算法设计与分析

Lecture 5: Divide and Conquer (分治)

曹刘娟

厦门大学信息学院计算机系

caoliujuan@xmu.edu.cn

Divide-and-Conquer

基本思想

- The divide-and-conquer (分治) algorithm divides an instance of a problem into two or more small instances. 将一个大问题切分成两个或者多个的子问题
- 化繁为简，分治是计算机科学要掌握的第二类重要思想
  - The small instance belongs to the same problem as the original instance. 子问题和原来问题是同一类型的问题
  - Assume that the small instance is easy to solve. 假设更加小的问题更加容易解决
  - Combine solutions to the small instances to solve the original instance. 将小问题的结果聚合起来
  - If the small instance is still difficult, divide again until it is easy.
- The divide-and-conquer is a top-down approach.
  - Recursion is usually adopted.
  - 分治经常和递归一起用，但不是必须。（亲同手足，互不分离，同时应用在算法设计）

Divide-and-Conquer

分治算法求解问题，遵循的三个步骤

The divide-and-conquer paradigm involves three steps at each level of the recursion:

- Divide the problem instance into a number of small instances. 将问题切分成为更小的部分
- Conquer the small instances by solving them recursively. If the sizes of small instances are small enough, just solve them without recursion. 求解更小的问题
- Combine (optional) the solutions to the small instances into the solution for the original instance. 将更小问题的答案合并起来

Analyzing Divide-and-Conquer Algorithms

分治算法的时间复杂度

- When an algorithm contains a recursive call to itself, its running time can often be described by a recursive equation (递归方程).
- We can easily solve them by the methods we have learned in Lecture 4.
- 我们可以用**递归方程的方法来分析分治算法的复杂度**

Analyzing Divide-and-Conquer Algorithms

分治算法的时间复杂度

- If the instance size is small enough, say $n \le c$ for some constant c, we can simply assume that the straightforward solution takes constant time $\Theta(1)$. 如果一个问题规模很小，那么直接认为算法的复杂度为$\Theta(1)$
- The running time of a divide-and-conquer algorithm is based on the three steps of the basic paradigm: 类似在递归里面学到的递归方程

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \le c \\ aT(n/b) + D(n) + C(n) & \text{otherwise} \end{cases}$$

  - $D(n)$: the cost of dividing into small instances. 将问题切割成为更小的问题的费用
  - $aT(n/b)$: conquer a small instances with each size n/b. 小问题总的费用
  - $C(n)$: the cost of combining the solutions of small instances. 将小问题答案合并起来的费用
  - $D(n)$ and $C(n)$ are usually merged into a function $f(n)$ for analysis convenience. 一般我们会将D(n)和C(n) 合并起来

MERGESORT 合并排序

也称归并排序

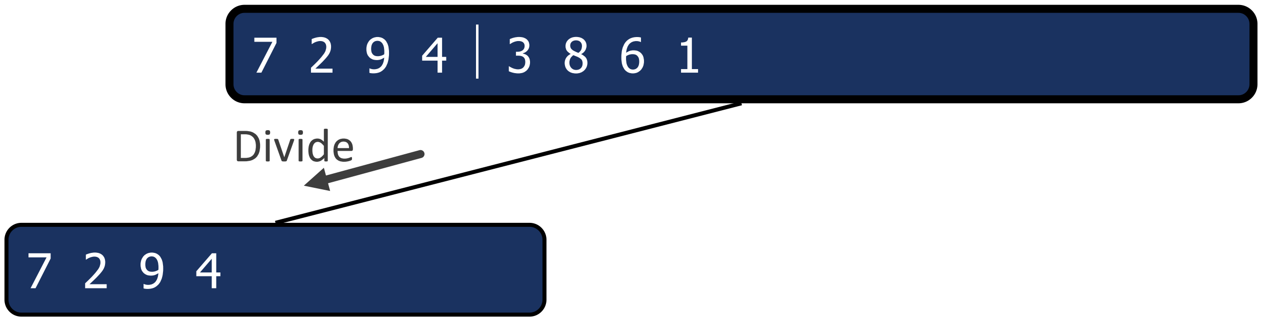
Mergesort 合并排序或归并排序

- Mergesort (合并排序) combines two sorted arrays into one sorted array.
- Given an array with n elements, Mergesort involves the following steps:
  1. Divide the array into two subarrays each with $n/2$ elements. 将数据切成两个原来一半大小的小数组
  2. Conquer each subarray by sorting it. Unless the array is sufficiently small, use recursion to do this. 逐个对小数组进行排序（递归划分），如果小数据足够的小，那么就可以直接求解
  3. Combine the solutions to the subarrays by merging them into a single sorted array. 将小问题答案合并起来

