COMP 2011: Data Structures

Lecture 6. Sorting Algorithms

Dr. CAO Yixin

yixin.cao@polyu.edu.hk

October, 2021

Review of Lecture 5

- Data structures are intrinsically recursive.
- Recursive algorithms are easy to write and easy to read.
- Recursive algorithms usually use more space, hence less efficient.
- Some recursive algorithms can be easily translated, but some are not.

Implementation of recursion

- When method A (caller) calls method B (callee), the return address of A is pushed onto the stack, and control is passed to B. When B finishes, the return address is popped off the stack and A resumes control. wikipedia and Quora.

- Recursive calls thus make the stack grow very fast.

  Optimized compilers are able to get rid of some recursions. (more details)

- A recursion can always be mechanically simulated by manually maintaining the call stack. But it is too error-prone.
  - When write iterative versions, we seldom do this way.
  - Instead, we take a problem-specific approach (e.g., Fibonacci).
- For recursion for divide and conquer, the bottom-up approach usually works.

Tip for designing recursive algorithms

Think what your algorithm does, instead of how it is done.

(We will see several examples today.)

Nothing is difficult,

as long as you keep doing practice, practice, practice.

Merge

Merging, the problem

8	10	14	89

32	38	50	77

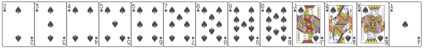
Merge two sorted arrays into a single sorted array.

5	6	7	8

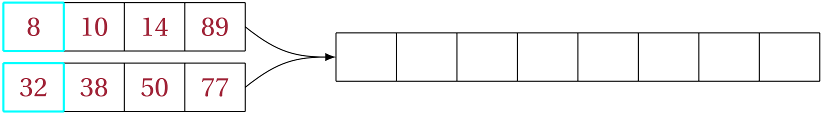
1	2	3	4

The ideas of bubble and insertion don't seem to be useful here.

How about selection?

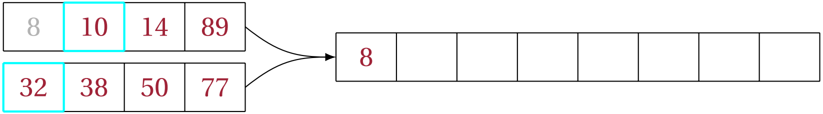


Merging, demonstration



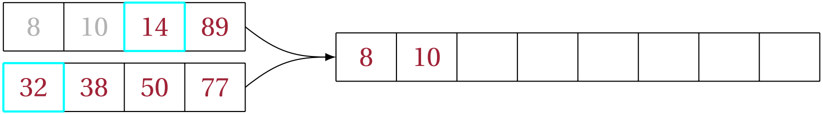
One of the first of the two arrays is a smallest.

Merging, demonstration

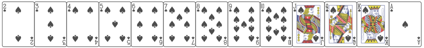


One of the first of the two arrays is a smallest.

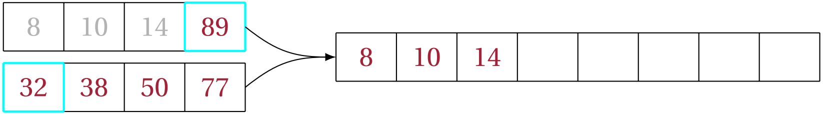
Merging, demonstration



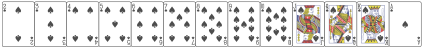
One of the first of the two arrays is a smallest.



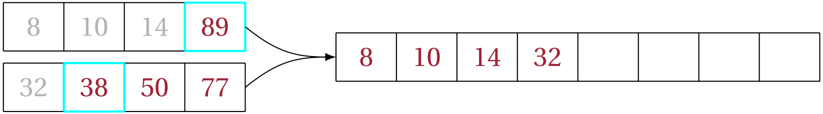
Merging, demonstration



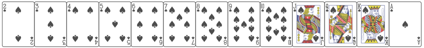
One of the first of the two arrays is a smallest.



