Object-oriented programming and data-structures

CS/ENGRD 2110
SUMMER 2018

Lecture 8: Sorting

http://courses.cs.cornell.edu/cs2110/2018su

Lecture 7 Recap

- Introduced a formal notation for analysing the runtime/space complexity of algorithms
- Went through examples of Big-O formulas/proofs
- Analysed complexity of ArrayList/LinkedList

Admin

- There will **not** be a prelim next week. Instead, future homeworks will include exam-style questions on the whole course to help you master the revision
- A3 will be released tomorrow evening and due next Tuesday
- HW5 has been released. It covers a lot of material and will get challenging at times. Start early!
- Please fill out the poll on Piazza. Thanks to those who already have.

This lecture

- Introduce Sorting Algorithms

- Derive and implement
  - Insertion sort
  - Selection sort
  - Merge sort
  - Quick sort

- Analyse their complexity

Why Sorting?

- Sorting is useful
  - Database indexing
  - Compressing data
  - Sorting TV channels, Netflix shows, Amazon products, etc.

- There are lots of ways to sort
  - There isn't one right answer
  - You need to be able to figure out the options and decide which one is right for your application.
  - Today, we'll learn about several different algorithms (and how to derive them)

Some Sorting Algorithms

- Insertion sort
- Selection sort
- Binary Sort
- Bubble Sort
- Merge sort
- Quick sort

Refining our analysis

- Worst-case
  - Complexity in worst possible scenario. Gives an upper bound on performance, but may only arise rarely
- Average-case
  - Analyse for an 'average' input. Problem here is that need to somehow know what “average” means”
- Best-case
  - What is the minimum number of operations that must be done in the best case scenario
- Amortized analysis
  - If expensive operation happens rarely, and lots of cheap operations happen frequently, may want to amortise total cost over all the operations to get average cost per operation.

Insertion Sort

- Let’s begin by looking through an example

- Consider the following array

3	6	4	5	1	2

Let’s sort it!

Insertion Sort

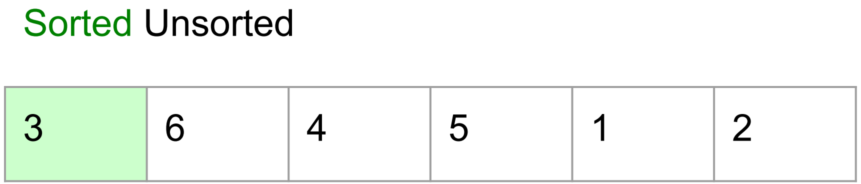
- Insertion sort iterates over the array, swapping pairs of integers until they are in the correct position

3	6	4	5	1	2

- Maintains the following invariant: **at round i, array[0,i] is sorted**

Insertion Sort

- Insertion sort iterates over the array, swapping pairs of integers until they are in the correct position



Sorted Unsorted

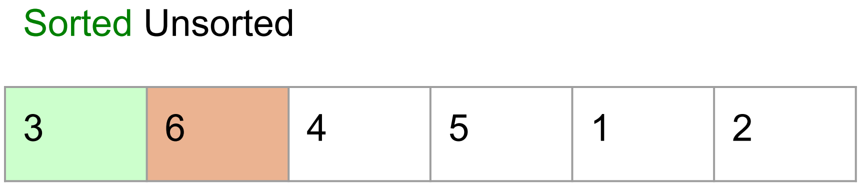
3	6	4	5	1	2

Round 0: Position 0 of the array is already sorted.

- Maintains the following invariant: **at round i, array[0,i] is sorted**

Insertion Sort

- Insertion sort iterates over the array, swapping pairs of integers until they are in the correct position



Sorted Unsorted

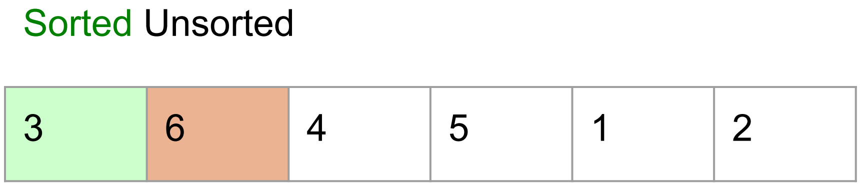
3	6	4	5	1	2

Round 0: Select element at position 1 array[i+1] and place to correct position in sorted array

- Maintains the following invariant: **at round i, array[0,i] is sorted**

Insertion Sort

- Insertion sort iterates over the array, swapping pairs of integers until they are in the correct position



Sorted Unsorted

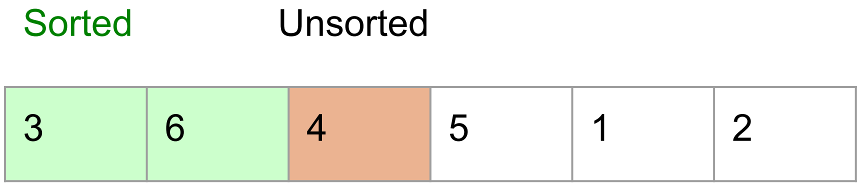
3	6	4	5	1	2

Round 0: Element does not move as it is greater than sorted array (6>3)

- Maintains the following invariant: **at round i, array[0,i] is sorted**

Insertion Sort

- Insertion sort iterates over the array, swapping pairs of integers until they are in the correct position



Sorted

Unsorted

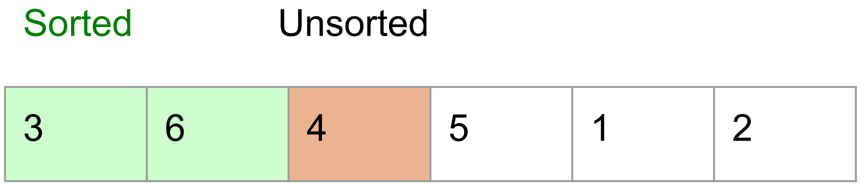
3	6	4	5	1	2

Round 1: select item at index 2, and **swap** it to the correct place.

- Maintains the following invariant: **at round i, array[0,i] is sorted**

Insertion Sort

- Insertion sort iterates over the array, swapping pairs of integers until they are in the correct position

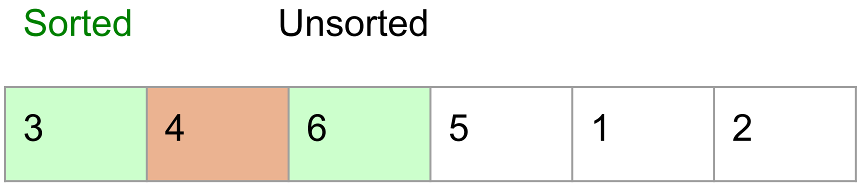


Round 1: 6>4, so swap 6 and 4

- Maintains the following invariant: **at round i, array[0,i] is sorted**

Insertion Sort

- Insertion sort iterates over the array, swapping pairs of integers until they are in the correct position



Sorted

Unsorted

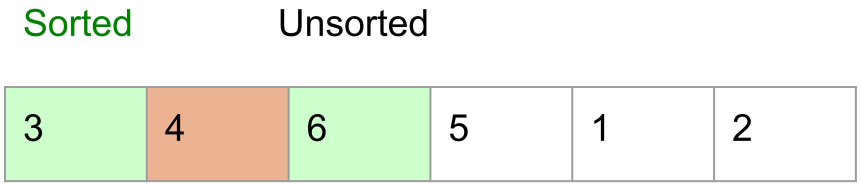
3	4	6	5	1	2

Round 1: 6>4, so swap 6 and 4

- Maintains the following invariant: **at round i, array[0,i] is sorted**

Insertion Sort

- Insertion sort iterates over the array, swapping pairs of integers until they are in the correct position



Sorted Unsorted

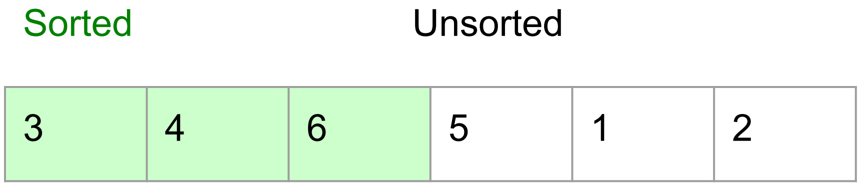
3	4	6	5	1	2

Round 1: 3<4, and all the elements that precede 3 are sorted, so 4 is in the correct position

- Maintains the following invariant: **at round i, array[0,i] is sorted**

Insertion Sort

- Insertion sort iterates over the array, swapping pairs of integers until they are in the correct position



Sorted Unsorted

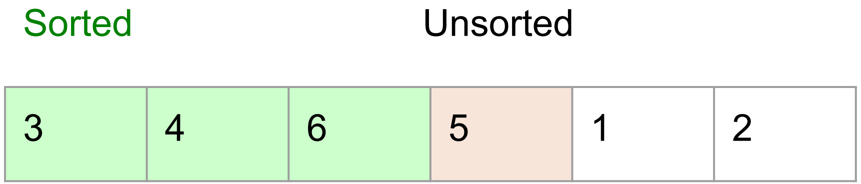
3	4	6	5	1	2

Round 1: 3<4, and all the elements that precede 3 are sorted, so 4 is in the correct position

- Maintains the following invariant: **at round i, array[0,i] is sorted**

Insertion Sort

- Insertion sort iterates over the array, swapping pairs of integers until they are in the correct position



Sorted

Unsorted

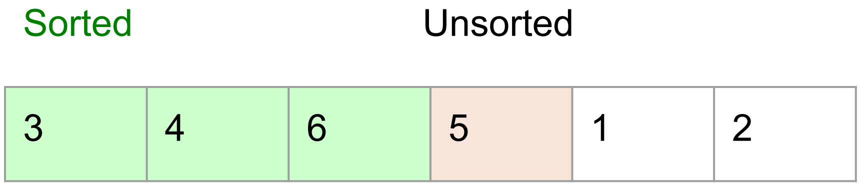
3	4	6	5	1	2

Round 2: Select element at position 3 in the array

- Maintains the following invariant: **at round i, array[0,i] is sorted**

Insertion Sort

- Insertion sort iterates over the array, swapping pairs of integers until they are in the correct position



Sorted

Unsorted

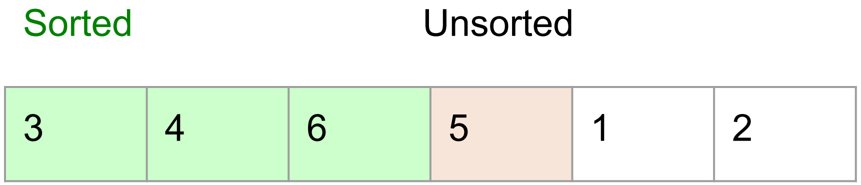
3	4	6	5	1	2

Round 2: Try to place it in the right position

- Maintains the following invariant: **at round i, array[0,i] is sorted**

Insertion Sort

- Insertion sort iterates over the array, swapping pairs of integers until they are in the correct position



Sorted

Unsorted

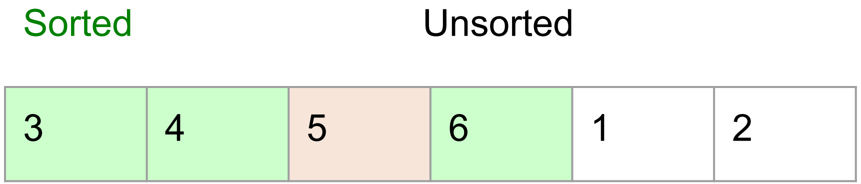
3	4	6	5	1	2

Round 2: 6>5, so swap the two elements

- Maintains the following invariant: **at round i, array[0,i] is sorted**

Insertion Sort

- Insertion sort iterates over the array, swapping pairs of integers until they are in the correct position



Sorted

Unsorted

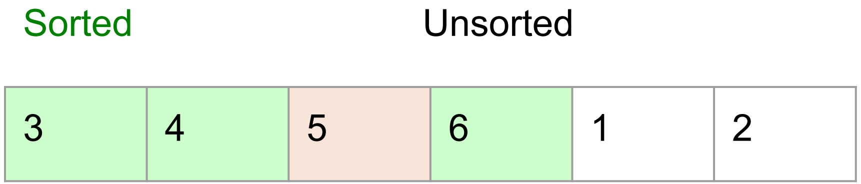
3	4	5	6	1	2

Round 2: 6>5, so swap the two elements

- Maintains the following invariant: **at round i, array[0,i] is sorted**

Insertion Sort

- Insertion sort iterates over the array, swapping pairs of integers until they are in the correct position



Sorted

Unsorted

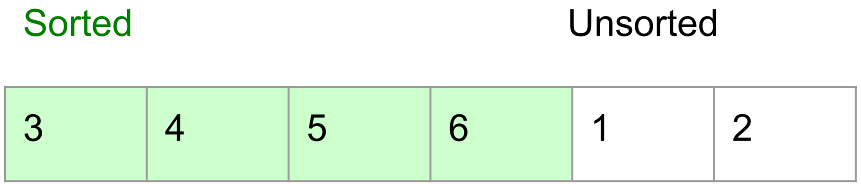
3	4	5	6	1	2

Round 2: 4<5, and the array array[0..1] is sorted, so 5 is in the correct position

- Maintains the following invariant: **at round i, array[0,i] is sorted**

Insertion Sort

- Insertion sort iterates over the array, swapping pairs of integers until they are in the correct position



Sorted

Unsorted

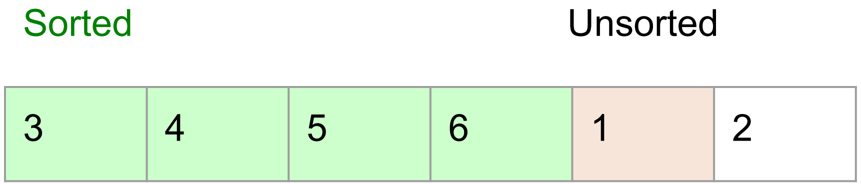
3	4	5	6	1	2

Round 2: 4<5, and the array array[0..1] is sorted, so 5 is in the correct position