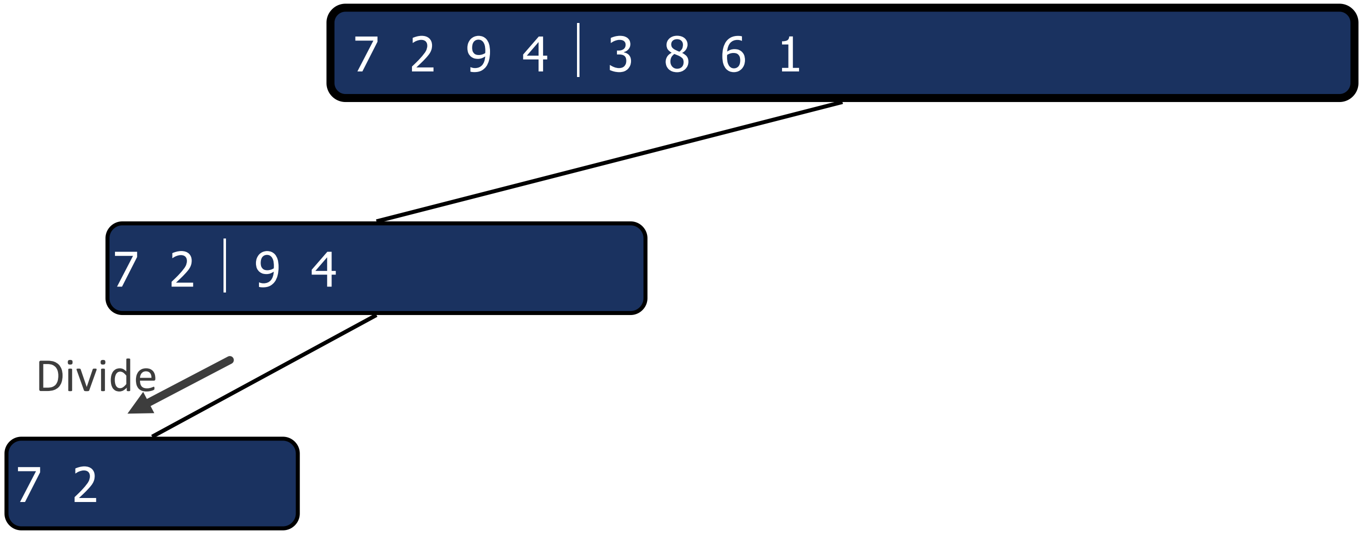
Mergesort Example

7 2 9 4 3 8 6 1

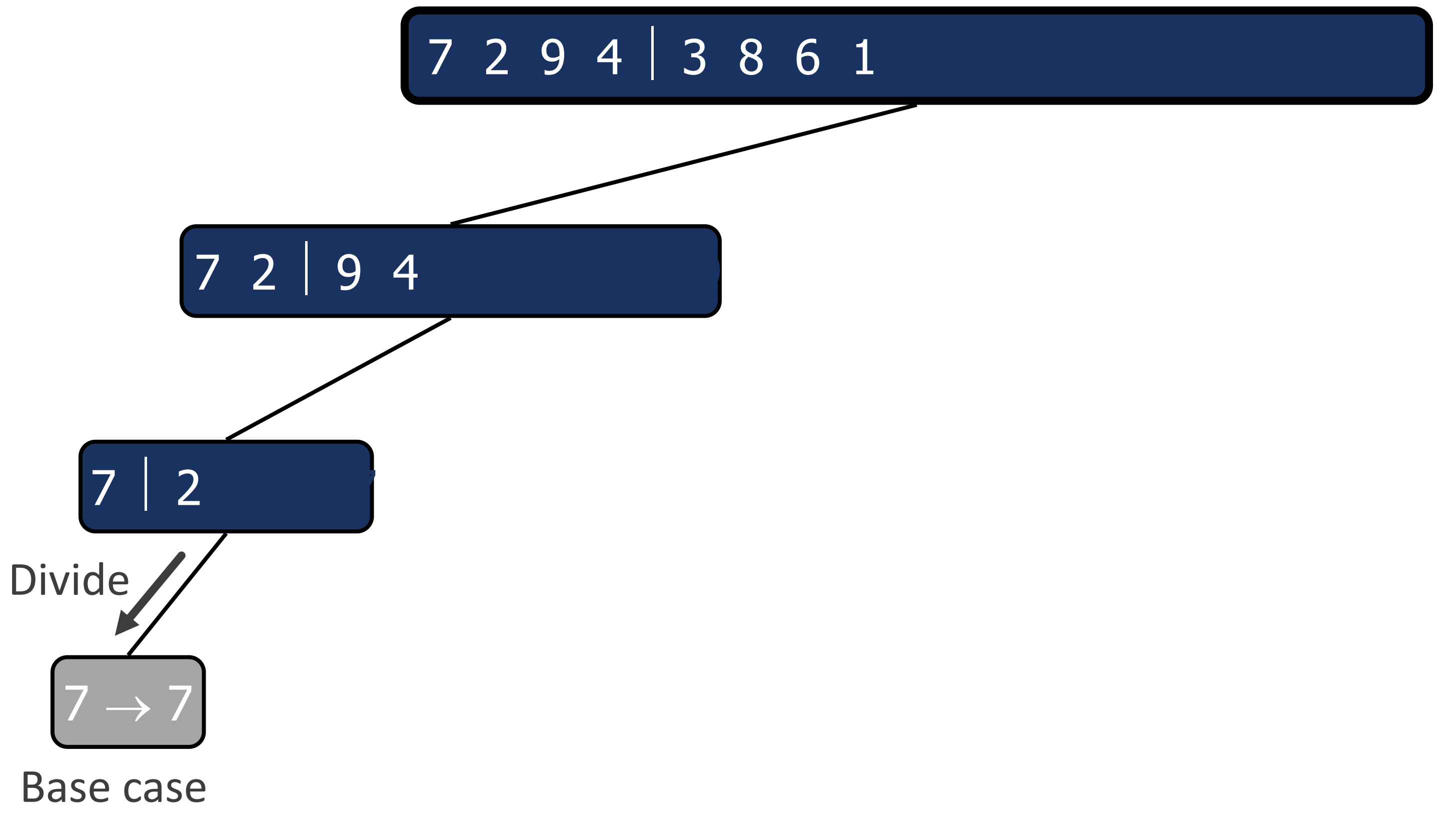
Mergesort Example



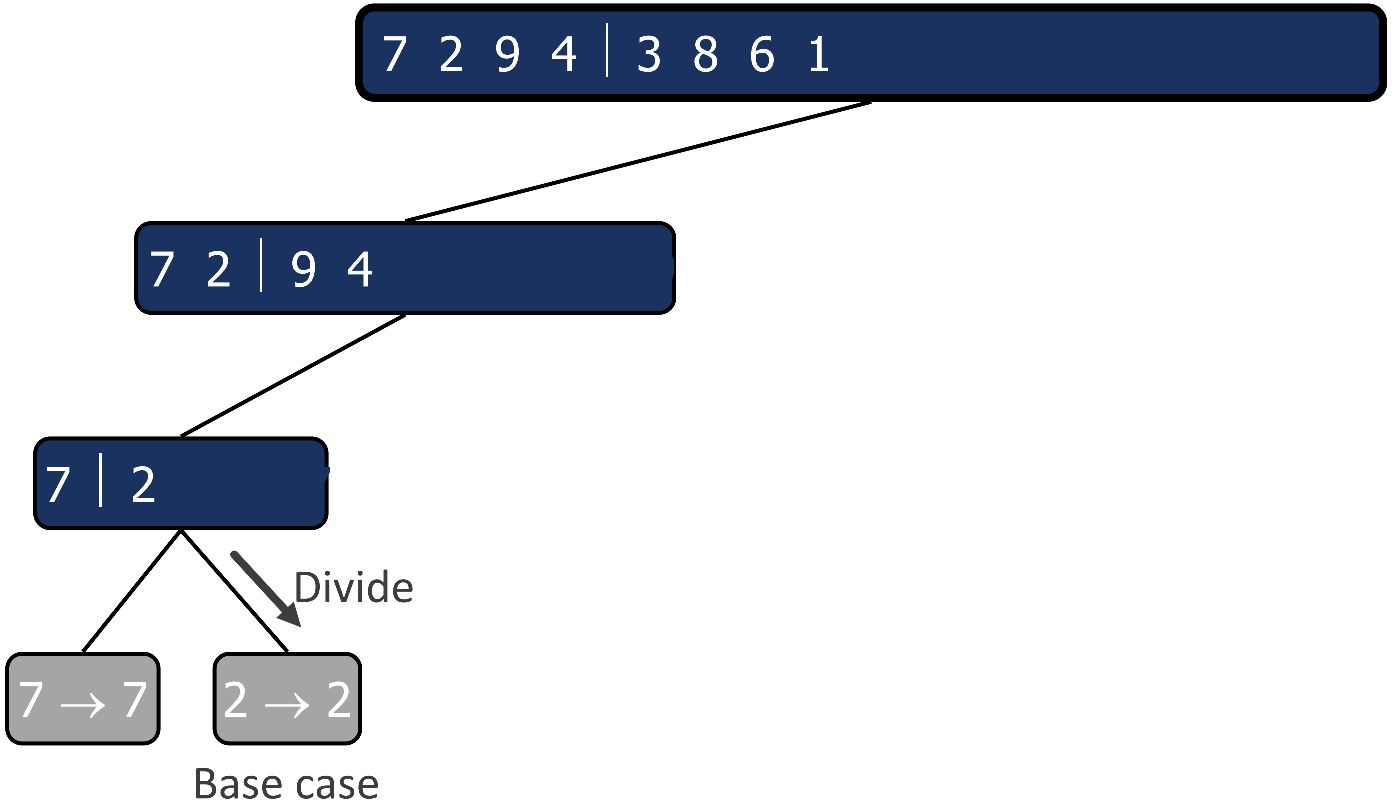
Mergesort Example



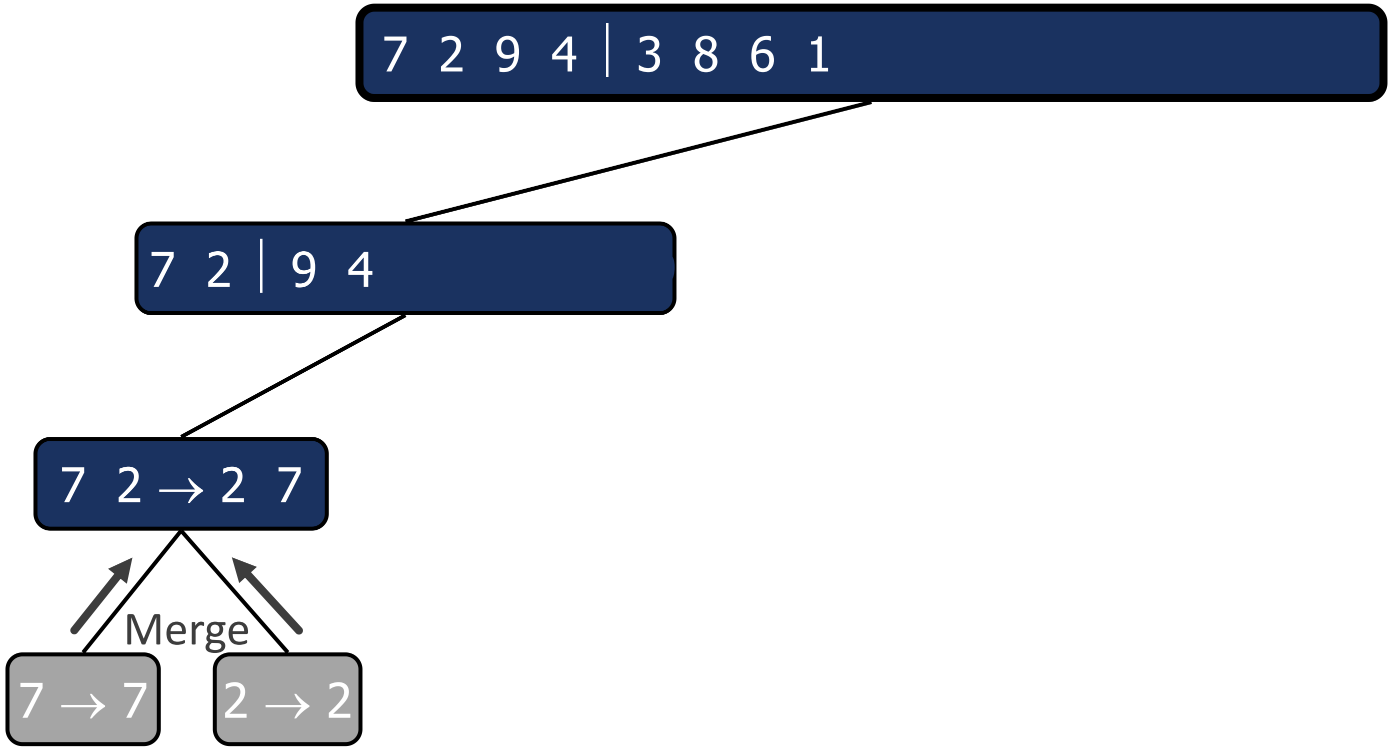
Mergesort Example



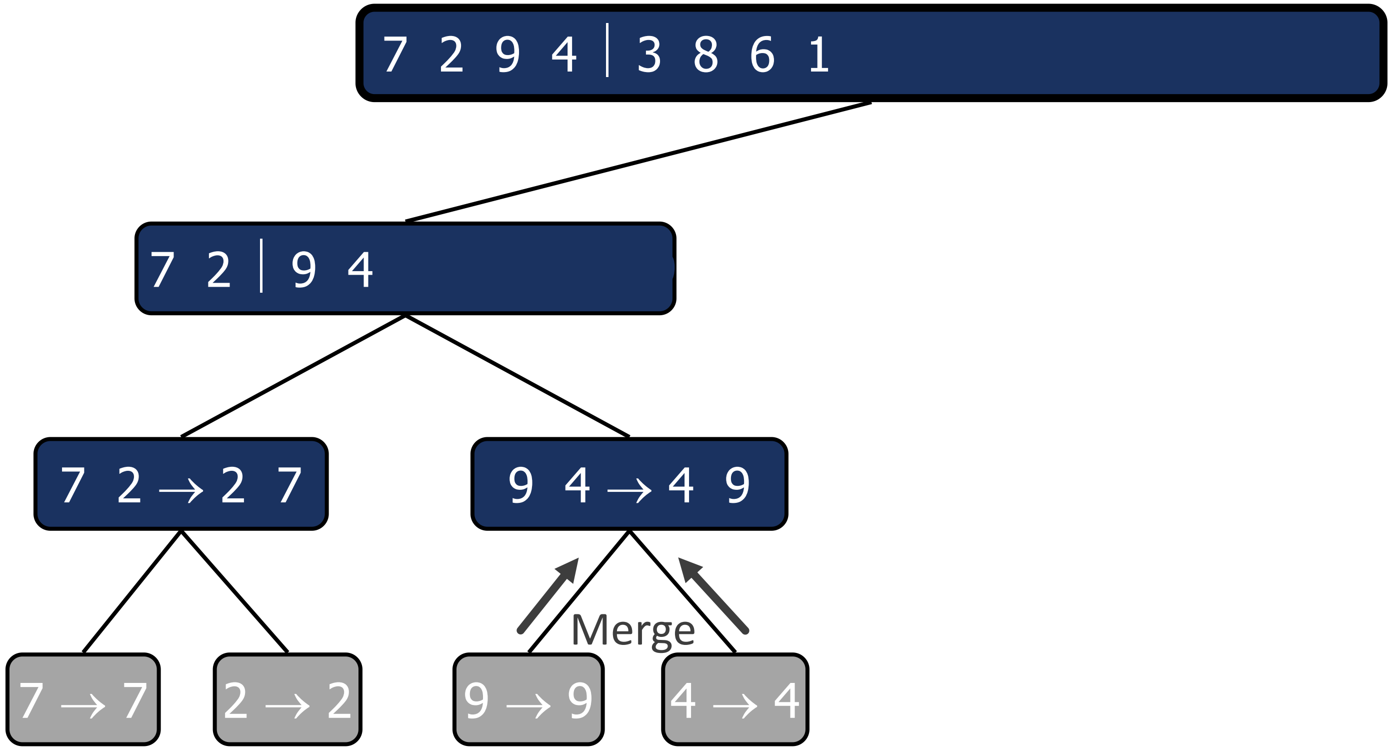
Mergesort Example



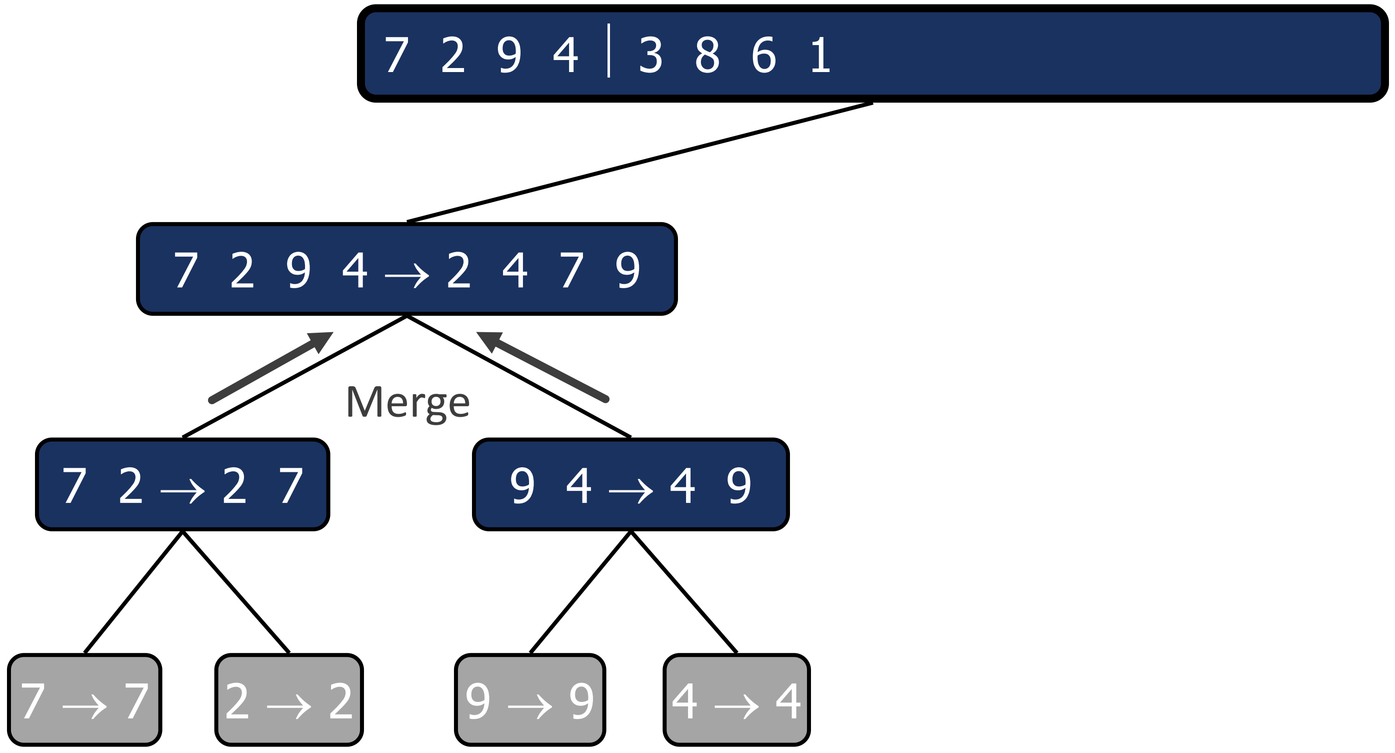
Mergesort Example



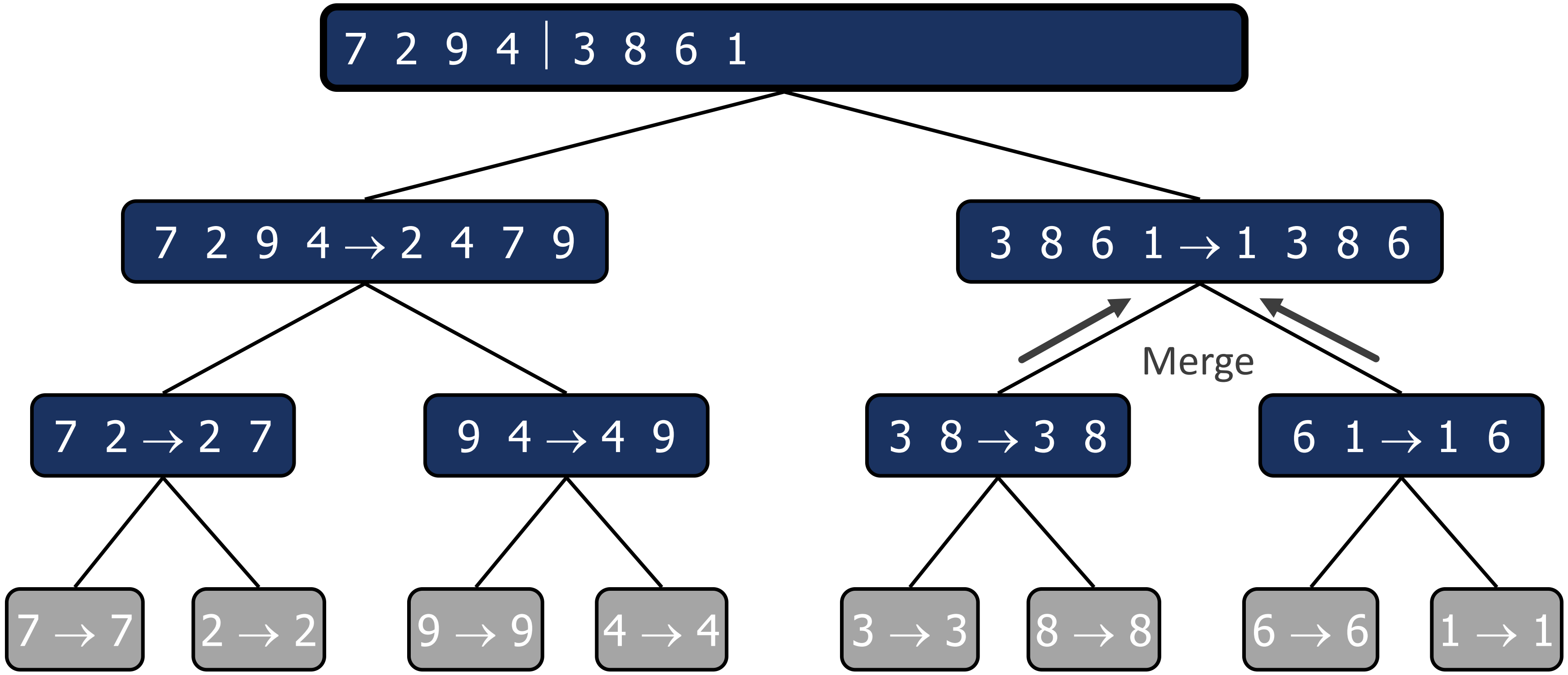
Mergesort Example



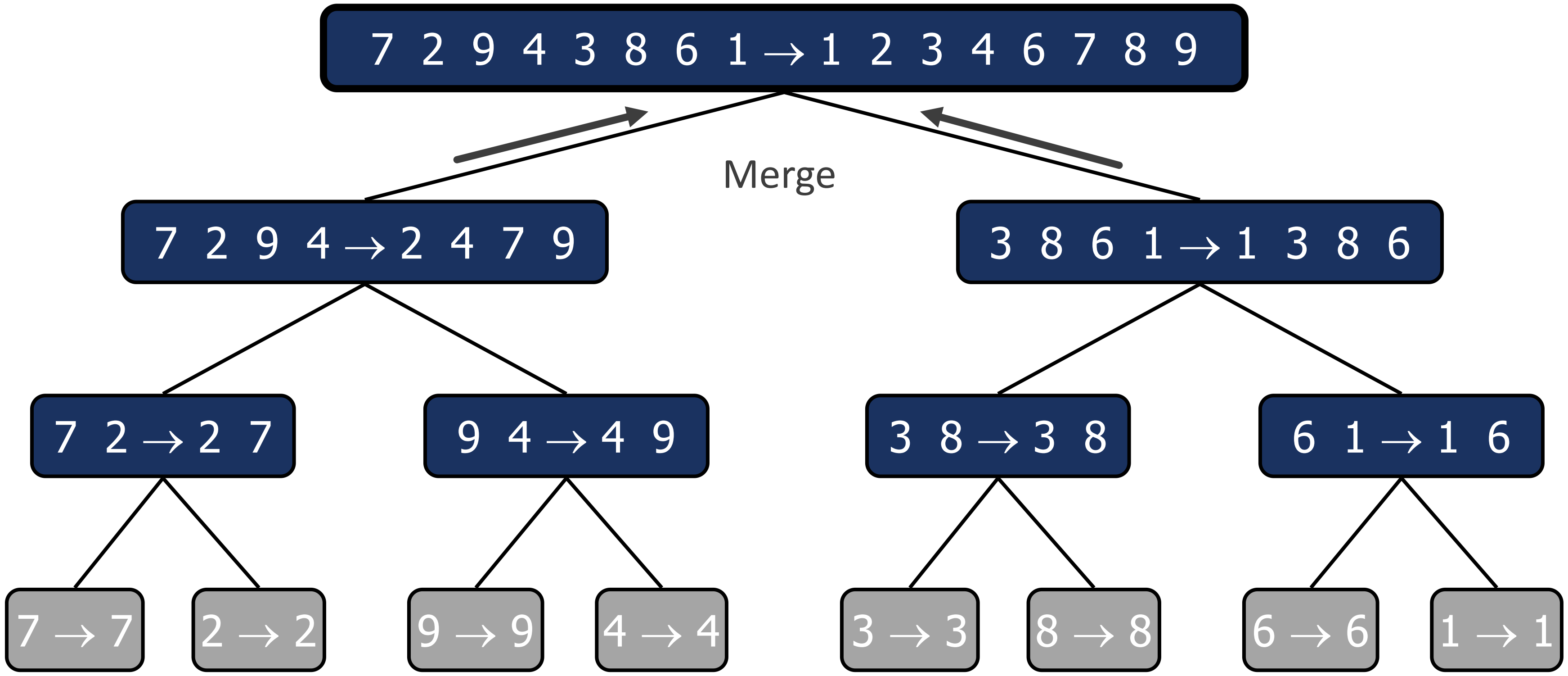
Mergesort Example



Mergesort Example



Mergesort Example



Mergesort Visualized Demo

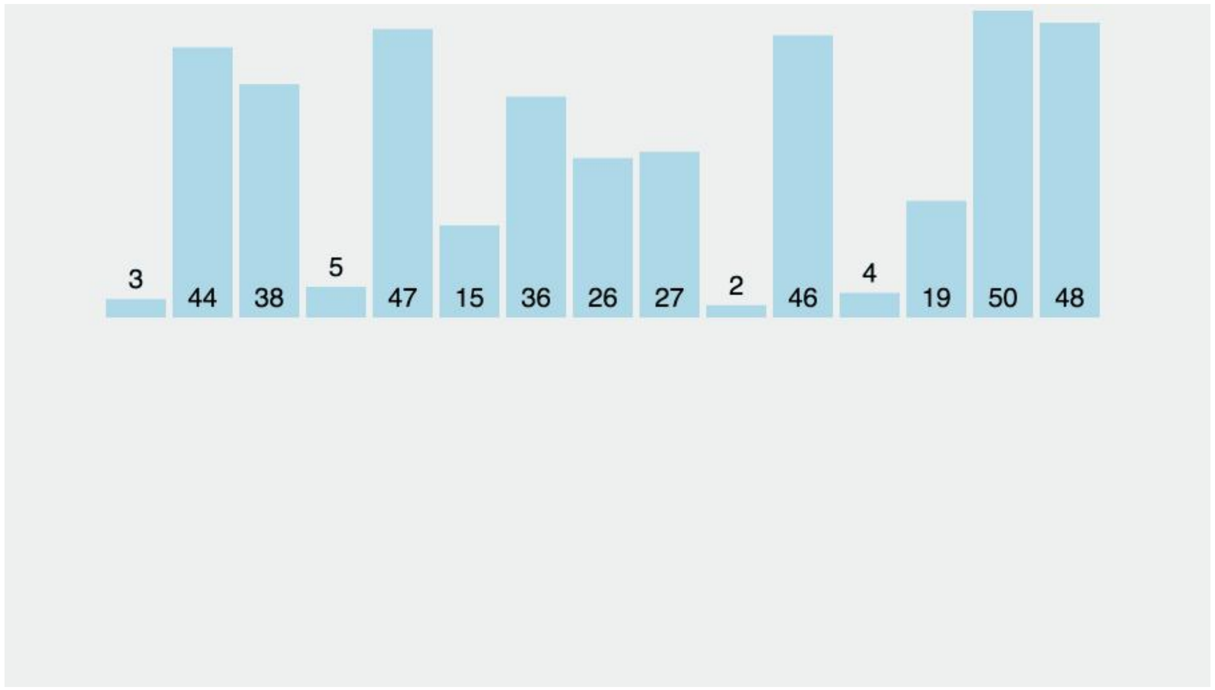


Image source: https://thumbs.gfycat.com/ZealousAdolescentBellsnake-size_restricted.gif

Mergesort

- Call $\text{MergeSort}(A, 1, \text{len}[A])$ for the sorting problem.
- Recursive call with different array index:
  - p: starting index
  - q: middle index
  - r: end index
- Exit condition: $p = r$, there is only one element.

```
MergeSort(A, p, r)
1  if p < r then