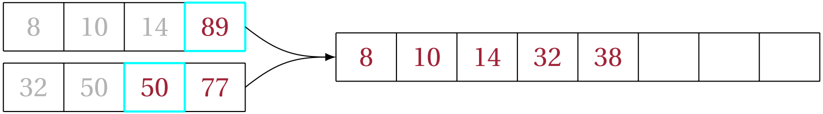
Merging, demonstration



One of the first of the two arrays is a smallest.

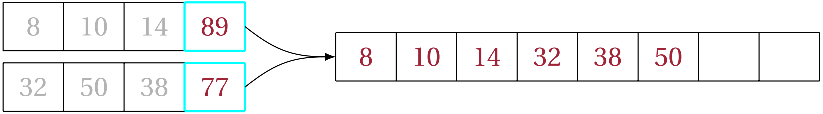


Merging, demonstration

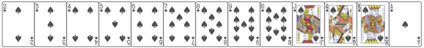


One of the first of the two arrays is a smallest.

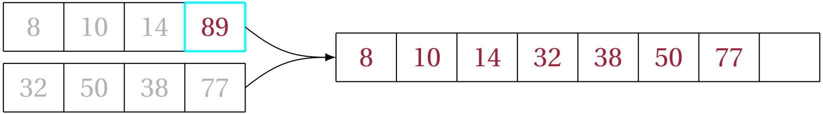
Merging, demonstration



One of the first of the two arrays is a smallest.



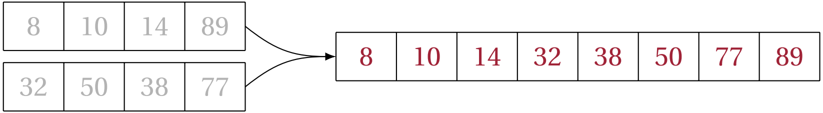
Merging, demonstration



One of the first of the two arrays is a smallest.

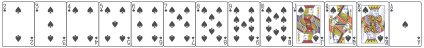
After one array is exhausted, copy all the leftovers of the other.

Merging, demonstration



One of the first of the two arrays is a smallest.

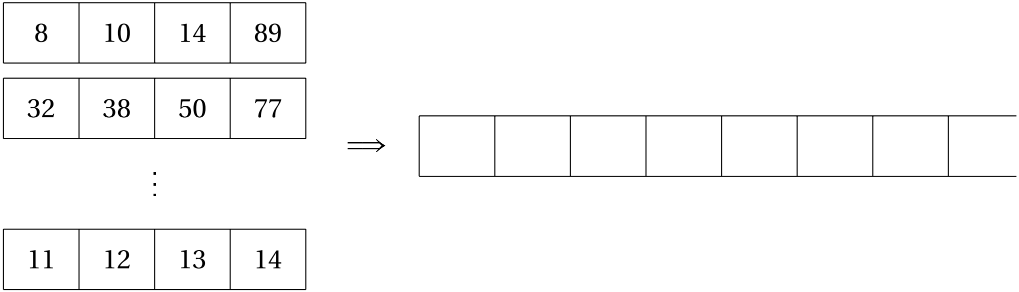
After one array is exhausted, copy all the leftovers of the other.



Merging two arrays, codes

```java
void mergeArrays(int[] a1, int[] a2, int[] a) {
    int i1 = 0, i2 = 0, i = 0;
    while (i1 < a1.length && i2 < a2.length)
        a[i++] = (a1[i1] <= a2[i2])? a1[i1++]:a2[i2++];

    // Don't forget the leftovers: we are not done yet!
    while (i1 < a1.length) a[i++] = a1[i1++];
    while (i2 < a2.length) a[i++] = a2[i2++];
}
```

Merge k sorted arrays

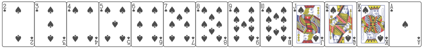


This is a very interesting problem. We will come back to it.

Instead of two input arrays, the two parts are in the same array.

5	6	7	8	1	2	3	4
0	1	2	3	4	5	6	7

left part: 5–8; right part: 1–4.



Instead of two input arrays, the two parts are in the same array.

