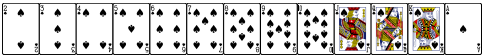
COMP 2011: Data Structures

Lecture 7. Quicksort

Dr. CAO Yixin

yixin.cao@polyu.edu.hk

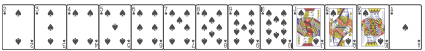
October, 2021



Review: mergesort

- Divide a big problem into smaller problems and solve each one separately.
- Binary search is an example, but mergesort shows the full power of this approach.
- They divide the array in the same way:
  - left half ($\lfloor \frac{n-1}{2} \rfloor$) and right half ($\lfloor \frac{n}{2} \rfloor$) for binary search; finding a peak
  - left half ($\lfloor \frac{n+1}{2} \rfloor$) and right half ($\lfloor \frac{n}{2} \rfloor$) for mergesort. finding max/min
- For iterative implementation: bottom-up.

- What are the best cases and worst cases of mergesort?
- Is there another way to divide an array?

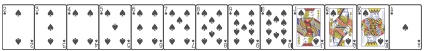


Review questions

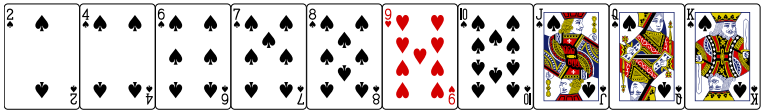
Of these sorting algorithms: bubble, insertion, selection, and mergesort, which are good for sorting an array that is

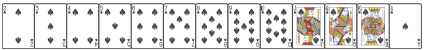
- sorted $1, 2, \ldots, n$
- reversely sorted $n, n-1, \ldots, 1$
- both sorted and reversely sorted $5, 5, \ldots, 5$
- almost sorted $1, 9, 3, 4, 5, 6, 7, 8, 2, 10$
- random

Demo Comparison@wikipedia



There is an array of n elements, which is originally sorted.

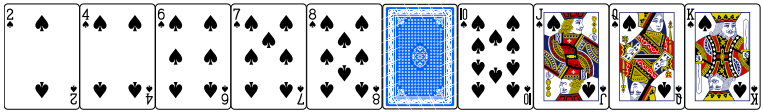


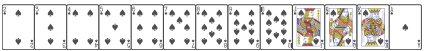


There is an array of n elements, which is originally sorted.

Now, the value of one element in it is modified, but you don't know which one.

Which sorting algorithm you want to use to make it sorted again?

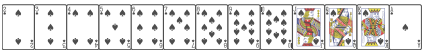




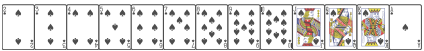
Overview

- Quicksort: Another sorting algorithm based on divide and conquer.
- Implementation.
  - easy if using extra space
  - can be made in-place, very nontrivial.
- widely used*: in-place but instable.
- several variations: the simple version is very inefficient.

$\star$: Quicksort is stupidly slow so it has never been used. What are commonly used are hybrid implementations of quicksort and insertion sort.

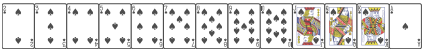


Partition



Ladies first

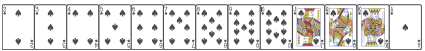
Rearrange the roster so all girls come before boys.



Ladies first

girls ⟶

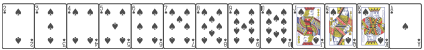
⟵ boys



Ladies first

girls ⟶

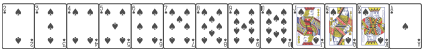
⟵ boys



Ladies first

girls ⟶

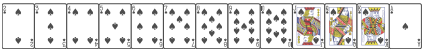
⟵ boys



Ladies first

girls ⟶

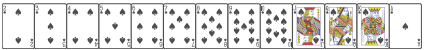
⟵ boys



Ladies first

girls ⟶

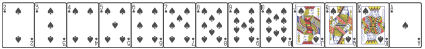
⟵ boys



Ladies first

girls ⟶

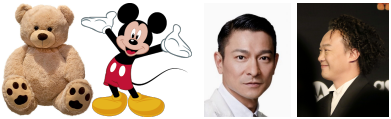
⟵ boys

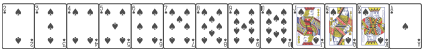


Ladies first

girls ⟶

⟵ boys

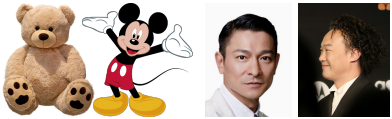


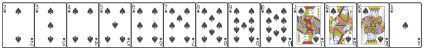


Ladies first

girls ⟶

⟵ boys

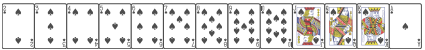




Ladies first

girls ⟶

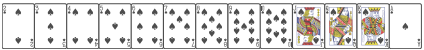
⟵ boys



Ladies first

girls ⟶

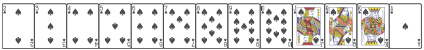
⟵ boys



Ladies first

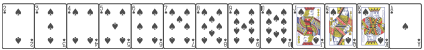
girls ⟶

⟵ boys

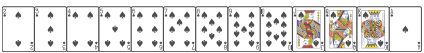


Ladies first

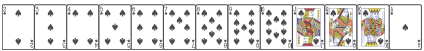
girls ⟶ ⟵ boys



```
1  void binarySort(Student[] a) {
2      int n = a.length;
3      Student[] b = new Student[n];
4      for (int i = 0; i < n; i++) b[i] = a[i];
5
6      int girls = 0, boys = n - 1;
7      for (int i = 0; i < b.length; i++) {
8          if (b[i].gender == 'F') a[girls++] = b[i];
9          else a[boys--] = b[i];
10     }
11 }
```



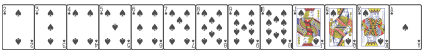
Can we do it in-place?



- Can you generalize the binary sort so that we allow non-binary gender?
- Can we sort student grades, where we have only A, B, C, and D?

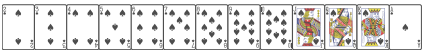
- In other words, we sort an array in which there are only three different keys?

They can be done in linear time, with the same appoach. But we cannot directly use the idea of binarySort.



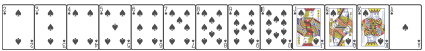
```
1  void sortGrades(Student[] a) {
2      int n = a.length;
3      Student[] copy = new Student[n];
4      for(int i = 0; i<n; i++) copy[i] = a[i];
5
6      int count[] = {0, 0, 0, 0};
7      for(Student c: a) count[c.grade - 'A']++;
8      int cur[] = {0, 0, 0, 0};
9      cur[1] = count[0];
10     cur[2] = cur[1] + count[1];
11     cur[3] = cur[2] + count[2];
12     for(int i = 0; i < n; i++)
13         a[cur[copy[i].grade - 'A']++] = copy[i];
14 }
```

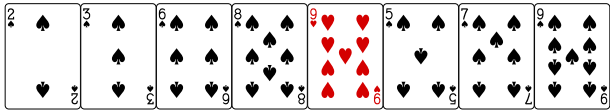
Quicksort

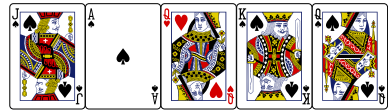


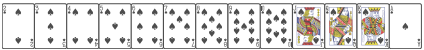
How old were you when you first play cards?

when my hands were too small to hold too many cards.







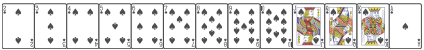


$17, 45, 38, 48, 18, 55$

< 60

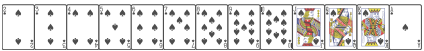
$65, 85, 85, 66, 71, 95, 60, 68, 96$

≥ 60



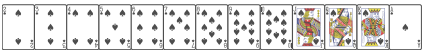
[35000, 18000, 56600, 19175, 32000, 46325, 17250, 56000, 15500, 600000]

[48.8, 52.1, 70.5, 82.4, 77.5, 51.1, 79.8, 97.3, 55.2, 45.2, 72.4, 68.4, 65.3]

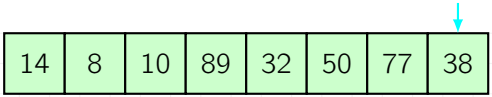


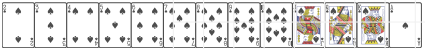
[35000, 18000, 56600, 19175, 32000, 46325, 17250, 56000, 15500, 600000]
HKD/m

