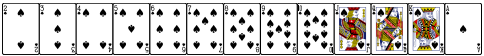
COMP 2011: Data Structures

Lecture 5. Recursion + Divide and Conquer

Dr. CAO Yixin

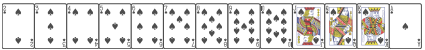
yixin.cao@polyu.edu.hk

September, 2021



Review of Lecture 4

- Queues and its implementation using an array *circularly*.
- Linked lists; boundary cases for its operations.
- Keep it simple (🦫); e.g., use `tail` only when it's absolutely necessary.
- We need to understand the way computers think, different from human beings. (Do not get everybody moving)

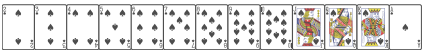


Overview

Typical computer science students study the *basic sorting algorithms* [including all the algorithms we will discuss] at least three times before they graduate.

Steven S. Skiena
"The Algorithm Design Manual"

Which sorting algorithm(s) Java library uses?



Overview

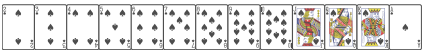
Typical computer science students study the *basic sorting algorithms* [including all the algorithms we will discuss] at least three times before they graduate.

Steven S. Skiena
"The Algorithm Design Manual"

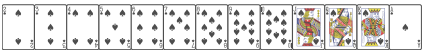
Which sorting algorithm(s) Java library uses?

formula: recursion + divide and conquer

Three keywords (🌍): comparisons, in-place, stable.



Recursion



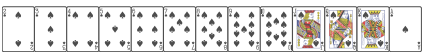
Recursion: A method calls itself

- Very natural mathematically, because many formulas are defined recursively. factorial; Fibonacci numbers (🐇); balanced parentheses; linked lists
- Terminates when a base case is reached.

```
long factorial(int a) {
  if (a <= 1) return 1;
  return a * factorial(a - 1);
}

long fibonacci(int n) {
  if (n <= 1) return n;
  return fibonacci(n - 2) + fibonacci(n - 1);
}
```

The most common bug!



Recursion: A method calls itself

- Very natural mathematically, because many formulas are defined recursively. factorial; Fibonacci numbers (🐇); balanced parentheses; linked lists
- Terminates when a base case is reached.

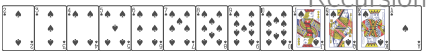
```
long factorial(int a) {
  if (a <= 1) return 1;
  return a * factorial(a - 1);
}

long fibonacci(int n) {
  if (n <= 1) return n;
  return fibonacci(n - 2) + fibonacci(n - 1);
}
```

Recursion can be avoided:
It can always be rewritten as iteration.

The most common bug!

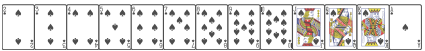
Recursion can be indirectly: $f1$ calls $f2$, and then $f2$ calls $f1$.



Examples

- The file-system in modern operating systems: folds in folds.
- program blocks (e.g., enclosed by { } in Java)
- multi-dimensional arrays in Java
- XML tags

  ...
- Many occurrences of recursions are used without being noticed.



Exercises

What these methods do?

```
1  long sum(int n) {
2      if (n <= 1) return n;
3      return sum(n - 1) + n;
4  }
5
6  void mystery1(String s) {
7      if (s.length() == 0) return;
8      System.out.print(s.charAt(0));
9      mystery1(s.substring(1));
10 }
11
12 void mystery2(String s) {
13     if (s.length() == 0) return;
14     mystery2(s.substring(1));
15     System.out.print(s.charAt(0));
16 }
```

Exercises

```
1  long sum(int n) {
2      if (n <= 1) return n;
3      return sum(n - 1) + n;
4  }
5
6  void mystery1(String s) {
7      if (s.length() == 0) return;
8      System.out.print(s.charAt(0));
9      mystery1(s.substring(1));
10 }
11
12 void mystery2(String s) {
13     if (s.length() == 0) return;
14     mystery2(s.substring(1));
15     System.out.print(s.charAt(0));
16 }
```

What these methods do?

What if we use `mystery1` in line 14 and `mystery2` in line 9?

