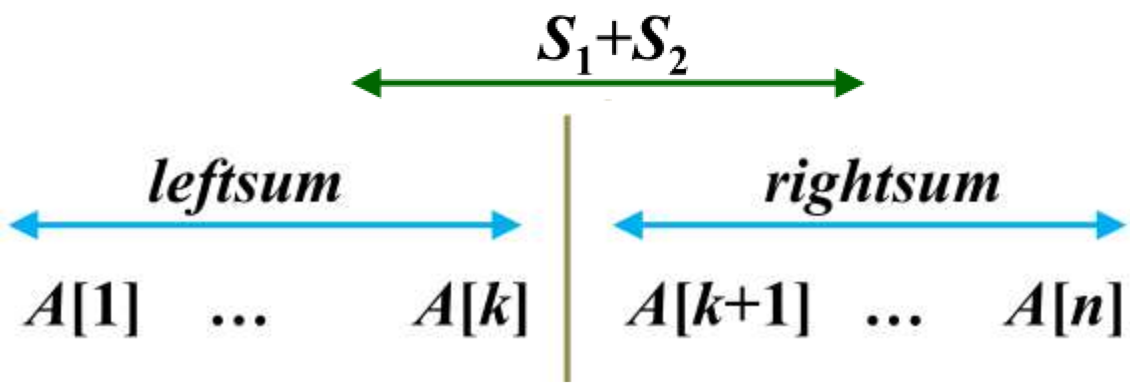

```
max_sum=-∞
for i=1 to n:
    sum = a[i]
    if sum>max_sum:   max_sum = sum
    for j = i+1 to n:
        sum += a[j]
        if sum > max_sum:
            max_sum = sum
```

$$sum = a_i + \ldots + a_j + a_{j+1}$$

$\Theta(n^2)$

分治法

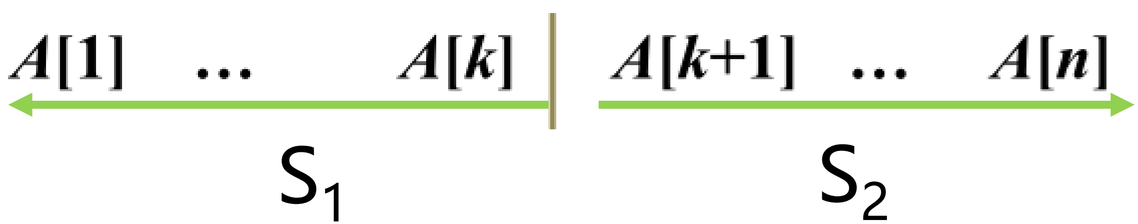
- 将序列分为一分为二
- 递归计算左、右子序列的最大子段和
- 跨越2子序列的最大子段和=S_1+S_2.
- max_sum = max(leftsum, rightsum, S_1+S_2)



```
MaxSubSum(A, left, right)
    if |A|==1: return A_1
    mid = (left+right)/2
    leftsum = MaxSubSum(A,left,mid)
    rightsum = MaxSubSum(A,mid+1,right)
    S1 = Aleft(A,left,mid)
    S2 = Aright(A,mid+1,right)
    return max(leftsum.rightsum,S1+S2)
```

跨边界的和

- 以mid为中心分别向两边计算和。
- 从mid向左出发，每次扩张一步，并且记录当前的值S_1，如果当前的和比上次的和大，就更新S_1，一直向左扩张到位置 Left。
- 从 mid+1向右出发，每次扩张一步，计算当前的和 为S_2，如果当前的值比上次的和大，那么，就更新S_2的值，一直向右扩张到位置Right。



$\Theta(n)$

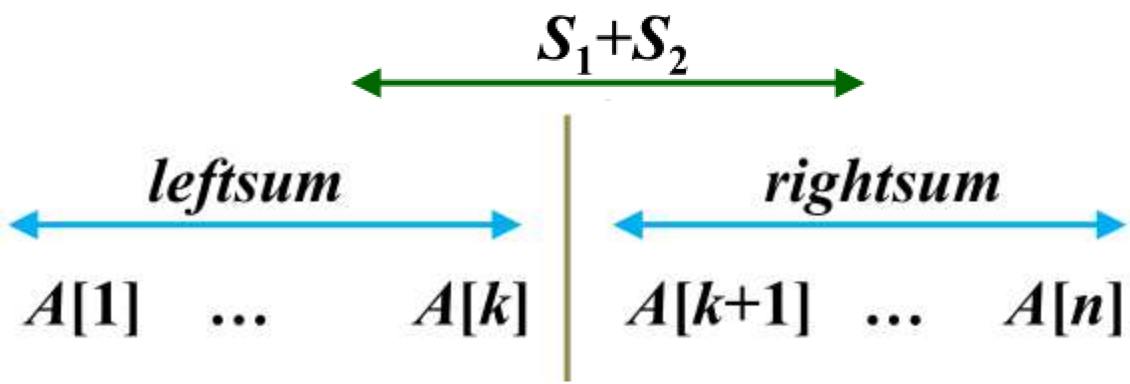
```
Aleft(A,left,mid):
   max_sum = -infinity
   sum=0
   for i = mid to left:
         sum= sum+A[i]
         if sum>=max_sum:
                  max_sum = sum
    return max_sum
```

时间复杂度

$T(n) = 2T(n/2)+n$

$T(1)=0$

$\Rightarrow$ $T(n) = \Theta(n\log n)$



有没有更好的算法?

- （动态规划）Kadane's algorithm
- 动态规划也是分治递归，通过存储子问题的解提高算法效率

最近点对

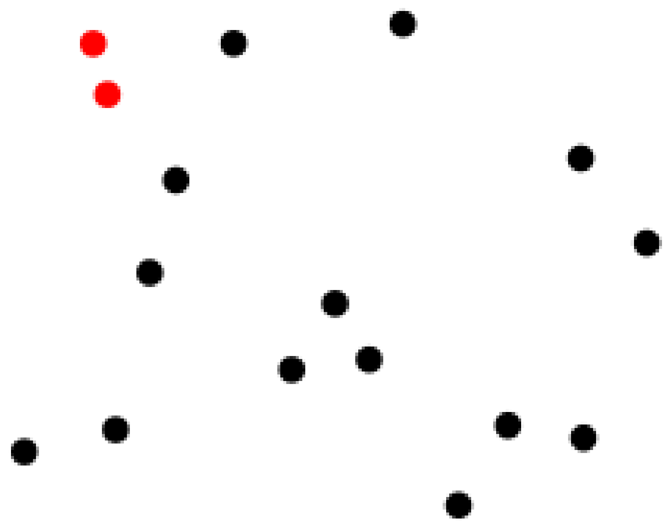
closest pair of points

Youtube频道： **hwdong**

博客: **hwdong-net**.github.io

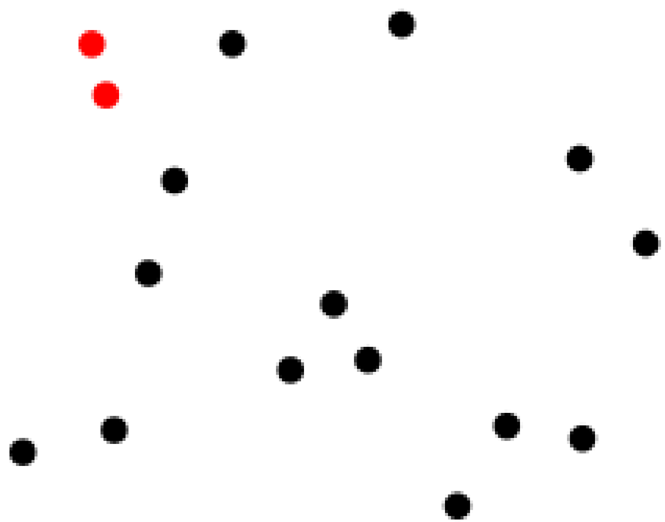
最近点对

- 是一个计算几何学的问题：给定欧氏空间中的n个点，找到它们之间距离最小的一对点。



最近点对

- 在计算机图形学、计算机视觉、地理信息系统、分子建模、空中交通管制等有很多应用



最近点对

- n个点的集合中，找出距离最近的两个点。
- 蛮力法：$\Theta(n^2)$
- 一维情形：
- 可以将这些点按坐标值排序，然后检查相邻2个点的距离。$\Theta(n\log_2 n)$
- 有没有更好的方法呢？