- Maintains the following invariant: **at round i, array[0,i] is sorted**

Insertion Sort

- Insertion sort iterates over the array, swapping pairs of integers until they are in the correct position



Sorted

Unsorted

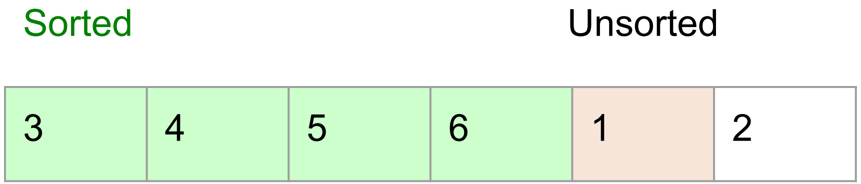
3	4	5	6	1	2

Round 3: Select element at position 4 (i+1) in the array. The first

- Maintains the following invariant: **at round i, array[0,i] is sorted**

Insertion Sort

- Insertion sort iterates over the array, swapping pairs of integers until they are in the correct position



Sorted

Unsorted

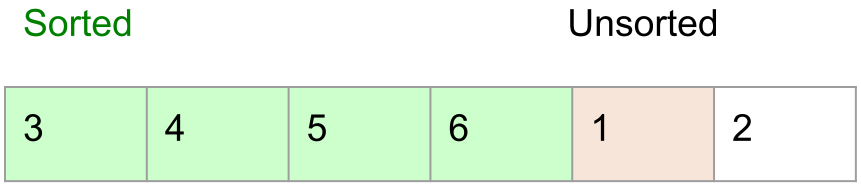
3	4	5	6	1	2

Round 3: Place it at the appropriate position in the sorted array

- Maintains the following invariant: **at round i, array[0,i] is sorted**

Insertion Sort

- Insertion sort iterates over the array, swapping pairs of integers until they are in the correct position



Sorted

Unsorted

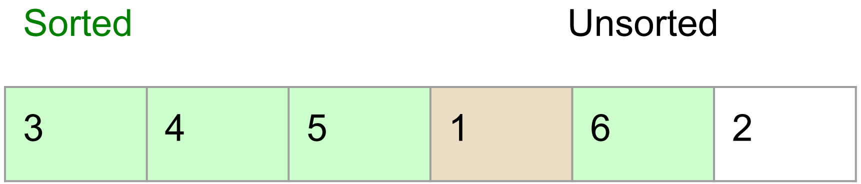
3	4	5	6	1	2

Round 3: 6>1, so swap

- Maintains the following invariant: **at round i, array[0,i] is sorted**

Insertion Sort

- Insertion sort iterates over the array, swapping pairs of integers until they are in the correct position



Sorted

Unsorted

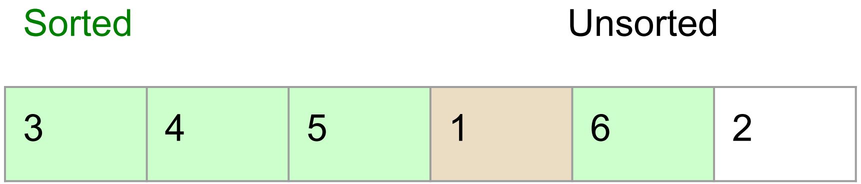
3	4	5	1	6	2

Round 3: 6>1, so swap

- Maintains the following invariant: **at round i, array[0,i] is sorted**

Insertion Sort

- Insertion sort iterates over the array, swapping pairs of integers until they are in the correct position



Sorted

Unsorted

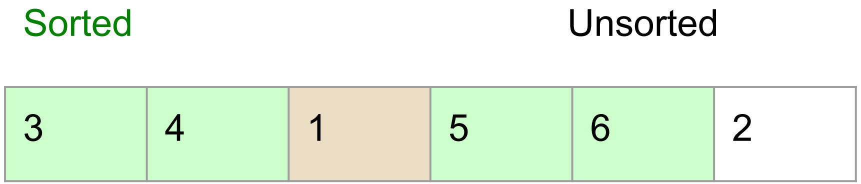
3	4	5	1	6	2

Round 3: 5>1, so swap

- Maintains the following invariant: **at round i, array[0,i] is sorted**

Insertion Sort

- Insertion sort iterates over the array, swapping pairs of integers until they are in the correct position


Sorted

Unsorted

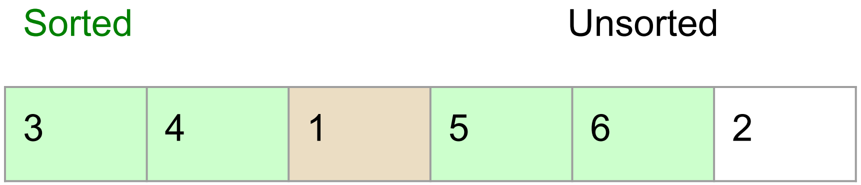
3	4	1	5	6	2

Round 3: 5>1, so swap

- Maintains the following invariant: **at round i, array[0,i] is sorted**

Insertion Sort

- Insertion sort iterates over the array, swapping pairs of integers until they are in the correct position



Sorted

Unsorted

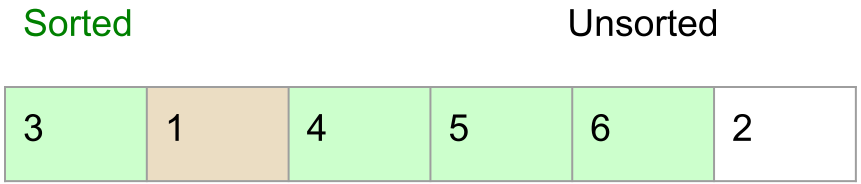
3	4	1	5	6	2

Round 3: 4>1, so swap

- Maintains the following invariant: **at round i, array[0,i] is sorted**

Insertion Sort

- Insertion sort iterates over the array, swapping pairs of integers until they are in the correct position



Sorted

Unsorted

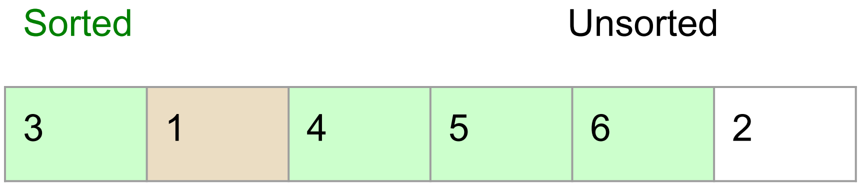
3	1	4	5	6	2

Round 3: 4>1, so swap

- Maintains the following invariant: **at round i, array[0,i] is sorted**

Insertion Sort

- Insertion sort iterates over the array, swapping pairs of integers until they are in the correct position



Sorted

Unsorted

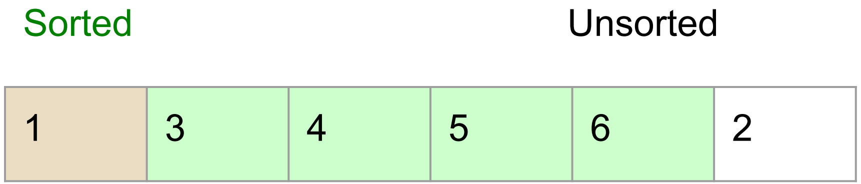
3	1	4	5	6	2

Round 3: 3>1, so swap

- Maintains the following invariant: **at round i, array[0,i] is sorted**

Insertion Sort

- Insertion sort iterates over the array, swapping pairs of integers until they are in the correct position



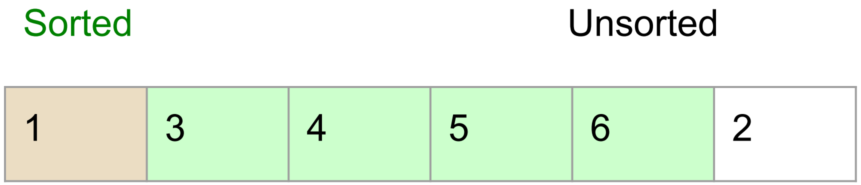
1	3	4	5	6	2

Round 3: 3>1, so swap

- Maintains the following invariant: **at round i, array[0,i] is sorted**

Insertion Sort

- Insertion sort iterates over the array, swapping pairs of integers until they are in the correct position



Sorted

Unsorted

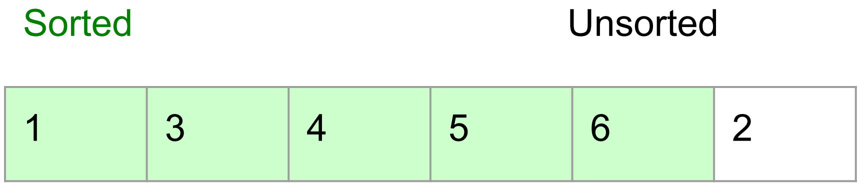
1	3	4	5	6	2

Round 3: Reached end of the array, so stop

- Maintains the following invariant: **at round i, array[0,i] is sorted**

Insertion Sort

- Insertion sort iterates over the array, swapping pairs of integers until they are in the correct position



Sorted Unsorted

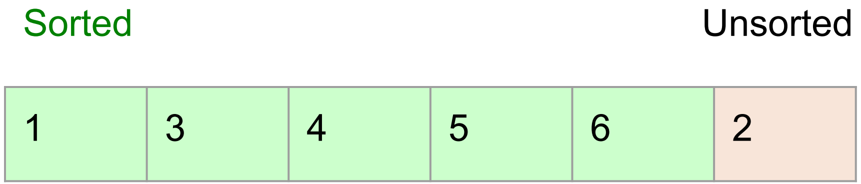
1	3	4	5	6	2

Round 3: Reached end of the array, so stop

- Maintains the following invariant: **at round i, array[0,i] is sorted**

Insertion Sort

- Insertion sort iterates over the array, swapping pairs of integers until they are in the correct position



Sorted

Unsorted

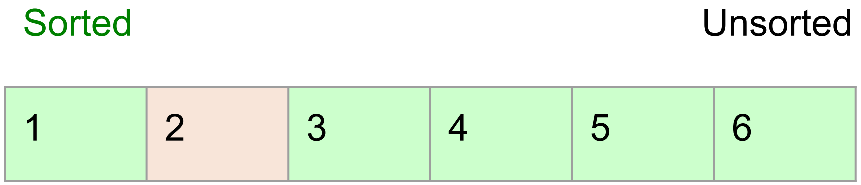
1	3	4	5	6	2

Round 4: Repeat process for 2

- Maintains the following invariant: **at round i, array[0,i] is sorted**

Insertion Sort

- Insertion sort iterates over the array, swapping pairs of integers until they are in the correct position



Sorted

Unsorted

1	2	3	4	5	6

Round 4: Repeat process for 2

- Maintains the following invariant: **at round i, array[0,i] is sorted**

Insertion Sort

- Insertion sort iterates over the array, swapping pairs of integers until they are in the correct position

Sorted Unsorted

1	2	3	4	5	6

Round 4: Array is fully sorted.

- Maintains the following invariant: **at round i, array[0,i] is sorted**

How to implement Insertion Sort?

```
// sort b[], an array of int
// inv: b[0..i] is sorted
for (int i= 0; i < b.length - 1; i= i+1) {
    // Push b[i+1] down to its sorted
    // position in b[0..i]
     int k= i+1;
     while (k > 0  &&  b[k-1] > b[k]) {
             swap(b,k-1,k);
             k--;
     }

}
```

Insertion Sort - Analysis

- How many comparisons does each round of the algorithm do?

Insertion Sort - Analysis

- How many comparisons does each round of the algorithm do?
    - Round 0, does at most 1
    - Round 1, at most 2. Round 2 at most 2, …
    - Round i does i+1 comparisons max

- How many rounds are there?

Insertion Sort - Analysis

- How many comparisons does each round of the algorithm do?
  - Round 0, does at most 1
  - Round 1, at most 2. Round 2 at most 2, …
  - Round i does i+1 comparisons max

- How many rounds are there?
  - N-1 rounds

Insertion Sort - Analysis

- How many comparisons does each round of the algorithm do?
  - Round 0, does at most 1
  - Round 1, at most 2. Round 2 at most 2, …
  - Round i does i+1 comparisons max

- How many rounds are there?
  - N-1 rounds

- How many comparisons in total?
  - n(n-1)/2

Insertion Sort - Analysis

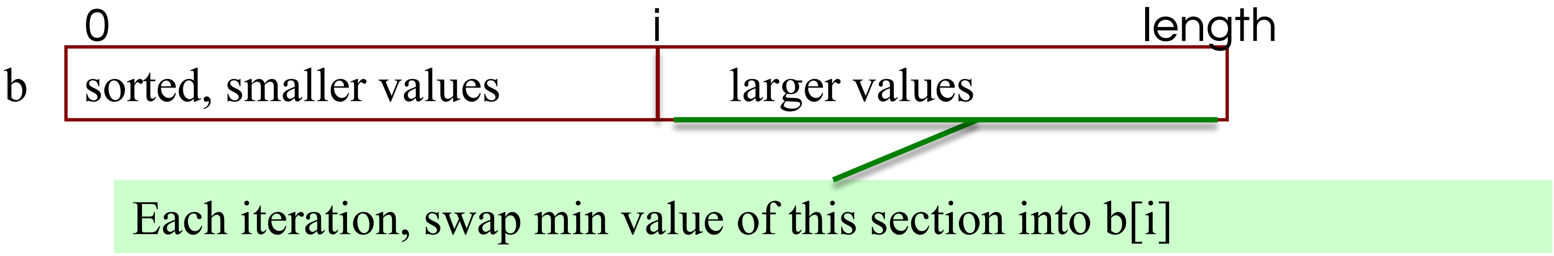
- Insertion sort is therefore O(n^2)

- In practice however, is it going to be expensive?
  - Can you think of scenarios where insertion sort is likely to perform well?

Selection Sort

- Selection sort has a similar invariant as insertion sort:
  - At round i, the positions before a[i] are already sorted

- Instead of swapping values, iterate over the unsorted array a[i..n-1] to find the minimum value, and place it in a[i].



Each iteration, swap min value of this section into b[i]

Selection Sort

- Let’s begin by looking through an example

- Consider the following array

3	6	4	5	1	2

Let’s sort it!