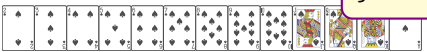
Binary search

```
1  int binarySearch(int[] a, int key) {
2      int l = 0, h = a.length - 1;
3      while (l <= h) {
4          int m = l + (h - l) / 2;
5          if (key == a[m]) return m;
6          if (key < a[m]) h = m - 1;
7          else l = m + 1;
8      }
9      return -1;
10 }
```

the iterative version

the recursive version

```
1  int bS(int[] a, int l, int h, int key) {
2      if(l > h) return -1;
3      int m = l + (h - l) / 2;
4      if (a[m] == key) return m;
5      return (key > a[m]) ? bS(a, m+1, h, key)
6          : bS(a, l, m - 1, key);
7  }
```

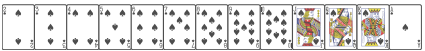


Greatest common divisor ()

The GCD of a and b is the largest integer that divides both a and b.
e.g., $\gcd(8, 12) = 4$.

48	18	12	$48 = 18 \times 2 + 12$
18	12	6	
12	6	0	
6			

Try to calculate $\gcd(162, 216)$ and $\gcd(2021, 2494)$ in this way.



Codes for greatest common divisor

the recursive version

```
1  int recursiveGCD(int a, int b) {
2      if (a * b == 0) return a + b;
3      int c = Math.min(a, b);
4      if ((a + b) % c == 0) return c;
5      return recursiveGCD((a + b) % c, c);
6  }
```

Codes for greatest common divisor

```java
int iterativeGCD(int a, int b) {
    while ( a * b != 0 ) {
        int temp = Math.min(a, b);
        a = (a + b) % temp;
        b = temp;
    }
    return a + b;
}
```

the iterative version

the recursive version

```java
int recursiveGCD(int a, int b) {
    if (a * b == 0) return a + b;
    int c = Math.min(a, b);
    if ((a + b) % c == 0) return c;
    return recursiveGCD((a + b) % c, c);
}
```

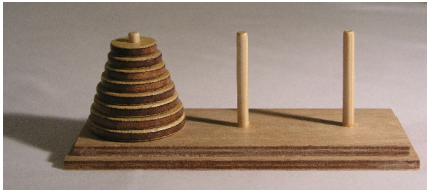
Towers of Hanoi (🌍)

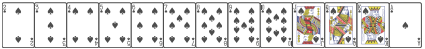
- Three rods and a number of disks of different diameters.
- Initially, all disks are in ascending order of size on one rod, smallest at the top.
- Objective: move the entire stack to another rod.

Rules

1. Only one disk can be moved at a time.
2. Only the uppermost disk on a stack can be moved.
3. No disk may be placed on top of a smaller disk.

demo 6 disks 7 disks 9 disks

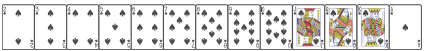




To move n disks from the left rod to the right rod

1. Move the top $n-1$ disks from the left rod to the center rod.
2. Move the largest disk from the left rod to the right rod.
3. Move the $n-1$ disks from the center rod to the right rod.

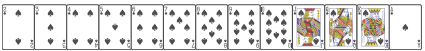
$$M(n) = \begin{cases} 1 & \text{if } n = 1, \\ 2M(n-1)+1 & \text{if } n > 1. \end{cases}$$



The codes

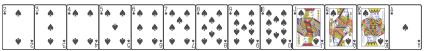
```
1  void move(int n, int from, int to) {
2      String[] rods = {"Left", "Center", "Right"};
3      if(n <= 1) {
4          System.out.println("disk␣1:␣" +
5              rods[from] + "␣->␣" + rods[to]);
6          return;
7      }
8      int via = 3 - from - to; // a small trick
9      move(n - 1, from, via);
10     System.out.println("disk␣" + n + ":␣" +
11         rods[from] + "␣->␣" + rods[to]);
12     move(n - 1, via, to);
13 }
```

Can we rewrite it without using recursion?



Summary

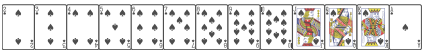
- A recursive method calls itself repeatedly, with different argument values each time.
- Some value of its arguments causes a recursive method to return without calling itself. This is called the *base case*.
- When the innermost instance of a recursive method returns, the process "unwinds" by completing pending instances of the method, going from the latest back to the original call. $f(4) \to f(3) \to f(2) \to f(1)$.
- A binary search can be carried out recursively by checking which half of a sorted range the search key is in, and then doing the same thing with that half.
- The towers of Hanoi puzzle can be solved recursively by
  1. Move the top $n-1$ disks from the left rod to the center rod .
  2. Move the largest disk from the left rod to the right rod .
  3. Move the $n-1$ disks from the center rod to the right rod .