5	6	7	8	1	2	3	4
0	1	2	3	4	5	6	7

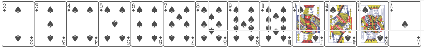
left part: 5–8; right part: 1–4.

Merging two parts, codes

```java
void mergeV1(int[] a, int low, int mid, int high) {
    int[] b = new int[mid - low + 1];
    int[] c = new int[high - mid];
    for (int i=0; i<=mid-low; i++) b[i]=a[low+i];
    for (int i=0; i<high-mid; i++) c[i]=a[mid+1+i];

    int i1 = 0, i2 = 0, i = low;
    while (i1 < b.length && i2 < c.length)
        a[i++] = (b[i1] <= c[i2])? b[i1++]:c[i2++];
    while (i1 < b.length) a[i++] = b[i1++];
    while (i2 < c.length) a[i++] = c[i2++];
}
```

Mergesort

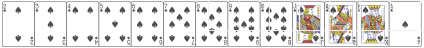
10	8	14	89	32	50	77	38

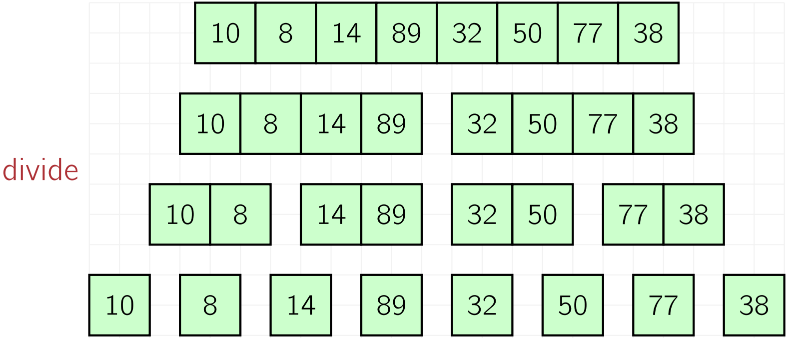


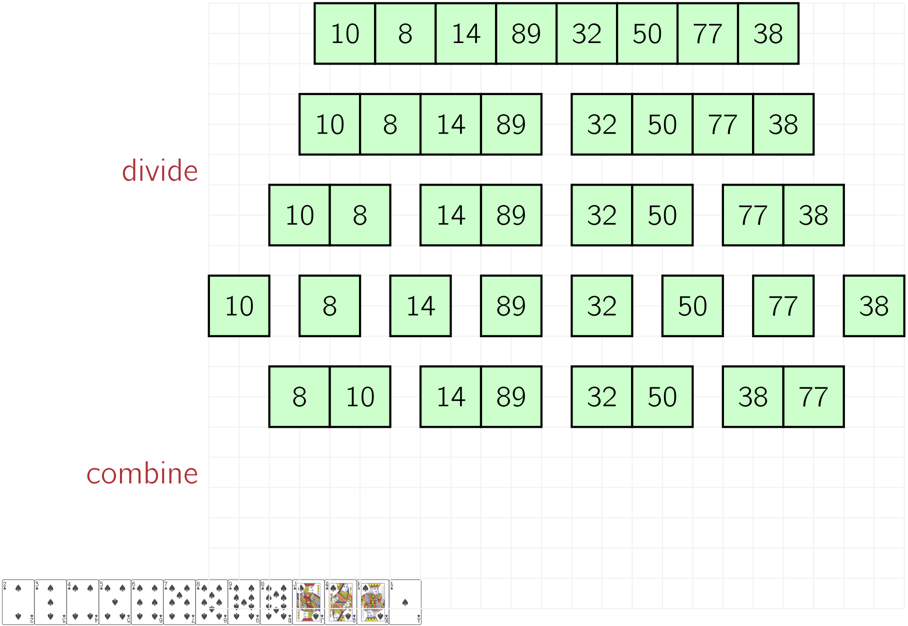
10	8	14	89	32	50	77	38

10	8	14	89

32	50	77	38

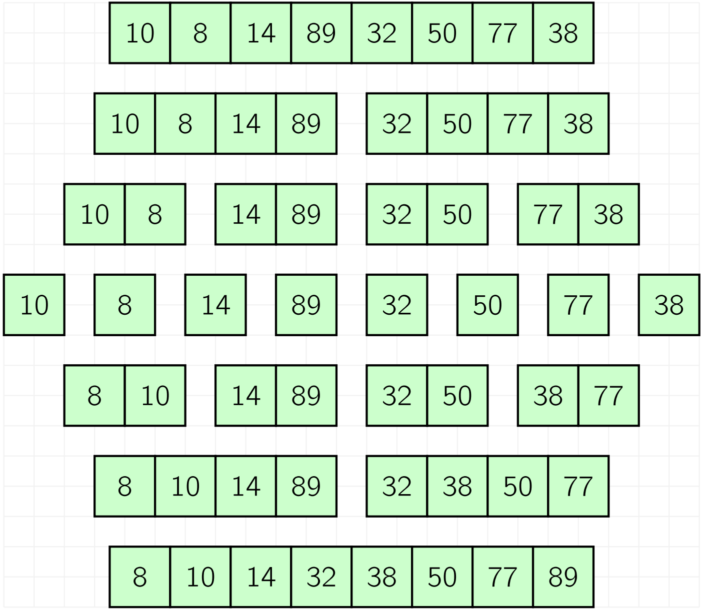



10 8 14 89 32 50 77 38
10 8 14 89
32 50 77 38
divide
10 8
14 89
32 50
77 38
10
8
14
89
32
50
77
38


divide
combine
10 8 14 89 32 50 77 38