Selection Sort

- Selection sort over the array, placing the minimum element of the unsorted array in a[i]

3	6	4	5	1	2

- Maintains the following invariant: **at round i, the positions before array[i] are sorted**

Selection Sort

- Selection sort over the array, placing the minimum element of the unsorted array in a[i]

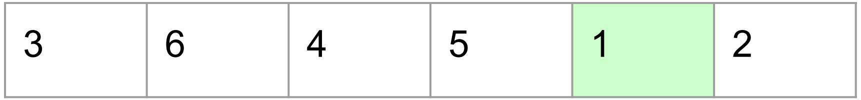
3	6	4	5	1	2

Round 0: Find the **minimum element** in a[i,n-1]

- Maintains the following invariant: **at round i, the positions before array[i] are sorted**

Selection Sort

- Selection sort over the array, placing the minimum element of the unsorted array in a[i]



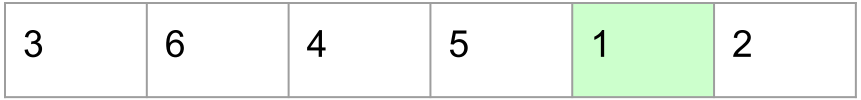
3	6	4	5	1	2

Round 0: Find the **minimum element** in a[i,n-1]

- Maintains the following invariant: **at round i, the positions before array[i] are sorted**

Selection Sort

- Selection sort over the array, placing the minimum element of the unsorted array in a[i]



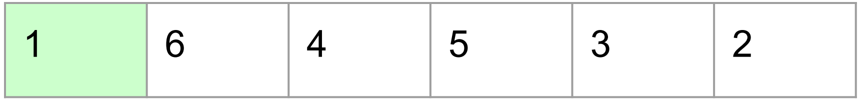
3	6	4	5	1	2

Round 0: Now swap with a[0] (a[i])

- Maintains the following invariant: **at round i, the positions before array[i] are sorted**

Selection Sort

- Selection sort over the array, placing the minimum element of the unsorted array in a[i]



1	6	4	5	3	2

Round 0: Now swap with a[0] (a[i])

- Maintains the following invariant: **at round i, the positions before array[i] are sorted**

Selection Sort

- Selection sort over the array, placing the minimum element of the unsorted array in a[i]

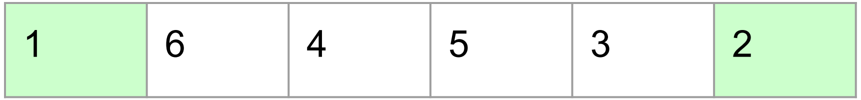
1	6	4	5	3	2

Round 1: Consider a[1] (= 6). Find the minimum in a[1,n-1]

- Maintains the following invariant: **at round i, the positions before array[i] are sorted**

Selection Sort

- Selection sort over the array, placing the minimum element of the unsorted array in a[i]



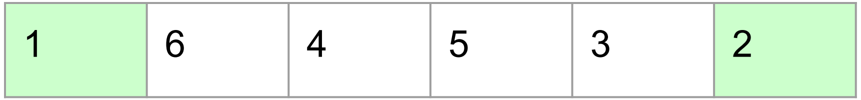
1	6	4	5	3	2

Round 1: Consider a[1] (= 6). Find the minimum in a[1,n-1]

- Maintains the following invariant: **at round i, the positions before array[i] are sorted**

Selection Sort

- Selection sort over the array, placing the minimum element of the unsorted array in a[i]



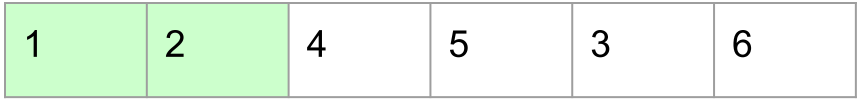
1	6	4	5	3	2

Round 1: Now swap with a[1]

- Maintains the following invariant: **at round i, the positions before array[i] are sorted**

Selection Sort

- Selection sort over the array, placing the minimum element of the unsorted array in a[i]



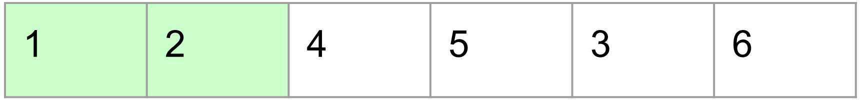
1	2	4	5	3	6

Round 1: Now swap with a[1]

- Maintains the following invariant: **at round i, the positions before array[i] are sorted**

Selection Sort

- Selection sort over the array, placing the minimum element of the unsorted array in a[i]



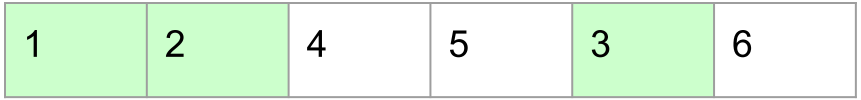
1	2	4	5	3	6

Round 2: Look at a[2]. Find minimum

- Maintains the following invariant: **at round i, the positions before array[i] are sorted**

Selection Sort

- Selection sort over the array, placing the minimum element of the unsorted array in a[i]



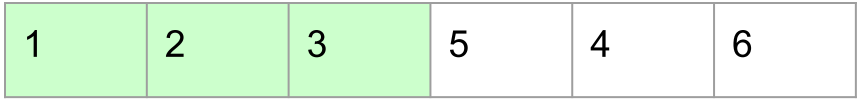
1	2	4	5	3	6

Round 3: Look at a[2]. Find minimum

- Maintains the following invariant: **at round i, the positions before array[i] are sorted**

Selection Sort

- Selection sort over the array, placing the minimum element of the unsorted array in a[i]



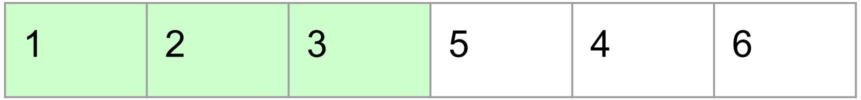
1	2	3	5	4	6

Round 2: Swap with a[2]

- Maintains the following invariant: **at round i, the positions before array[i] are sorted**

Selection Sort

- Selection sort over the array, placing the minimum element of the unsorted array in a[i]



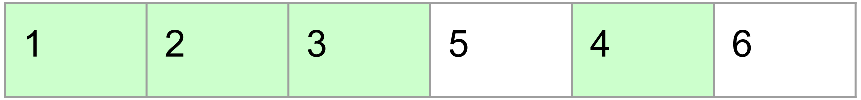
1	2	3	5	4	6

Round 3: Look at a[3]

- Maintains the following invariant: **at round i, the positions before array[i] are sorted**

Selection Sort

- Selection sort over the array, placing the minimum element of the unsorted array in a[i]



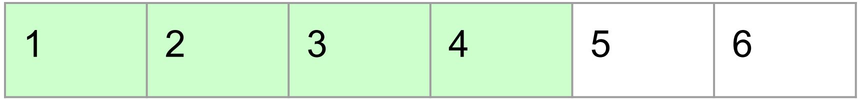
1	2	3	5	4	6

Round 3: Look at a[3]. Find minimum

- Maintains the following invariant: **at round i, the positions before array[i] are sorted**

Selection Sort

- Selection sort over the array, placing the minimum element of the unsorted array in a[i]



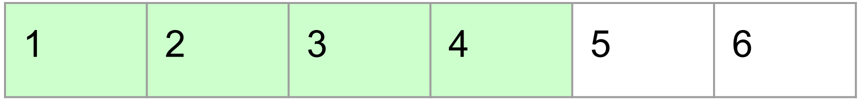
1	2	3	4	5	6

Round 3: Swap with a[3]

- Maintains the following invariant: **at round i, the positions before array[i] are sorted**

Selection Sort

- Selection sort over the array, placing the minimum element of the unsorted array in a[i]



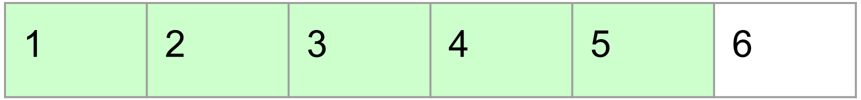
1	2	3	4	5	6

Round 4: Look at a[4]. Find minimum

- Maintains the following invariant: **at round i, the positions before array[i] are sorted**

Selection Sort

- Selection sort over the array, placing the minimum element of the unsorted array in a[i]



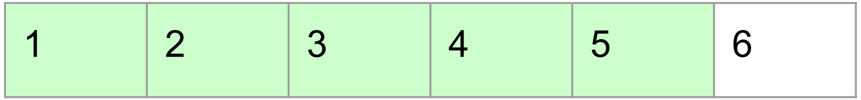
1	2	3	4	5	6

Round 4: Swap with itself

- Maintains the following invariant: **at round i, the positions before array[i] are sorted**

Selection Sort

- Selection sort over the array, placing the minimum element of the unsorted array in a[i]



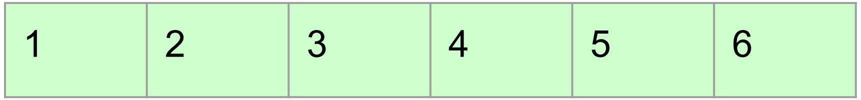
1	2	3	4	5	6

Round 5: Look at a[5].

- Maintains the following invariant: **at round i, the positions before array[i] are sorted**

Selection Sort

- Selection sort over the array, placing the minimum element of the unsorted array in a[i]



1	2	3	4	5	6

Array is sorted!

- Maintains the following invariant: **at round i, the positions before array[i] are sorted**

How to implement Selection Sort?

```
// sort b[], an array of int
// inv: positions before b[i] are sorted
for (int i= 0; i < b.length - 1; i= i+1) {
    //  Find the smallest element in
    //a [i..end] and swap it into a[i]
     int k= i+1;
     while (k > 0  &&  b[k-1] > b[k]) {
            swap(b,k-1,k);
            k--;
     }

}
```

Selection Sort - Analysis

- How many operations does each round of the algorithm do?

Selection Sort - Analysis

- How many comparisons does each round of the algorithm do?
  - Round 0, n (+1 swap)
  - Round 1, n-1 (+ 1 swap)
  - Round i does n - i comparisons (+ 1 swap)

- How many rounds are there?

Selection Sort - Analysis

- How many comparisons does each round of the algorithm do?
  - Round 0, n (+1 swap)
  - Round 1, n-1 (+ 1 swap)
  - Round i does n - i comparisons (+ 1 swap)

- How many rounds are there?
  - n

Selection Sort - Analysis

- How many comparisons does each round of the algorithm do?
    - Round 0, n (+1 swap)
    - Round 1, n-1 (+ 1 swap)
    - Round i does n - i comparisons (+ 1 swap)

- How many rounds are there?
    - N

- How many comparisons in total?

Selection Sort - Analysis

- How many comparisons does each round of the algorithm do?
  - Round 0, n (+1 swap)
  - Round 1, n-1 (+ 1 swap)
  - Round i does n - i comparisons (+ 1 swap)

- How many rounds are there?
  - N

- How many comparisons in total?
  - n(n+1)/2

Selection Sort - Analysis

- Selection sort is therefore O(n^2)
- In practice however, is it going to be expensive?
- What is likely to be faster ?
  - Insertion sort?
  - Selection sort?

Merge Sort

- Could recursion save the day?

- Instead of processing one large big array, what if we recursively subdivided each array into smaller arrays, **sorted those subarrays,** and then **merged** the sorted arrays together at the end?

- What would be the base case of merge sort?

Merge Sort

- Could recursion save the day?

- Instead of processing one large big array, what if we recursively subdivided each array into smaller arrays, **sorted those subarrays,** and then **merged** the sorted arrays together at the end?

- What would be the base case of merge sort?

Merge Sort

- Let’s begin by looking through an example
- Consider the following array

3	6	4	5	1	2

Let’s sort it!

Merge Sort

3	6	4	5	1	2

Let’s first partition the array into two smaller arrays

Merge Sort

3	6	4	5	1	2

Let’s first partition the array into two smaller arrays

3	6	4

5	1	2

Merge Sort

3	6	4	5	1	2

Let’s first partition the array into two smaller arrays

3	6	4

5	1	2

And again

Merge Sort

3	6	4	5	1	2

Let's first partition the array into two smaller arrays

3	6	4

5	1	2

And again

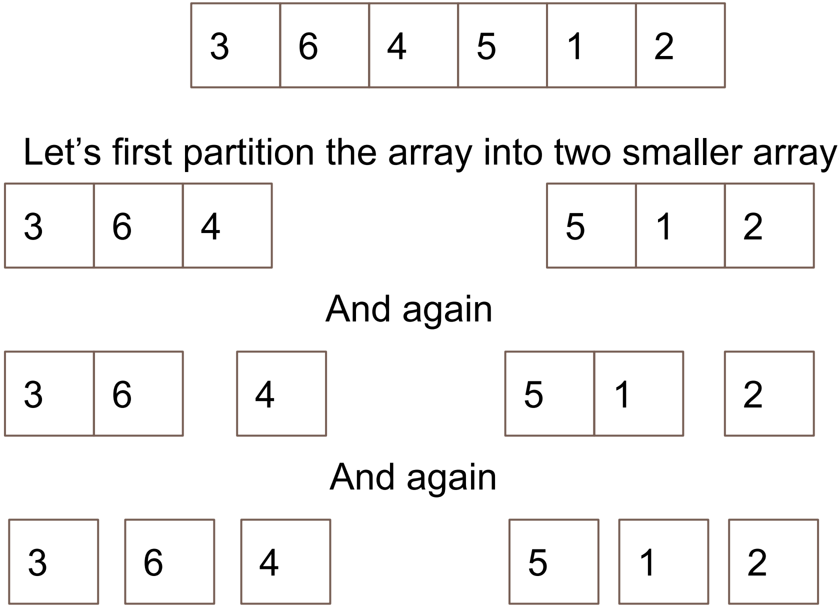
3	6

4

5	1

2

Merge Sort



3 6 4 5 1 2

Let’s first partition the array into two smaller arrays

3 6 4　　5 1 2

And again

3 6　4　　5 1　2

And again

3　6　4　　5　1　2

Merge Sort

3 6 4 5 1 2

Let’s first partition the array into two smaller arrays

3 6 4   5 1 2

And again

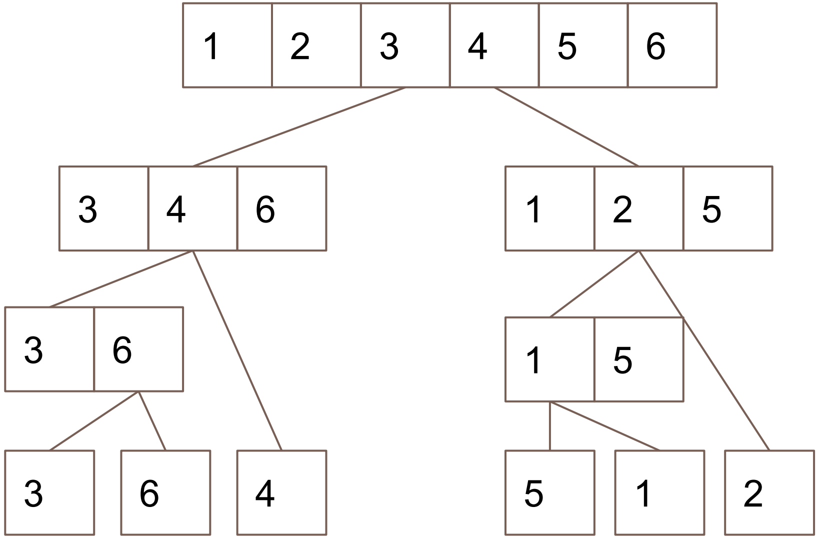
3 6   4   5 1   2

And again

3   6   4   5   1   2

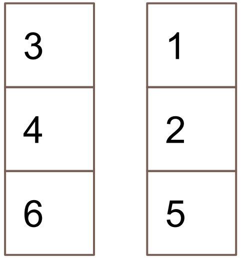
Arrays of size 1 are sorted. Base case!