Recursion vs. iteration

- Sometimes it is easy to rewrite a recursive algorithm into an iterative one.
- But sometimes it's not: Some problems ($n!$) and some procedures (towers of Hanoi) are impossible/difficulty to be expressed in a non-recursive way.
- Recursion is not always good
  - The recursive `Fibonacci` is insanely slow. Why?
  - It's hard to debug recursive methods.

Which of the three sorting algorithm we've discussed can be written as recursion?

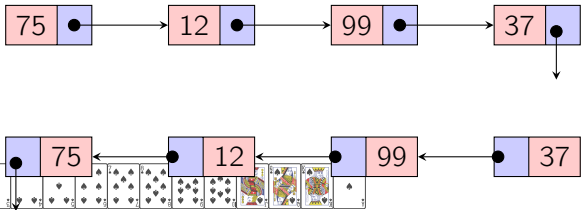


Recursion vs. iteration

- Sometimes it is easy to rewrite a recursive algorithm into an iterative one.
- But sometimes it's not: Some problems ($n!$) and some procedures (towers of Hanoi) are impossible/difficulty to be expressed in a non-recursive way.
- Recursion is not always good
  - The recursive `Fibonacci` is insanely slow. Why?
  - It's hard to debug recursive methods.

```
1  void recursiveSelection(int[] a, int begin) {
2      if (begin == a.length - 1) return;
3      int n = a.length;
4      int min = begin;
5      for (int i = begin+1; i <
6          if (a[min] > a[i]) mi
7      swap(a, begin, min);
8      recursiveSelection(a, begin+1);
9  }
```

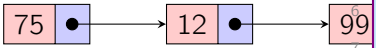
Which of the three sorting algorithm we've discussed can be written as recursion?

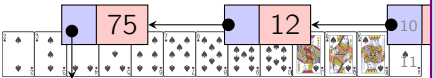

75
12
99
37
75
12
99
37

```
Node recursiveReverse() {




}
```






```
1  void reverse() {
2      if (                          ) return;
3      Node ind1 = head, ind2 = head.next, in
4      ind1.next = null;
// don't forget this!
5      while (ind2.next != null) {
6                =     ;
7                =     ;
8                =         ;
9                   =      ;
10     }
11     ind2.next = ind1;
```

fill the spaces

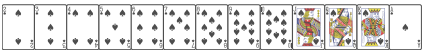
Implementation of recursion

- 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)

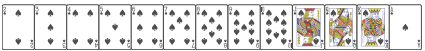
- A recursion can always be mechanically simulated by manually maintaining the call stack. But it is too error-prone.
  - When write iterative versions, we seldom do this way.
  - Instead, we take a problem-specific approach (e.g., Fibonacci).



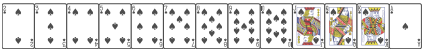
Recursive calls eat space

```
void iter(int n) {
  while (true) { n++; }
}
```

```
void rec(int n) {
  rec(n++);
}
```

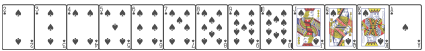


Divide and Conquer



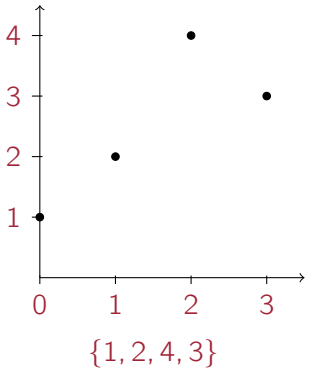
Problems to be discussed

- To find a maximum element from an array, we need $n-1$ comparisons. How many different ways you can do that?
- How about finding a local maximal, i.e., a number that is not smaller than its neighbor(s)?
- What's the smallest number of comparisons you need to find both a maximum element and a minimum element?
- What's the smallest number of comparisons you need to find two largest elements?

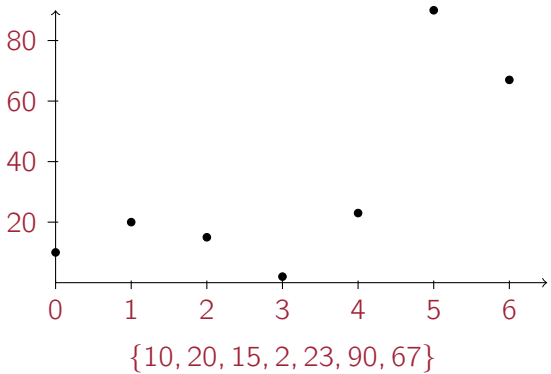


Finding a peak

A peak: An array element that is not smaller than its neighbor(s).