2      q ← ⌊(p + r)/2⌋
3      MergeSort(A, p, q)
4      MergeSort(A, q + 1, r)
5      Merge(A, p, q, r)
```

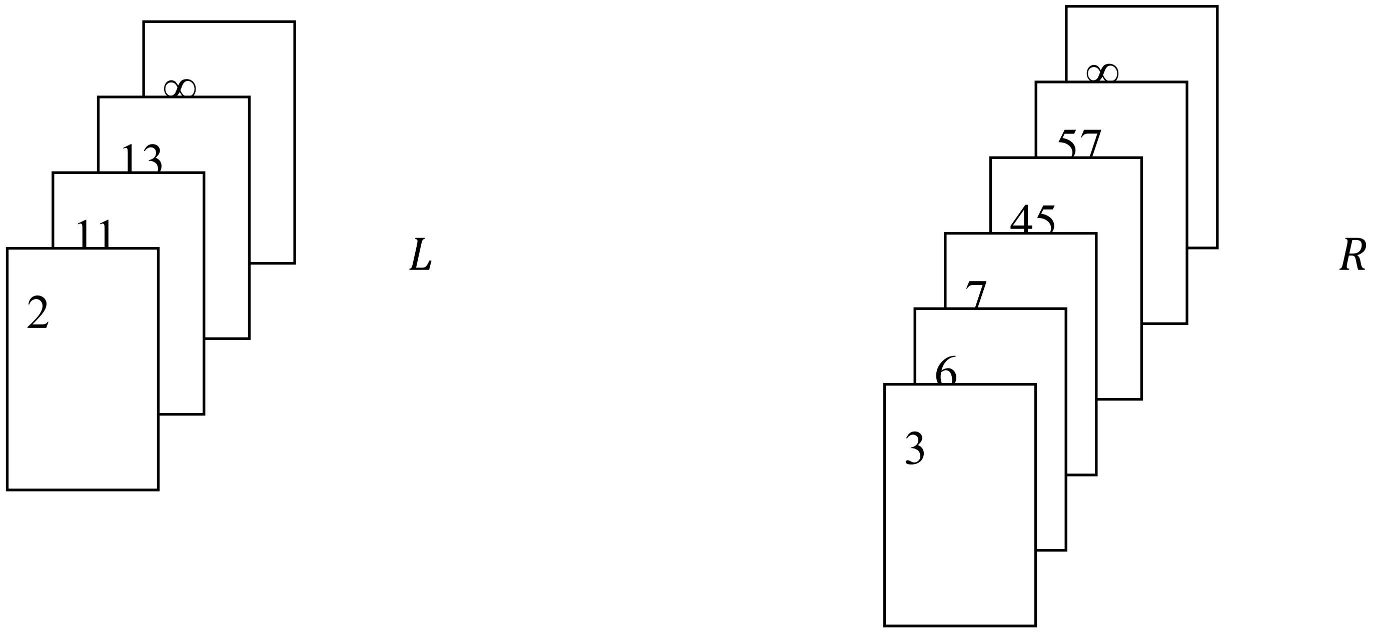
Mergesort

```
Merge($A, p, q, r$)
1   $n_1 \leftarrow q - p + 1$
2   $n_2 \leftarrow r - q$
3   for $i \leftarrow 1$ to $n_1$ do
4       $L[i] \leftarrow A[p + i - 1]$
5   for $j \leftarrow 1$ to $n_2$ do
6       $R[j] \leftarrow A[q + j]$
7   $L[n_1 + 1] \leftarrow \infty$
8   $R[n_2 + 1] \leftarrow \infty$
```

```
9   $i \leftarrow 1$
10  $j \leftarrow 1$
11  for $k \leftarrow p$ to $r$ do
12      if $L[i] \leq R[j]$ then
13          $A[k] \leftarrow L[i]$
14          $i \leftarrow i + 1$
15      else  $A[k] \leftarrow R[j]$
16          $j \leftarrow j + 1$
```

- Line 1-6: L and R are used to store two sorted subarrays with size n_1 and n_2.
- Line 7-8: Assign infinity at the end of L and R for comparison convenience.
- Line 9-16: For each index from p to r, compare one by one and increase the index of the array with smaller element.