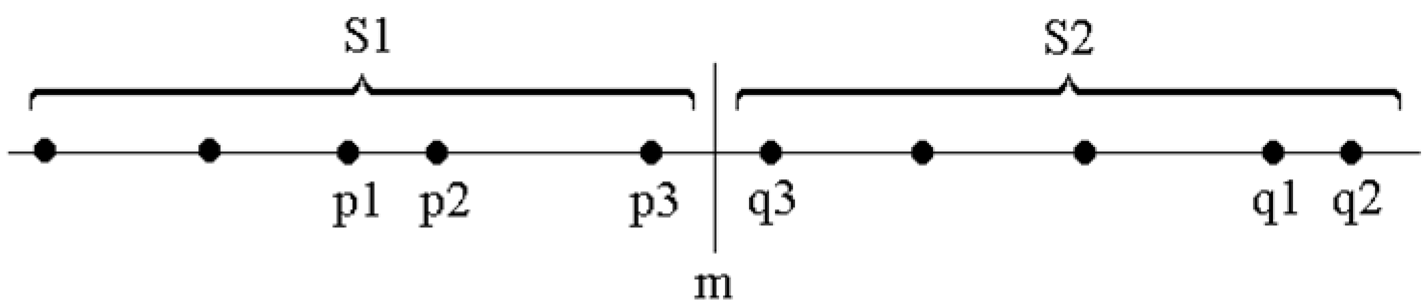
分治法

- 用中值点m将点集一分为二
- 递归求S_1、 S_2的最近点对(p_1,p_2)、(q_1,q_2)。
- 求跨越m的最小点对(p_3,q_3)，如何找？

$p_3 \in (m-d,m]$，$q_3 \in (m,m+d]$

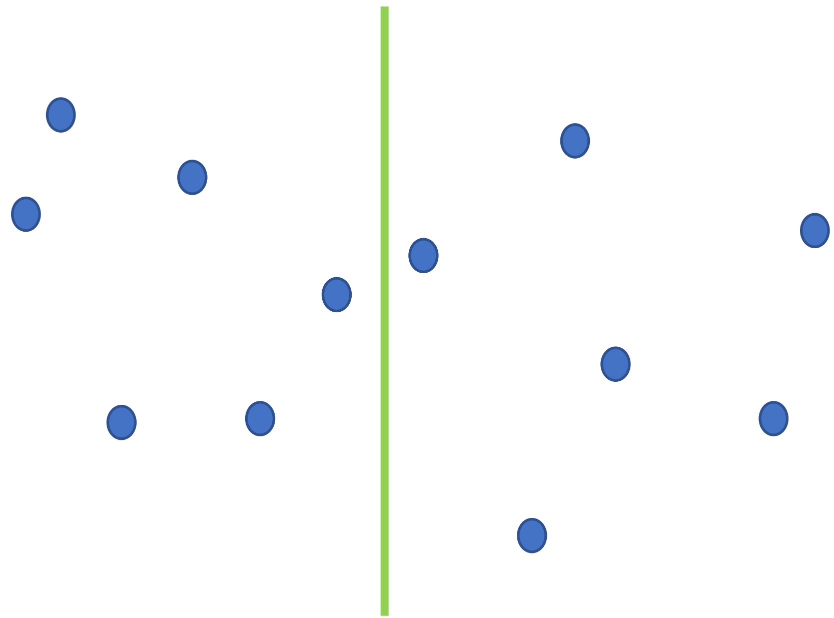
$d=\min\{|p_1-p_2|,|q_1-q_2|\}$

- **p_3**是S_1的最大坐标点， **q_3**是S_2最小坐标点



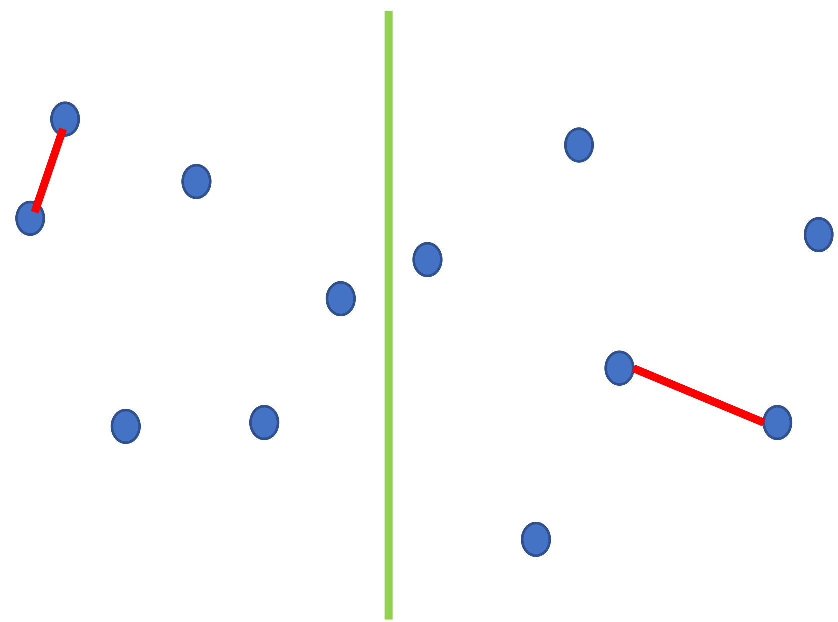
平面点集的最近点对

- 用一根线将平面一分为二，左右两侧各一半点



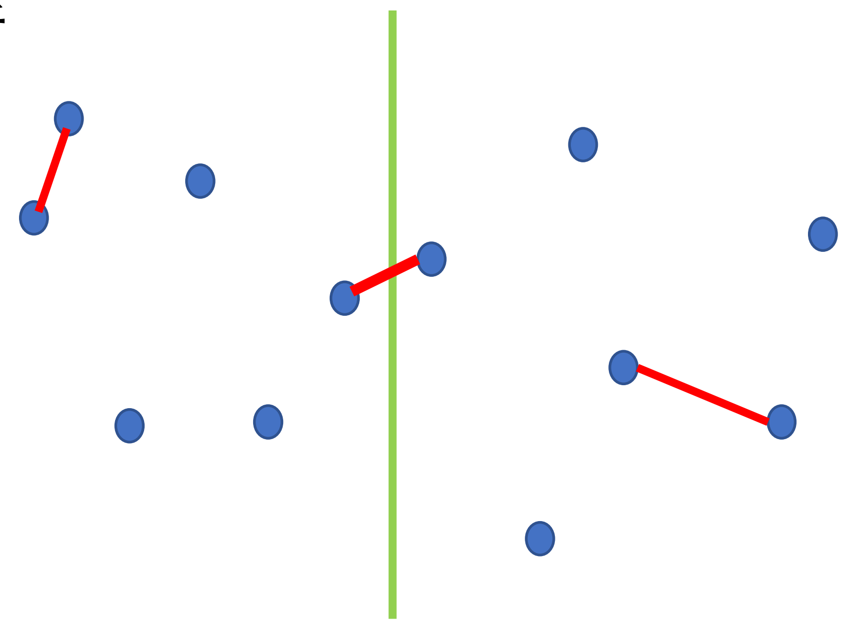
平面点集的最近点对

- 用一根线将平面一分为二，左右两侧各一半点
- 递归对每个子集求最近点对(p_1,p_2)、(q_1,q_2)



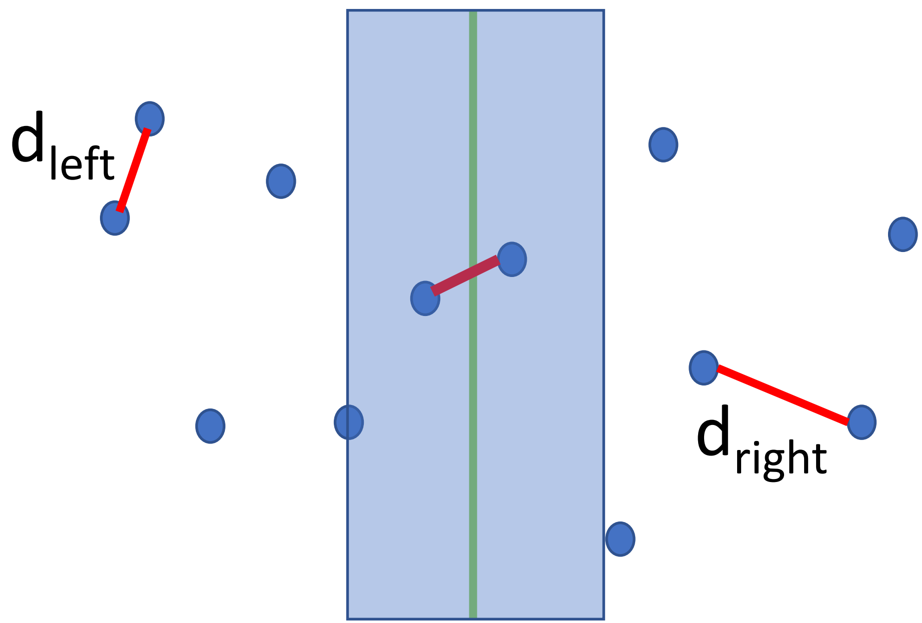
平面点集的最近点对

- 用一根线将平面一分为二，左右两侧各一半点
- 递归对每个子集求最近点对(p_1,p_2)、(q_1,q_2)
- 确定跨越子集的最近点对(p_3,q_3)
- 返回3个点对的最短者