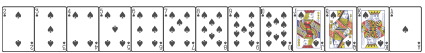
$\{1, 2, 4, 3\}$



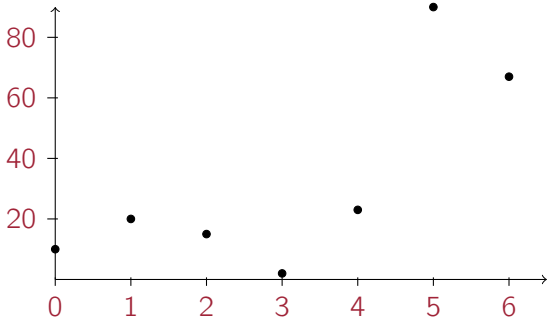
$\{10, 20, 15, 2, 23, 90, 67\}$

A peak doesn't need to be the highest point.

The highest point in Hong Kong is not The Peak (554 m), but Tai Mo Shan (957 m).



The key observation



$\{10, 20, 15, 2, 23, 90, 67\}$

Can you write the algorithm?

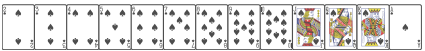
- Every nonempty array has a peak.
  - If $a[0]$ isn't a peak, then $a[0] < a[1]$; if $a[1]$ isn't, then $a[1] < a[2]$; ...; then $a[n-1]$ is.
- For any $a[i]$ that is not a peak, try to check this against any i.
  - if $a[i] < a[i-1]$, then there is a peak to the left of $a[i]$;
  - otherwise, $a[i] < a[i+1]$, and there is a peak to the right of $a[i]$;
- So a peak can be found in a similar way as binary search.

Comparisons of elements

- For loop conditions, we compare indices. Inside the loop, we compare elements.
- A comparison of two indices (not too large numbers) is consider a primitive step.
- But the comparison of two objects usually needs a lot of operations.

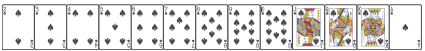
  e.g., long strings and pictures.
- To make class comparable, we need to implement the Comparable interface.

```
class Student implements Comparable<Student>{
    String name;
    String id;
    String major;
    String email;
    public int compareTo(Student s2) {
        return name.compareTo(s2.name);
    }
}
```



Finding maximum by divide and conquer

- Find *the* maximum value; find *a* maximum element.
  - An array can have many maximum elements, but they must have the same value.
  - e.g., in $\{11, 19, 23, 4, 12, 23, 7\}$, both $a[2]$ and $a[5]$ are maximum elements.
  - Articles (a, an, the) are very important in English: compare "read a book" and "read the book."
  - There are no articles in Chinese, so some students use them in an arbitrary way, or not use at all.
- Each iteration of selection sort finds a maximum element in a range.
- It takes $n - 1$ comparisons.

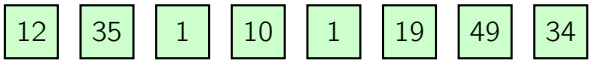


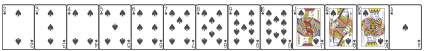
Finding the maximum by divide and conquer

We can solve it by divide and conquer:

1. partition a into two parts,
2. find a maximum element from each part; and
3. compare the two maximum elements and return the larger.

Base case: only one element.



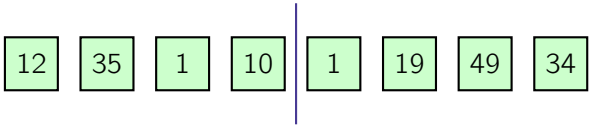


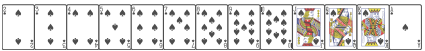
Finding the maximum by divide and conquer

We can solve it by divide and conquer:

1. partition a into two parts,
2. find a maximum element from each part; and
3. compare the two maximum elements and return the larger.

Base case: only one element.



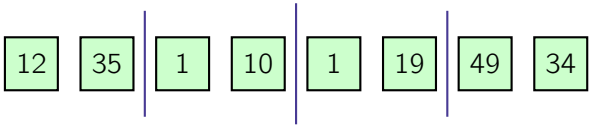


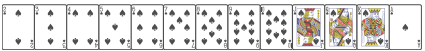
Finding the maximum by divide and conquer

We can solve it by divide and conquer:

1. partition a into two parts,
2. find a maximum element from each part; and
3. compare the two maximum elements and return the larger.