[48.8, 52.1, 70.5, 82.4, 77.5, 51.1, 79.8, 97.3, 55.2, 45.2, 72.4, 68.4, 65.3]
KG

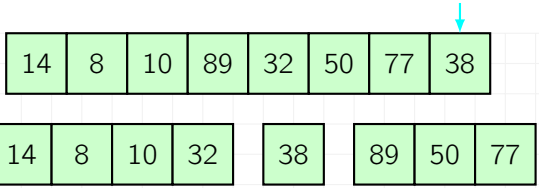


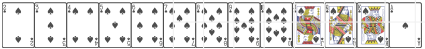
Demonstration of quicksort



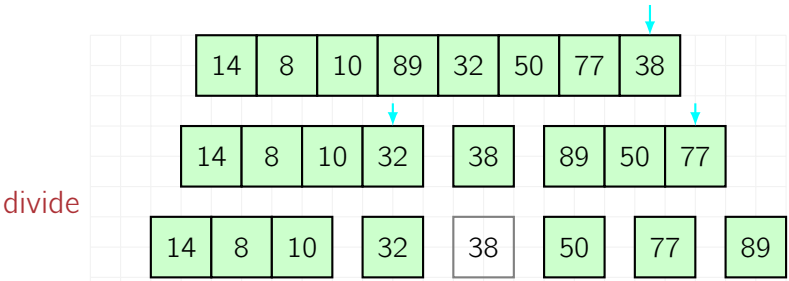


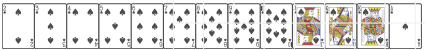
Demonstration of quicksort



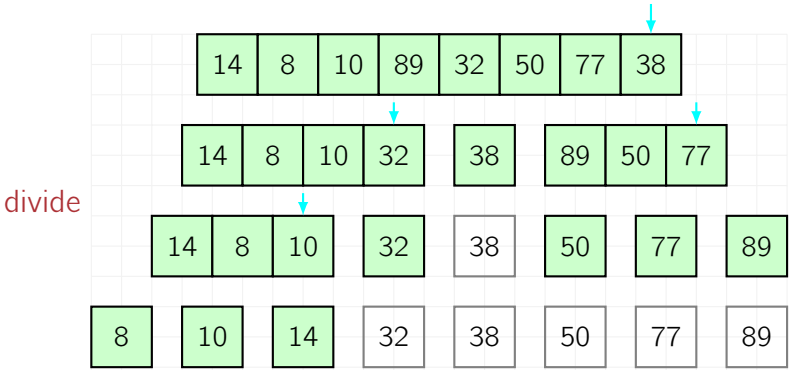


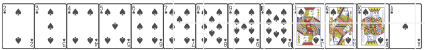
Demonstration of quicksort



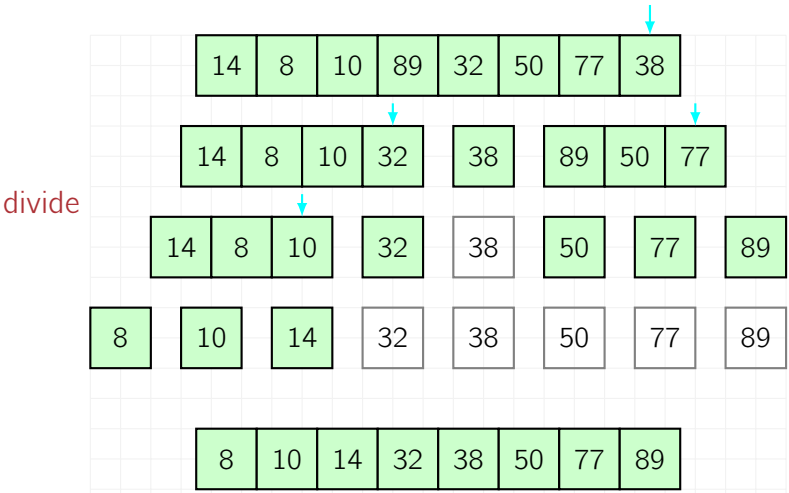


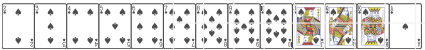
Demonstration of quicksort





Demonstration of quicksort



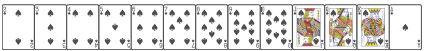


Quicksort ()

In every step, we choose a pivot p, and divide the array into two parts such that

- every element in the left is no larger than p and
- every element in the right is larger than p.

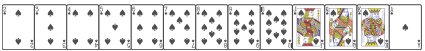
Where is p itself?



Recursive quicksort

```
1  void naive(int[] a, int low, int high) {
2      if(low >= high) return;
3      int pivot = a[high];
4      int[] b = new int[high - low + 1];
5      for (int i=0; i<b.length; i++) b[i] = a[low+i];
6
7      int l = low, r = high;
8      for (int i = 0; i < b.length; i++) {
9          if (b[i] <= pivot) a[l++] = b[i];
10         else a[r--] = b[i];
11     }
12     naive(a, low, l - 2); // also ok r - 1.
13     naive(a, r + 1, high);   // also ok r.
14 }
```

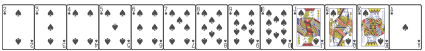
Common mistake: use $l-1$ instead of $l-2$ in line 12



Analysis of quicksort

It's a divide-and-conquer algorithm, so we consider #levels and work at each level.

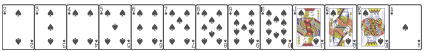
- The work of each partition is proportional to the number of elements under concern, so each level uses $O(n)$ time.
- The running time of the whole algorithm depends on #levels (iterations),
- which depends on whether the partitions are balanced,
- which depends on the pivot values p chosen in each step.
- So, if we always choose the last element as pivot, the worst cases are _____?
- What are the best cases?



Analysis of quicksort

It's a divide-and-conquer algorithm, so we consider #levels and work at each level.

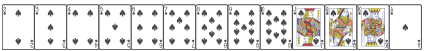
- The work of each partition is proportional to the number of elements under concern, so each level uses $O(n)$ time.
- The running time of the whole algorithm depends on #levels (iterations),
- which depends on whether the partitions are balanced,
- which depends on the pivot values p chosen in each step.
- So, if we always choose the last element as pivot, the worst cases are _____?
- What are the best cases?



Analysis of quicksort

It's a divide-and-conquer algorithm, so we consider #levels and work at each level.

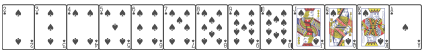
- The work of each partition is proportional to the number of elements under concern, so each level uses $O(n)$ time.
- The running time of the whole algorithm depends on #levels (iterations),
- which depends on whether the partitions are balanced,
- which depends on the pivot values p chosen in each step.
- So, if we always choose the last element as pivot, the worst cases are _____?
- What are the best cases?



Analysis of quicksort

It's a divide-and-conquer algorithm, so we consider #levels and work at each level.

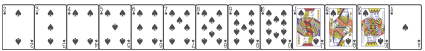
- The work of each partition is proportional to the number of elements under concern, so each level uses $O(n)$ time.
- The running time of the whole algorithm depends on #levels (iterations),
- which depends on whether the partitions are balanced,
- which depends on the pivot values p chosen in each step.
- So, if we always choose the last element as pivot, the worst cases are _____?
- What are the best cases?



Analysis of quicksort

It's a divide-and-conquer algorithm, so we consider #levels and work at each level.

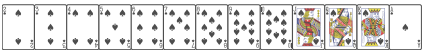
- The work of each partition is proportional to the number of elements under concern, so each level uses $O(n)$ time.
- The running time of the whole algorithm depends on #levels (iterations),
- which depends on whether the partitions are balanced,
- which depends on the pivot values p chosen in each step.
- So, if we always choose the last element as pivot, the worst cases are _____?
- What are the best cases?



Analysis of quicksort

It's a divide-and-conquer algorithm, so we consider #levels and work at each level.

- The work of each partition is proportional to the number of elements under concern, so each level uses $O(n)$ time.
- The running time of the whole algorithm depends on #levels (iterations),
- which depends on whether the partitions are balanced,
- which depends on the pivot values p chosen in each step.
- So, if we always choose the last element as pivot, the worst cases are sorted arrays.
- What are the best cases?

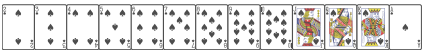


Analysis of quicksort

It's a divide-and-conquer algorithm, so we consider #levels and work at each level.

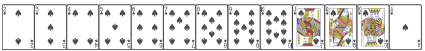
- The work of each partition is proportional to the number of elements under concern, so each level uses $O(n)$ time.
- The running time of the whole algorithm depends on #levels (iterations),
- which depends on whether the partitions are balanced,
- which depends on the pivot values p chosen in each step.
- So, if we always choose the last element as pivot, the worst cases are sorted arrays.
- What are the best cases?

How to find good pivots?