Mergesort



A

2	3	6	7	11	13	45	57

The Correctness of Merge

```
11  for $k \leftarrow p$ to $r$ do
12      if $L[i] \le R[j]$ then
13          $A[k] \leftarrow L[i]$
14          $i \leftarrow i + 1$
15      else $A[k] \leftarrow R[j]$
16          $j \leftarrow j + 1$
```

- 证明合并排序里面Merge算法的正确性
- Loop invariant: **循环不变量**
  - At the start of each iteration of the for loop in Lines 12-17, the subarray $A[p \ldots k-1]$ contains the $k-p$ smallest elements of L and R, in sorted order.
  - $L[i]$ and $R[j]$ are the smallest elements of their arrays that have not been copied back into A.
- We show that this loop invariant holds
  - prior to the first iteration of the loop; 循环开始之前
  - after the kth iteration of the loop; 第k次循环之后
  - when the loop terminates. 循环结束之后

The Correctness of Merge

```
11  for $k \leftarrow p$ to $r$ do
12      if $L[i] \leq R[j]$ then
13          $A[k] \leftarrow L[i]$
14          $i \leftarrow i + 1$
15      else  $A[k] \leftarrow R[j]$
16          $j \leftarrow j + 1$
```

Initialization 初始步

- Prior to the first iteration of the loop, we have $k = p$, so that the subarray $A[p...p-1]$ is empty. 最开始的时候，A[p p-1]是空数组，成立
- Both $L[i]$ and $R[j]$ are the smallest elements of their arrays that have not been copied back into A. 这两个元素本来就是排好序的，并且放在各自的第一个位置

The Correctness of Merge

```
for $k \leftarrow p$ to $r$ do
    if $L[i] \le R[j]$ then
        $A[k] \leftarrow L[i]$
        $i \leftarrow i + 1$
    else $A[k] \leftarrow R[j]$
        $j \leftarrow j + 1$
```

Maintenance 归纳步骤

- Hypothesis: **Before the kth iteration**, the loop invariant holds.

  假设k次迭代之前，循环不变量是成立的。

- Let us first suppose that $L[i] \le R[j]$. Then $L[i]$ is the smallest element not yet copied back into A. 假设L[i]比较小，那么L[i]会被拷贝进A

- Because $A[p...k-1]$ contains the $k-p$ smallest elements, after Line 13 copies $L[i]$ into $A[k]$, the subarray $A[p...k]$ will contain the $k-p+1$ smallest elements. 最开始的时候 A[p.. k-1]有 k-p个最小的元素，经过第13行，那么A[p..k]就有k-p+1个元素了

- Before the next iteration, k and i are increased by 1. 在下个迭代之前，k和i都加1
  - $A[p...k]$ contain the $k-p+1$ smallest elements.
  - $L[i+1]$ and $R[j]$ are the smallest elements of their arrays that have not been copied back into A.

- Therefore, **before the $(k+1)$th iteration**, the loop invariant holds. 经过上面的检查，循环不变量成立

The Correctness of Merge

```
11  for $k \leftarrow p$ to $r$ do
12      if $L[i] \leq R[j]$ then
13          $A[k] \leftarrow L[i]$
14          $i \leftarrow i + 1$
15      else  $A[k] \leftarrow R[j]$
16          $j \leftarrow j + 1$
```

Termination 终止步

- At termination, $k = r + 1$.
- By the loop invariant, the subarray $A[p...k-1]$, which is $A[p...r]$, contains the $k - p = r - p + 1$ smallest elements of L and R, in sorted order.

结束的时候A[p...k-1]包含了r-p+1个从L和R里面最小的元素，因此循环不变量得证

- Therefore, the loop invariant holds. Merge correctly merges two sorted arrays into one sorted array.

Time Complexity of Mergesort

```
11  for $k \leftarrow p$ to $r$ do
12      if $L[i] \le R[j]$ then
13          $A[k] \leftarrow L[i]$
14          $i \leftarrow i + 1$
15      else  $A[k] \leftarrow R[j]$
16          $j \leftarrow j + 1$
```

合并排序时间复杂度

- No matter how different the input is, Merge always does $r - p + 1 = n$ times of key comparison.
  - For sorting algorithms, we usually only count the number of key comparisons.
- So the recursive equation is:

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

- By the master method case 2, we have $f(n) = n = \Theta(n) = \Theta(n^{\log_2 2})$. （公式法）
- Therefore, $T(n) = \Theta(n \lg n)$ for best-, worst- and average-case.

- Mergesort的缺点:需要额外O(n)的空间来进行存储

Classroom Exercise （课堂练习）

- Write each step of Mergesort to sort the following array:

$$\langle 27,\ 10,\ 12,\ 20,\ 25,\ 13,\ 15,\ 22 \rangle$$

Classroom Exercise

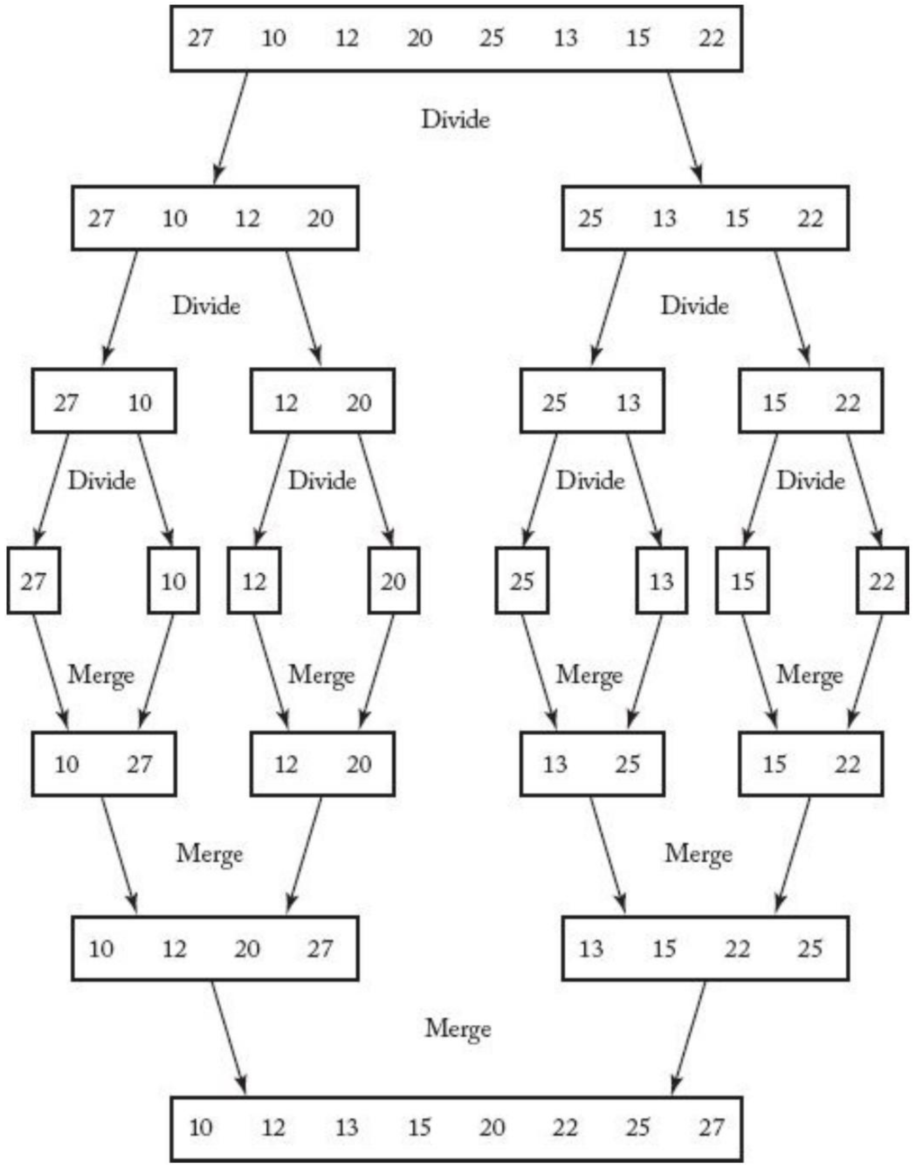


Image source: Figure 2.2, Richard E. Neapolitan, Foundations of Algorithms (5th Edition), Jones & Bartlett Learning, 2014

Classroom Exercise课堂练习

- 一共有25个运动员，在操场上有5个跑道，一次最多只能跑5个运动员。最少需要几次比赛，能够确定跑得最快的前3名运动员

Classroom Exercise课堂练习

Solution

在这个问题中，我们只需要返回前三个元素即可。

我们可以将运动员列表分为5组，每组5人。
首先进行5次比赛以确定每组内的排名（调用mergeSort函数来对每部分进行排序）

第6次比赛：把每组的第一名拿出来比赛。
假设第6次比赛结果的排名为A, B, C, D, E
A为第一，B为第二， C为第三，D为第4， E为第5

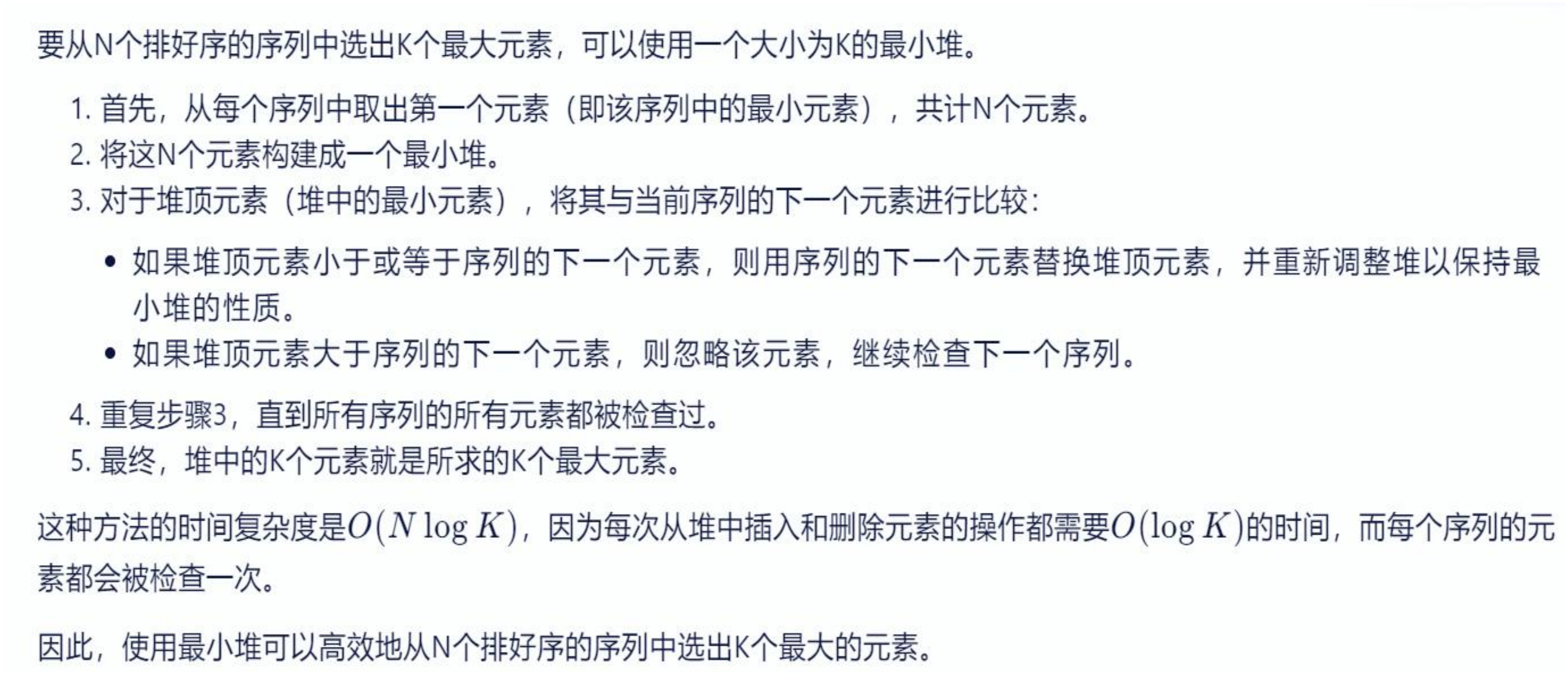
第7次比赛：把A所在小组的第2名，第3名， B所组在第2名，和C拿出来比赛
选出第二，第三

共7次比赛

思考题

- 从N个排好序的序列中选出K个最大元素?
  - 假定A1, A2.., AN是排好序的序列，如何最快得从中选出K个最大的元素

Solution



要从N个排好序的序列中选出K个最大元素，可以使用一个大小为K的最小堆。

1. 首先，从每个序列中取出第一个元素（即该序列中的最小元素），共计N个元素。
2. 将这N个元素构建成一个最小堆。
3. 对于堆顶元素（堆中的最小元素），将其与当前序列的下一个元素进行比较：
   - 如果堆顶元素小于或等于序列的下一个元素，则用序列的下一个元素替换堆顶元素，并重新调整堆以保持最小堆的性质。
   - 如果堆顶元素大于序列的下一个元素，则忽略该元素，继续检查下一个序列。
4. 重复步骤3，直到所有序列的所有元素都被检查过。
5. 最终，堆中的K个元素就是所求的K个最大元素。

这种方法的时间复杂度是$O(N \log K)$，因为每次从堆中插入和删除元素的操作都需要$O(\log K)$的时间，而每个序列的元素都会被检查一次。

因此，使用最小堆可以高效地从N个排好序的序列中选出K个最大的元素。

QUICKSORT

Quicksort 快速排序

- Mergesort splits the array first, and then combines them by merging.

合并排序先将数组分为两个子数组，之后再将排序好的子数组合并起来

- Can we roughly sort the array first, and then split it?