10 8 14 89
32 50 77 38
10 8
14 89
32 50
77 38
10
8
14
89
32
50
77
38
8 10
14 89
32 50
38 77

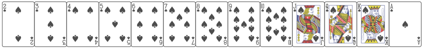
divide

combine



Recursive mergesort: Version 1

```
void rMergesortV1(int[] a, int low, int high) {
    if (high <= low) return;
    int mid = low + (high - low) / 2;
    rMergesortV1(a, low, mid);
    rMergesortV1(a, mid + 1, high);
    mergeV1(a, low, mid, high);
}
```

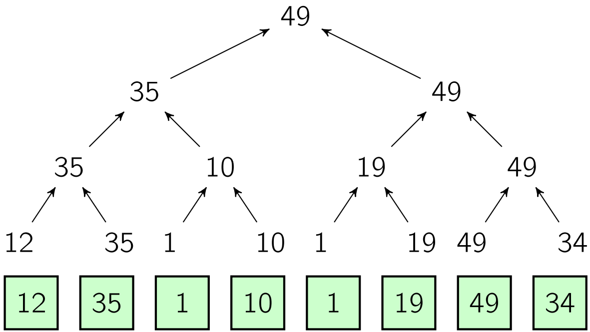


```java
void rMergesortV2(int[] a) {
    if (a.length <= 1) return;
    int n = a.length;

    //step 1: partition (almost) evenly;
    int[] b = new int[(n + 1) / 2];
    int[] c = new int[n / 2];
    for (int i=0; i<(n+1)/2; i++) b[i] = a[i];
    for (int i=0; i<n/2; i++) c[i] = a[(n+1)/2+i];

    //step 2: recursively sort the two subarrays;
    rMergesortV2(b);
    rMergesortV2(c);

    //step 3: and then merge the sorted subarrays.
    mergeArrays(b, c, a);
}
```

Pay attentions to the comments in `comp2011\lec6\Sorting.java`.