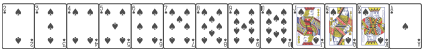
The worst cases

$$n \to n-1 \to n-2 \to n-3 \to \cdots \to 1.$$



The worst cases: sorted arrays

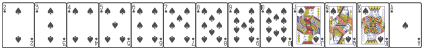
0	1	2	3	4	5	6	7	8	9



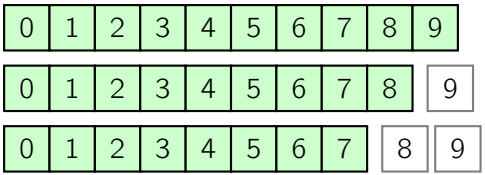
The worst cases: sorted arrays

0	1	2	3	4	5	6	7	8	9

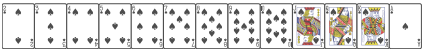
0	1	2	3	4	5	6	7	8	9



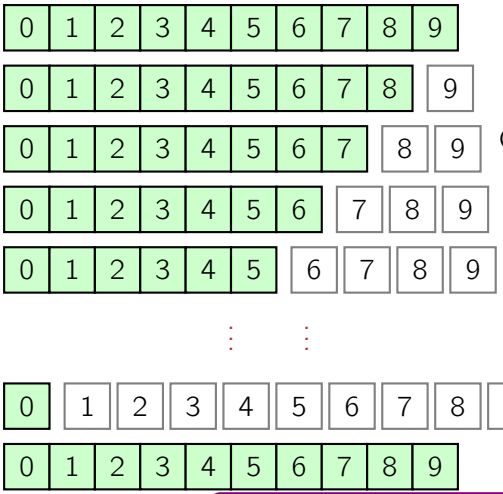
The worst cases: sorted arrays



Only one part in every partition step.



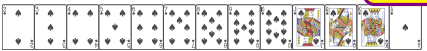
The worst cases: sorted arrays



Only one part in every partition step.

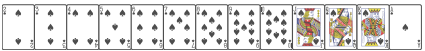
The execution is exactly the same as selection sort (selecting the largest element)

Write the execution of quicksort on a reversely sorted array.



The best cases

- In the worst cases, quicksort takes $\Theta(n^2)$ time.
- What are the best cases for mergesort and quicksort?

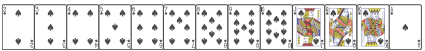


The best cases

- In the worst cases, quicksort takes $\Theta(n^2)$ time.

- What are the best cases for mergesort and quicksort?
- The best cases of quicksort behave similar as mergesort:
  - each partition is almost even
  - $\log n$ levels
  - $O(n)$ work at each level
  - e.g., $1,3,2,6,7,5,4,12,13,15,14,10,11,9,8^\star$. Try to write the execution process.
- Even in the best case, the running time is $\Theta(n \log n)$.

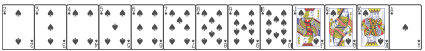
$\star$: This is only good to our naïve implementation.



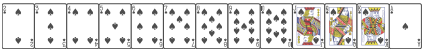
Comparisions of mergesort and quicksort

	mergesort	quicksort
divide	$O(1)$	$\Theta(n)$
combine	$\Theta(n)$	0
levels	$\Theta(\log n)$	$\log n \sim n-1$
best running time	$\Theta(n \log n)$	$\Theta(n \log n)$
average running time	$\Theta(n \log n)$	$\Theta(n \log n)$
worst running time	$\Theta(n \log n)$	$\Theta(n^2)$

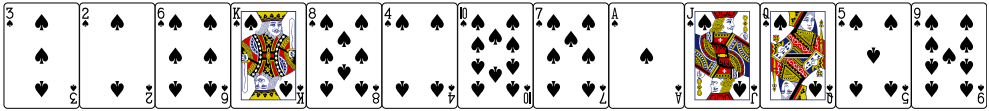
Demo1 Demo2



In-place Implementations

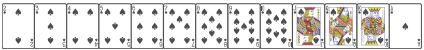


In quicksort, the only thing we need to do is to divide.

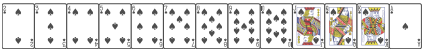


We always use the last element as pivot, so 9.

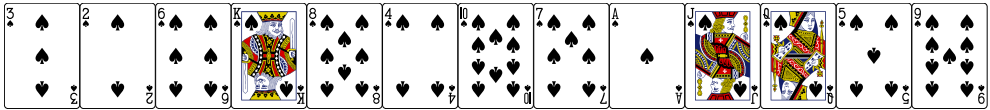
Optional and strongly suggested.

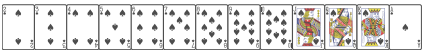


Lomuto partition scheme

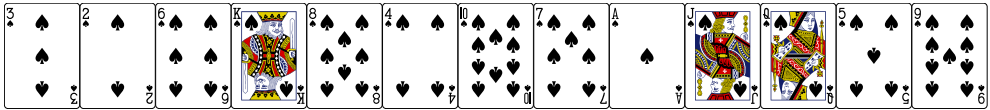


1

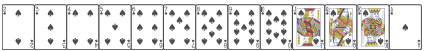




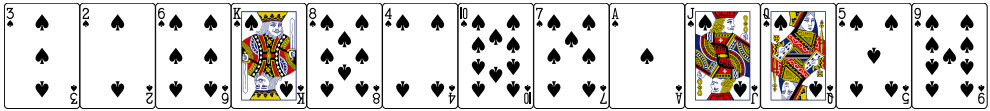
1



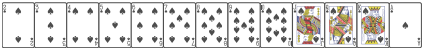
We need to maintain two indices:
i points to the card under consideration;
j points to the first large ($> p$) card.
Both initially point to the first card.



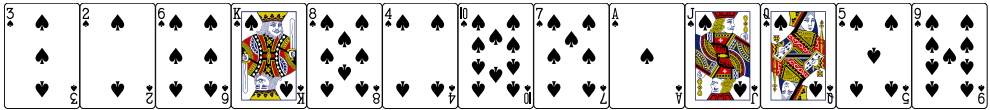
1



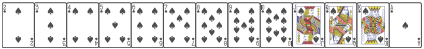
If $a[i] \leq$pivot, then we swap it with $a[j]$ and increase j by 1.
i is always increased by 1 in each iteration.



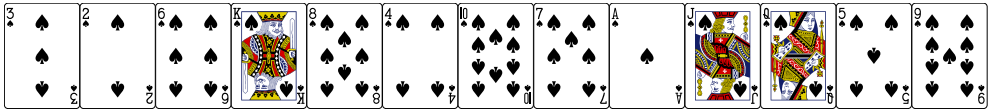
1



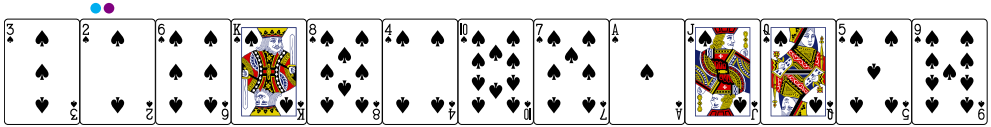
You may want to avoid the stupid swapping of $a[i]$ and $a[j]$ when $i == j$, but adding `if (i != j)` is a bad idea. WHY?