  可不可以先将数组粗粗的排下序，之后再对数组进行分割?

  - E.g. put small elements on the left, and large element on the right. 将比较小的元素放在数组左边，比较大元素放在数组右边
  - If we can do in this way, we don’t need to merge.

被誉为20世纪最好的10个算法之一

Quicksort

- **Quicksort (快速排序)** is developed by British computer scientist Charles Antony Richard Hoare (Tony Hoare) in 1962.
- You can know the main property of Quicksort by its name – quick!
- When implemented well, it can be about two or three times faster than Mergesort.
- 1980年获图灵奖
- 2000年因在计算机科学与教育方面的杰出贡献，获得英国王室颁赠爵士头衔
- 第一位因为发明算法而被封为爵士的计算机科学家

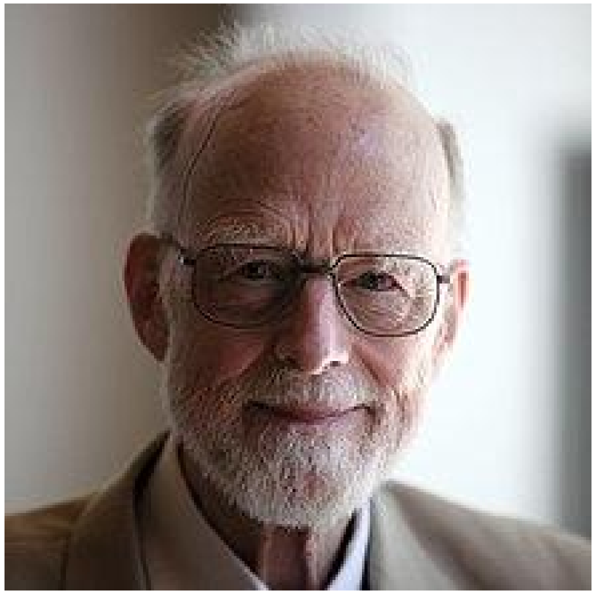
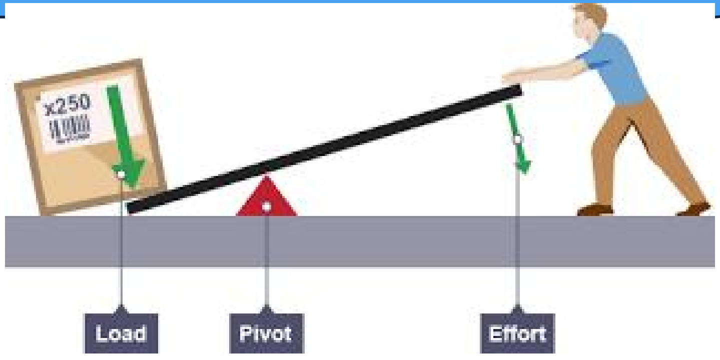

Tony Hoare in 2011

Image source: https://en.wikipedia.org/wiki/Tony_Hoare

Quicksort



Steps:步骤

- Randomly select a pivot (支点) element.
  - Conventional use the first or last item.

  选择数组里面第一个元素（或者最后一个，或者随机一个）当做支点

- Put all the elements smaller than the pivot element on its left, and all the elements greater than the pivot element on its right.

  通过移动支点和移动元素，将数组里面比这个支点小的元素放到这个支点的左边，比支点大的元素放到支点右边

- Recursively sort the left subarray and right subarray. 采用分治算法，不断递归左右子数组，不需要合并
  - Each subarray is sorted after recursion call. Therefore, there’s no need to combine the results.

Image source: http://mrtremblaycambridge.weebly.com/p15-turning-on-a-pivot.html

Quicksort Visualized Demo

https://visualgo.net/en/sorting

Image source: https://en.wikipedia.org/wiki/Quicksort

Quicksort

- Call QuickSort(A, 1, len[A]) for the sorting problem.
- Recursive call with different array index:
    - p: starting index
    - q: pivot index
    - r: end index
- Exit condition: $p = r$, there is only one element.

```
QuickSort($A$, $p$, $r$)
1   if $p < r$ then
2       $q \leftarrow$ Partition($A$, $p$, $r$)
3       QuickSort($A$, $p$, $q - 1$)
4       QuickSort($A$, $q + 1$, $r$)
```

Quicksort

- Call QuickSort(A, 1, len[A]) for the sorting problem.
- Recursive call with different array index:
  - p: starting index
  - q: pivot index
  - r: end index
- Exit condition: $p = r$, there is only one element.

```
MergeSort($A, p, r$)
1  if $p < r$ then
2      $q \leftarrow \lfloor (p+r)/2 \rfloor$
3      MergeSort($A, p, q$)
4      MergeSort($A, q+1, r$)
5      Merge($A, p, q, r$)
```

```
QuickSort($A, p, r$)
1  if $p < r$ then
2      $q \leftarrow$ Partition($A, p, r$)
3      QuickSort($A, p, q-1$)
4      QuickSort($A, q+1, r$)
```

和mergesort很像, 但是递归调用的顺序不一样, quick是先操作, 后递归.

Quicksort

步骤：

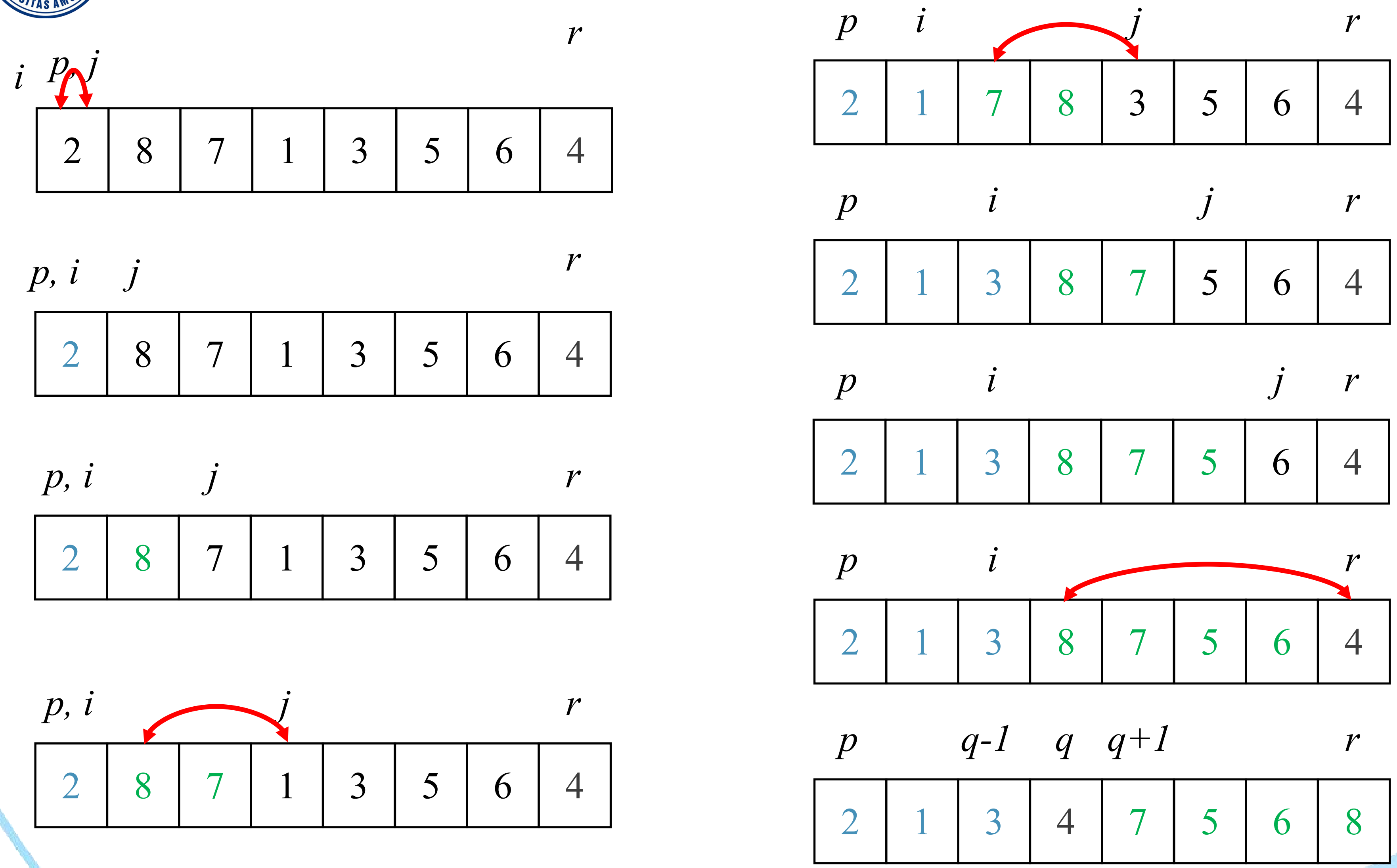
- Line 1: Simply select the last element $A[r]$ as the pivot. 选择最后一个元素当做支点
- Line 2: Use i to store the index for switching. i 放到数据的开头位置
- Line 3-6: iterate over j to find elements smaller than $\boldsymbol{pivot}$ and switch them to the front.

3-6行 从头开始遍历整个数据，如果有元素(坐标为j)比支点小，就把这个元素和第i个位置元素互换，之后i进位

- Line 7: put **pivot** at the proper position. 将支点放到 i+1位置（经过循环找到支点位置）

i理解为，比支点小的元素，都放在数组前i个

```
Partition($A, p, r$)
1  $pivot \leftarrow A[r]$
2  $i \leftarrow p - 1$
3  for $j \leftarrow p$ to $r - 1$ do
4      if $A[j] \le pivot$ then
5          $i \leftarrow i + 1$
6          $A[i] \leftrightarrow A[j]$
7  $A[i + 1] \leftrightarrow A[r]$
8  return $i + 1$
```


i p, j r
2 8 7 1 3 5 6 4
p, i j r
2 8 7 1 3 5 6 4
p, i j r
2 8 7 1 3 5 6 4
p, i j r
2 8 7 1 3 5 6 4
p i j r
2 1 7 8 3 5 6 4
p i j r
2 1 3 8 7 5 6 4
p i j r
2 1 3 8 7 5 6 4
p i r
2 1 3 8 7 5 6 4
p q-1 q q+1 r
2 1 3 4 7 5 6 8

Time Complexity of Quicksort 时间复杂度

- In Partition, each element in A is compared with the pivot except itself.

 在切割函数里面，每个元素都和支点比较了一次，一共有n-1次比较

- Therefore, the number of comparisons in Partition is $n-1$

```
Partition($A$, $p$, $r$)
1  $pivot \leftarrow A[r]$
2  $i \leftarrow p-1$
3  for $j \leftarrow p$ to $r-1$ do
4      if $A[j] \leq pivot$ then
5          $i \leftarrow i+1$
6          $A[i] \leftrightarrow A[j]$
7  $A[i+1] \leftrightarrow A[r]$
8  return $i+1$
```

Worst-Case Time Complexity of Quicksort

- **The worst-case** occurs when the array is already sorted (in either nondecreasing or nonincreasing order). 最坏情况复杂度 （数据已经排好序或者排了逆序）
- In each recursion step, **the pivot** element is always the smallest or largest item.

  在每次递归调用时，支点都是最大或者最小的元素
  - Thus, n elements are divided into $n-1$ and 0 elements during recursive call.
- The recursive equation is:

$$T(n) = T(n-1) + T(0) + n - 1$$

- Using recursion tree, we can easily get

$$T(n) = n(n-1)/2 = \Theta(n^2)$$

Worst-Case Time Complexity of Quicksort

- The closer the input array is to being sorted, the closer we are to **the worst-case** performance. 如果数组已经接近排好序了，那么算法的性能会更差
  - Because the pivot can’t fairly separate two subarrays.
  - Recursion loses it power.
- How to wisely choose the pivot? 选择哪一个数来当支点?
  - Random. 随机选择? （简单易行,稳定性不足）
  - Median of $A[1]$, $A[\lfloor n/2 \rfloor]$, and $A[n]$. Safe to avoid the worst-case but more comparisons are needed. 还是选择 第一个元素，中间元素，最后元素，这三个元素里面的中值（median）（性能好，额外代价）
- What will be the best case?