Analysis

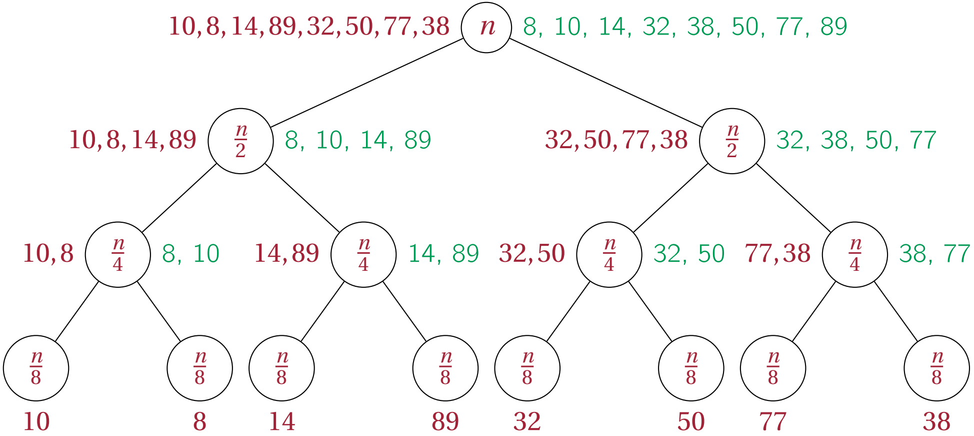
Iterative mergesort



Similar as finding maximum by divide and conquer

- Sort every pair by merging two trivial parts.
- Sort every quadruple (four) by merging two pairs.
- ...
- Sort a by merging two parts.

The execution of mergesort



What is the pattern?

- Sort every pair by merging two trivial parts. $\log n$
- Sort every quadruple (four) by merging two pairs. $\log n - 1$
- ... $\vdots$
- Sort a by merging two parts. 1

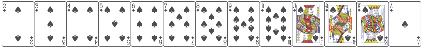
At level $j = 1, 2, \ldots, \log n$, there are ________ subproblems, each of size ________.

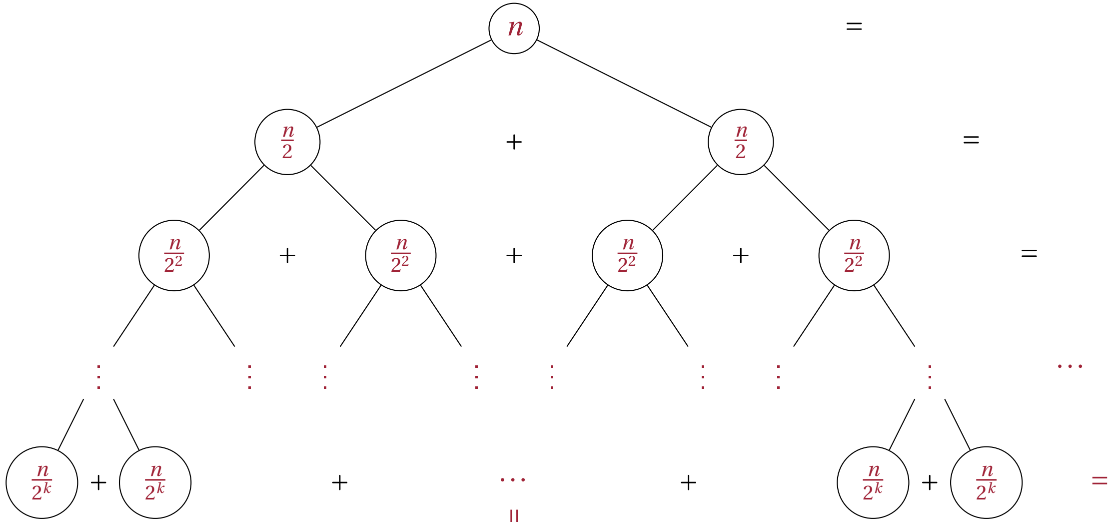
A. 2^j and 2^j

B. $\frac{n}{2^j}$ and $\frac{n}{2^j}$

C. 2^j and $\frac{n}{2^j}$

D. $\frac{n}{2^j}$ and 2^j





$$\Downarrow$$

$$O\left(\sum_{i=0}^{k} 2^i \cdot \frac{n}{2^i}\right) = O\left(\sum_{i=0}^{k} n\right) = O(k \cdot n) \quad \Leftrightarrow$$

For a general n, there exists k such that $2^k \leq n < 2^{k+1}$.