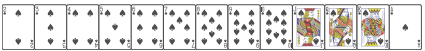


1



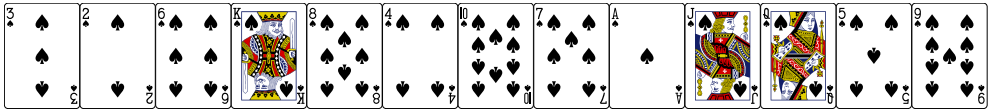
2



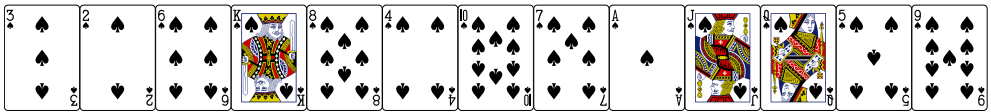


likewise…

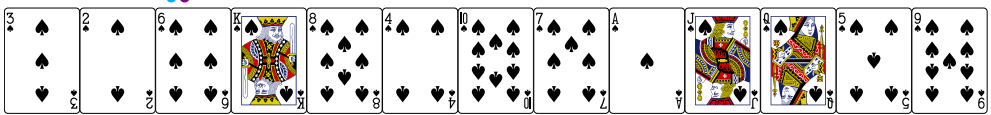
1

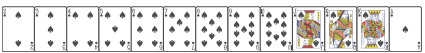


2



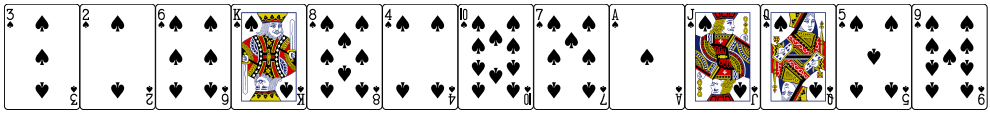
3



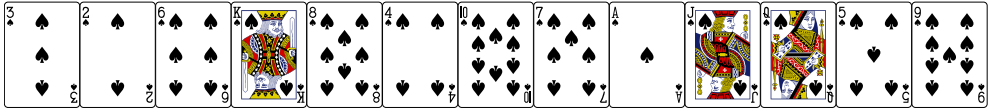


likewise…

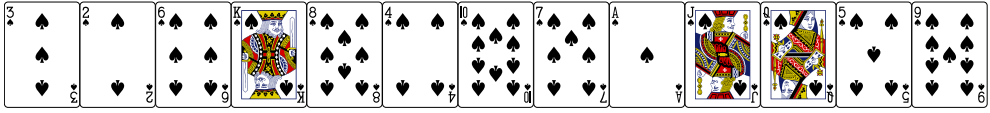
1



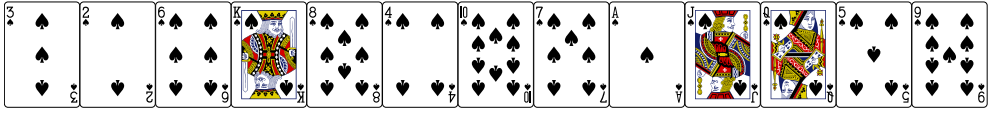
2



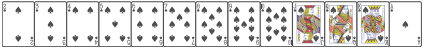
3



4

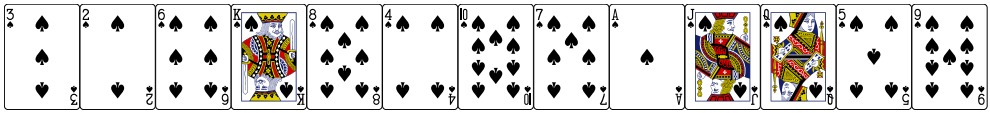


This time we don't swap, so i moves but j stays.

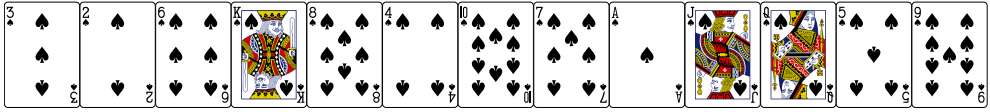



1
2
3
4
5

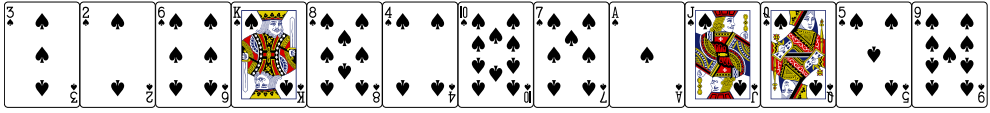
1



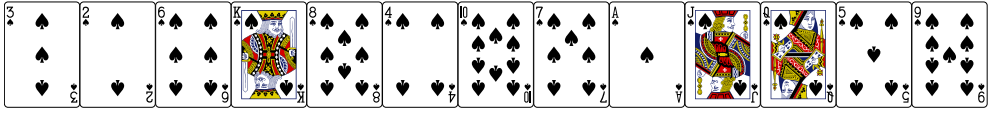
2



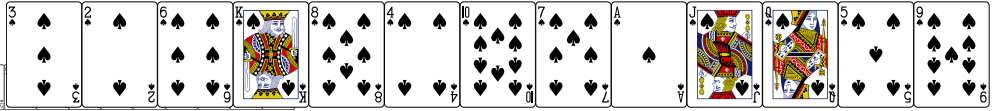
3

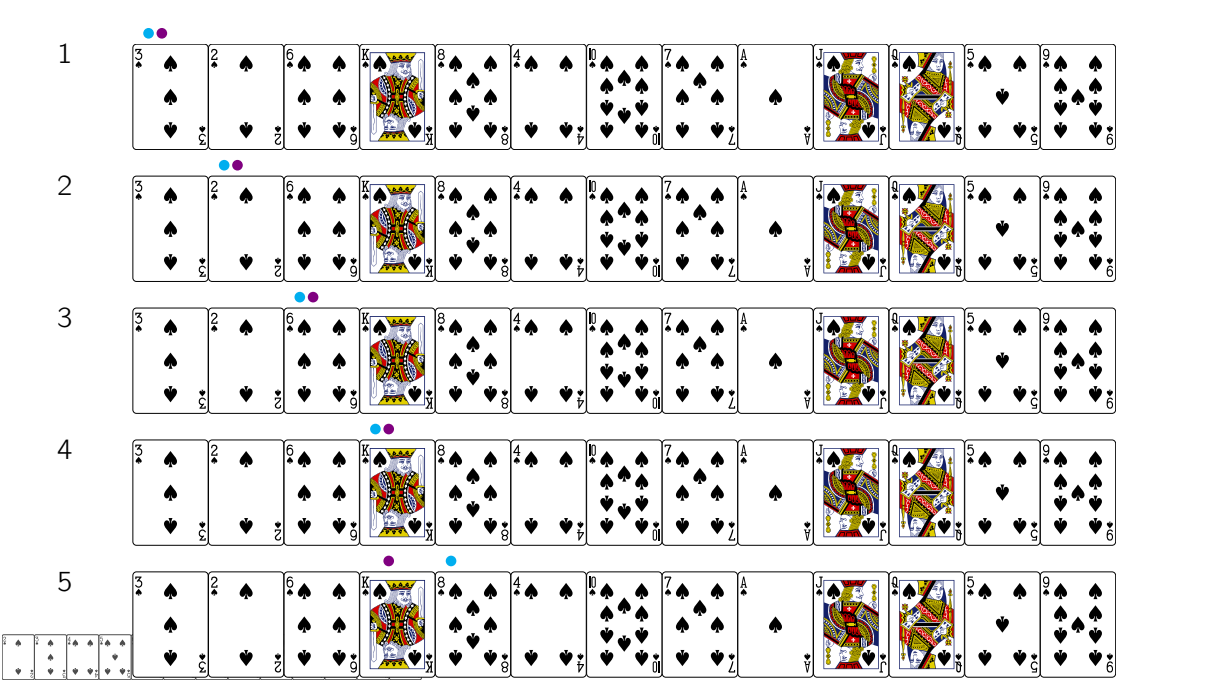


4

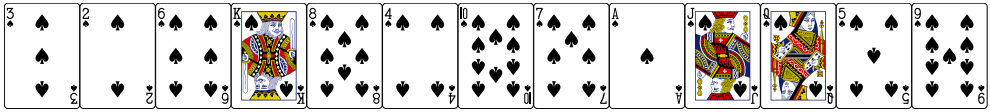


5

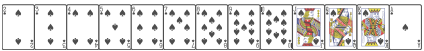



1
2
3
4
5

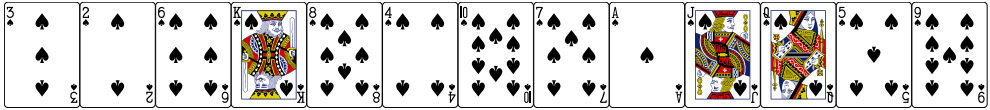
5



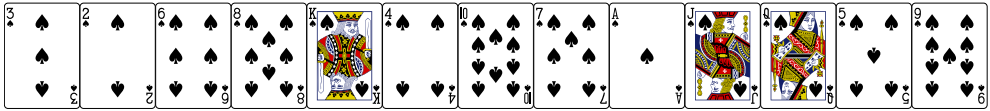
We need to swap, so both i and j move.

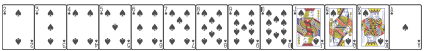


5



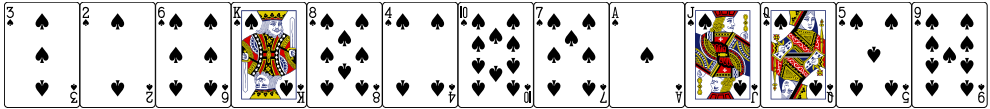
6



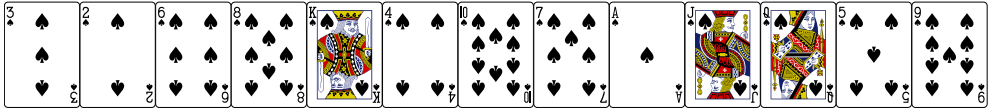


again

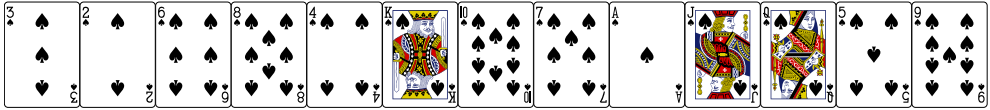
5

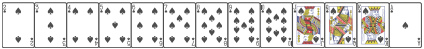


6



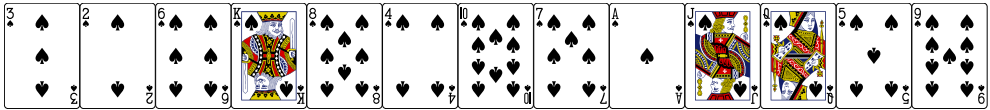
7



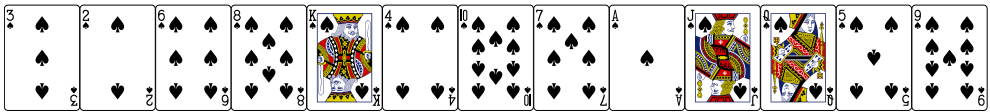


no swap, so...