Merge Sort

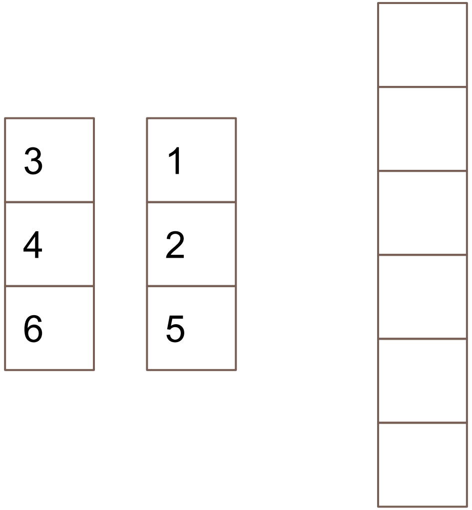


Now merge sorted arrays back together

How to merge two sorted arrays?

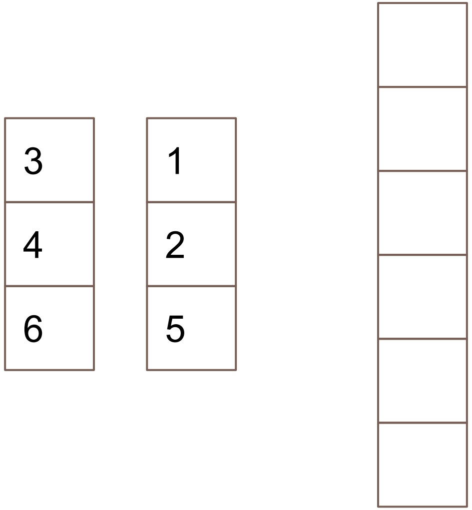


How to merge two sorted arrays?



Step 1: Create an array of size a.length + b.length

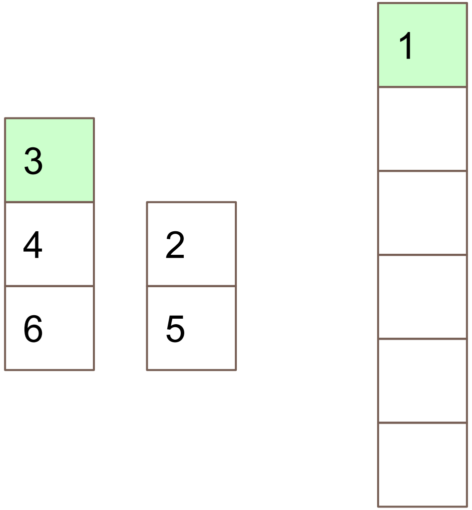
How to merge two sorted arrays?



Step 1: Create an array of size a.length + b.length

Step 2: Where can we find the smallest element of the new array?

How to merge two sorted arrays?

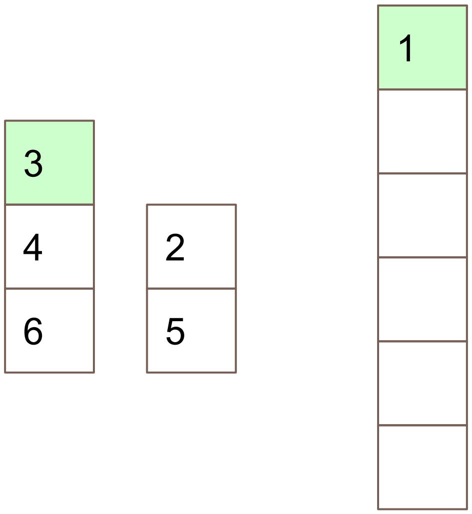


Step 1: Create an array of size a.length + b.length

Step 2: Where can we find the smallest element of the new array?
It is either the smallest element of a or the smallest element of b

Step 3: Where can we find the second smallest element of the new array?
Depending on what we chose last, it is either the first element of a/b or the second element of a/b

How to merge two sorted arrays?



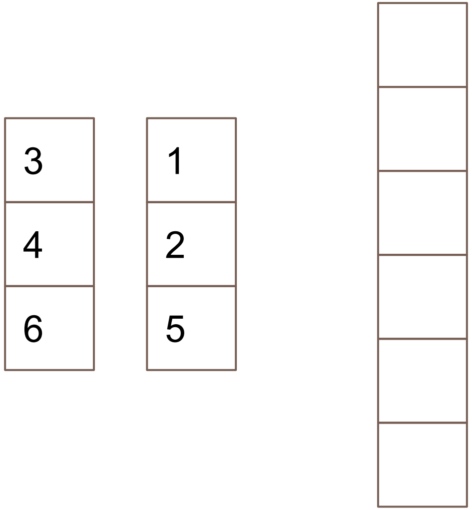
Step 1: Create an array of size a.length + b.length

Step 2: Where can we find the smallest element of the new array?
It is either the smallest element of a or the smallest element of b

Step 3: Where can we find the second smallest element of the new array?
Depending on what we chose last, it is either the first element of a/b or the second element of a/b

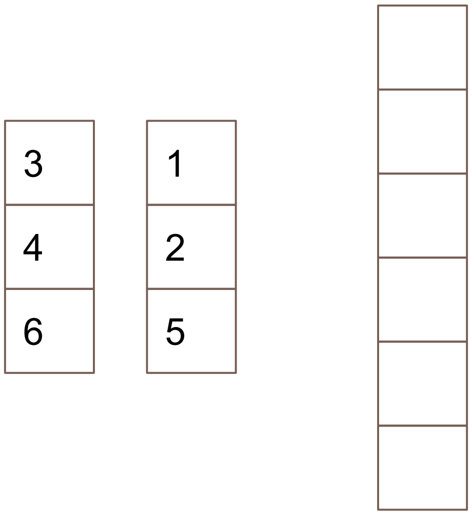
Keep two indices **headA** and **headB**. When select an element of **a,** increameant **headA**. When select an element of **b**, increment **headB.** Then test for a[headA]<=b[headB] at every iteration

How to merge two sorted arrays?



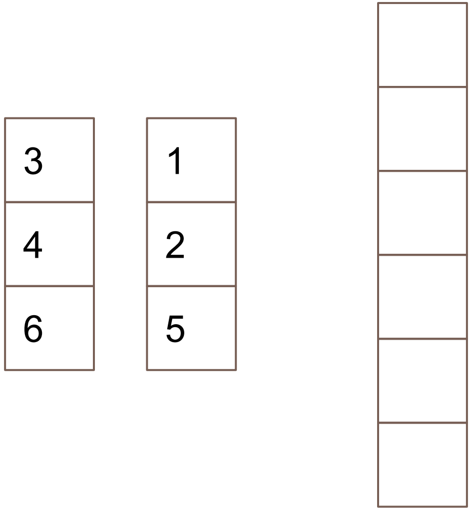
```
headA = 0;
headB = 0;
```

How to merge two sorted arrays?



```
headA = 0;
headB = 0;
```

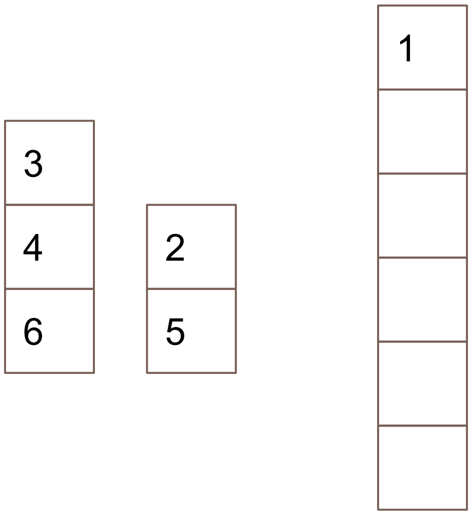
How to merge two sorted arrays?



```
headA = 0;
headB = 0;

a[headA] <= b[headB]?
        No -> result[0] = b[headB]; headB++
```

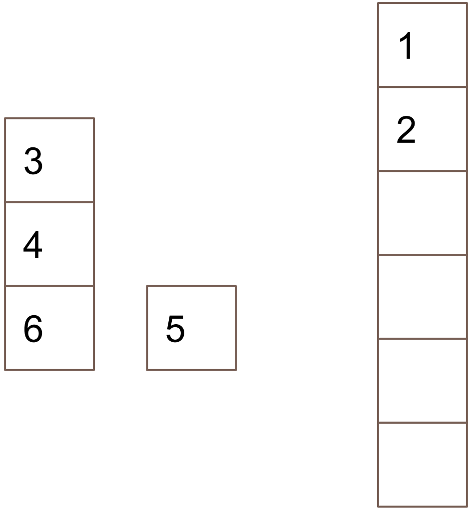
How to merge two sorted arrays?



```
headA = 0;
headB = 1;

a[headA] <= b[headB]?
        No -> result[1] = b[headB]; headB++
```

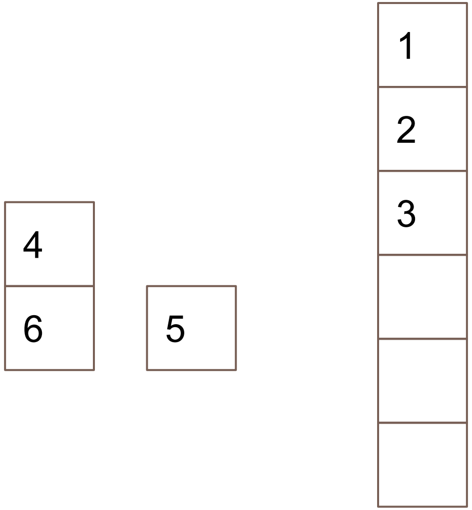
How to merge two sorted arrays?



```
headA = 0;
headB = 2;

a[headA] <= b[headB]?
        Yes -> result[2] = a[headA]; headA++
```

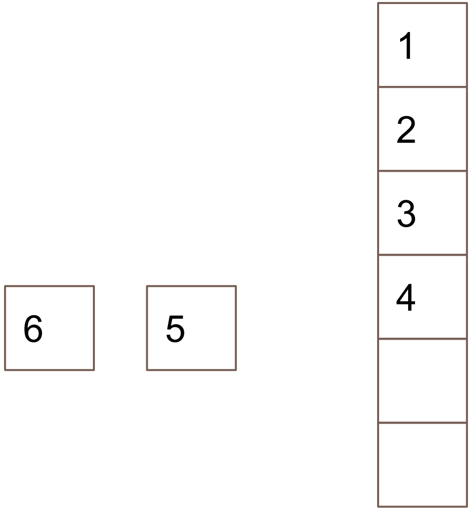
How to merge two sorted arrays?



```
headA = 1;
headB = 2;

a[headA] <= b[headB]?
        Yes -> result[3] = a[headA]; headA++
```

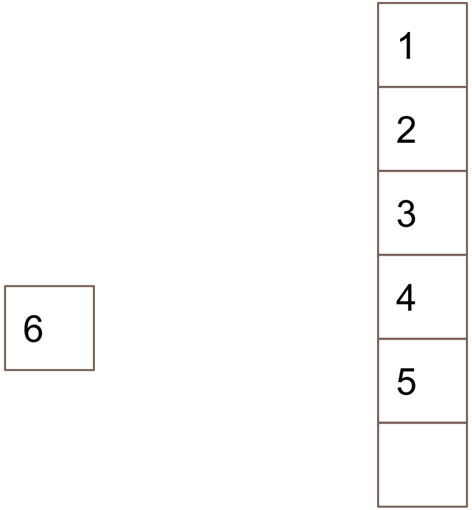
How to merge two sorted arrays?



```
headA = 2;
headB = 2;

a[headA] <= b[headB]?
        Yes -> result[3] = a[headA]; headA++
```

How to merge two sorted arrays?



```
headA = 1;
headB = 2;

a[headA] <= b[headB]?
        No -> result[3] = b[headB]; headB++
```

How to merge two sorted arrays?

1
2
3
4
5
6

```
headA = 2;
headB = 2;

headB >= b.length.
        result[3] = a[headA]; headA++
```

Merge sort complexity analysis

- And more generally: how do we analyse the complexity of a recursive algorithm.