Base case: only one element.



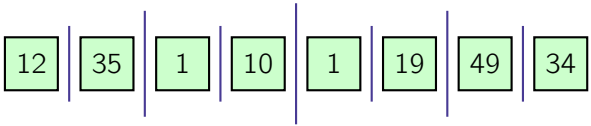


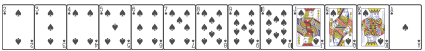
Finding the maximum by divide and conquer

We can solve it by divide and conquer:

1. partition a into two parts,
2. find a maximum element from each part; and
3. compare the two maximum elements and return the larger.

Base case: only one element.





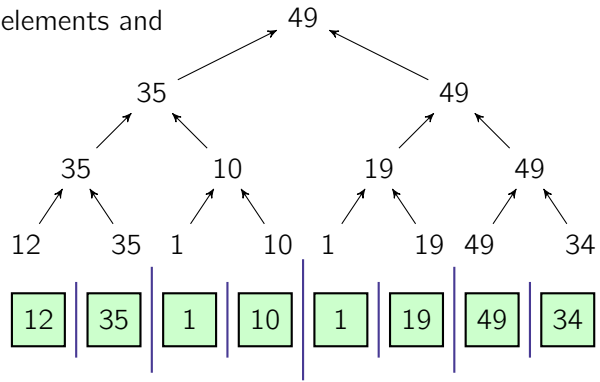
Finding the maximum by divide and conquer

We can solve it by divide and conquer:

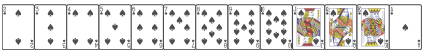
1. partition a into two parts,
2. find a maximum element from each part; and
3. compare the two maximum elements and return the larger.

Base case: only one element.

Finding a minimum is similar.

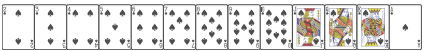


Try to write the codes.
(Recursive version is easier.)



The recursive version

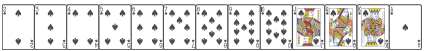
```
1  int max(int[] a, int low, int high){
2      if (low >= high) return a[low];
3      int mid = low + (high - low) / 2;
4      int m1 = max(a, low, mid);
5      int m2 = max(a, mid+1, high);
6      return (m1 > m2)? m1:m2;
7  }
```



Iterative implementation

0	1	2	3	4	5	6	7	8	9
8	10	14	89	32	50	38	77	7	11

$n = 10$.

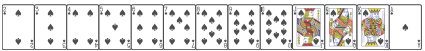


Iterative implementation

After finding the larger of the first pair, number at index 1 is useless.

0	1	2	3	4	5	6	7	8	9
8	10	14	89	32	50	38	77	7	11
10	10	14	89	32	50	38	77	7	11

$n = 10$.

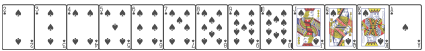


Iterative implementation

After finding the larger of the first pair, number at index 1 is useless.

0	1	2	3	4	5	6	7	8	9
8	10	14	89	32	50	38	77	7	11
10	10	14	89	32	50	38	77	7	11
10	89	14	89	32	50	38	77	7	11

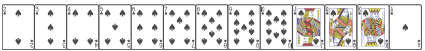
$n = 10$.



Iterative implementation

After finding the larger of the first pair, number at index 1 is useless.

0	1	2	3	4	5	6	7	8	9	
8	10	14	89	32	50	38	77	7	11	$n = 10$.
10	10	14	89	32	50	38	77	7	11	
10	89	14	89	32	50	38	77	7	11	
10	89	50	89	32	50	38	77	7	11	
10	89	50	77	32	50	38	77	7	11	
10	89	50	77	11	50	38	77	7	11	$n = 5$.

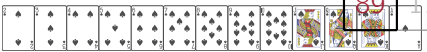


Iterative implementation

After finding the larger of the first pair, number at index 1 is useless.

Similarly.