选择快速排序算法中的支点

在选择快速排序算法中的支点（pivot）时，有多种策略可以考虑。以下是几种常见策略的概述以及它们的优缺点：

1. **随机选择**：

 - 优点：简单易行，可以在一定程度上避免最坏情况的发生。

 - 缺点：虽然有可能选到好的支点，但仍然存在一定的概率选到差的支点，导致性能不稳定。

2. **选择第一个元素作为支点**：

 - 优点：实现简单，没有额外的计算开销。

 - 缺点：如果数据已经是有序或接近有序的，会导致最坏情况发生，此时快速排序的时间复杂度退化为O(n^2)。

3. **选择中间元素作为支点**：

 - 优点：有较大概率选到一个合适的支点，减少分区不平衡的可能性。

 - 缺点：需要额外的比较和交换操作，增加了一些计算成本。

4. **三数取中法**：

 - 优点：选取\(A[1]\)，\(A[lfloor n/2 \rfloor]\)，\(A[n]\)三个位置的元素，取这三个元素的中值作为支点，这种方法在统计上更有可能接近中位数，减少了支点选取得太过极端的情况，从而避免了最坏情况的发生。

 - 缺点：需要更多的比较来确定中值，增加了计算成本。

综合来看，如果对算法的稳定性有较高要求，建议使用三数取中法来选择支点。尽管这会增加一些比较操作，但能够有效避免最坏情况的发生，并且在平均情况下也能得到较好的性能。如果对算法的性能要求不是极端苛刻，并且数据分布相对均匀，那么随机选择支点也是一个不错的选择，因为它简单且易于实现。

在实际应用中，还可以根据具体问题的数据特征和需求来选择最适合的支点选择策略。

Best-Case Time Complexity of Quicksort

- **The best-case** occurs when the Partition almost evenly splits the array: 最好的情况是切割的时候，每次正好均匀的将数据切分为两半
  - Array has odd number of elements: Both subarrays have $\lfloor n/2 \rfloor$ elements.
  - Array has even number of elements: One subarrays has $\lfloor n/2 \rfloor$ elements and another has $n/2$.
- In both case, the size of the subarray is no more than $n/2$.
- The recursive equation is: **（公式法）**

$$T(n) = 2T(n/2) + n - 1 = O(n\lg n).$$

最好情况的时间复杂度也是 O(nlgn)

Basis for comparison	Quick Sort	Merge Sort
The partition of elements in the array	The splitting of a array of elements is in any ratio, not necessarily divided into half.	In the merge sort, the array is parted into just 2 halves (i.e. n/2).
Worst case complexity	O(n^2)	O(nlogn)
Works well on	It works well on smaller array	It operates fine on any size of array
Speed of execution	It work faster than other sorting algorithms for small data set like Selection sort etc	It has a consistent speed on any size of data
Additional storage space requirement	Less(In-place)	More(not In-place)
Efficiency	Inefficient for larger arrays	More efficient
Sorting method	Internal	External
Stability	Not Stable	Stable
Preferred for	for Arrays	for Linked Lists
Locality of reference	good	poor
Major work	The major work is to partition the array into two sub-arrays before sorting them recursively.	Major work is to combine the two sub-arrays after sorting them recursively.
Division of array	Division of an array into sub-arrays may or may not be balanced as the array is partitioned around the pivot.	Division of an array into sub array is always balanced as it divides the array exactly at the middle.
Method	Quick sort is in- place sorting method.	Merge sort is not in – place sorting method.
Merging	Quicksort does not need explicit merging of the sorted sub-arrays; rather the sub-arrays rearranged properly during partitioning.	Merge sort performs explicit merging of sorted sub-arrays.
Space	Quicksort does not require additional array space.	For merging of sorted sub-arrays, it needs a temporary array with the size equal to the number of input elements.

Average-Case Time Complexity of Quicksort

- The worst-case of Quicksort is no faster than insertion sort (also $\Theta(n^2)$), and slower than Mergesort ($\Theta(n\log n)$).
- The best-case of Quicksort is slower than the best-case of insertion sort ($\Theta(n)$).
- How dare it name itself “quick”? 快速排序的平均时间复杂度比较快
  - The average-case behavior earns its name!

Average-Case Time Complexity of Quicksort

快速排序平均时间复杂度

- To analyze the average-case time complexity, we can add randomization.
  - Randomly permutate the input array (uniform distributed input).输入随机
  - Randomly choose the pivot item. 支点选择随机

Average-Case Time Complexity of Quicksort

- By randomization, now the probability of pivot being any item in the array is $1/n$.

假设第p个小的元素被选为支点，则两个子问题规模为p-1和n-p

$$T(n) = \sum_{p=1}^{n} \frac{1}{n}([T(p-1) + T(n-p)] + n - 1)$$

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

$$nT(n) = 2\sum_{p=1}^{n} T(p-1) + n(n-1) \text{ (multiply by } n\text{)}$$

$$(n-1)T(n-1) = 2\sum_{p=1}^{n-1} T(p-1) + (n-1)(n-2) \text{ (apply to } n-1\text{)}$$

Average-Case Time Complexity of Quicksort

$nT(n) - (n-1)T(n-1) = 2T(n-1) + 2(n-1)$ (subtraction) *相减*

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

Harmonic series
调和级数

- Let $a_n = \frac{T(n)}{n+1}$,

递归树方法

$$a_n = a_{n-1} + \frac{2(n-1)}{n(n+1)} = \sum_{i=1}^{n} \frac{2(i-1)}{i(i+1)} \approx 2\sum_{i=1}^{n} \frac{1}{i} \approx 2\ln n.$$

- Therefore, $T(n) \approx (n+1)2\ln n = (n+1)2\ln 2 \lg n \approx 1.38(n+1)\lg n = \Theta(n\lg n)$.

$\ln n = \ln 2 \lg n$ 换底公式

LARGE INTEGER MULTIPLICATION

大整数乘法

Large Integer Multiplication 大整数乘法

- Suppose that we need to do arithmetic operations on integers whose size is very large. 大整数乘法运算
- In cryptography (密码学) and network security, encryption and decryption need to multiply very large numbers. 在密码学领域，加密和解密经常需要乘以非常大的整数
- How to do arithmetic for those large integers? 平时c语言默认的乘法是支持long或者long long数据的乘法，没有支持更大的数相乘
- 我们这里考虑的是两个二进制数的乘法

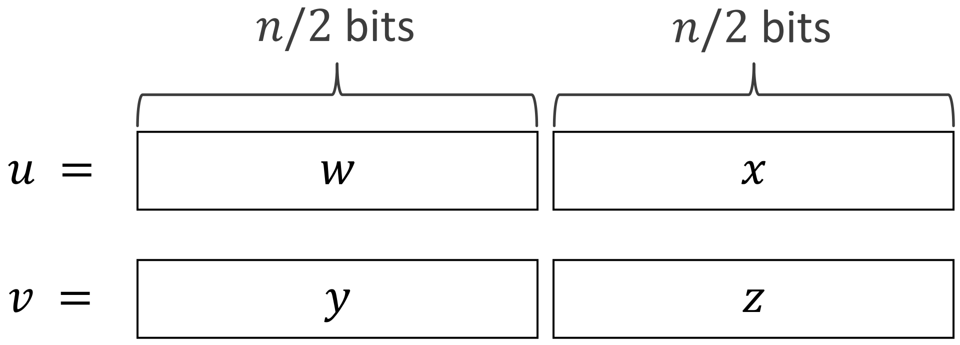
Large Integer Multiplication

- <span style="color:red">Tradition algorithm</span> is nothing but what we have learned in primary school.
- It takes $\Theta(n^2)$ bit operations. 如果我们按照小学时学的乘法来算的话，复杂度是$\Theta(n^2)$，一共有$\Theta(n^2)$次乘法，$\Theta(n^2)$次加法

```
      ×    ************
           ************
     ____________________________
           ************
          ************
         ************
        ************
       ************
      ************
     ************
    ************
   ************
  ************
 ************
************
     ______________________________
************************
```

Large Integer Multiplication

- Use n-bit binary representation for u and v. 这里假设u，v都是用二进制来表示的
- We can use divide-and-conquer: Each integer is divided into two parts of $n/2$ bits each. 采用分治来计算，将每个大整数，都切成两半



Example:
$(110011)_2$
$= (110)_2 \times 2^3 + (011)_2$

- Therefore, integers u and v can be represented as:

$$u = w2^{n/2} + x, \quad v = y2^{n/2} + z.$$

- Then, we have:

$$uv = \left(w2^{n/2} + x\right)\left(y2^{n/2} + z\right) = wy2^n + (wz + xy)2^{n/2} + xz.$$

Large Integer Multiplication

$$uv = (w2^{n/2} + x)(y2^{n/2} + z) = wy2^n + (wz + xy)2^{n/2} + xz.$$

- We need 4 recursive multiplications with size $n/2$ to calculate

$$wy2^n + (wz + xy)2^{n/2} + xz$$

  - Multiply with 2^n is to simply shift by n bits to the left with cost $\Theta(n)$.
  - 3 times of addition is also with cost $\Theta(n)$.
- The recursive equation is:

$$T(n) = \begin{cases} \Theta(1) & n = 1 \\ 4T(n/2) + cn & n > 1 \end{cases}$$

- By the **master method**, we get $T(n) = \Theta(n^2)$.
- It is still quadratic. Why?
- 用公式法分析下，哪些部分是可以改进的？哪个是a, b,哪个是f(n)

$n^{\lg 4}$

Large Integer Multiplication

- We decompose the instance of n into 4 small instances with size $n/2$.
- If we can decrease 4 to 3, by the master method we get $T(n) = \Theta(n^{\lg 3})$.
- As before, we need to calculate

$$wy,\ wz + xy,\ xz$$

- If instead we set

$$r = (w + x)(y + z) = wy + (wz + xy) + xz$$

we have

$$wz + xy = r - wy - xz$$

- Then, we only need to calculate

$$r,\ wy,\ xz$$

Large Integer Multiplication

```
Multiply2int($u$, $v$)
1  if $|u| = |v| = 1$ then return $uv$
2  else
3          Split $u$ into $w$ and $x$; split $v$ into $y$ and $z$
4          $A_1 \leftarrow$ Multiply2int($w$, $y$)
5          $A_2 \leftarrow$ Multiply2int($x$, $z$)
6          $A_3 \leftarrow$ Multiply2int($w + x$, $y + z$)
7      return $A_1 2^n + (A_3 - A_1 - A_2)2^{n/2} + A_2$
```

- The above method yields the following recursive equation:

$$T(n) = 3T(n/2) + cn.$$

- By the master method, we get $T(n) = \Theta(n^{\lg 3}) \approx \Theta(n^{1.59})$.

用额外的加减法来替换掉乘法。在计算机里面乘法操作比加法操作慢不少

Classroom Exercise （课堂练习）

Write the pseudocode of binary search algorithm: 写出二分查找的伪代码

- Given a sorted array A, 给定一个排序好的数组
  - If x equals the middle item, quit. 如果x等于中间元素，那么找到，退出
  - Otherwise, compare x with the middle item.
    - If x is smaller, search the left subarray. 如果x比中间元素小，就搜索左数组
    - If x is greater, search the right subarray. 如果x比中间元素大，就搜索右数组
- What is the time complexity? 时间复杂度是多少?

试着写下递归版本和非递归版本

Classroom Exercise （课堂练习）

Solution:

```
BinarySearch($A$, $x$)
1 $p \leftarrow 1$
2 $r \leftarrow n$
3 $k \leftarrow 0$
4 while $p <= r$ and $k = 0$ do
5     $m \leftarrow \lfloor (p+r)/2 \rfloor$
6     if $A[m] = x$ then return $m$
7     else if $x < A[m]$ then $r \leftarrow m - 1$
8         else $p \leftarrow m + 1$
9 return 0
```

```
RecursiveBinarySearch($A$, $p$, $r$)
1 if $p > r$ then return 0
2 else
3     $m \leftarrow \lfloor (p+r)/2 \rfloor$
4     if $A[m] = x$ then return $m$
5     else if $x < A[m]$ then
6         RecursiveBinarySearch($A$, $p$, $m - 1$)
7     else
8         RecursiveBinarySearch($A$, $m + 1$, $r$)
```

Divide-and-conquer algorithm is not necessarily implemented by recursion.

Classroom Exercise

- The recursive equation is:

$$T(n) = T(n/2) + 1.$$

- By using **master method case 2**, we have $T(n) = \Theta(\lg n)$.

MATRIX MULTIPLICATION

矩阵乘法

Matrix Multiplication 矩阵乘法

- Given matrices A and B with size $n \times n$, compute the matrix product $C = AB$.

  给定两个数组都是$n \times n$，计算两个矩阵相乘的结果
- The formula we have learned in linear algebra for doing this is:

$$C(i,j) = \sum_{k=1}^{n} A(i,k)B(k,j)\,.$$

- Calculating each $C(i,j)$ takes $O(n)$. Thus, calculating total $n \times n$ elements in C takes $O(n^3)$.

Matrix Multiplication

- Suppose we want to product C of two 2×2 matrices, A and B, That is,

把每个矩阵都分成为4个更小的矩阵，然后再把结果结合起来

$$\begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}.$$

- 如果这个矩阵太大了，单独的一台计算机无法存储，就必须将矩阵进行切割，将切割之后的一块块放在不同的计算机上
- 想下我们平时用的APP里面哪个有可能用到非常大的矩阵呢？
- The divide-and-conquer version consists of computing C as defined by the following equation:

$$C = \begin{bmatrix} a_{11}b_{11} + a_{12}b_{21} & a_{11}b_{12} + a_{12}b_{22} \\ a_{21}b_{11} + a_{22}b_{21} & a_{21}b_{12} + a_{22}b_{22} \end{bmatrix}.$$

原问题划分为8个子问题的乘积

Matrix Multiplication

- Will it be better?
- The cost of multiplying two $n \times n$ matrices consists of:
  - 8 times the cost of multiplying two $n/2 \times n/2$ matrices;
  - 4 times the cost of adding two $n/2 \times n/2$ matrices.
- The recursive equation is:

$$T(n) = \begin{cases} \Theta(1) & n = 1 \\ 8T(n/2) + 4(n/2)^2 & n > 1 \end{cases}$$

- Make use of the **master method,** $T(n) = \Theta(n^3)$. And thus this method is no faster than the ordinary one.
- What can we do?