5



6



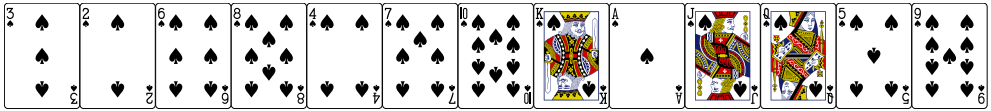
7

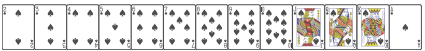
8

swap again


5
6
7
8
9

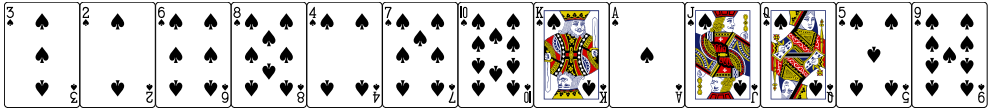
9



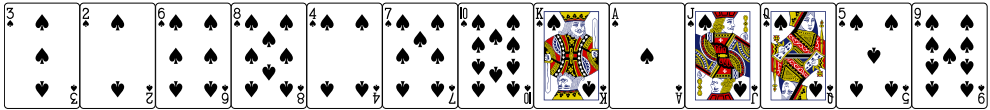


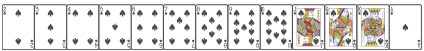
no swap

9



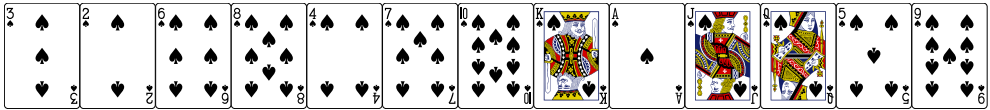
10



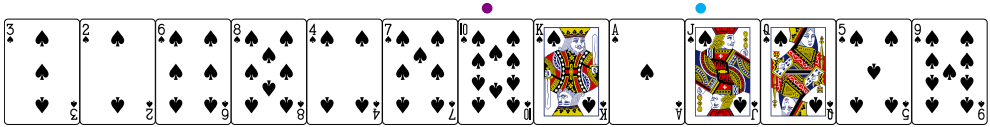


no swap

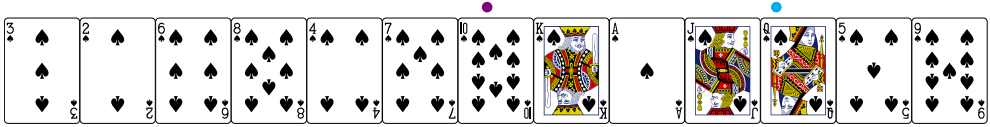
9

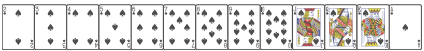


10



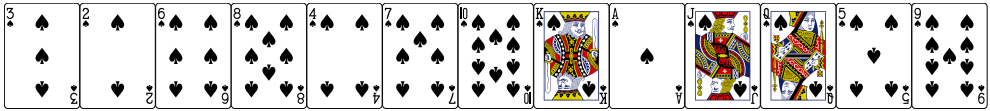
11



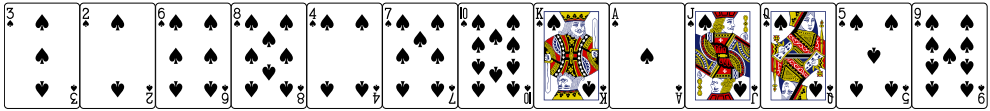


no swap

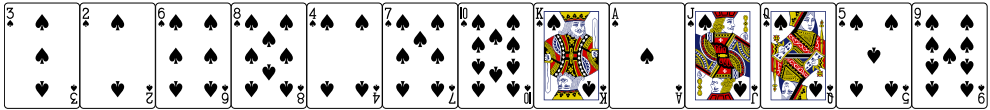
9



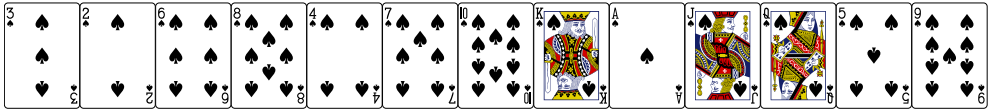
10

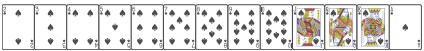


11



12

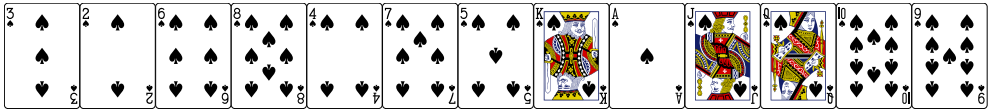


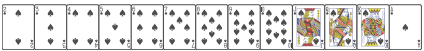


swap again

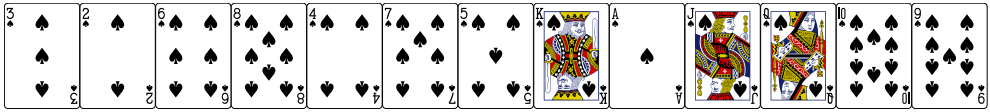

9
10
11
12
13

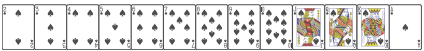
13





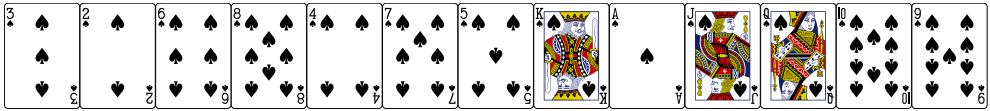
13



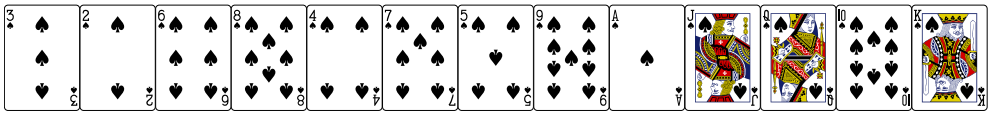


The last one must be swapped. WHY?

13

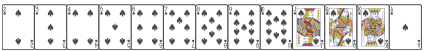


14

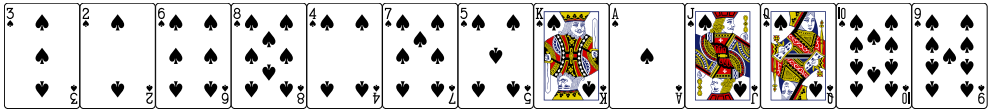


j

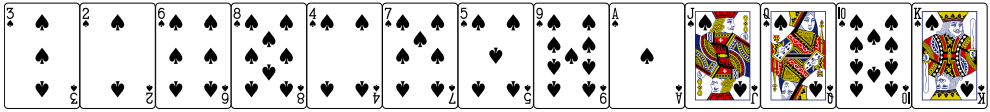
The array is divided into two parts, what are they?



13

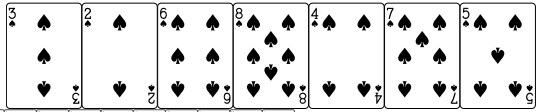


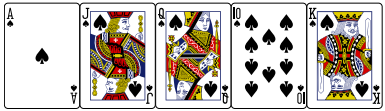
14

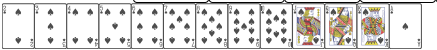


j

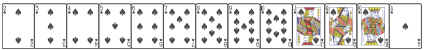
The array is divided into two parts, what are they?



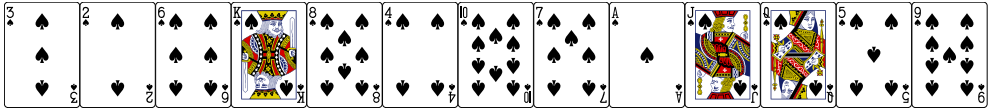




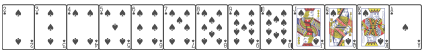
Hoare partition scheme



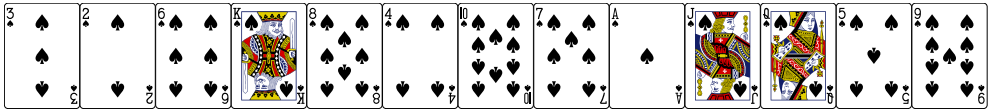
1



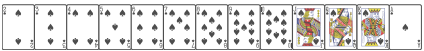
The original partition scheme by C.A.R. Hoare, the “inventor” of quicksort.