The work of sorting n elements is between sorting 2^k elements $O(2^k \log 2^k)$
and sorting 2^{k+1} elements $O(2^{k+1} \log 2^{k+1})$

$$O(2^k \log 2^k) = O(2^{k+1} \log 2^{k+1}) = O(nlogn).$$

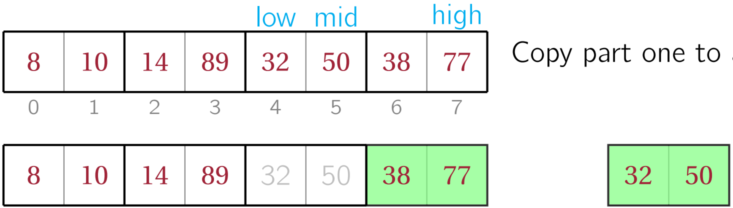
n	$\log n$	$n \log n$	n^2
2	1	2	4
4	2	8	16
8	3	24	64
16	4	64	256
32	5	160	1024
64	6	384	4096
128	7	896	16384
256	8	2048	65536

Improvement

Merging, a better way

				low	mid		high
8	10	14	89	32	50	38	77
0	1	2	3	4	5	6	7

Merging, a better way



Copy part one to a temp position.

Merging, a better way

				low	mid		high
8	10	14	89	32	50	38	77
0	1	2	3	4	5	6	7

Copy part one to a temp position.

Treated as two “arrays.”

8	10	14	89	32	50	38	77

32	50

8	10	14	89	32	50	38	77

32	50

Merging, a better way

				low	mid		high
8	10	14	89	32	50	38	77
0	1	2	3	4	5	6	7

Copy part one to a temp position.

Treated as two “arrays.”

8	10	14	89	32	50	38	77
8	10	14	89	32	50	38	77
8	10	14	89	32	38	38	77

32	50
32	50
32	50

Merging, a better way

Copy part one to a temp position.

Treated as two “arrays.”

				low	mid		high
8	10	14	89	32	50	38	77
0	1	2	3	4	5	6	7

8	10	14	89	32	50	38	77	32	50
8	10	14	89	32	50	38	77	32	50
8	10	14	89	32	38	38	77	32	50
8	10	14	89	32	38	50	77	32	50

Done.

Merging, a better way

				low	mid		high
8	10	14	89	32	50	38	77
0	1	2	3	4	5	6	7

Copy part one to a temp position.

Treated as two "arrays."

8	10	14	89	32	50	38	77	32	50
8	10	14	89	32	50	38	77	32	50
8	10	14	89	32	38	38	77	32	50
8	10	14	89	32	38	50	77	32	50

Done.

Exercise: do the next merging.

8	10	14	89	32	38	50	77	8	10	14	89

Mergesort: the standard version

```
void merge(int[] a, int low, int mid, int high) {
    int[] temp = new int[mid - low + 1];
    for(int i=0; i<temp.length; i++) temp[i]=a[low+i];
    int i = 0, j = mid+1, k = low;
    while (i < temp.length && j <= high)
        a[k++] = temp[i] <= a[j]?temp[i++]:a[j++];
    while (i < temp.length) a[k++] = temp[i++];
}
```

```
void mergesort(int[] a, int low, int high) {
    if (high < 1 + low) return;
    int mid = low + (high - low) / 2;
    mergesort(a, low, mid);
    mergesort(a, mid+1, high);
    merge2(a, low, mid, high);
}
```

Mergesort: the standard version

```java
void merge(int[] a, int low, int mid, int high) {
    int[] temp = new int[mid - low + 1];
    for(int i=0; i<temp.length; i++) temp[i]=a[low+i];
    int i = 0, j = mid+1, k = low;
    while (i < temp.length && j <= high)
        a[k++] = temp[i] <= a[j]?temp[i++]:a[j++];
    while (i < temp.length) a[k++] = temp[i++];
}
```

There is no need to deal with the leftovers of the second part. Why?

```java
void mergesort(int[] a, int low, int high) {
    if (high < 1 + low) return;
    int mid = low + (high - low) / 2;
    mergesort(a, low, mid);
    mergesort(a, mid+1, high);
    merge2(a, low, mid, high);
}
```