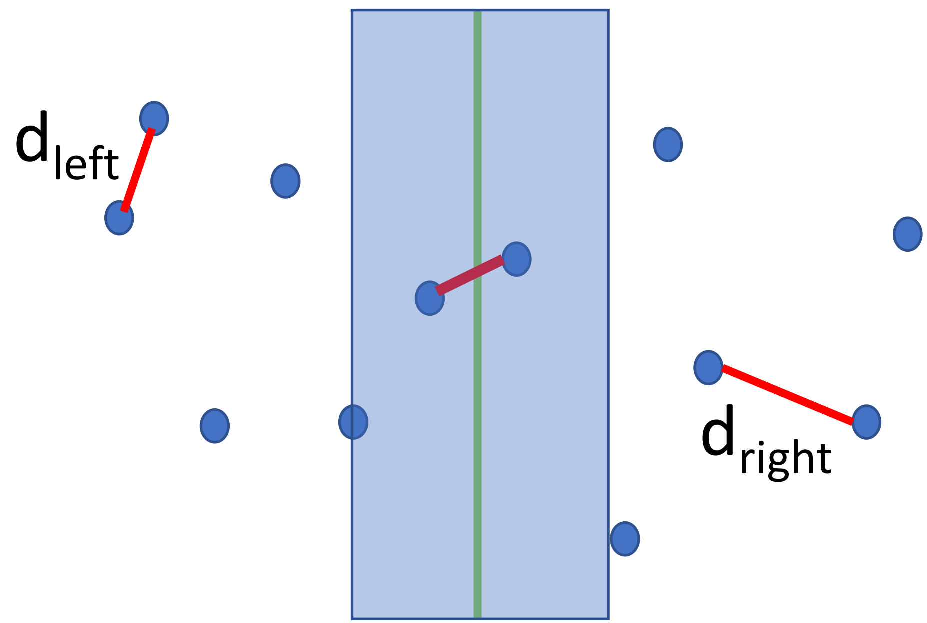
平面点集的最近点对

- 如何求确定跨越子集的最近点对(p_3,q_3)?
- 设$d = \min(d_{left},d_{right})$
- 若$dist(p_i,q_j)<d$，则(p_i,q_j)必然在直线m的d的带状范围里

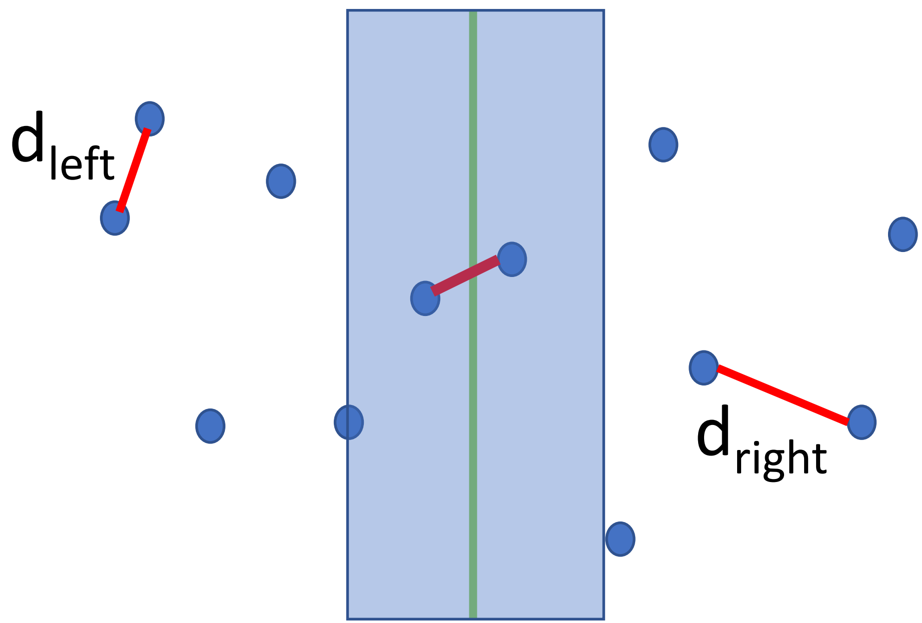


平面点集的最近点对

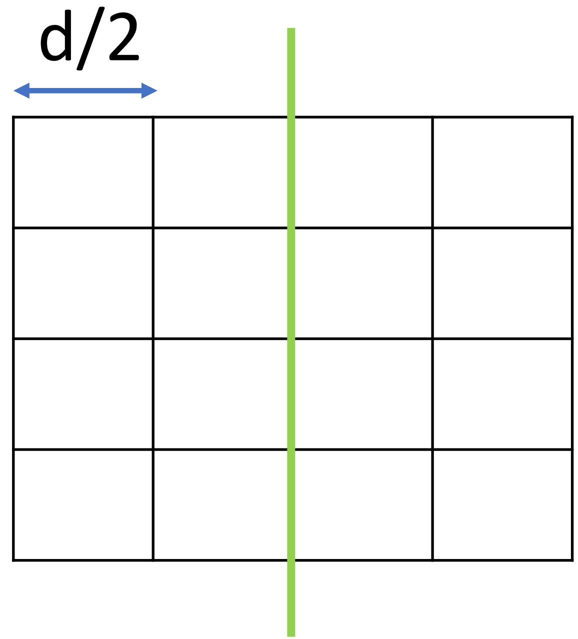
- 设S_y是在带状区域里按照y坐标排序的所有点。 S_y也可能是整个点集.
- 怎么办?