Merge sort complexity analysis

- And more generally: how do we analyse the complexity of a recursive algorithm.

- First: what is the complexity of the function that merges two sorted arrays?

Merge sort complexity analysis

- And more generally: how do we analyse the complexity of a recursive algorithm.

- First: what is the complexity of the function that merges two sorted arrays?
  - One pass over the arrays, so O(n)

Merge sort complexity analysis

- And more generally: how do we analyse the complexity of a recursive algorithm.

- First: what is the complexity of the function that merges two sorted arrays?
  - One pass over the arrays, so O(n)

- When algorithm contains recursive call, can describe its running time by a **recurrence equation** that describes running time of a problem of size n in terms of running time on smaller inputs.

Recurrence Relations

- Let T(n) be the running time on a problem of size n.

- If the problem is small enough (ex: base case), say n<= c for some constant c, solving the base case takes constant time O(1)

- Assume that the recursive call yields ***a*** subproblems, each of which is **1/b** of the size of the original. It takes time **T(n/b)** to solve one subproblem of size **n/b** and so it takes **aT(n/b)** to solve **a** of them. If it takes **D(n)** to subdivide the arrays, and **C(n)** to combine them, then we have the following recurrence rel

$$T(n) = O(1) \text{ if } n <= c$$
$$T(n) = aT(n/b) + D(n) + C(n) \text{ otherwise}$$

Recurrence Relation for Merge Sort

- In the general case:
  - $T(n) = O(1)$ if $n <= c$

    $T(n) = aT(n/b) + D(n) + C(n)$ otherwise

- For merge sort:

Recurrence Relation for Merge Sort

- In the general case:
  - $T(n) = O(1)$ if $n <= c$
    $T(n) = aT(n/b) + D(n) + C(n)$ otherwise

- For merge sort:
  - $a = 2$, $b = 2$

Recurrence Relation for Merge Sort

- In the general case:
  - T(n) = O(1) if n<= c

    T(n) = aT(n/b) + D(n) + C(n) otherwise

- For merge sort:
  - a = 2, b = 2
  - D(n) is O(1) (computes middle of the subarray)

Recurrence Relation for Merge Sort

- In the general case:
  - $T(n) = O(1)$ if $n <= c$
    $T(n) = aT(n/b) + D(n) + C(n)$ otherwise

- For merge sort:
  - $a = 2$, $b = 2$
  - $D(n)$ is $O(1)$ (computes middle of the subarray)
  - $C(n)$ is $O(n)$ (merge procedure of sorted arrays)

Recurrence Relation for Merge Sort

- In the general case:
  - $T(n) = O(1)$ if $n <= c$
    $T(n) = aT(n/b) + D(n) + C(n)$ otherwise

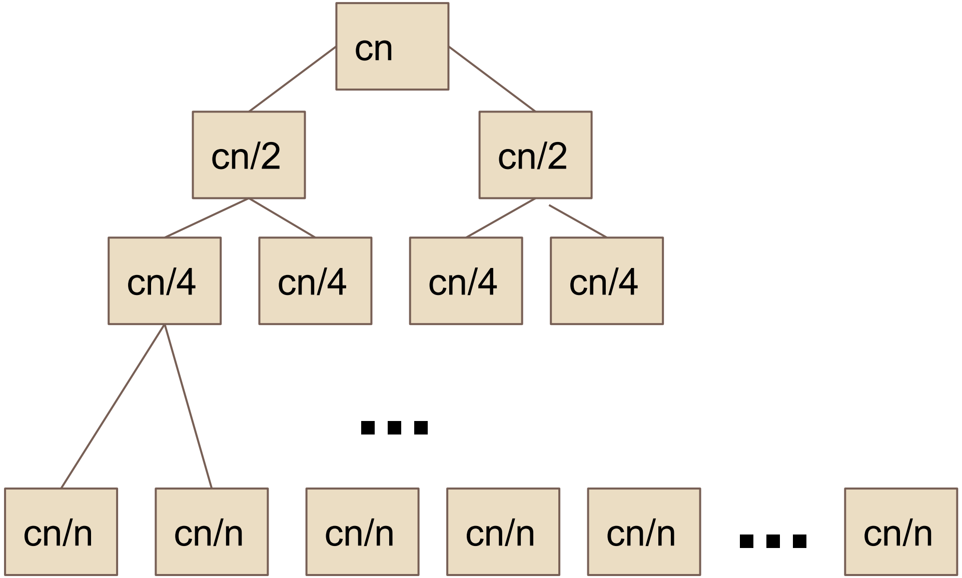
- For merge sort:
  - $a = 2$, $b = 2$
  - $D(n)$ is $O(1)$ (computes middle of the subarray)
  - $C(n)$ is $O(n)$ (merge procedure of sorted arrays)

- $T(n) = c$ if $n = 1$
  $T(n) = 2T(n/2) + cn$ if $n > 1$

(For simplicitly lets assume that n is a power of 2)

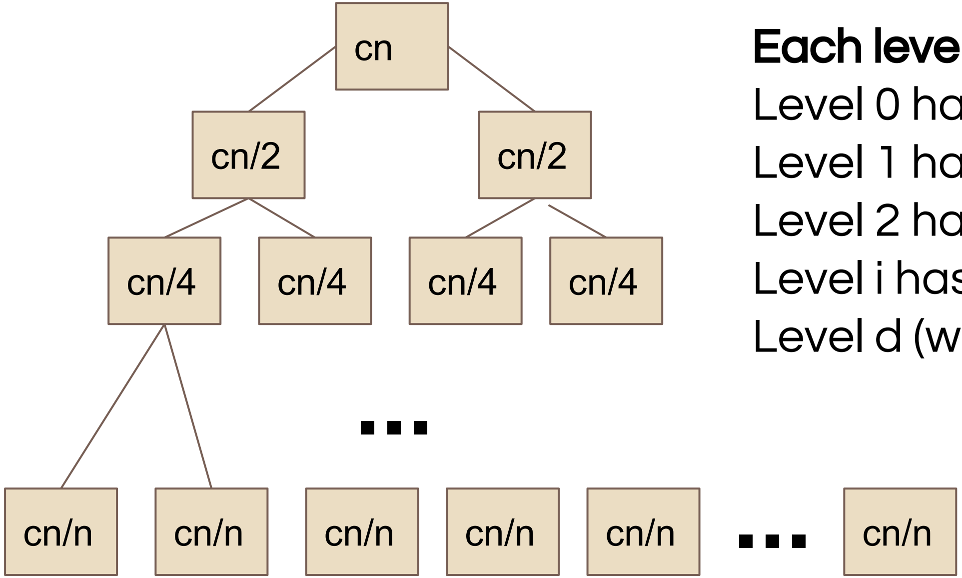
How do we solve the recurrence?

- Use a **recurrence tree**
  - Graphically lay out the cost of each level of the recursion in a tree-structure.



How do we solve the recurrence?

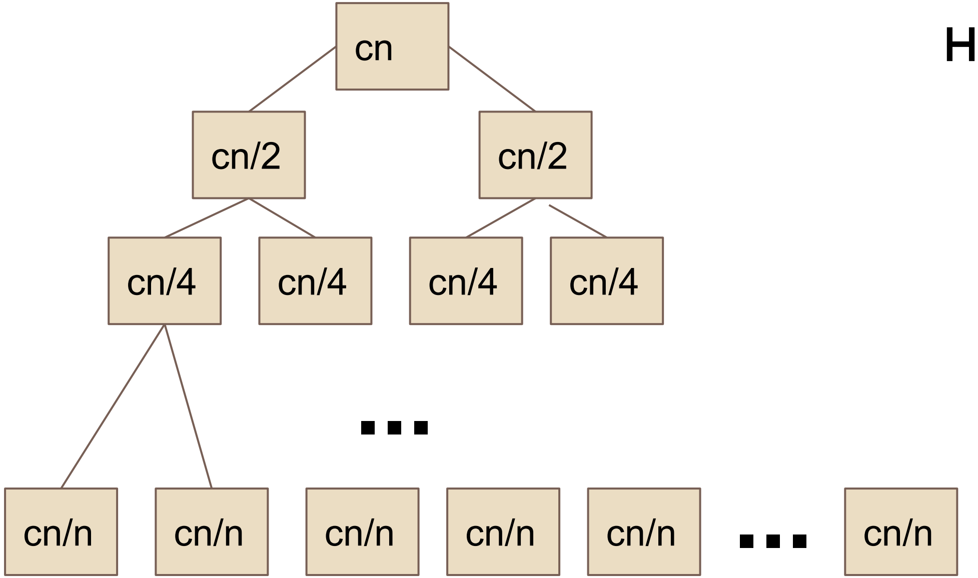
- Use a **recurrence tree**
  - Graphically lay out the cost of each level of the recursion in a tree-structure.



Each level in the tree has exactly cn cost:
Level 0 has cn
Level 1 has cn/2 + cn/2 = cn
Level 2 has cn/4 + cn/4 + cn/4 + cn/4 = cn
Level i has (i+1) * cn/2^i = cn
Level d (where n = 2^d) has n * cn/2^d = n * c = cn

How do we solve the recurrence?

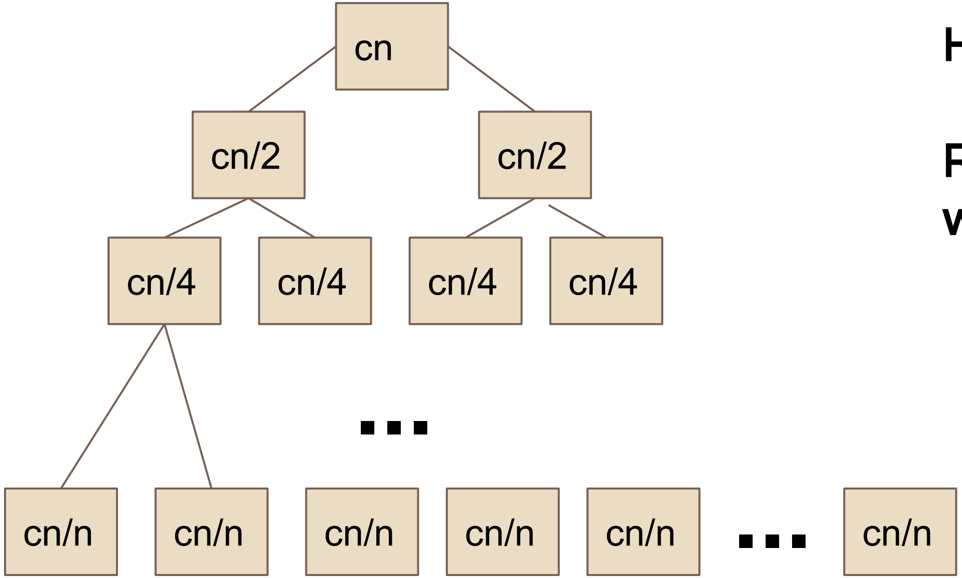
- Use a **recurrence tree**
  - Graphically lay out the cost of each level of the recursion in a tree-structure.



How many levels are there?

How do we solve the recurrence?

- Use a **recurrence tree**
  - Graphically lay out the cost of each level of the recursion in a tree-structure.

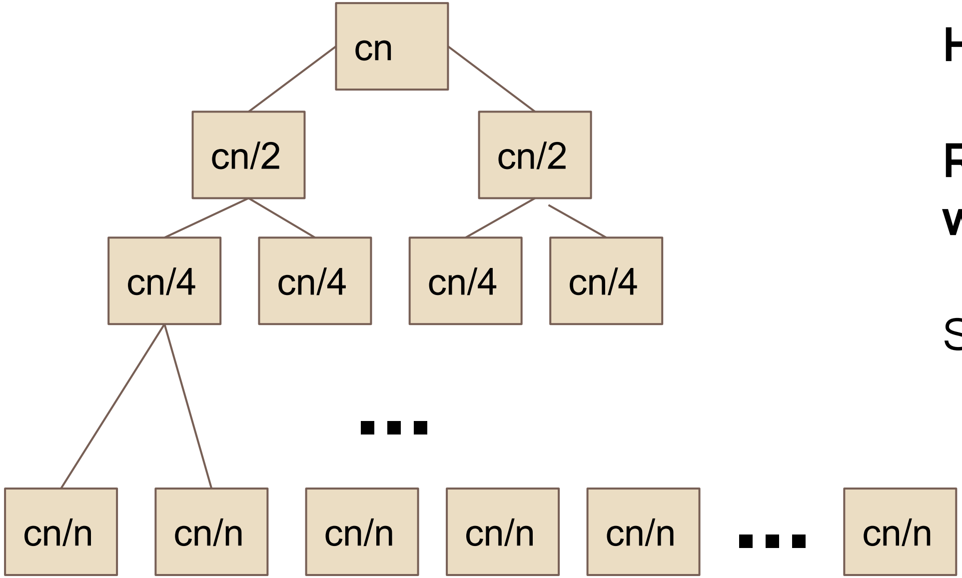


How many levels are there?

Remember that the recursion stops when the input size is 1

How do we solve the recurrence?

- Use a **recurrence tree**
  - Graphically lay out the cost of each level of the recursion in a tree-structure.



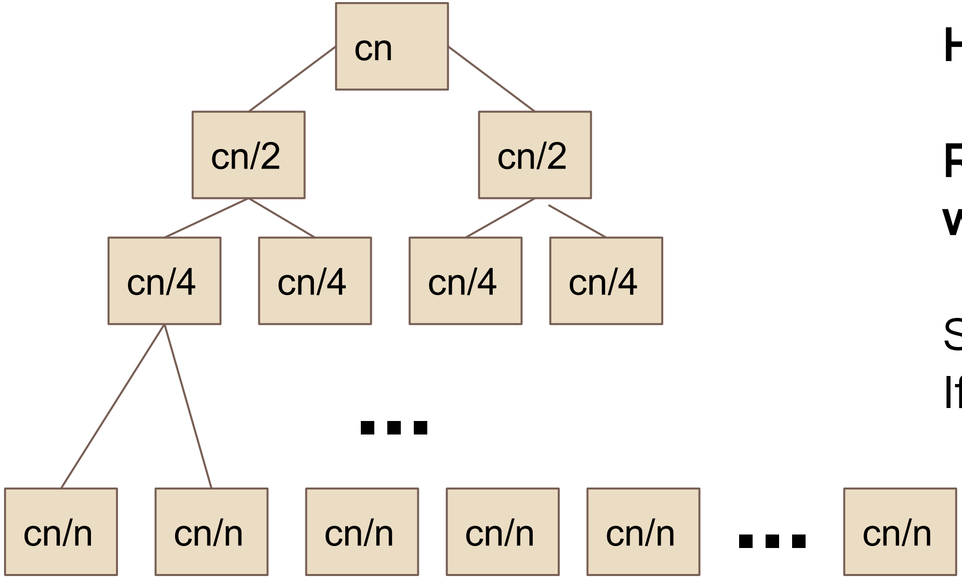
How many levels are there?