0	1	2	3	4	5	6	7	8	9	
8	10	14	89	32	50	38	77	7	11	$n = 10$.
10	10	14	89	32	50	38	77	7	11	
10	89	14	89	32	50	38	77	7	11	
10	89	50	89	32	50	38	77	7	11	
10	89	50	77	32	50	38	77	7	11	
10	89	50	77	11	50	38	77	7	11	$n = 5$.
89	77	11	77	11	50	38	77	7	11	$n = 3$.
89	11	11	77	11	50	38	77	7	11	$n = 2$.
89	11	11	77	11	50	38	77	7	11	$n = 1$. Done.



The codes for the iterative version

```
1  int max(int[] a, int low, int high){
2      if (low >= high) return a[low];
3      int mid = low + (high - low) / 2;
4      int m1 = max(a, low, mid);
5      int m2 = max(a, mid+1, high);
6      return (m1 > m2)? m1:m2;
7  }
```

the recursive version

the iterative version

```
1  int max(int[] a) {
2      int n = a.length;
3      int[] b = new int[n];
4      for (int i = 0; i < n; i++) b[i] = a[i];
5      while (n > 1) {
6          for (int j = 0; j < n/2; j ++)
7              b[j] = (b[j*2]>b[j*2+1])?b[j*2]:b[j*2+1];
8          if (n != n/2*2) b[n/2] = b[n-1];
9          n = (n+1)/2;
10     }
       return b[0];
   }
```

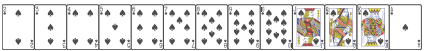
Finding maximum and minimum

- We know how to find *a* maximum element from an array, and how to find *a* minimum element from an array.

- Now the question is to find both the maximum and minimum.
- We can find a maximum first, then a minimum, with _____ comparisons.
- Or, we can find them simultaneously, scanning the array only once, with _____ comparisons.

Can we save some comparisons?

```
1  for (int i = 2; i < a.length; i++) {
2      if (a[i] > a[ans[0]]) ans[0] = i;
3      if (a[i] < a[ans[1]]) ans[1] = i;
4  }
```



Finding maximum and minimum

- We know how to find *a* maximum element from an array, and how to find *a* minimum element from an array.

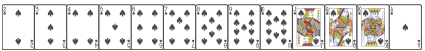
- Now the question is to find both the maximum and minimum.
- We can find a maximum first, then a minimum, with _____ comparisons.
- Or, we can find them simultaneously, scanning the array only once, with _____ comparisons.

Can we save some comparisons?

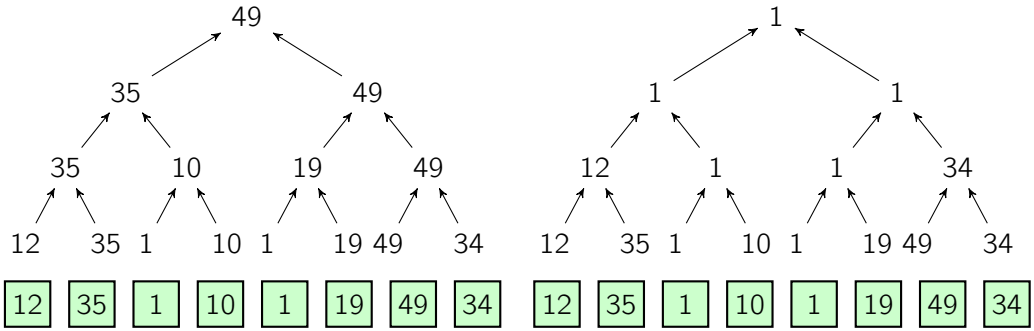
```
1  for (int i = 2; i < a.length; i++) {
2      if (a[i] > a[ans[0]]) ans[0] = i;
3      if (a[i] < a[ans[1]]) ans[1] = i;
4  }
```

Adding `else` to line 3 may help, but not in the worst case.

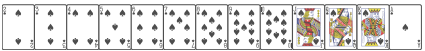
Consider, e.g., $5, 2, 4, 3$.



Finding maximum and finding minimum

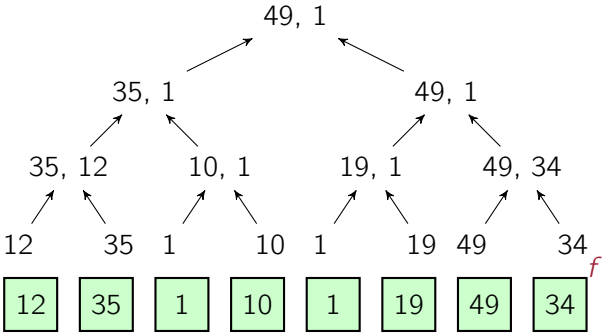


The operations at the lowest level are the ame.