- 定理：设S_y的点为$(s_1, s_2, s_3,\dots, s_m)$，如果$dist(s_i, s_j)<d$，则 $j-i<11$

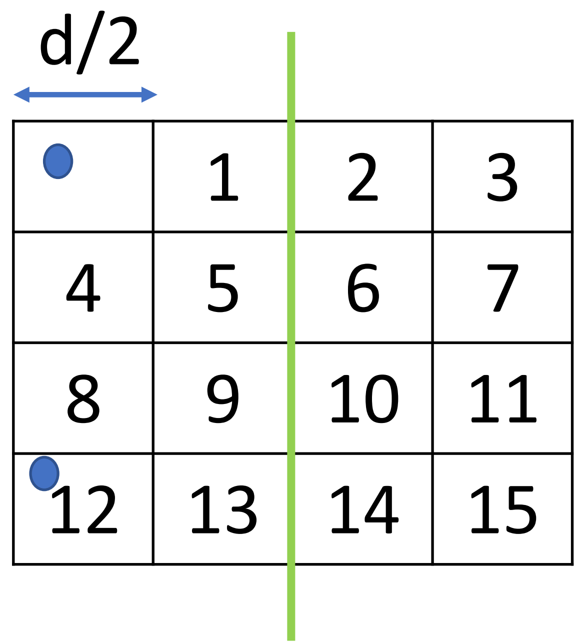


- 将带状区域分割成d/2的正方形块
- 每个正方形块中至多只有1个点



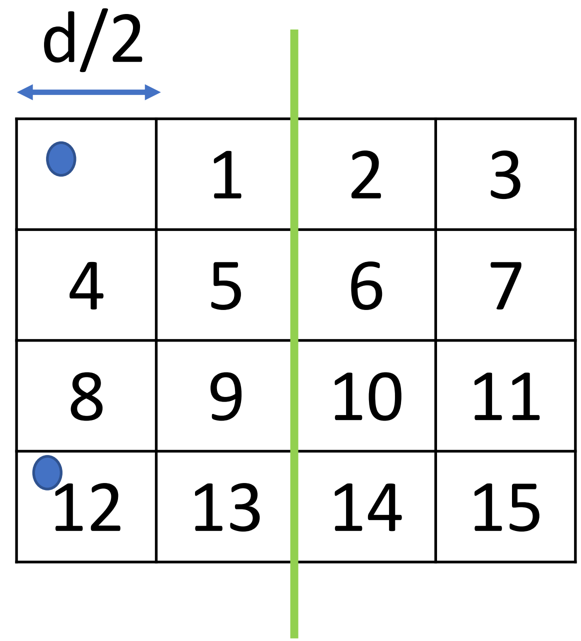
假如2点的$j-i \geq 12$，则它们必然超过2行，

从而$dist(s_i, s_j) \geq 2d/2=d$

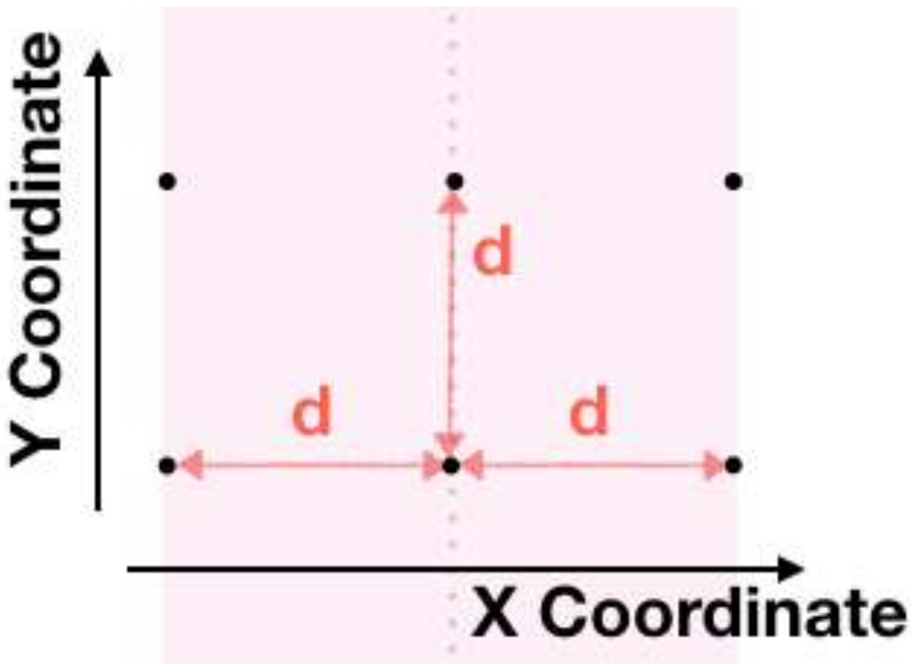


因此，每个点最多只需要和12个点计算距离及比较，线性时间。

```
for i = 1 to |S_y|:
    for k = 1 to 12:
        d = min(dist(s_i, s_{i+k}),d)
return d
```



- 每个点最多和8个点距离不超过d.



```
ClosestPair(P):
    if|P|==1: return INFINITY
    if |P|==2: return dist(P[1],P[2])
    计算分割中位线L                              O(nlog2n)
    d1 = ClosestPair(Pleft):                    } 2T(n/2)
    d2 = ClosestPair(Pright):                   }
    d= min(d1,d2)
    保留距离L不超过d的点                          O(n)
    Sy = 对剩余点集按y坐标排序                    O(nlog2n)
    for i = 1 to |Sy|:                          }
        for k = 1 to 12:                        } O(n)
            d = min(dist(si, si+k),d)           }
    return d
```

Notation: $O(n\log_2 n)$, $2T(n/2)$, $O(n)$, $|S_y|$, $\mathrm{dist}(s_i, s_{i+k})$

时间复杂度

- $T(n)<2T(n/2)+O(n\log n)$
- 总的时间: $T(n) = O(n\log^2 n)$
- 能否达到$T(n) = O(n\log n)$?
- 在递归算法里不每次排序，而是一次性排序好

ClosestPair(Px,Py):

if|Px|==1: return INFINITY

if |Px|==2: return dist(Px[1],Px[2])

d1 = ClosestPair(LeftHalf(Px,Py))

d2 = ClosestPair(RightHalf(Px,Py))

2T(n/2)

d= min(d1,d2)

Sy =保留距离L不超过d的点 O(n)

for i = 1 to $|S_y|$:

for k = 1 to 12:

d = min(dist(s_i, s_{i+k}),d)

O(n)

return d

时间复杂度

- 分割点集的次数： $O(\log n)$
- 合并： $O(n)$
- 递归求解子问题： $T(n)<2T(n/2)+cn$
- 总的时间: $T(n) = O(n\log n)$