Remember that the recursion stops when the input size is 1

So if n = 2^1, there would be 2 levels

How do we solve the recurrence?

- Use a **recurrence tree**
  - Graphically lay out the cost of each level of the recursion in a tree-structure.



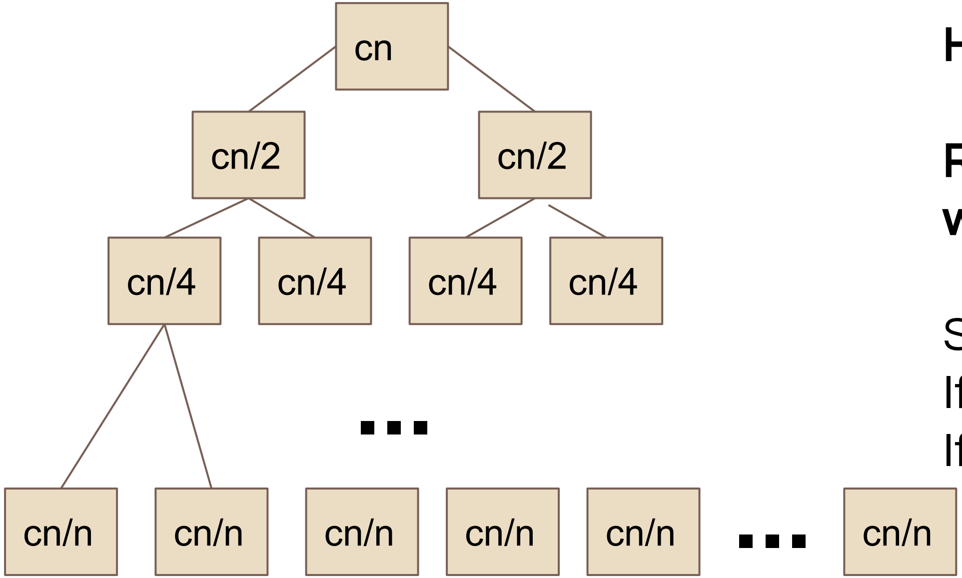
How many levels are there?

Remember that the recursion stops when the input size is 1

So if n = 2^1, there would be 2 levels
If n = 2^2, there would be 3 levels

How do we solve the recurrence?

- Use a **recurrence tree**
  - Graphically lay out the cost of each level of the recursion in a tree-structure.



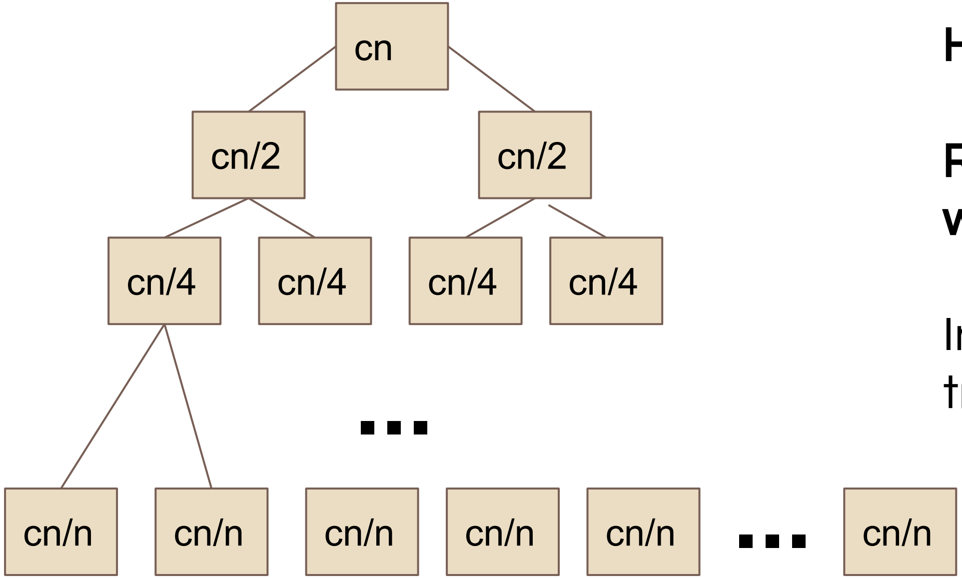
How many levels are there?

Remember that the recursion stops when the input size is 1

So if n = 2^1, there would be 2 levels
If n = 2^2, there would be 3 levels
If n = 2^3, there would be 4 levels

How do we solve the recurrence?

- Use a **recurrence tree**
  - Graphically lay out the cost of each level of the recursion in a tree-structure.



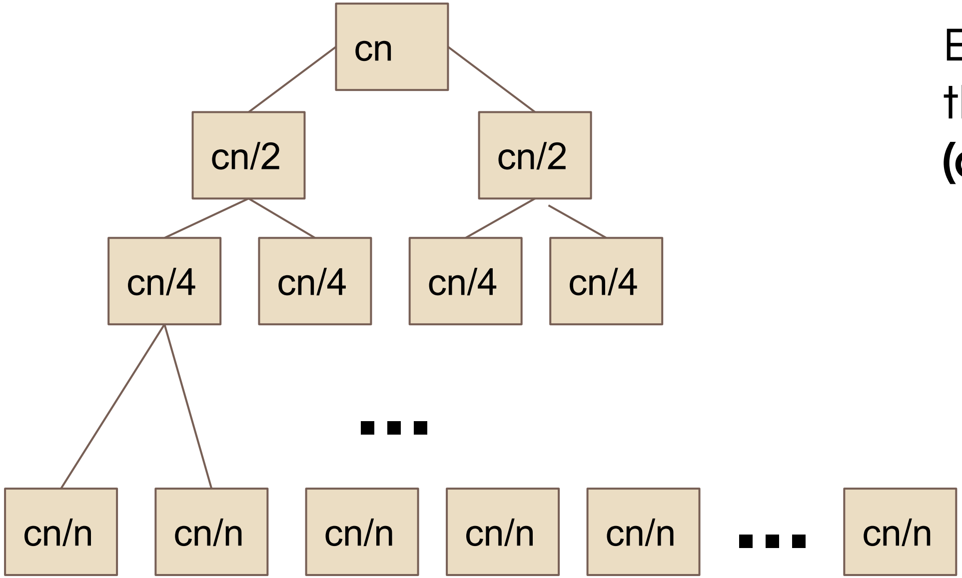
How many levels are there?

Remember that the recursion stops when the input size is 1

In general, there are lg(n) + 1 levels in the tree.

How do we solve the recurrence?

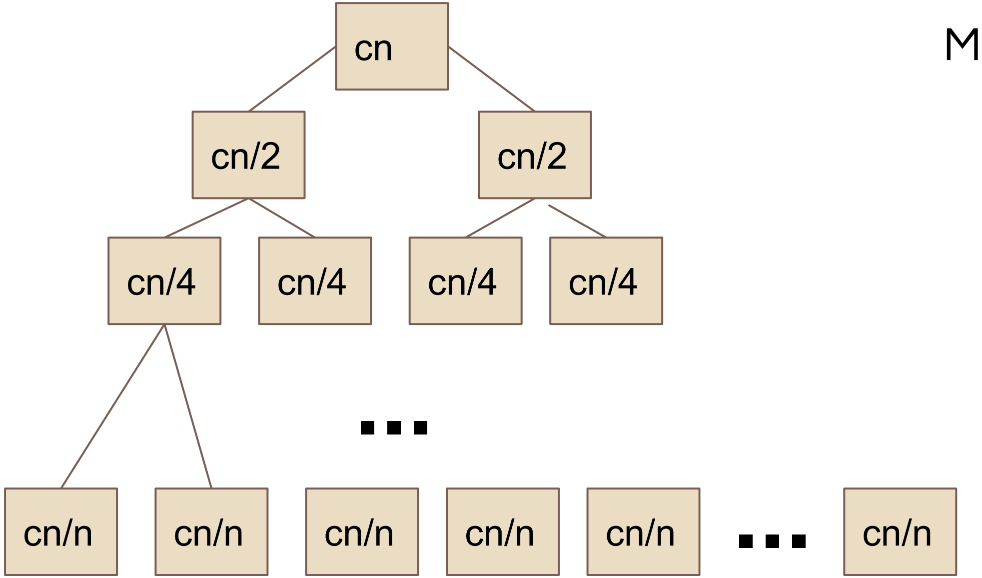
- Use a **recurrence tree**
  - Graphically lay out the cost of each level of the recursion in a tree-structure.



Each level of the tree does **cn** work and there are **lg(n) + 1** levels of the tree
(cn)(lg(n) + 1) = cn* lg(n) + cn

How do we solve the recurrence?

- Use a **recurrence tree**
  - Graphically lay out the cost of each level of the recursion in a tree-structure.



Merge sort is **O(nlgn)**

Merge sort vs others

- Insertion sort and selection sort have worse time complexity than merge sort.
- But, they have better space complexity as they can sort the data **in-place** whereas merge sort requires additional arrays (merge sort has **O(n)** space complexity)
- For small inputs, insertion sort is often faster!

Best of both worlds?

- Can we design an algorithm that sorts arrays in-place but with O(nlgn) complexity?

Best of both worlds?

- Can we design an algorithm that sorts arrays in-place but with O(nlgn) complexity?

- The answer is almost!
  - **Quicksort** sorts arrays in place and has O(nlgn) complexity in the best-case, but O(n^2) in the worst-case.

Quicksort

- Can we design an algorithm that sorts arrays in-place but with O(nlgn) complexity?
- The answer is almost!
  - **Quicksort** sorts arrays in place and has O(nlgn) complexity in the best-case, but O(n^2) in the worst-case.

Quicksort - Cute Story

Quicksort developed by Tony Hoare (he's currently 83, still works at Microsoft Research)

Developed Quicksort in 1958. But he could not explain it to his colleage, so he gave up on it.

Later, he saw a draft of the new language Algol 58 (which became Algol 60). It had recursive procedures, for the first time, in a procedural programming language. "Ah!". he said. "I know how to write it better now". 15 minutes later, his colleague also understood it.

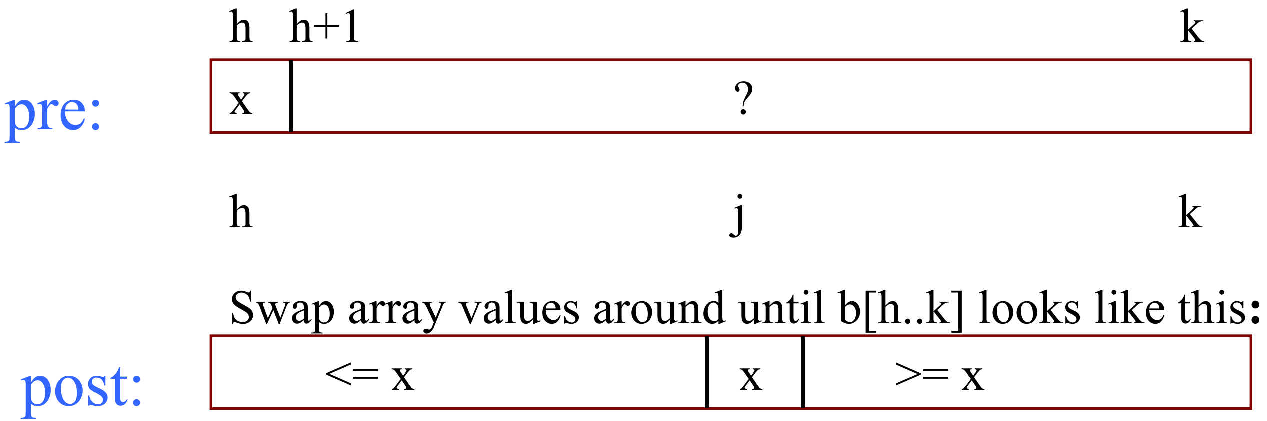
Fun fact: at university, we had a course called **Hoare Logic**, based on what he invented. He attended our lectures a few times :-). Nothing like attending an entire course named after you.

Quicksort - Key Idea

- Quicksort is **recursive** like merge sort.
- Unlike merge sort, however, quicksort first **processes** the array before partitioning the array in two.
- This processing allows quicksort to have better space complexity
- Idea is to pick a **pivot element** and partition the array into those bigger than the pivot and those smaller than the pivot, calling quicksort recursively on each side of the array.