Summary

- Merging two sorted arrays means to create a third array that contains all the elements from both arrays in sorted order.
- In mergesort, 1-element subarrays of a large array are merged into 2-element subarrays, 2-element subarrays are merged into 4-element subarrays, and so on until the entire array is sorted.
- Mergesort *always* takes $O(n \log n)$ time.
- Mergesort requires a workspace of size $\frac{n}{2}$. The first sorting not in-place.
- Mergesort makes two recursive calls to itself.
- To translate it to the iterative version, we take the bottom-up approach (similar as finding maximum).

- Timsort ( ) avoids the pitfalls of mergesort.

Stability of Sorting Algorithms

Sorting a table

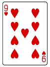
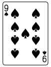
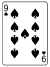
Lab	Student name	ID
LAB001	Chan Eason	
LAB003	Chan Jennifer	
LAB003	Cheung Jacky	
LAB002	Ho Denise	
LAB001	Man Janice	
LAB001	Peng Eddie	
LAB003	Sit Fiona	
LAB002	Tang Gloria	
LAB002	Tse Kay	
LAB002	Yung Joey	

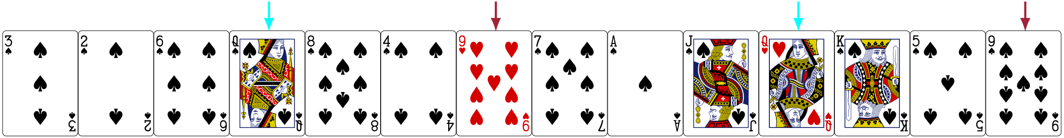
Lab	Student name	ID
LAB001	Chan Eason	
LAB001	Man Janice	
LAB001	Peng Eddie	
LAB002	Ho Denise	
LAB002	Tang Gloria	
LAB002	Tse Kay	
LAB002	Yung Joey	
LAB003	Chan Jennifer	
LAB003	Cheung Jacky	
LAB003	Sit Fiona	

Already sorted by name. If we sort by lab, what results do you expect.

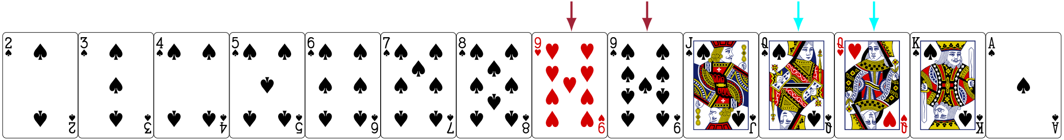
Stability ( )

The relative order of two equal items (having the same key) will be preserved.

If  came before  in the input, it will also come before  in the output.

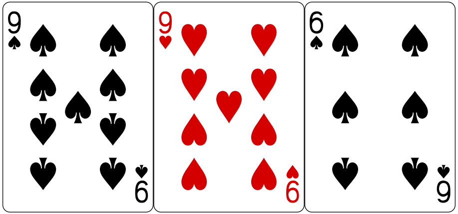


After sorted by a stable sorting algorithm

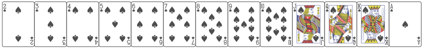


Are sorting algorithms always stable?

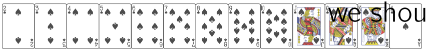
If not, which sorting algorithms is not stable?



Try the sorting algorithm on this array, and see the order of two 9's.



If we sort the patients by their urgent levels,
we should maintain “first come, first served.”



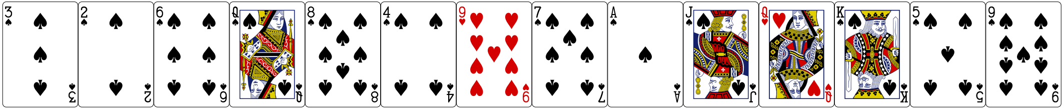
Why do we need stability?

In a more general situation:

- We have a lot of objects (tasks, patients) of different priority,
- When sorting them by priority, we want first come, first serve.
- If an instable sorting algorithm is used, we mess up the data.