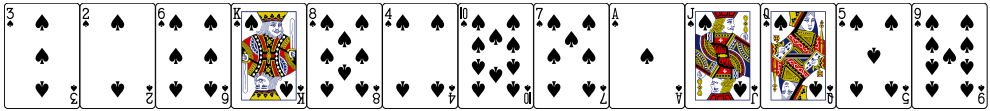
1



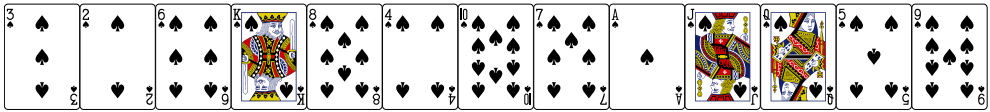
We maintain also two indices:
i points to the first large card ($> p$);
j points to the last small card ($\leq p$).
Initially `i = low` and `j = high`.

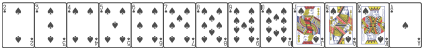


1



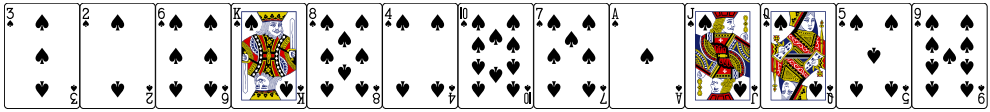
2



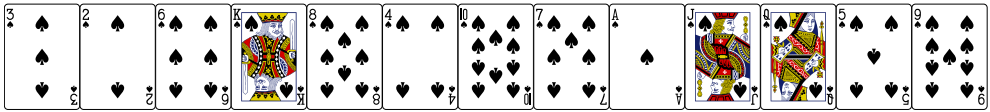


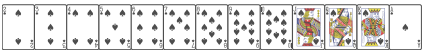
i is increased till...the king.

1



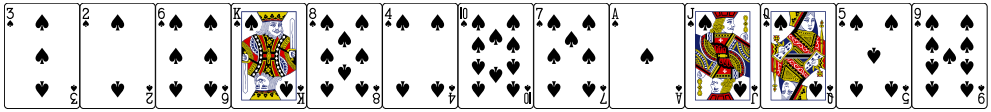
2



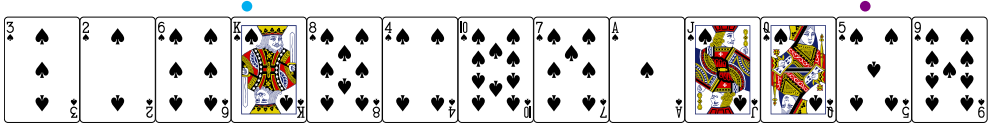


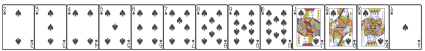
j is decreased till...?

1



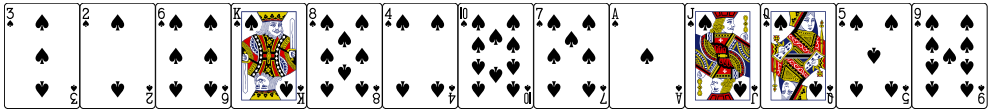
2



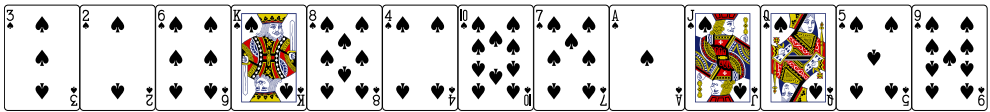


swap them.

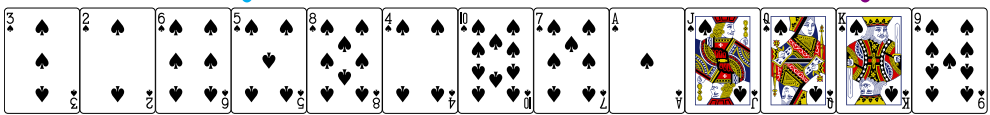
1

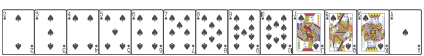


2

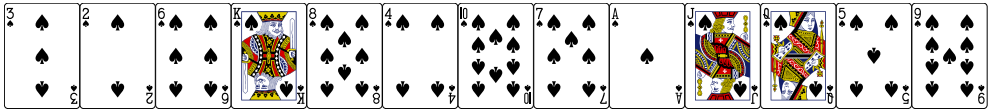


3

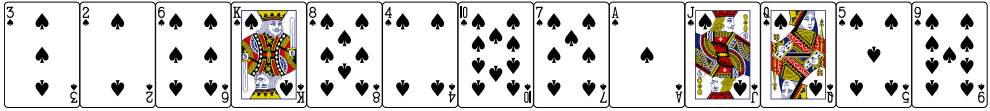




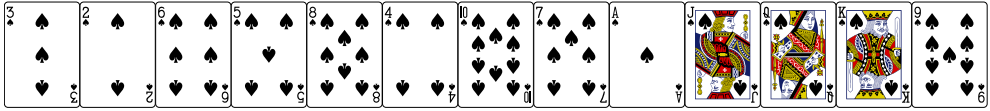
1

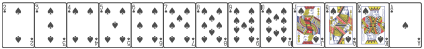


2



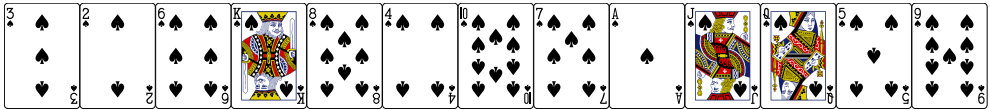
3



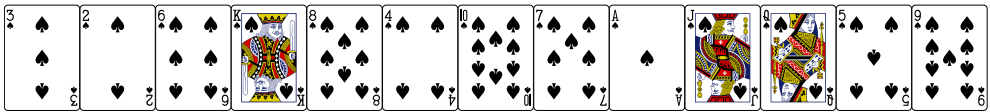


again, increased i and decrease j.