Quicksort - Partitioning

- Pick a **pivot** (any element in the array) and partition the array sy such that all elements smaller than the pivot are to the left of the pivot, all the elements greater than the pivot are to the right.



h h+1 k

pre: x ?

h j k

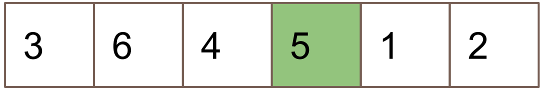
Swap array values around until b[h..k] looks like this:

post: <= x x >= x

x is called the pivot

Quicksort - Example

- Let’s sort this array (again)

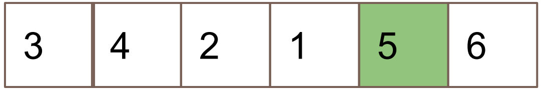


3	6	4	5	1	2

Select 5 as the pivot

Quicksort - Example

- Let’s sort this array (again)

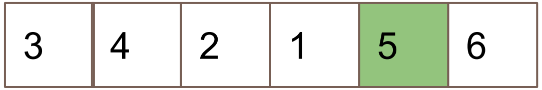


3	4	2	1	5	6

Partition the array

Quicksort - Example

- Let’s sort this array (again)

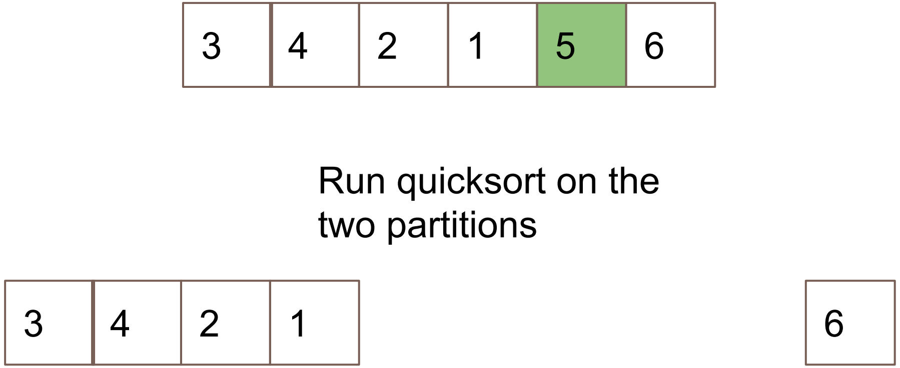


3	4	2	1	5	6

Run quicksort on the
two partitions

Quicksort - Example

- Let’s sort this array (again)



3	4	2	1	5	6

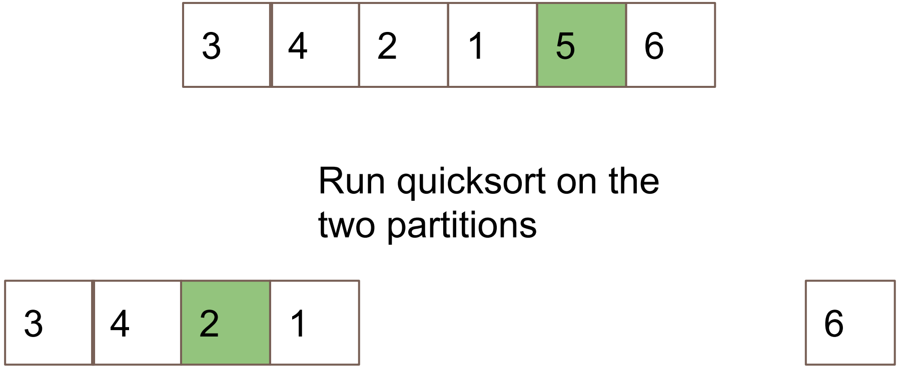
Run quicksort on the two partitions

3	4	2	1

6

Quicksort - Example

- Let’s sort this array (again)



3	4	2	1	5	6

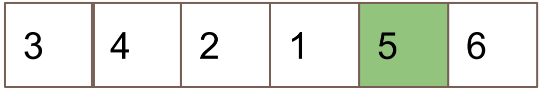
Run quicksort on the two partitions

3	4	2	1

6

Quicksort - Example

- Let’s sort this array (again)



3	4	2	1	5	6

Run quicksort on the two partitions

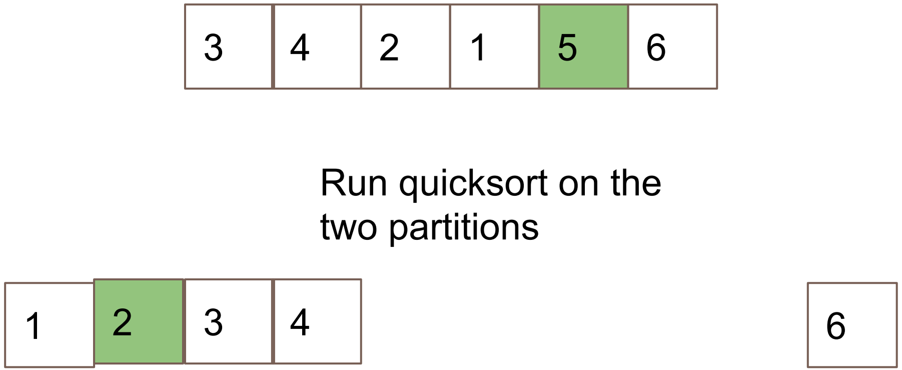
3	4	2	1

6

Partition Array

Quicksort - Example

- Let’s sort this array (again)



3	4	2	1	🟩 5	6

Run quicksort on the two partitions

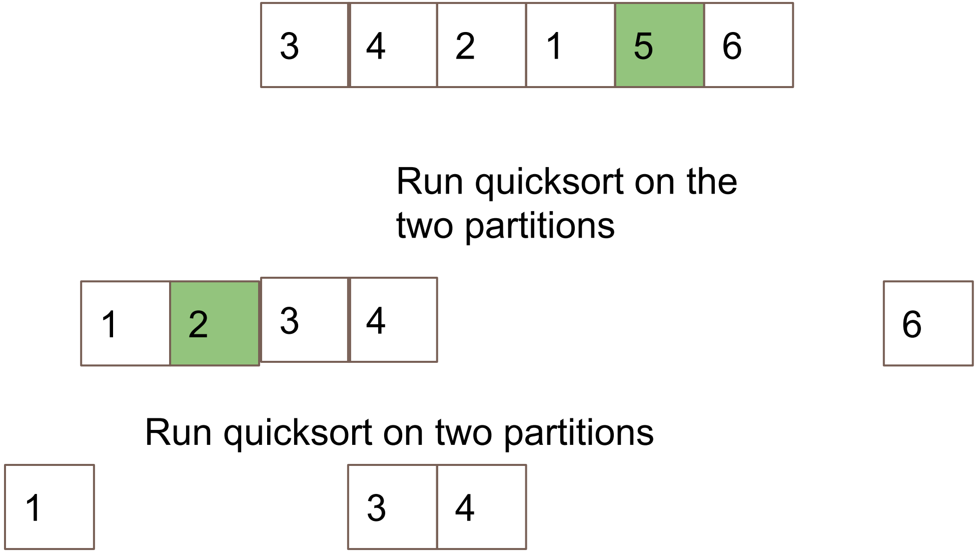
1	🟩 2	3	4

6

Partition Array

Quicksort - Example

- Let’s sort this array (again)

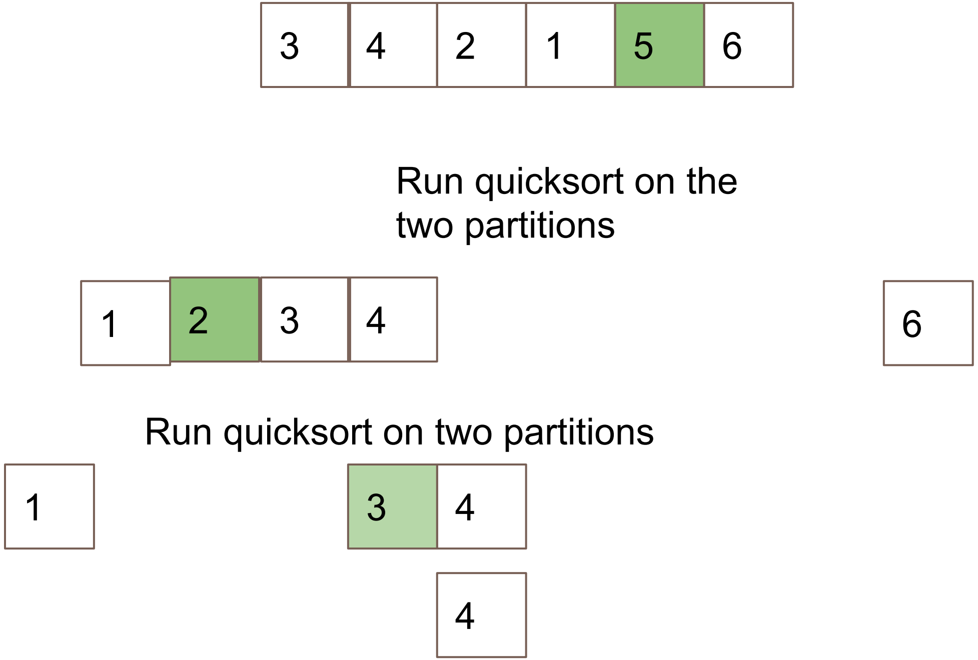


Run quicksort on the two partitions

Run quicksort on two partitions

Quicksort - Example

- Let’s sort this array (again)

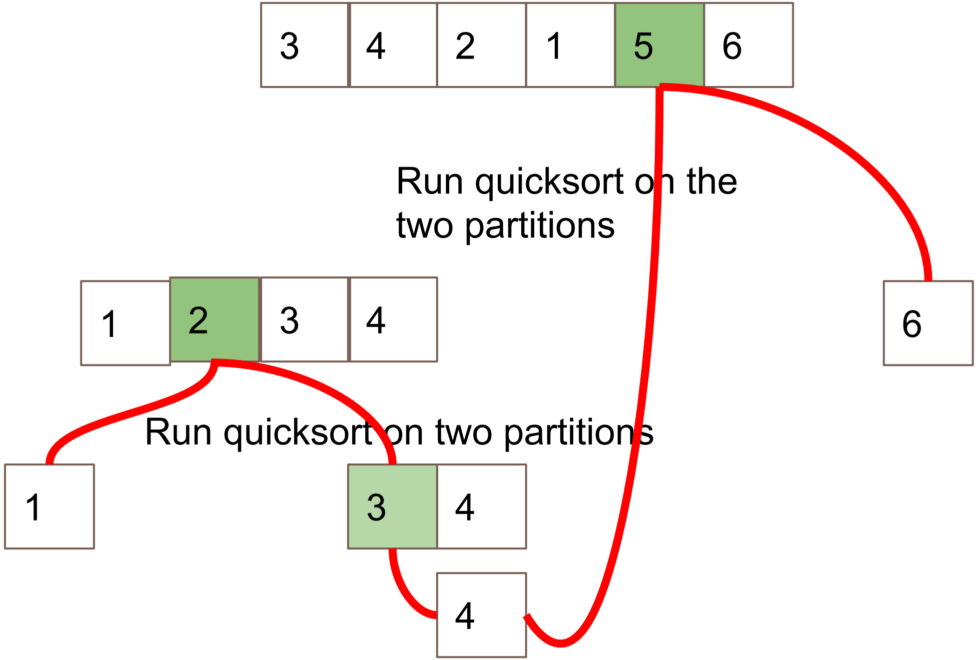


Run quicksort on the two partitions

Run quicksort on two partitions

Quicksort - Example

- Let's sort this array (again)



Run quicksort on the two partitions

Run quicksort on two partitions

When we reach the base case, no need to merge. The array is already sorted!

Back to the partition algorithm

- The secret sauce of quicksort is its partitioning algorithm that **partitions the array in place**

- Partition function as it executes, partitions the array into four regions:

p	i	j	A[r]
<=x	> x	unrestricted	x

Define several terms: p, the beginning index of the array that we want to partition. **i** is the start of the region for which >x. j is the end of the region for which >x. a[r] is the pivot x

Back to the partition algorithm

- The secret sauce of quicksort is its partitioning algorithm that **partitions the array in place**

- Partition function as it executes, partitions the array into four regions:

p i j A[r]

<=x > x unrestricted x

Goes in a loop from p to r-1, and maintains the following invariant for each element a[k] in the array

  - If p <= k < =i, then a[k] <= x
  - If i+1 <=k<=j-1, then A[k]>x

Back to the partition algorithm

- Let’s sort this array

2	8	7	1	3	5	6	4

Goes in a loop from p to r-1, and maintains the following invariant for each element a[k] in the array

- If p <= k < =i, then a[k] <= x
- If i+1 <=k<=j-1, then A[k]>x

Back to the partition algorithm

- Let’s sort this array

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

Initialise i to p-1, j to p, and select the last element as the pivot

Goes in a loop from p to r-1, and maintains the following invariant for each element a[k] in the array

- If p <= k < =i, then a[k] <= x
- If i+1 <=k<=j-1, then A[k]>x

Back to the partition algorithm

- Let’s sort this array

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

Now loop from j = p to j = r-1

Goes in a loop from p to r-1, and maintains the following invariant for each element a[k] in the array

- If $p <= k < =i$, then $a[k] <= x$
- If $i+1 <=k<=j-1$, then $A[k]>x$

Back to the partition algorithm

- Let’s sort this array

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

If A[j] <= x, increment a[i] to indicate that there is now one element that is < x. Then swap A[j] with A[i]

Goes in a loop from p to r-1, and maintains the following invariant for each element a[k] in the array

- If p <= k < =i, then a[k] <= x
- If i+1 <=k<=j-1, then A[k]>x

Back to the partition algorithm

- Let’s sort this array

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

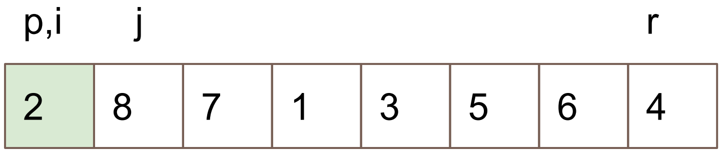
If A[j] <= x, increment a[i] to indicate that there is now one element that is < x. Then swap A[j] with A[i]

Goes in a loop from p to r-1, and maintains the following invariant for each element a[k] in the array

- If p <= k < =i, then a[k] <= x
- If i+1 <=k<=j-1, then A[k]>x

Back to the partition algorithm

- Let’s sort this array



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

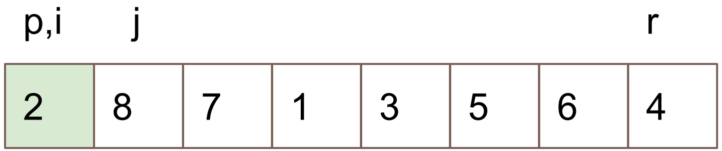
If A[i]> x, then do not change i, and simply increment j. The partition between a[i+1] and a[j-1] denotes the values that are greater than the pivot