- 我们要探索更好的方式

Matrix Multiplication with Strassen's algorithm

- **Strassen‘s algorithm (斯特拉森算法)** reduces the number of multiplications from 8 to 7.
- Strassen determined that if we let

$$
\begin{aligned}
m_1 &= (a_{11} + a_{22})(b_{11} + b_{22}) \\
m_2 &= (a_{21} + a_{22})b_{11} \\
m_3 &= a_{11}(b_{12} - b_{22}) \\
m_4 &= a_{22}(b_{21} - b_{11}) \\
m_5 &= (a_{11} + a_{12})b_{22} \\
m_6 &= (a_{21} - a_{11})(b_{11} + b_{12}) \\
m_7 &= (a_{12} - a_{22})(b_{21} + b_{22})
\end{aligned}
$$

Less multiplication, at expense of more addition and subtraction.
这种方法采用了更少的乘法，代价在于更多的加法和减法

the product C is given by

$$C = \begin{bmatrix} m_1 + m_4 - m_5 + m_7 & m_3 + m_5 \\ m_2 + m_4 & m_1 + m_3 - m_2 + m_6 \end{bmatrix}$$

Matrix Multiplication with Strassen's algorithm

- To multiply two 2×2 matrices, Strassen's method requires 7 multiplications and 18 additions/subtractions.
  - The standard method requires 8 multiplications and 4 additions/subtractions.
  - Use 14 more additions/subtractions to save 1 multiplication. It that worthy?
- Recursive equation:

$$T(n) = \begin{cases} \Theta(1) & n = 1 \\ \mathbf{7}T(n/2) + \mathbf{18}(n/2)^2 & n > 1 \end{cases}$$

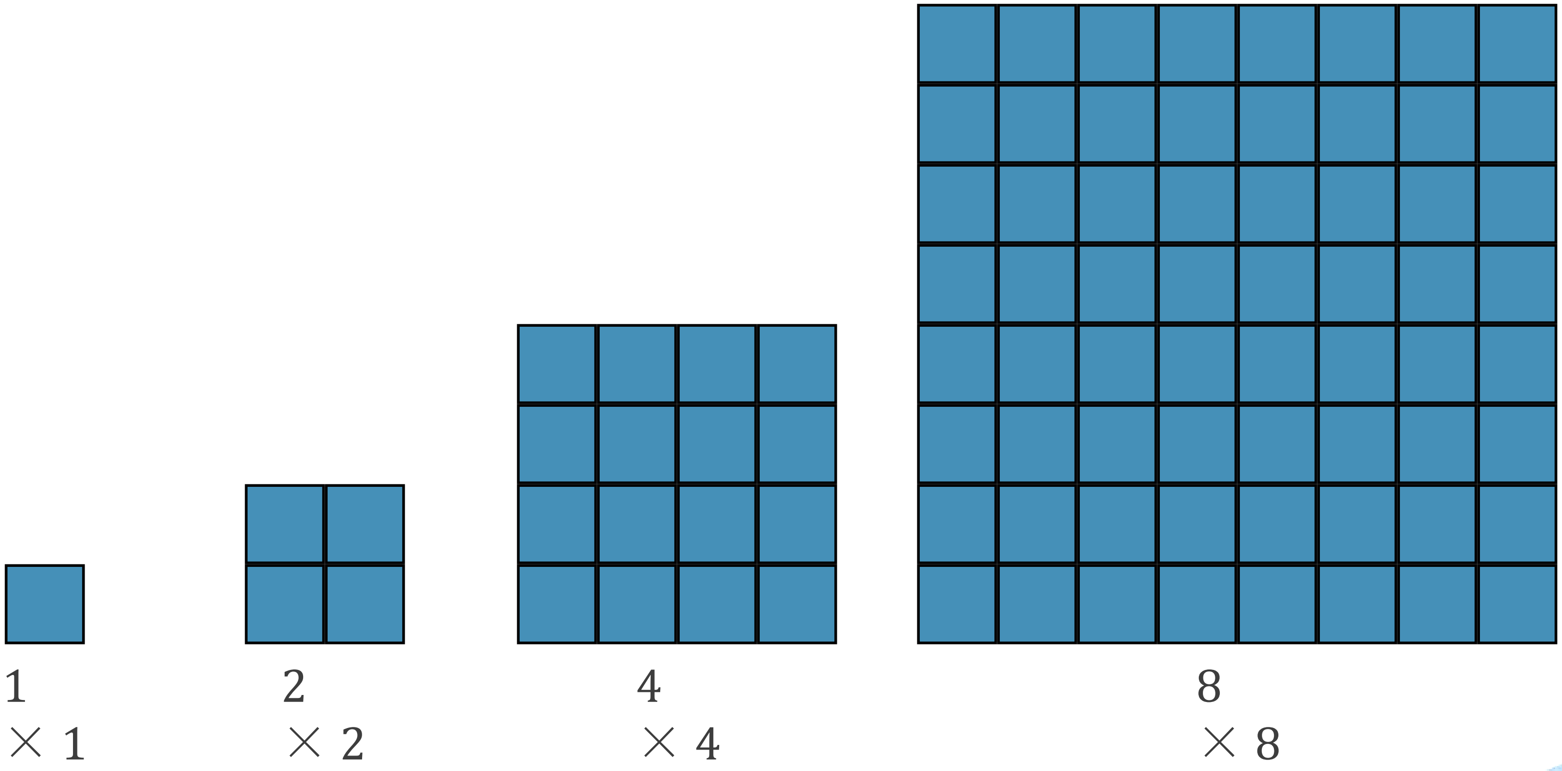
- Use the **master method case 1**, $f(n) = \frac{18}{4}n^2 = O(n^{\log_2 7 - \epsilon}) \approx O(n^{2.81-\epsilon})$ for $\epsilon \approx 0.81$.
- Therefore, we have $T(n) = \Theta(n^{2.81})$.

DEFECTIVE CHESSBOARD

残缺棋盘

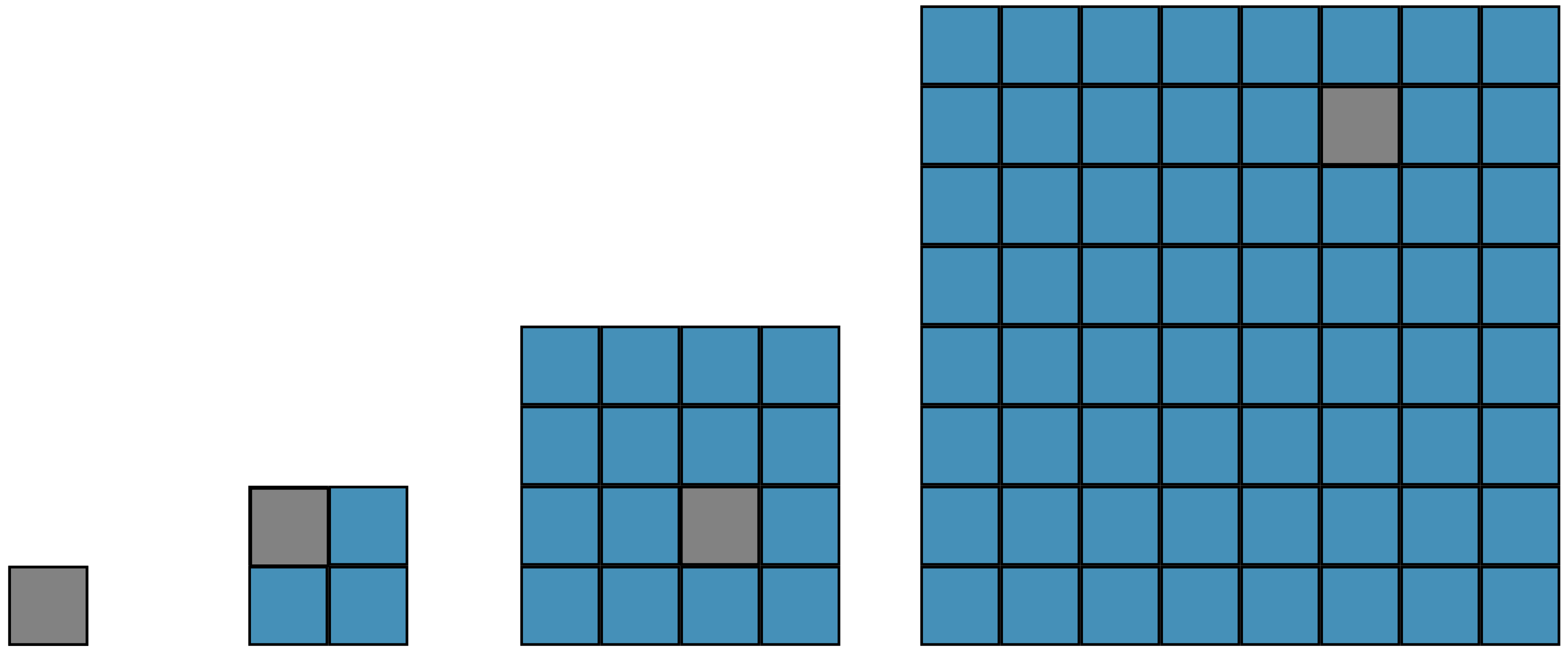
Defective Chessboard 残缺棋盘

- A chessboard is an $2^n \times 2^n$ grid, for $n \geq 0$:



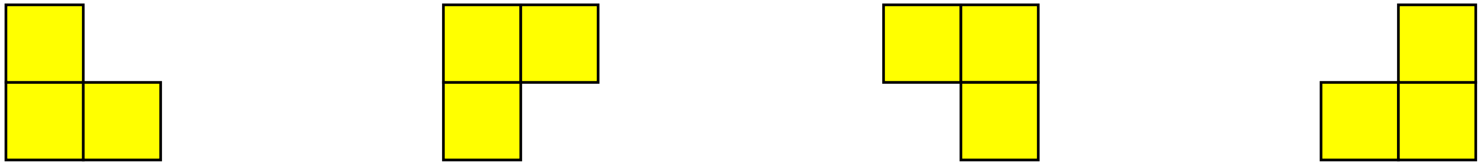
Defective Chessboard

- A defective chessboard (残缺棋盘) is chessboard that has one unavailable (defective) position. 残缺棋盘上有一个格子是少了的



Defective Chessboard

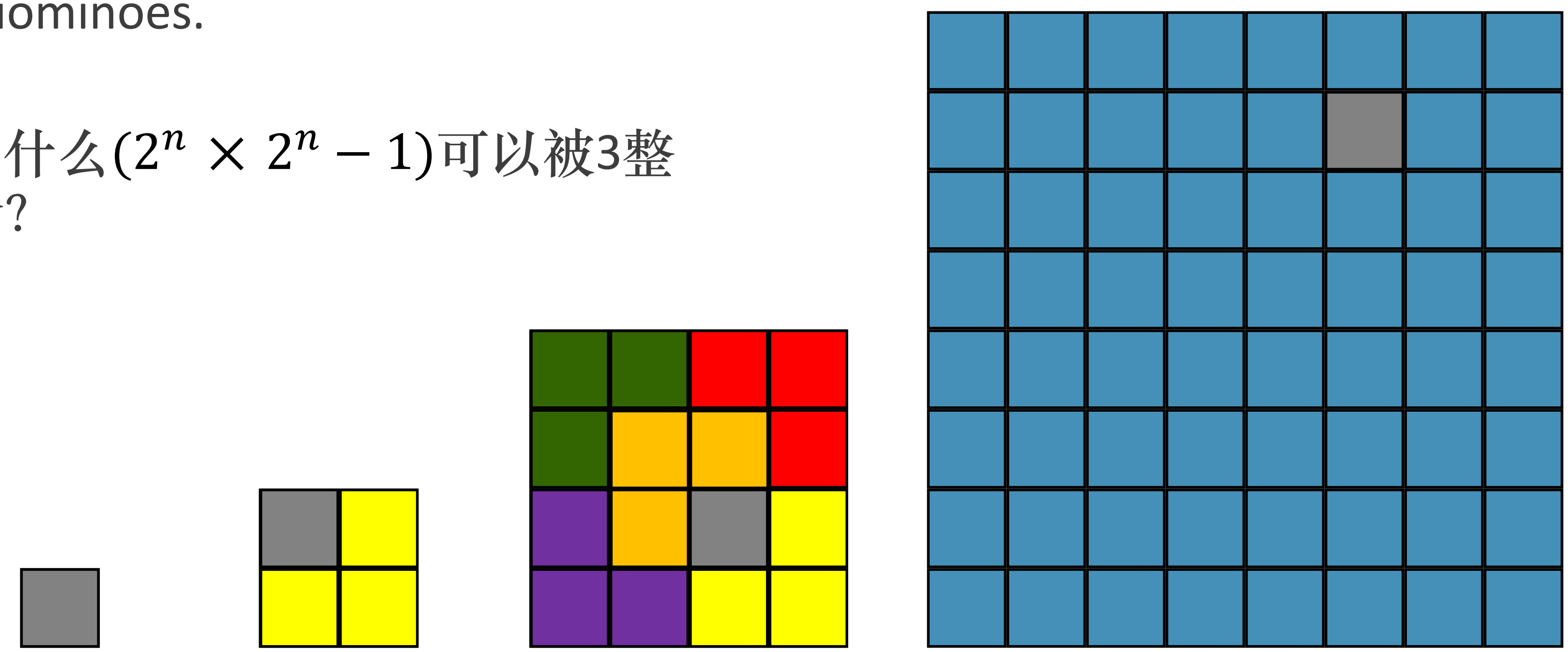
- A **triomino (三格板)** is an L shaped object that can cover three squares of a chessboard.
- A triomino has four orientations. 三格板有4个不同的方向
- You can use infinite number of triominoes. 三格板的数量是无限的



Defective Chessboard

- Task: Place triominoes on an $2^n \times 2^n$ $(n \geq 1)$ defective chessboard so that all $2^n \times 2^n - 1$ nondefective positions are covered. 用三格板覆盖所有非残缺的位置
- Totally, we place $(2^n \times 2^n - 1)/3$ triominoes.