So stability is very important.

Stability of basic sorting algorithms



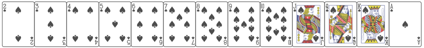
Bubble sort is stable

- Each time an element is moved by one position.
- It never swap two elements of the same value.

Insertion sort is stable

- We only move elements larger than the current.

Selection sort is not stable

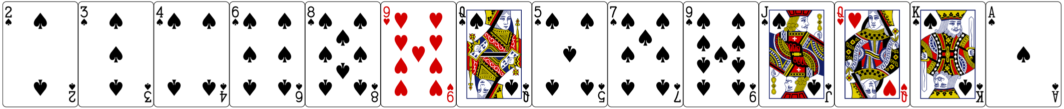


Mergesort

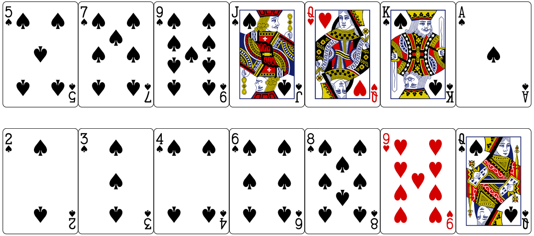
In mergesort

- Elements in the same part never change order
- Two elements change order only when they are merged from two different parts into one part.

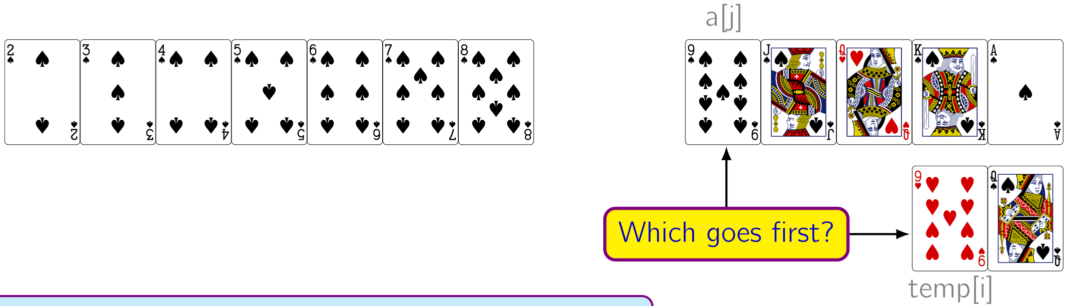
Mergesort



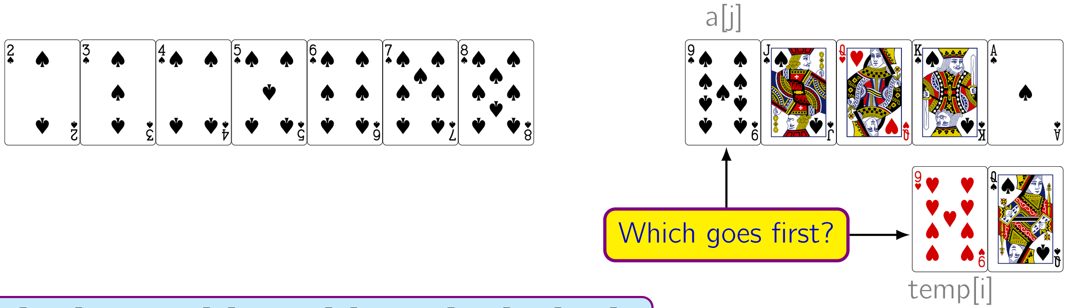
Mergesort



Mergesort

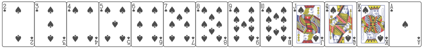



```
a[k++] = temp[i] <= a[j]?temp[i++]:a[j++];
```




```
a[k++] = temp[i] <= a[j]?temp[i++]:a[j++];
```

Of two equal elements, the one from `temp` takes precedence.
`temp` is a copy of the left part. So mergesort is stable.



Not stable if <= is replaced by <!

Correct implementations of bubble sort, insertion sort, and mergesort are stable.

Week 7: Quicksort