Goes in a loop from p to r-1, and maintains the following invariant for each element a[k] in the array

- If p <= k < =i, then a[k] <= x
- If i+1 <=k<=j-1, then A[k]>x

Back to the partition algorithm

- Let’s sort this array



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

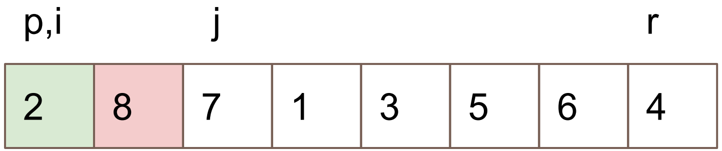
If A[i]> x, then do not change i, and simply increment j. The partition between a[i+1] and a[j-1] denotes the values that are greater than the pivot

Goes in a loop from p to r-1, and maintains the following invariant for each element a[k] in the array

- If p <= k < =i, then a[k] <= x
- If i+1 <=k<=j-1, then A[k]>x

Back to the partition algorithm

- Let’s sort this array



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

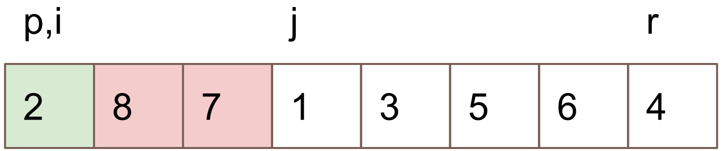
If A[i]> x, then do not change i, and simply increment j. The partition between a[i+1] and a[j-1] denotes the values that are greater than the pivot

Goes in a loop from p to r-1, and maintains the following invariant for each element a[k] in the array

- If p <= k < =i, then a[k] <= x
- If i+1 <=k<=j-1, then A[k]>x

Back to the partition algorithm

- Let’s sort this array



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

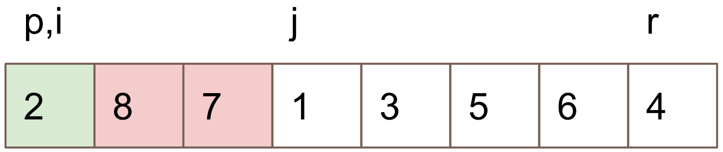
If A[j] <= x, increment a[i] to indicate that there is now one element that is < x. Then swap A[j] with A[i]

Goes in a loop from p to r-1, and maintains the following invariant for each element a[k] in the array

- If p <= k < =i, then a[k] <= x
- If i+1 <=k<=j-1, then A[k]>x

Back to the partition algorithm

- Let’s sort this array



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

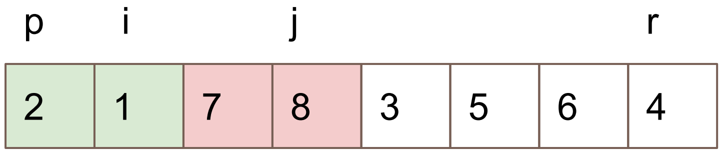
i = 0 + 1, so swap a[1] (= 8) with a[j] = 1.

Goes in a loop from p to r-1, and maintains the following invariant for each element a[k] in the array

- If p <= k < =i, then a[k] <= x
- If i+1 <=k<=j-1, then A[k]>x

Back to the partition algorithm

- Let’s sort this array



p	i		j				r
2	1	7	8	3	5	6	4

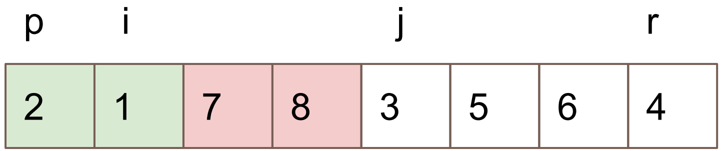
i = 0 + 1, so swap a[1] (= 8) with a[j] = 1.

Goes in a loop from p to r-1, and maintains the following invariant for each element a[k] in the array

- If p <= k < =i, then a[k] <= x
- If i+1 <=k<=j-1, then A[k]>x

Back to the partition algorithm

- Let's sort this array



p	i			j			r
2	1	7	8	3	5	6	4

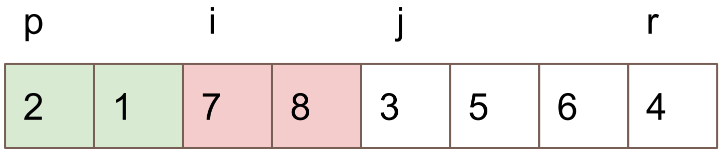
If A[j] <= x, increment a[i] to indicate that there is now one element that is < x. Then swap A[j] with A[i]

Goes in a loop from p to r-1, and maintains the following invariant for each element a[k] in the array

- If p <= k < =i, then a[k] <= x
- If i+1 <=k<=j-1, then A[k]>x

Back to the partition algorithm

- Let’s sort this array



p		i		j			r
2	1	7	8	3	5	6	4

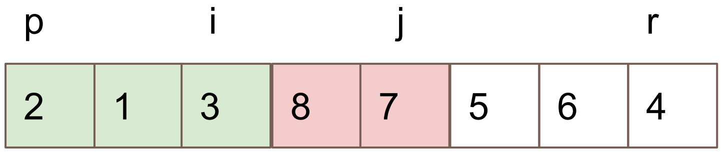
i = 1 + 1, so swap a[2] (= 7) with a[j] = 3.

Goes in a loop from p to r-1, and maintains the following invariant for each element a[k] in the array

- If p <= k < =i, then a[k] <= x
- If i+1 <=k<=j-1, then A[k]>x

Back to the partition algorithm

- Let’s sort this array



p		i		j			r
2	1	3	8	7	5	6	4

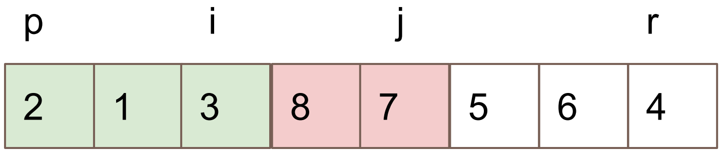
i = 1 + 1, so swap a[2] (= 7) with a[j] = 3.

Goes in a loop from p to r-1, and maintains the following invariant for each element a[k] in the array

- If p <= k < =i, then a[k] <= x
- If i+1 <=k<=j-1, then A[k]>x

Back to the partition algorithm

- Let’s sort this array



p		i		j			r
2	1	3	8	7	5	6	4

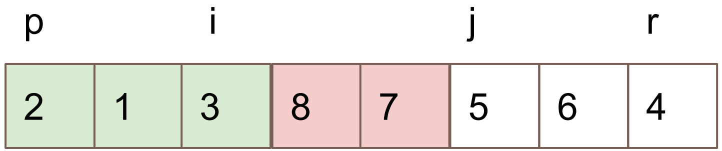
i = 1 + 1, so swap a[2] (= 7) with a[j] = 3.

Goes in a loop from p to r-1, and maintains the following invariant for each element a[k] in the array

- If p <= k < =i, then a[k] <= x
- If i+1 <=k<=j-1, then A[k]>x

Back to the partition algorithm

- Let's sort this array



p		i			j		r
2	1	3	8	7	5	6	4

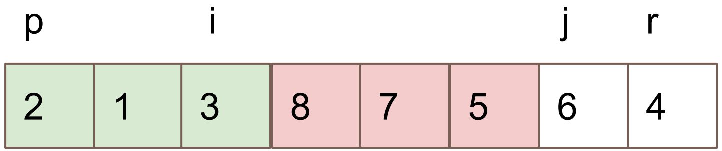
If A[i]> x, then do not change i, and simply increment j. The partition between a[i+1] and a[j-1] denotes the values that are greater than the pivot

Goes in a loop from p to r-1, and maintains the following invariant for each element a[k] in the array

- If p <= k < =i, then a[k] <= x
- If i+1 <=k<=j-1, then A[k]>x

Back to the partition algorithm

- Let’s sort this array



p		i				j	r
2	1	3	8	7	5	6	4

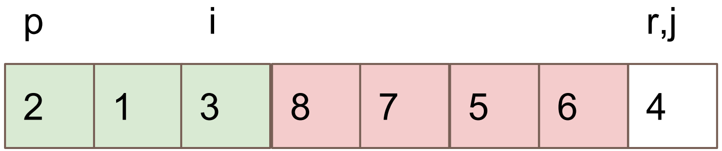
If A[i]> x, then do not change i, and simply increment j. The partition between a[i+1] and a[j-1] denotes the values that are greater than the pivot

Goes in a loop from p to r-1, and maintains the following invariant for each element a[k] in the array

- If p <= k < =i, then a[k] <= x
- If i+1 <=k<=j-1, then A[k]>x

Back to the partition algorithm

- Let’s sort this array



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

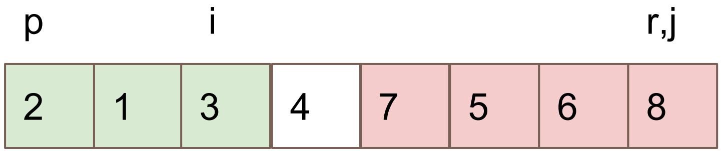
When j = r, exchange a[i+1] (i=2, so 8) with A[r]

Goes in a loop from p to r-1, and maintains the following invariant for each element a[k] in the array

- If p <= k < =i, then a[k] <= x
- If i+1 <=k<=j-1, then A[k]>x

Back to the partition algorithm

- Let’s sort this array



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

Important point: the pivot is now in the **correct location** for the sorted array. Now recurse on a[p,i] and a[i+2, r]

Goes in a loop from p to r-1, and maintains the following invariant for each element a[k] in the array

- If p <= k < =i, then a[k] <= x
- If i+1 <=k<=j-1, then A[k]>x

Quicksort Pseudocode

```
Quicksort(a,p,r):
   if p < r
         q = partition(a,p,r)
         quicksort(a,p,q-1)
         quicksort(a,q+1,r)
```

```
Partition(a,p,r):
   x = a[r]
   i = p-1
   For j = p to r-1
         If a[j] <=x
               I = i+1
               swap(a[i],a[j])
    Swap(a[i+1], a[r])
    return i+1
```

Quicksort complexity analysis

- What is the complexity of the partition function?

Quicksort complexity analysis

- What is the complexity of the partition function?
  - O(n)

Quicksort complexity analysis

- What is the complexity of the partition function?
  - O(n)

- What about for Quicksort?

Quicksort complexity analysis

- What is the complexity of the partition function?
  - O(n)

- What about for Quicksort?
  - Let’s write the recurrence relation

- $T(n) = c$ if $n = 1$

Quicksort complexity analysis

- What is the complexity of the partition function?
  - O(n)

- What about for Quicksort?
  - Let's write the recurrence relation

- $T(n) = c$ if $n = 1$
- $T(n) = T(n_1) + T(n_2) + O(n)$ where n_1 is array size before pivot, n_2 after

Quicksort complexity analysis

- $T(n) = c$ if $n = 1$
- $T(n) = T(n_1) + T(n_2) + O(n)$ where n_1 is array size before pivot, n_2 after

- If choose pivot such that exactly in the middle of the array: $n_1 = n_2 = n/2$
  - $T(n) = 2T(n/2) + O(n)$

Quicksort complexity analysis

- $T(n) = c$ if $n = 1$
- $T(n) = T(n_1) + T(n_2) + O(n)$ where n_1 is array size before pivot, n_2 after

- If choose pivot such that exactly in the middle of the array: $n_1 = n_2 = n/2$
  - $T(n) = 2T(n/2) + O(n)$
  - Same as merge sort $O(n\lg n)$

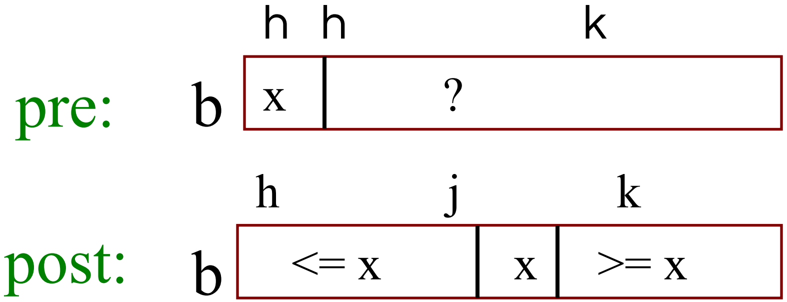
- But what if pivot is such that it is always the last element of the array?
  - $T(n) = O(n) + T(n-1) + O(1)$

Quicksort complexity analysis

- If pivot is such that it is always the last element of the array?
  - $T(n) = O(n) + T(n-1) + O(1)$
  - $N + N - 1 + N-2 + \ldots + 2 + 1$

- Quicksort has O(n^2) complexity in the worst-case
  - Because we always choose the last element as the pivot, arises when input is sorted already

How to choose pivot?



Choosing pivot

Ideal pivot: the median, since it splits array in half

But computing is O(n), quite complicated

Popular heuristics: Use

- first array value (not so good)
- middle array value (not so good)
- Choose a random element (not so good)
- median of first, middle, last, values (often used)!

Can we do better?

- Do algorithms with better than O(nlgn) complexity exist?
  - Yes and no

- For **comparison** based algorithms (ak: when you actually compare to elements in the array a[i]<a[j], O(nlgn) is actually optimal!

- But there are algorithms that are not comparison based!

Non-comparison based sorting

- Counting sort
    - Assumes that each of the n input elements is an integer in range (0,k)
    - Determines, for each input element x, the number of elements less than x. Uses this information to place element x directly into its position in the output array
- Radix sort
    - Used by the card-sorting machines you see in computer museums
    - Sorts integers by their digits (starting from the least significant one)
- Bucket sort
    - Assumes data is uniformly generated.
    - Creates different buckets and assumes buckets will be mostly empty

Sorting in Java

- Java.util.Arrays has a method Sort()
  - implemented as a collection of overloaded methods
  - for primitives, Sort is implemented with a version of quicksort
  - for Objects that implement Comparable, Sort is implemented with mergesort

- Tradeoff between speed/space and stability/performance guarantees