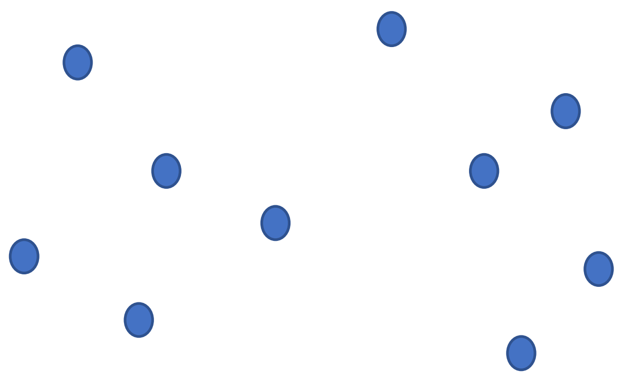
凸包问题

Youtube频道： **hwdong**

博客： **hwdong-net**.github.io

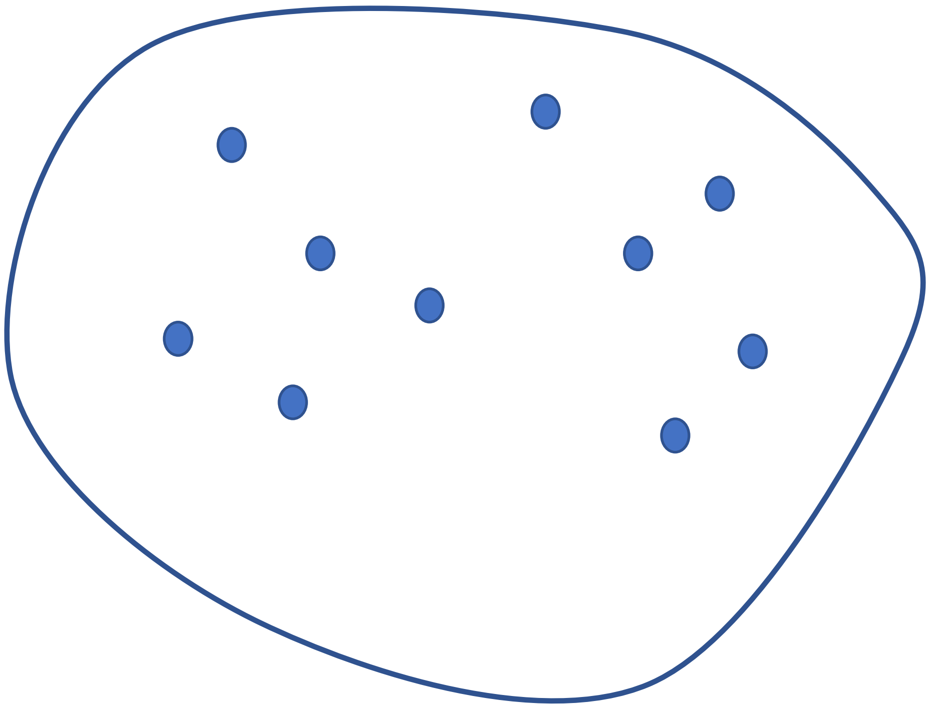
凸包问题

- 点集凸包是包围该点集的最小凸多边形



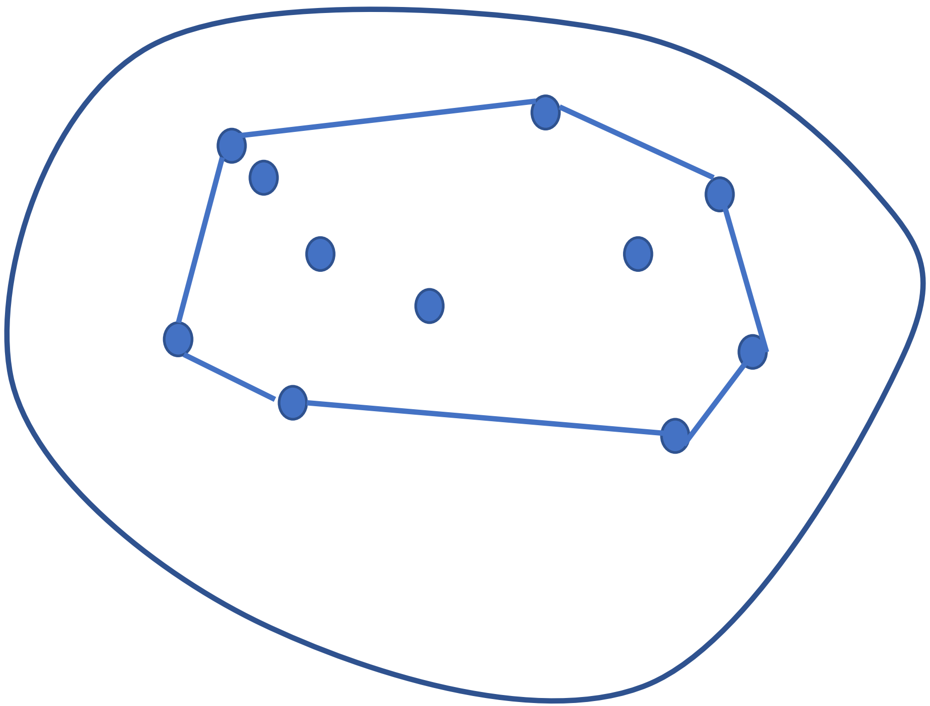
凸包问题

- 点集凸包是包围该点集的最小凸多边形



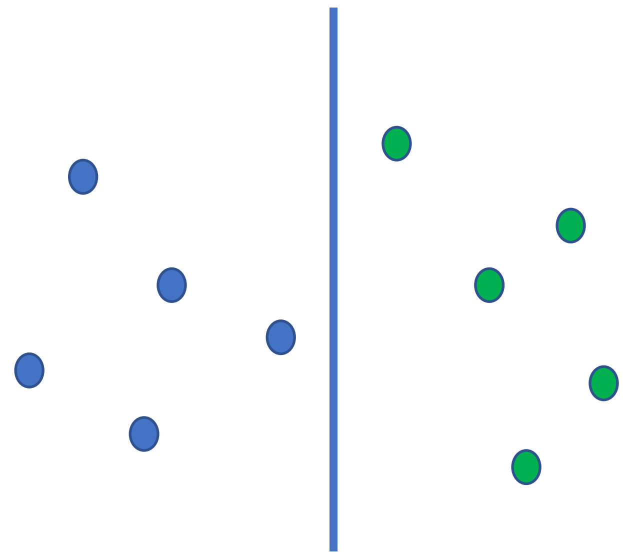
凸包问题

- 点集凸包是包围该点集的最小凸多边形



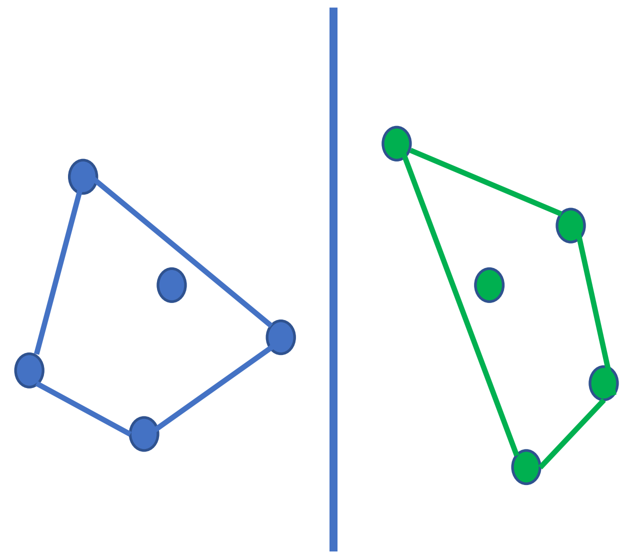
凸包问题：分治法

- 用中值x将点集分为左右2部分。



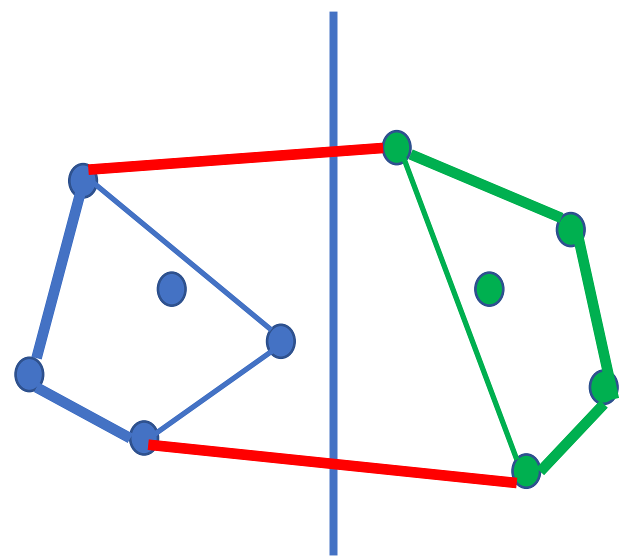
凸包问题：分治法

- 用中值x将点集分为左右2部分。
- 递归求左右点集的凸包



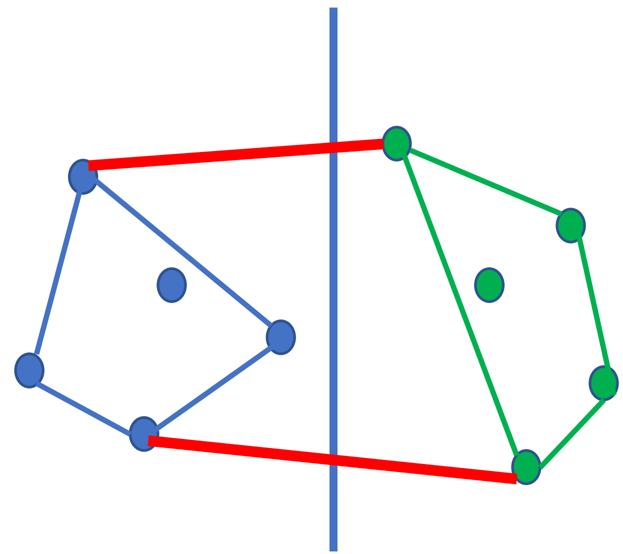
凸包问题：分治法

- 用中值x将点集分为左右2部分。
- 递归求左右点集的凸包。
- 合并左右2个凸包



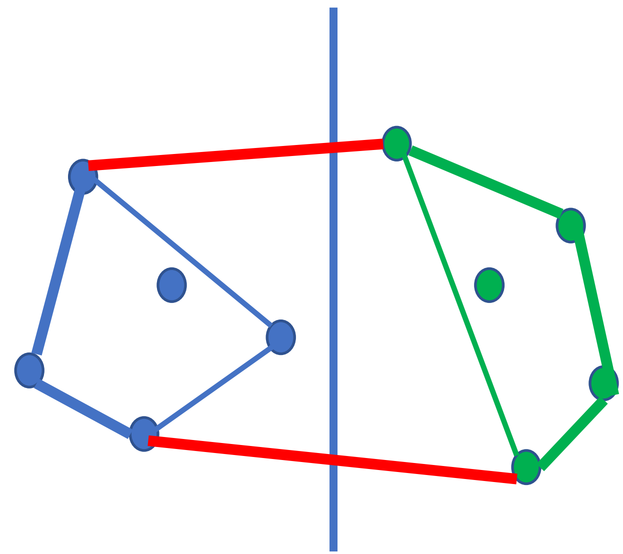
合并凸包

- 找到最上、最下的切线



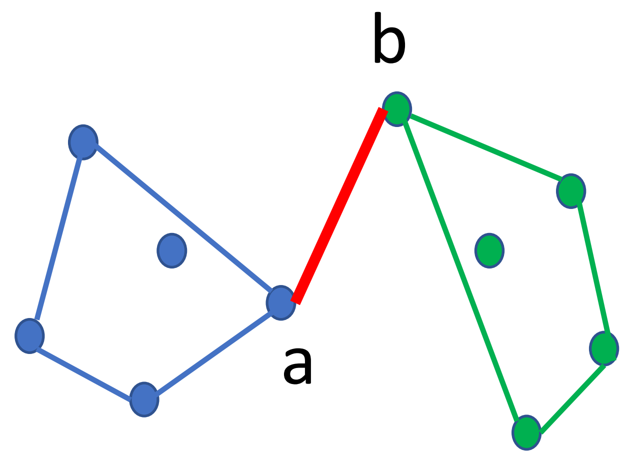
合并凸包

- 找到最上、最下的切线
- 根据切线合并凸包



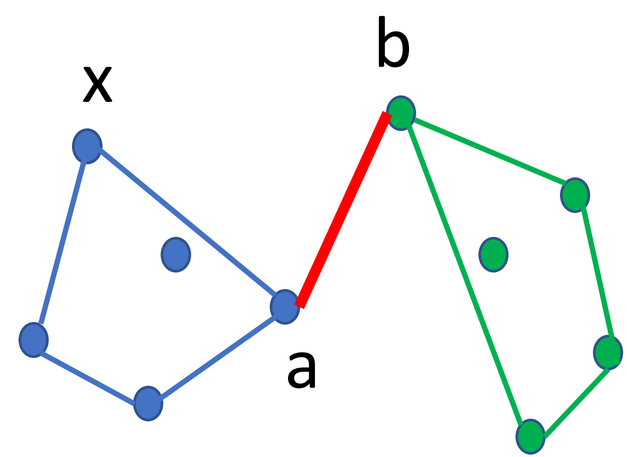
寻找最上切边

- a,b分别是凸包A,B的最右、最左顶点



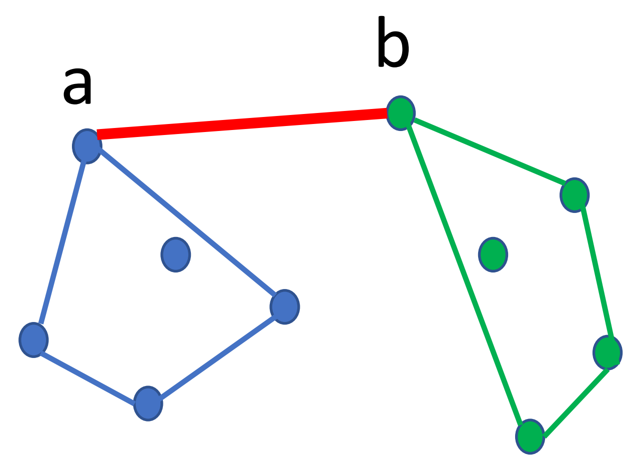
寻找最上切边

- a,b分别是凸包A,B的最右、最左顶点