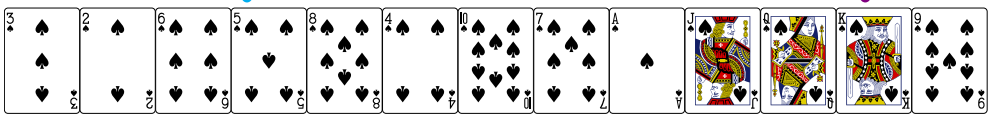
1



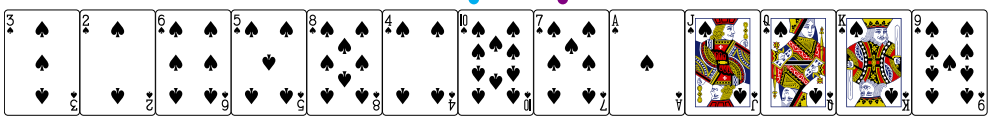
2

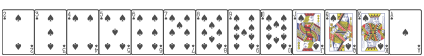


3

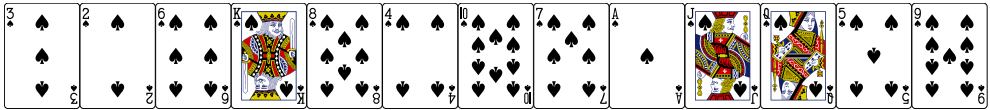


4

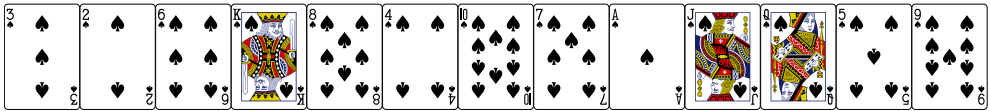




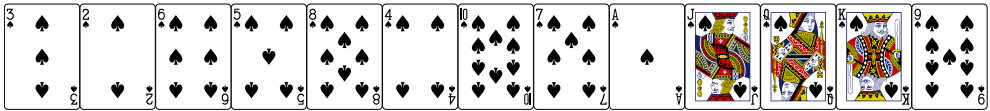
1



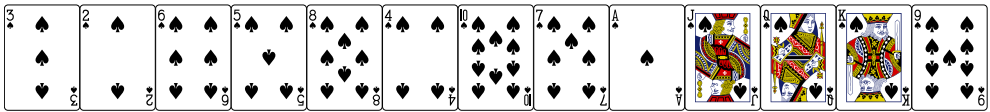
2

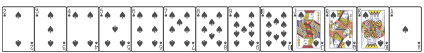


3

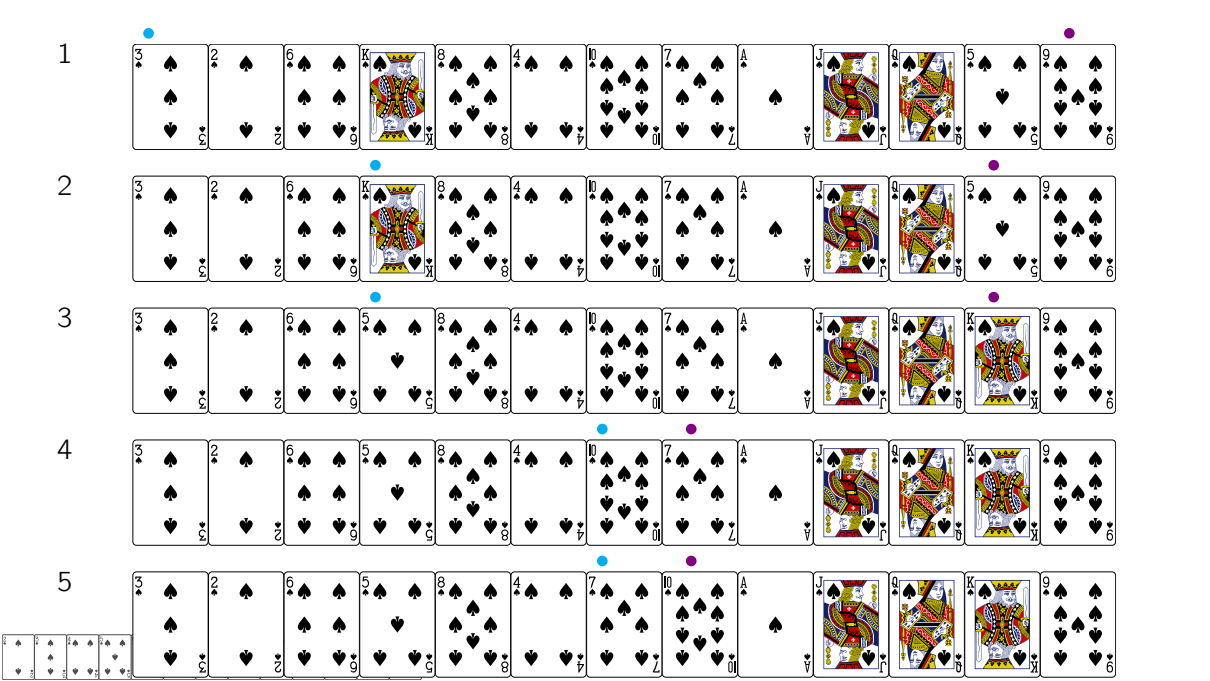


4

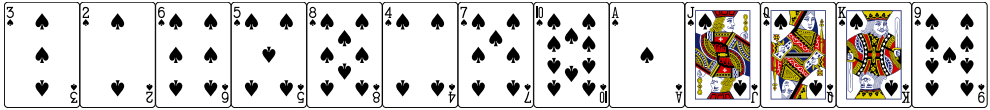


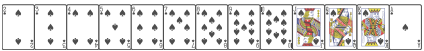


swap and repeat...

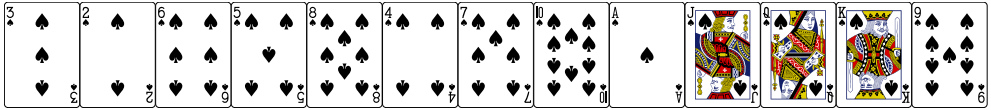

1
2
3
4
5

5

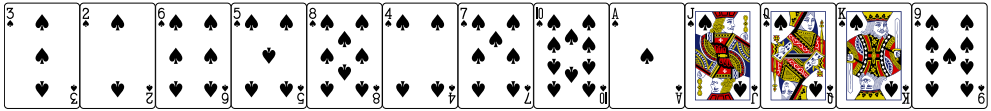


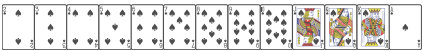


5



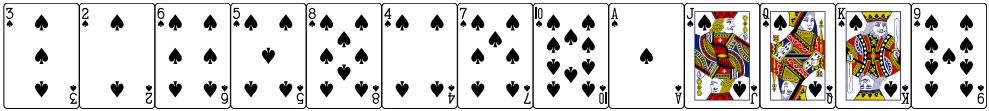
6



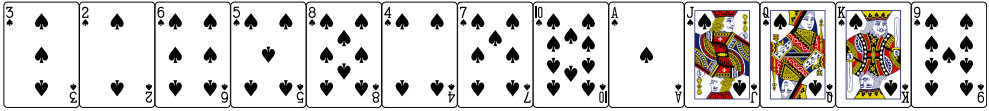


Where should j go to?

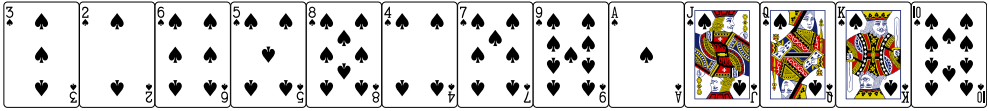
5

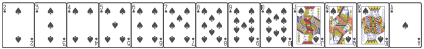


6



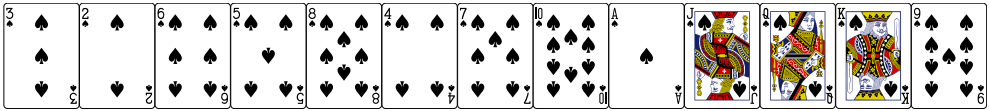
6



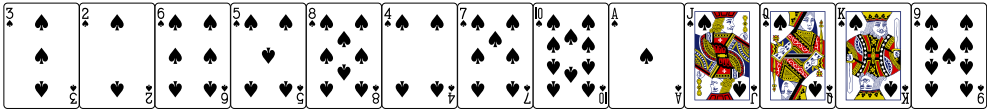


Nowhere, we swap a[j] with the pivot, and then stop.

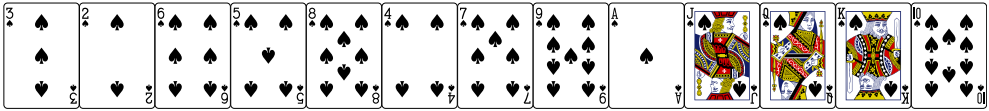
5



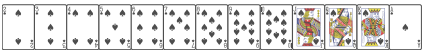
6



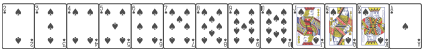
6



There are many many variations of this old algorithm. Try to think as many as possible different implementations



Improvements

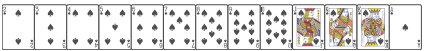


Real-life "quicksort"

Choice of "good" pivots (🌑)

- Random pivot: choosing a random index for the pivot.
  The worst case remains the same, but you shouldn't be always unlucky.
- Median-of-three pivot: choosing the median of the first, middle and last element of the partition for the pivot (recommended by Sedgewick).
  good for sorted array, but bad cases still exist. $1, x, \ldots, x, 1, y, \ldots, y, 1$
- Dual pivots; i.e., partition into three parts. Since J2SE 7 (2007)
- Use insertion sort when the partition is small.

Why not $3, 4, \ldots$ pivots?

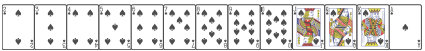


Real-life "quicksort"

Choice of "good" pivots (🎲)

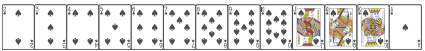
- Random pivot: choosing a random index for the pivot.
  The worst case remains the same, but you shouldn't be always unlucky.
- Median-of-three pivot: choosing the median of the first, middle and last element of the partition for the pivot (recommended by Sedgewick).
  good for sorted array, but bad cases still exist. $1, x, \ldots, x, 1, y, \ldots, y, 1$
- Dual pivots; i.e., partition into three parts. Since J2SE 7 (2007)
- Use insertion sort when the partition is small.

Why not $3, 4, \ldots$ pivots?



Summary

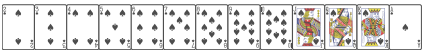
- Choose a pivot p, and then partition the array to two parts, $\leq p$ and $> p$, followed by two recursive calls.
- A naïve implementation is to use a temporary array.
- After the partition, the pivot is in neither part. It's already in correct position!
- In the Hoare partitioning scheme,
  - two indices, each in its own while loop, start at opposite ends of the array and step toward each other, looking for items that need to be swapped.
  - When an index finds an item that needs to be swapped, its while loop exits.
  - When both while loops exit, the items are swapped.
  - When both while loops exit, and the indices have met or passed each other, the partition is complete.
  - You need extra tess to prevent the indices running off the ends of the array.
- In a more advanced version of quicksort, the pivot can be chosen in a smart way.
- Change to insertion sort when the subarray is small enough.