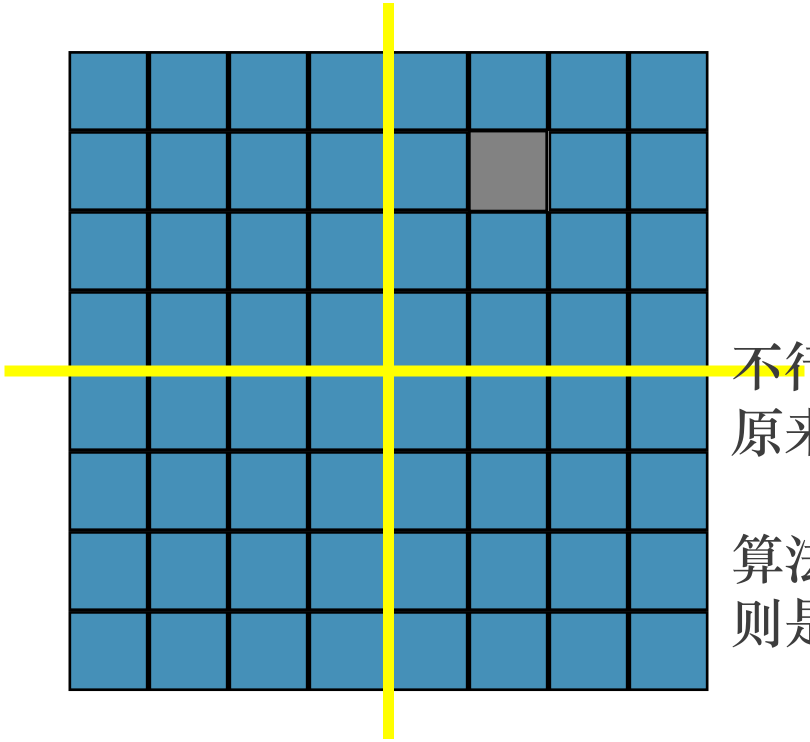
为什么$(2^n \times 2^n - 1)$可以被3整除?



Defective Chessboard

- How to use divide-and-conquer?

可不可以直接就4等分的方式来分治?



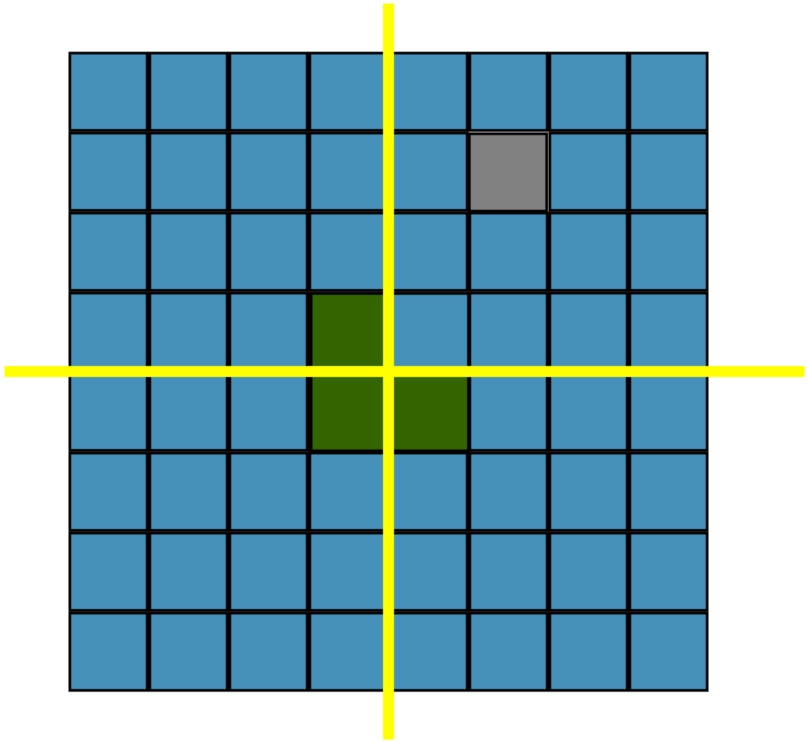
不行，分治的子问题要和原来的问题是一个类型的

算法里面一个很重要的原则是**等价性**

- If we divide it into 4 $2^{n-1} \times 2^{n-1}$ chessboard, only one is defective.

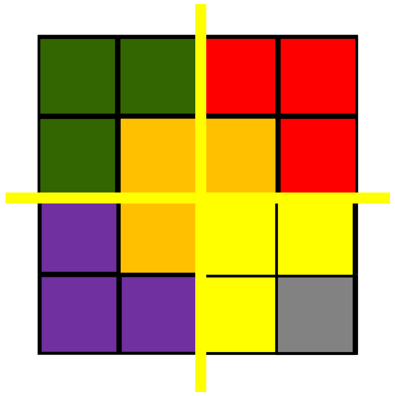
Defective Chessboard

- Put one triomino at their common corner, which makes all 4 small chessboards have a defective position. 将一个3元板放在中间，这样每个小棋盘就有格子有一个缺陷点了



Defective Chessboard

- Then, simply recursively solve this problem.



棋盘左上角坐标是(1,1)

```
TileBoard($tr, tc, dr, dc, size$)
1    if $size = 1$ return ok
2    $tile \leftarrow tile + 1$; $t \leftarrow tile$   当前三元板的编号
3    $s \leftarrow size/2$
4    if $dr < tr + s$ and $dc < tc + s$ then
5        TileBoard($tr, tc, dr, dc, s$)
6    else  $Board[tr + s - 1, tc + s - 1] \leftarrow t$
7        TileBoard($tr, tc, tr + s - 1, tc + s - 1, s$)
8    if $dr < tr + s$ and $dc \geq tc + s$ then
9        TileBoard($tr, tc + s, dr, dc, s$)
10   else  $Board[tr + s - 1, tc + s] \leftarrow t$
11       TileBoard($tr, tc + s, tr + s - 1, tc + s, s$)
12   if $dr \geq tr + s$ and $dc < tc + s$ then
13       TileBoard($tr + s, tc, dr, dc, s$)
14   else  $Board[tr + s, tc + s - 1] \leftarrow t$
15       TileBoard($tr + s, tc, tr + s, tc + s - 1, s$)
16   if $dr \geq tr + s$ and $dc \geq tc + s$ then
17       TileBoard($tr + s, tc + s, dr, dc, s$)
18   else  $Board[tr + s, tc + s] \leftarrow t$
19       TileBoard($tr + s, tc + s, tr + s, tc + s, s$)
```

Left top

Right top

Left bottom

Right bottom

When $size = 1$, the board is 1×1.

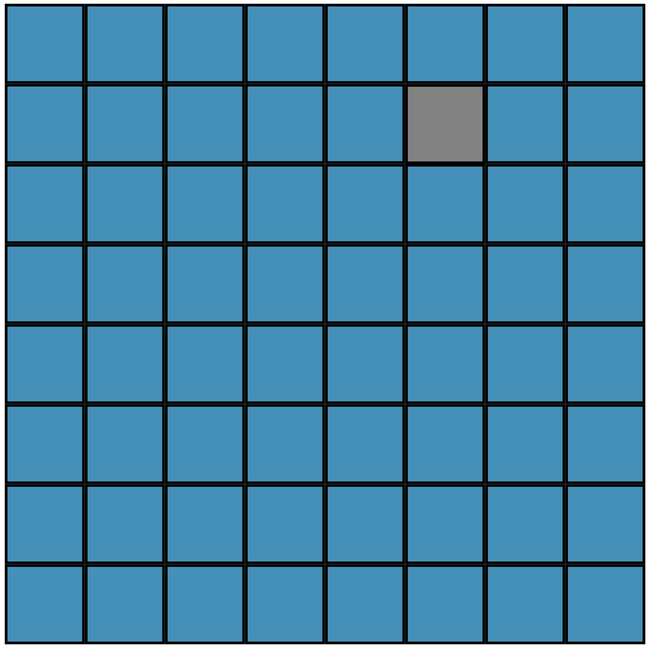
Check if defective position in this region.(在的话，递归调用)

Record triomino t on the board.
否则将棋盘上的这个格子标志为三元板的标号，再递归调用

tr: row of left-upper square
tc: column of left-upper square
dr: row of defective square
dc: column of defective square
$tile$: accumulated triomino number
t: current triomino number

Classroom Exercise （课堂练习）

Write down the triomino number in the following defective chessboard. 在下面的各自里面写上对应的三格板的序号



Classroom Exercise

Solution:

3	3	4	4	8	8	9	9
3	2	2	4	8		7	9
5	2	6	6	10	7	7	11
5	5	6	1	10	10	11	11
13	13	14	1	1	18	19	19
13	12	14	14	18	18	17	19
15	12	12	16	20	17	17	21
15	15	16	16	20	20	21	21

Time Complexity

注意棋盘的大小是 $2^n \times 2^n \ (n \geq 1)$

- The recursive equation is:

$$T(n) = \begin{cases} \Theta(1) & n = 1 \\ 4T(n-1) + c & n > 1 \end{cases}$$

where

$$\begin{aligned} & 4T(n-1) + c \\ = {} & 4[4T(n-2) + c] + c \\ = {} & 4^2T(n-2) + 4c + c \\ = {} & 4^3T(n-3) + 4^2c + 4c + c \\ = {} & 4^{n-1}T(1) + 4^{n-2}c + \ldots + 4c + c \\ = {} & \Theta(4^{n-1}) \end{aligned}$$

DETERMINING THRESHOLD

什么时候不用分治?

Determining Thresholds

- For matrix multiplication and large integer multiplication, when n is small, using standard algorithm will be even faster.

对于矩阵乘或者大数乘，当n更小的时候，用标准的方法来算其实更快

- For Mergesort, using recursive method on small array will also be slower than quadratic sorting algorithm like exchange sort.

对于合并排序，当数据量很少时，他的速度是慢于选择排序的

- How to determine the threshold?

Determining Thresholds

- If we have the recursive equation of Mergesort measured by computational time:

$$T(n) = 32n\lg n \ \mu s$$

and selection sort takes

$$T(n) = \frac{n(n-1)}{2}\mu s$$

- We can compare and get the threshold:

$$\frac{n(n-1)}{2} < 32n\lg n$$

$$n < 591.$$

When Not to Use Divide-and-Conquer

- 什么时候觉得分治法不适用
- An instance of size n is divided into two or more instances each almost of size n.
  - nth Fibonacci term: $T(n) = T(n-1) + T(n-2) + 1$. 分治的子问题和原问题差不多的情况
  - Worst-case Quicksort is also not acceptable: $T(n) = T(n-1) + n - 1$.
- An instance of size n is divided into almost n instances of size n/c, where c is a constant. 或者那么多子问题合起来，还比原问题更大
  - E.g. $T(n) = T(n/2) + T(n/2) + \ldots + T(n/2)$.

Conclusion

After this lecture, you should know:

- What is the key idea of divide-and-conquer.
- How to divide a big problem instance into several small instances.
- How to use recursion to design a divide-and-conquer algorithm.
- How Mergesort and Quicksort work and what are their complexity.

Homework （课后作业）

- Page 63-65

  5.1

Experiment （编程作业，都用分治法）

- LeetCode 35 (二分查找)
- LeetCode 50 (幂函数)
- LeetCode 53 (求最大子数组和，用分治方法，时间复杂度O(nlgn))
- LeetCode 69 (计算Sqrt)
- LeetCode 215 (计算第K个元素，提示用QuickSort的partition函数)

有问题欢迎随时跟我讨论

Classroom Exercise （课堂练习）

What is the best case input for Quicksort when $n = 12$?

Classroom Exercise （课堂练习）

Solution:

The best case occurs when each pivot evenly split the array:

$$1,\ 2,\ 4,\ 5,\ 3,\ 7,\ 8,\ 10,\ 12,\ 11,\ 9,\ 6$$

Think: how to write a best-case input generator for Quicksort?