- 循环：
-     if a的逆时针邻居x如果在ab上方，则a = x
-



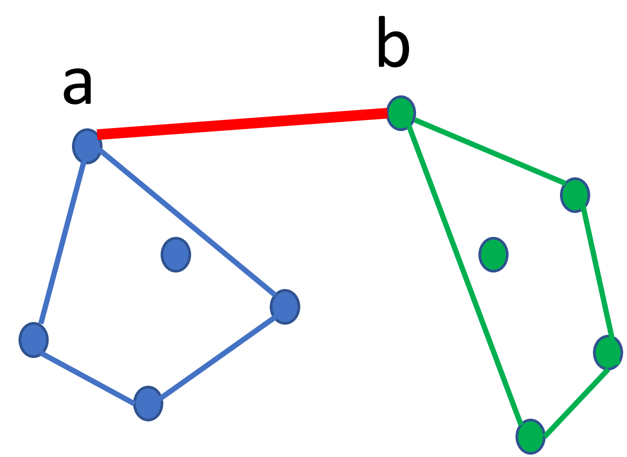
寻找最上切边

- a,b分别是凸包A,B的最右、最左顶点
- 循环：
-     if a的逆时针邻居x如果在ab上方，则a = x
-



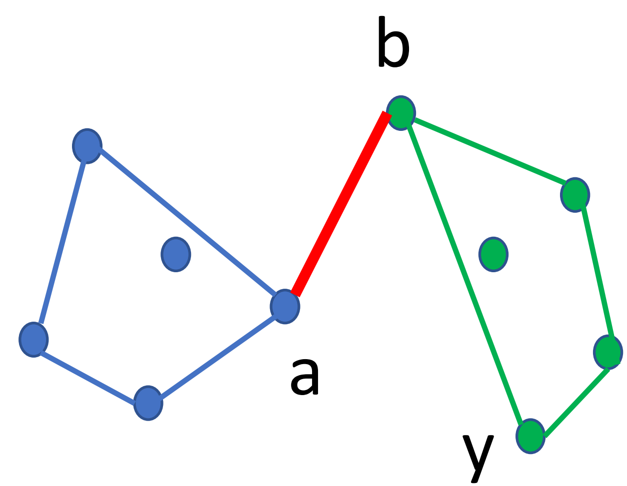
寻找最上切边

- a,b分别是凸包A,B的最右、最左顶点
- 循环：
-     if a的逆时针邻居x如果在ab上方，则a = x
-     else if b的顺时针邻居y如果在ab上方，则 b=y
-     else 退出循环



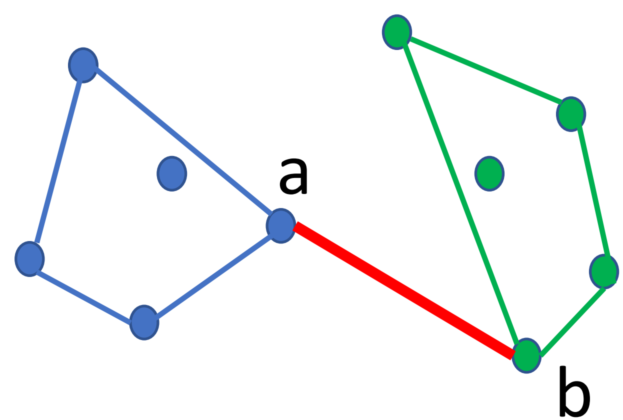
寻找最下切边

- a,b分别是凸包A,B的最右、最左顶点
- 循环：
- 	if b的逆时针邻居y如果在ab下方，则 b=y
-



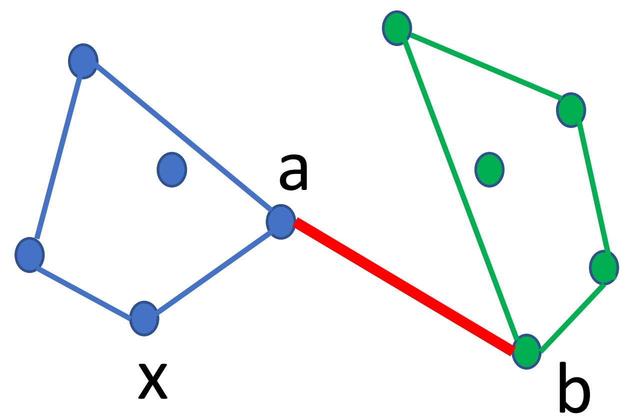
寻找最下切边

- a,b分别是凸包A,B的最右、最左顶点
- 循环:
-     if b的逆时针邻居y如果在ab下方，则 b=y
-



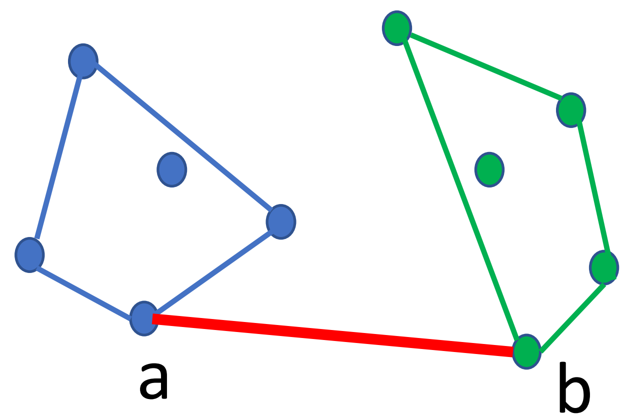
寻找最下切边

- a,b分别是凸包A,B的最右、最左顶点
- 循环：
- if b的逆时针邻居y如果在ab下方，则 b=y
- else if a的顺时针邻居x如果在ab下方，则 a = x
-



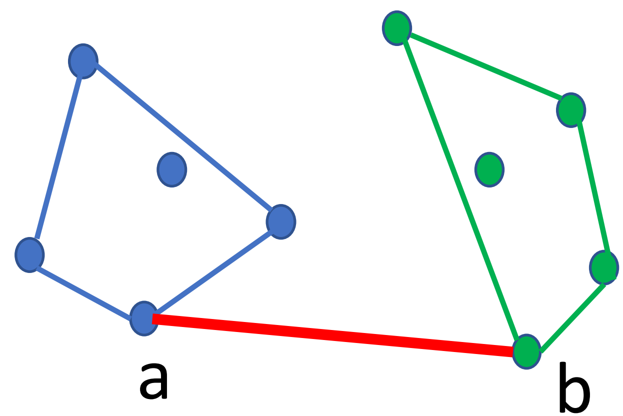
寻找最下切边

- a,b分别是凸包A,B的最右、最左顶点
- 循环：
-     if b的逆时针邻居y如果在ab下方，则 b=y
-     else if a的顺时针邻居x如果在ab下方，则 a = x
-