Warning

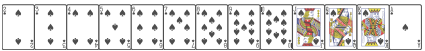
In this subject, by quicksort we always mean an in-place implementation, with the last element as the pivot.

In other context, you need to be careful on the meaning of quicksort.



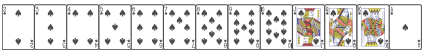
Why is sorting worth so much attention?

- Sorting is the basic building block that many other algorithms are built around. By understanding sorting, we obtain an amazing amount of power to solve other problems.
- Most of the interesting ideas used in the design of algorithms appear in the context of sorting, such as divide-and-conquer, data structures, and randomized algorithms.
- Computers have historically spent more time sorting than doing anything else. A quarter of all mainframe cycles were spent sorting data [Knuth]. Sorting remains the most ubiquitous combinatorial algorithm problem in practice.
- Sorting is the most thoroughly studied problem in computer science. Literally dozens of different algorithms are known, most of which possess some particular advantage over all other algorithms in certain situations.



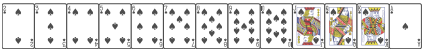
True or false

- Quicksort is quick(er than other sorting algorithms).

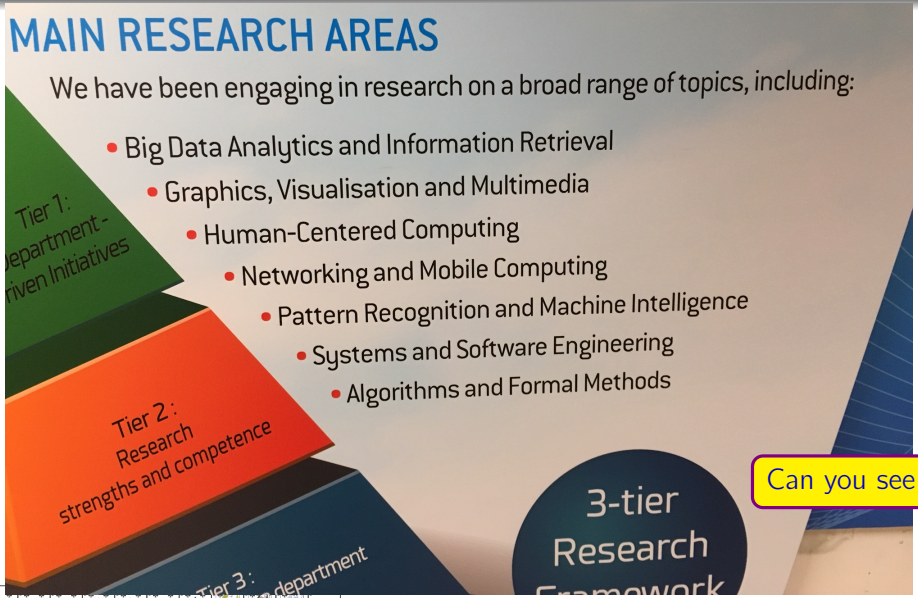


True or false

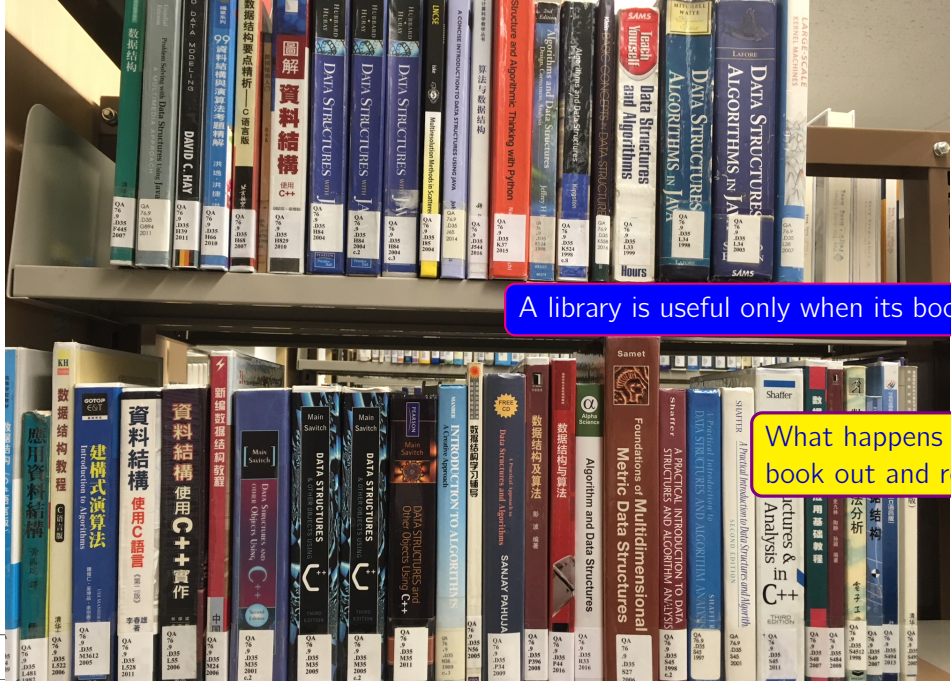
- Quicksort is quick(er than other sorting algorithms).
- Democratic People's Republic of Korea (North) is more democratic than Republic of Korea (South).



Exhibition board out of GO



Can you see the problem?

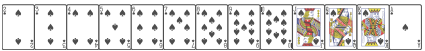

A library is useful only when its books are sorted.
What happens if you check a book out and return it later?

The question again

bubble, insertion, selection, mergesort, quicksort

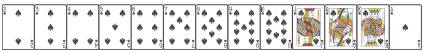
Which sorting algorithms are good for sorting an array that is

- sorted $1, 2, \ldots, n$
- reversely sorted $n, n-1, \ldots, 1$
- both sorted and reversely sorted $5, 5, \ldots, 5$
- almost sorted $1, 9, 3, 4, 5, 6, 7, 8, 2, 10$
- random



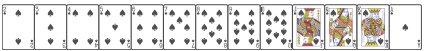
You sort (almost) sorted arrays more often than you thought

- Sorting twice doesn't make your array more "sorted," and can be 10 times time consuming than sorting once!
- We don't sort a sorted array often, at least not intentionally.
- But, how about unintentionally?
  - You're calling method `customerList()` written by Ben.
  - Since the list was not sorted, you sort it before using.
  - Later, Ben thinks it is more convenient to sort it.
  - Now, you're sorting a sorted array.
  - (Is it a good idea to remove the sorting in your side?)
- Most of the time, we are sorting an "almost sorted" array.
  - Even in a busy website as TMall, only millions of users transact on a normal day.
  - A small number (10^7) of changes among a huge number (10^{10}) of items.
  - It's similar as sorted.
- Insertion is the best; cocktail sort (), bidirectional bubble sort, is not too bad.



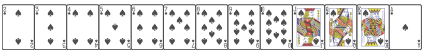
You sort (almost) sorted arrays more often than you thought

- Sorting twice doesn't make your array more "sorted," and can be 10 times time consuming than sorting once!
- We don't sort a sorted array often, at least not intentionally.
- But, how about unintentionally?
  - You're calling method `customerList()` written by Ben.
  - Since the list was not sorted, you sort it before using.
  - Later, Ben thinks it is more convenient to sort it.
  - Now, you're sorting a sorted array.
  - (Is it a good idea to remove the sorting in your side?)
- Most of the time, we are sorting an "almost sorted" array.
  - Even in a busy website as TMall, only millions of users transact on a normal day.
  - A small number (10^7) of changes among a huge number (10^{10}) of items.
  - It's similar as sorted.
- Insertion is the best; cocktail sort (), bidirectional bubble sort, is not too bad.



You sort (almost) sorted arrays more often than you thought

- Sorting twice doesn't make your array more "sorted," and can be 10 times time consuming than sorting once!
- We don't sort a sorted array often, at least not intentionally.
- But, how about unintentionally?
  - You're calling method `customerList()` written by Ben.
  - Since the list was not sorted, you sort it before using.
  - Later, Ben thinks it is more convenient to sort it.
  - Now, you're sorting a sorted array.
  - (Is it a good idea to remove the sorting in your side?)
- Most of the time, we are sorting an "almost sorted" array.
  - Even in a busy website as TMall, only millions of users transact on a normal day.
  - A small number (10^7) of changes among a huge number (10^{10}) of items.
  - It's similar as sorted.
- Insertion is the best; cocktail sort (), bidirectional bubble sort, is not too bad.



Week 8: Mid-term
Week 9: Trees