Finding maximum and minimum by divide and conquer

- Two comparisons for two parts.
  - compare the left max and right max;
  - compare the left min and right min;
- But *only one* comparison in the base case.

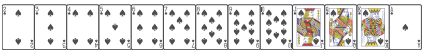


n	$\#comparisons$
2	1
4	4
8	10
16	22

$f(2n) = 2f(n) + 2 \Rightarrow f(n) = 1.5n - 2$

Write the codes.

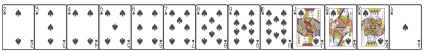
A third approach: Compare in Pairs; see geeksforgeeks.



Finding a second largest element

- If the array has two largest elements, then we return one of them.
  $9, 9, 6$: return 0 or 1, the index of one of the 9's.
- If the array has a unique maximum element, then we return an element with the second largest value.
  $9, 6, 6$: return 1 or 2, the index of one of the 6's.

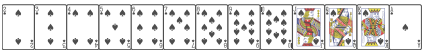
Although we are only explicitly asked for a second largest,
we have to find a largest and a second largest elements.



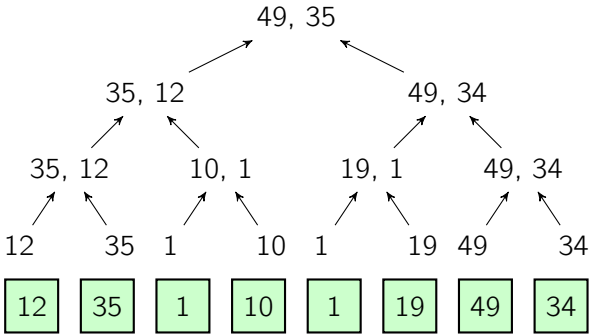
Finding a second largest element

- A naive idea is to find a largest, and then a second largest.
- This problem appears to be similar as the previous one (finding max/min).

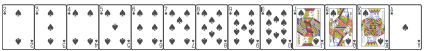
- If you have tried, you should know this one is more complicated.
  If you haven't, try to do it now.



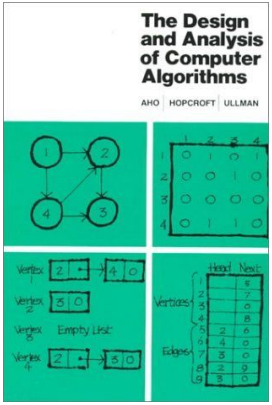
The comparison tree

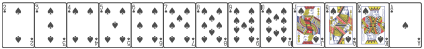


Very similar to the previous one.



Perhaps the most important principle
for the good algorithm designer
is to refuse to be content.

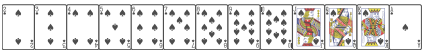




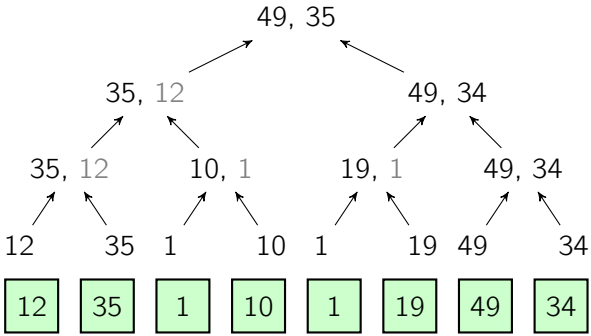
Interestingly, this problem can be solved using only $n + \log n$ comparisons.

$$n + \log n = 2^{30} + 30 \text{ for } n = 2^{30}.$$

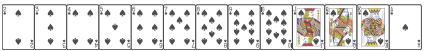
The trick is to take a lazy approach.
Instead of calculating two values for each recursive step, only the maximum.



Unnecessary calculations

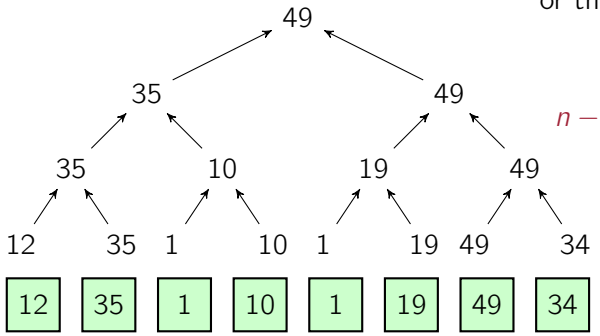


This gray numbers are never used.