寻找最下切边

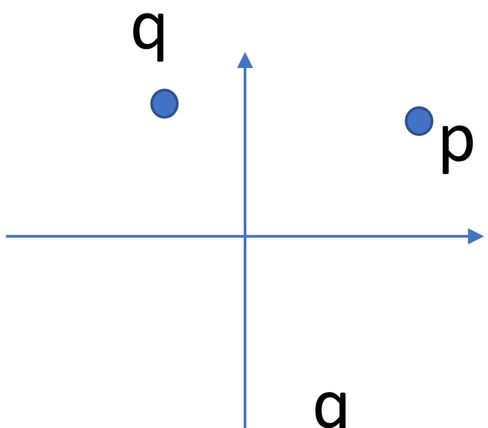
- a,b分别是凸包A,B的最右、最左顶点
- 循环：
- 　if b的逆时针邻居y如果在ab下方，则 b=y
- 　else if a的顺时针邻居x如果在ab下方，则 a = x
- 　else 退出循环



逆时针排列多边形顶点

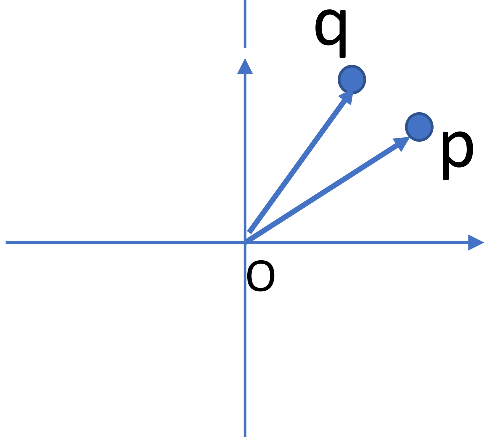
- 对顶点排序
- 比较顶点大小：

  1）在不同象限



  2）叉积

  OP x OQ > 0?



时间复杂度

- 用中值x将点集分为左右2部分。 $O(n\log n)$
- 递归求左右点集的凸包。 $2T(n/2)$
- 合并左右2个凸包 $O(n)$

$$T(n) = 2T(n/2)+O(n\log n)+O(n)$$

先在外面排好序，再递归求解凸包

$$T(n) = 2T(n/2)+O(n) \Rightarrow T(n) = O(n\log n)$$

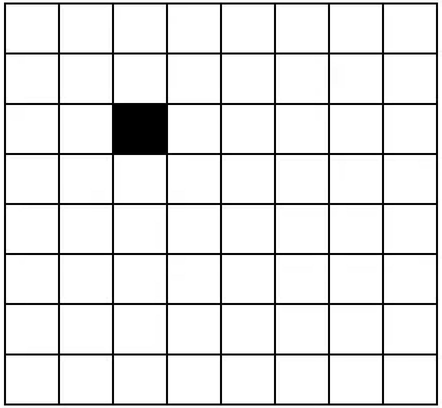
棋盘覆盖

Youtube频道： **hwdong**

博客： **hwdong-net**.github.io

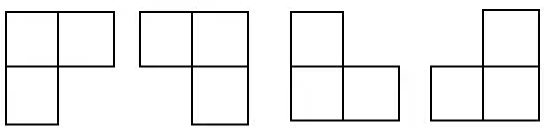
棋盘覆盖

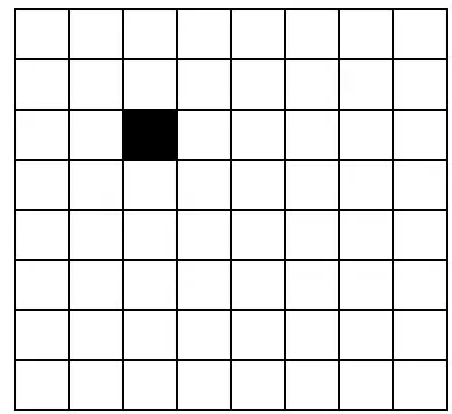
- 在一个 2^k x 2^k 个方格组成的棋盘中，恰有一个方格与其他方格不同，称该方格为一特殊方格，且称该棋盘为一**特殊棋盘**。



棋盘覆盖

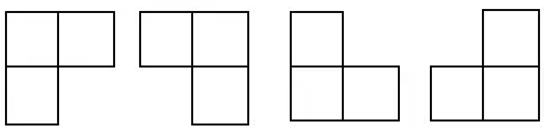
- 在棋盘覆盖问题中，要用图示的4种不同形态的L型骨牌覆盖给定的特殊棋盘上除特殊方格以外的所有方格，且任何2个L型骨牌不得重叠覆盖。

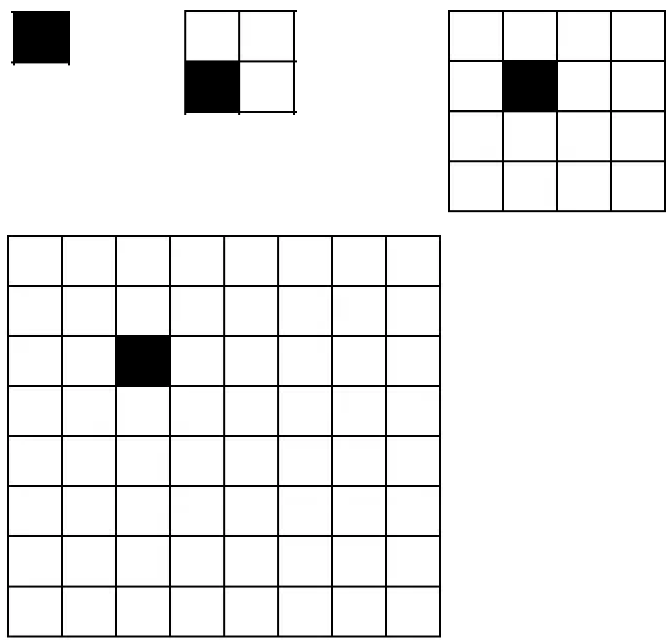




棋盘覆盖

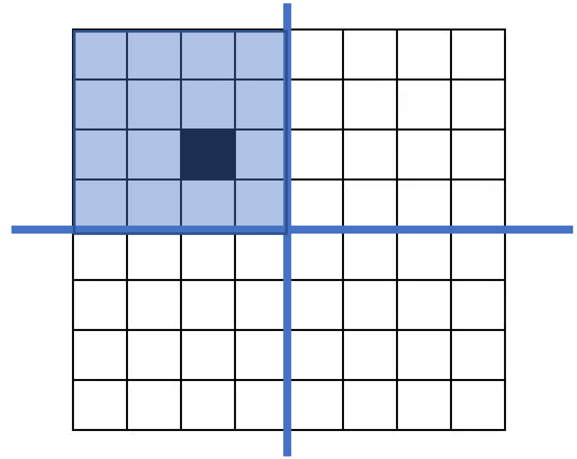
- 在棋盘覆盖问题中，要用图示的4种不同形态的L型骨牌覆盖给定的特殊棋盘上除特殊方格以外的所有方格，且任何2个L型骨牌不得重叠覆盖。





分治法

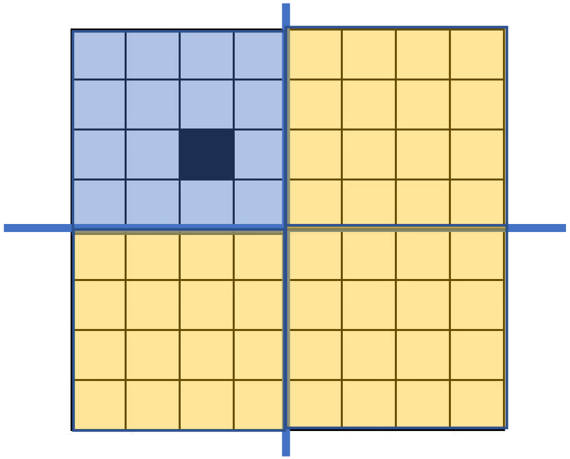
- 分成4个子问题



是子问题吗？

分治法

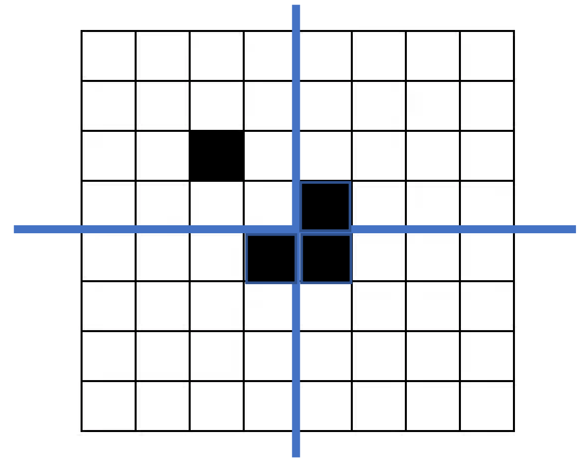
- 分成4个子问题



是子问题吗？

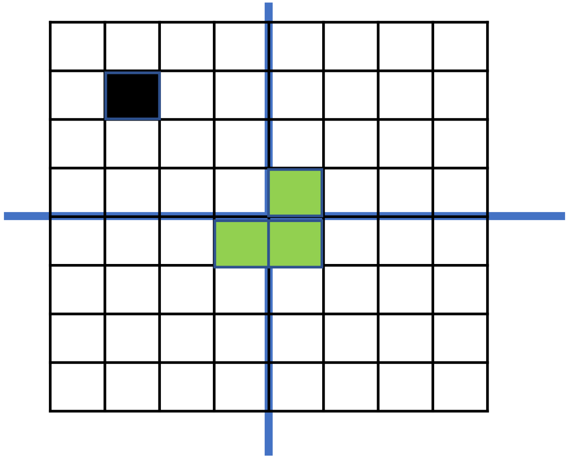
分治法

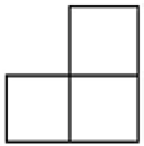
- 分成4个子问题



现在都是子
问题了

```
chessBoard(board, L, R, T, B, x, y)
    if(R<=L or B<=T) return;
    x_m = (L+R)/2; y_m = = (T+B)/2;
    if x<=x_m and y<=y_m:
        覆盖左L型骨
      chessBoard(board,L,x_m,T,y_m,x,y)
      chessBoard(board,L,x_m,y_m+1,B,x_m, y_m+1)
      chessBoard(board,x_m+1,R,y_m+1,B,x_m+1, y_m+1)
      chessBoard(board, x_m+1,R,T, y_m, x_m+1, y_m)
    else if  x<=x_m and y>y_m:
        …
    else if  >x_m and y>y_m:
        …
    else
        …
```





时间复杂度

$T(0) = 0$

$T(k) = 4T(k-1)+O(1)$

$\Rightarrow$ $T(k) = O(4^k)$

$n=2^k$

$T(n)=O(n^2)$

最大最小值问题

Youtube频道： **hwdong**

博客： **hwdong-net**.github.io

求最大与最小值

7 6 3 9 4 1 2 8

- 分成2个子问题：求最大值、求最小值
- 在n个元素中求最大值，共n-1次比较
- 在剩下元素中求最小值，共n-2次比较。
- 总的时间复杂度是2n-3.

- 能否改进呢?

- 改进:

7 6 3 9 4 1 2 8

max=7

min = 6

```
if a[0]>a[1]:
    max = a[0]
    min = a[1]
else:
    max = a[1]
    min = a[0]
```

- 改进:

7 6 3 9 4 1 2 8

i=2

max=7

min = 6

```
if a[i]>max:
    max = a[i]
else if a[i]<min:
    min = a[i]
```

- 改进:

7 6 3 9 4 1 2 8

↑

i=2

max=7

min = 3

```
if a[i]>max:
    max = a[i]
else if a[i]<min:
    min = a[i]
```

- 改进:

7 6 3 9 4 1 2 8

i=3

max=7

min = 3

```
if a[i]>max:
    max = a[i]
else if a[i]<min:
    min = a[i]
```

- 改进:

7 6 3 9 4 1 2 8

↑
i=3

max=9

min = 3

```
if a[i]>max:
    max = a[i]
else if a[i]<min:
    min = a[i]
```

- 改进：

7 6 3 9 4 1 2 8

i=3

$$T(n) \leq 2(n-1)+1$$

```
if n==1:
    max = min = a[0]
   return max,min
if a[0]>a[1]:
    max = a[0]
    min = a[1]
else:
    max = a[1]
    min = a[0]
for i=2 to n
   if a[i]>max:
      max = a[i]
   else if a[i]<min:
      min = a[i]
```

- 分组:

7 6 3 9 4 1 2 8 5

max = 7

min = 6

- 分组：

7 6 3 9 4 1 2 8 5

max = 7 9

min = 6 3

- 分组:

7 6 3 9 4 1 2 8 5

max = 7 9 4

min = 6 3 1

- 分组:

7 6 3 9 4 1 2 8 5

max = 7 9 4 8

min = 6 3 1 2

- 分组:

7 6 3 9 4 1 2 8 5

max = 7 9 4 8

min = 6 3 1 2

Max = 9

- 分组:

7 6 3 9 4 1 2 8 $\boxed{5}$

max = 7 9 4 8

min = $\boxed{6 \quad 3 \quad 1 \quad 2}$

Max = 9

Min = 1

$$T(n) = \lfloor n/2 \rfloor + 2(\lceil n/2 \rceil - 1) = n + \lceil n/2 \rceil - 2$$

$$= \lceil 3n/2 \rceil - 2$$

问: 需要辅助空间吗?

- 分组:

7 6 3 9 4 1 2 8

7 6

max= 7

min = 6

```
if n==1:
   min = max = a[0]
if a[0]>a[1]:
     max = a[0]
     min = a[1]
else:
     max = a[1]
     min = a[0]
```

- 分组:

7 6 3 9 4 1 2 8

7 6

7 6 3 9

max=9

min = 3

```
if a[i]>a[i+1]:
    if a[i]>max: max = a[i]
    if a[i+1]<min: min= a[i+1]
else:
    if a[i+1]>max: max = a[i+1]
    if a[i]<min: min= a[i]
```

- 分组:

7 6 3 9 4 1 2 8

7 6

7 6 3 9

7 6 3 9 4 1

max=9

min = 1

```
if a[i]>a[i+1]:
    if a[i]>max: max = a[i]
    if a[i+1]<min: min= a[i+1]
else:
    if a[i+1]>max: max = a[i+1]
    if a[i]<min: min= a[i]
```

- 分组:

7 6 3 9 4 1 2 8

7 6

7 6 3 9

7 6 3 9 4 1 2 8

max=9

min = 1

```
if a[i]>a[i+1]:
    if a[i]>max: max = a[i]
    if a[i+1]<min: min= a[i+1]
else:
    if a[i+1]>max: max = a[i+1]
    if a[i]<min: min= a[i]
```

- 分组:

7 6 3 9 4 1 2 8 11

7 6

7 6 3 9

7 6 3 9 4 1 2 8

7 6 3 9 4 1 2 8 11

max=9

min = 1

```
if a[i]>a[i+1]:
    if a[i]>max: max = a[i]
    if a[i+1]<min: min= a[i+1]
else:
    if a[i+1]>max: max = a[i+1]
    if a[i]<min: min= a[i]
```

```
if n==1:
   min = max = a[0]
if a[0]>a[1]:
     max = a[0]
     min = a[1]
else:
     max = a[1]
     min = a[0]
```

```
i=2
while i<n:
     if a[i]>a[i+1]:
         if a[i]>max: max = a[i]
         if a[i+1]<min: min=a[i+1]
     else:
         if a[i+1]>max: max = a[i+1]
         if a[i]<min: min= a[i]
```

```
if n%2==1:
    if a[n-1]<min:
        min = a[n-1]
    if a[n-1]>max:
        max = a[n-1]
```

n偶数： 1+3*((n-2)/2)

n奇数： 1+3*((n-3)/2)+2

分治递归求最大最小问题

- 将数组A从中间一分为2个子数组AL和AR
- 递归地求解AL的最大最小、递归地求解AR的最大最小
- AL最大和AR最大的大者作为A的最大，AL最小和AR最小的小者作为A的最小.

```
get_maxmin(A,L,R):
    if L==R:        return A[L],A[R]
    if R==L+1:
        if A[L]>A[R]: return A[L],A[R]
        else: return A[R],A[L]
    ML,mL = get_maxmin(A,L,m):
    MR,mR = get_maxmin(A,m+1,R):
    if ML>MR:              M = ML
    else:  M = MR
    if mL<mR:              min = mL
    else:  min = mR
    return M,min
```

- 时间复杂度

$$T(n)=\begin{cases} O(1) & n=2 \\ 2T(n/2)+2 & n>2 \end{cases}$$

设$n=2^k$，则

$$\begin{aligned} T(n) &= T(2^k) = 2T(2^{(k-1)}) + 2 \\ &= 2(2T(2^{(k-2)}) + 2) + 2 = 2^2 T(2^{(k-2)}) + 2^2 + 2 \\ &= 2^2(2T(2^{(k-3)}) + 2) + 2^2 + 2 = 2^3 T(2^{(k-3)}) + 2^3 + 2^2 + 2 \\ &= \ldots \\ &= 2^{k-1} + 2^{k-1} + \cdots + 2^2 + 2 \\ &= 3 \cdot 2^{k-1} - 2 = 3n/2 - 2 \end{aligned}$$

关注我

博客：hwdong.net

Youtube频道