Finding a second largest element, the smart way

This page is advanced and optional.

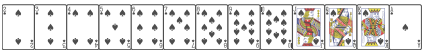


Either the largest of the left or the second of the right.

$n - 1 + \lceil \log n \rceil - 1 = n + \lceil \log n \rceil - 2.$

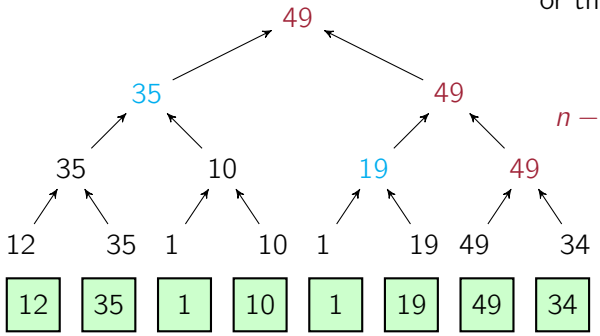
Write the codes.

Hint: we need to store all the comparison results.



Finding a second largest element, the smart way

This page is advanced and optional.

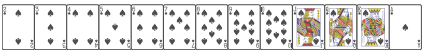


Either the largest of the left or the second of the right.

$n - 1 + \lceil \log n \rceil - 1 = n + \lceil \log n \rceil - 2.$

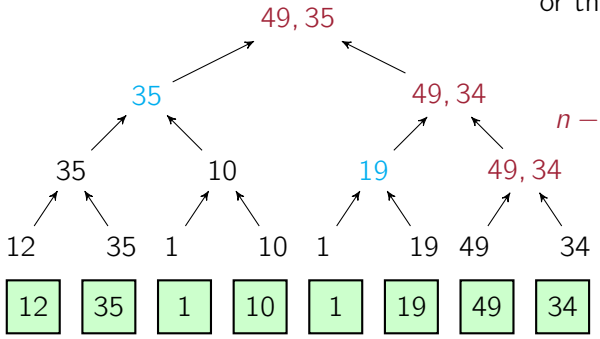
Write the codes.

Hint: we need to store all the comparison results.



Finding a second largest element, the smart way

This page is advanced and optional.

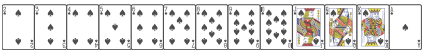


Either the largest of the left or the second of the right.

$n - 1 + \lceil \log n \rceil - 1 = n + \lceil \log n \rceil - 2.$

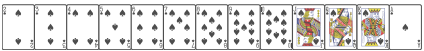
Write the codes.

Hint: we need to store all the comparison results.



Bubble sort, insertion sort, and selection sort are slow,
because they do too many repetitive comparisons.

For which of them,
you can find a way to avoid some unnecessary comparisons?
(possibly sacrificing some space)



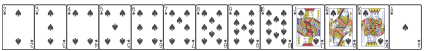
Data structures are intrinsically recursive.

```java
class Node<T> {
    T element;
    Node<T> next;
}
```

Java

```c
struct Node {
    int data;
    struct Node* next;
};
```

C

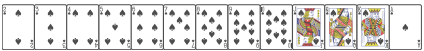


```haskell
maximum' :: (Ord a) => [a] -> a
maximum' [] = error "empty list"
maximum' [x] = x
maximum' (x:xs) = max x (maximum' xs)
```

Finding max. in Haskell (link)

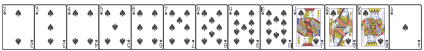
`Ord`(erable) is similar to `Comparable` in Java.

> “ Iteration in functional languages is usually accomplished via recursion. ”
>
> Wikipedia (🌍)



```
roster
.stream()
.filter(
    p -> p.getGender() == Person.Sex.MALE
        && p.getAge() >= 18
        && p.getAge() <= 25)
.map(p -> p.getEmailAddress())
.forEach(email -> System.out.println(email));
```

The Java Tutorial on Lambda Expressions

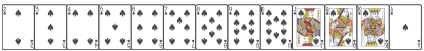


```
int peak(int[] a, int low, int high) {
}


int max(int[] a, int low, int high) {
}


int maxmin(int[] a, int low, int high) {
}


int second(int[] a, int low, int high) {
}
```



Week 6: Mergesort