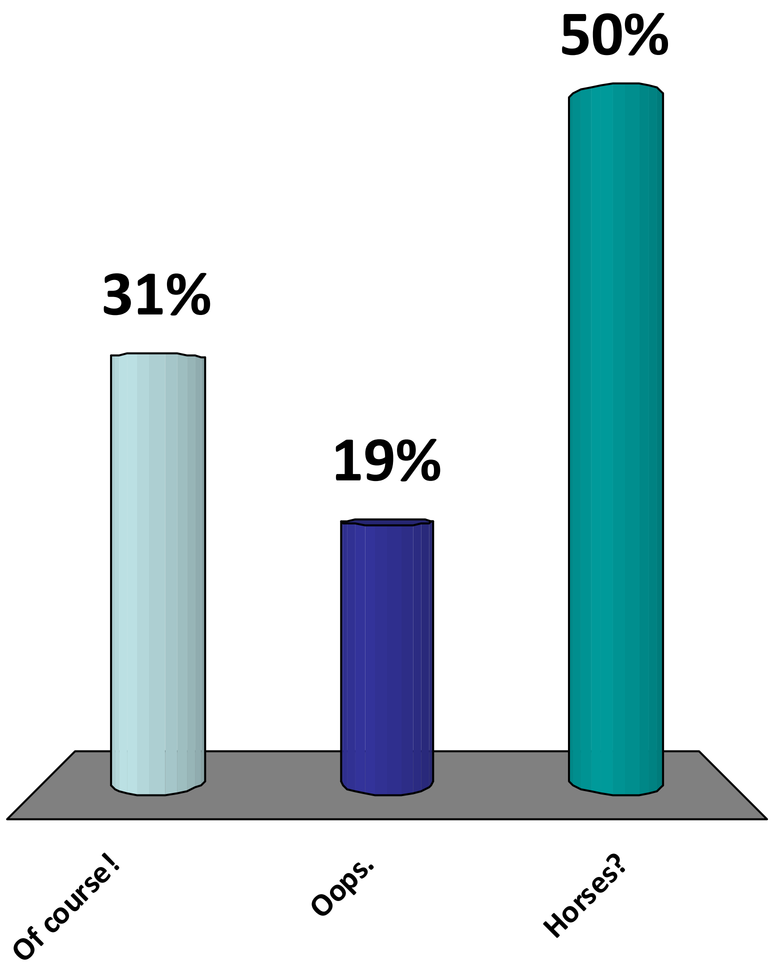
CS2020
Data Structures and Algorithms

Welcome!

I remembered to bring my clicker to class?

1. Of course!
2. Oops.
3. Horses?



Administrativia

Problem Set 2

- Due last night

Problem Set 3

Problem Set 3

Exercise: Sorting

Problem Set 3

Problem 1: Sorting Detective

- Six suspicious sorting algorithms
  - Investigate the mysterious sorting code.
  - Identify each sorting algorithm.
  - Find the criminal: Dr. Evil!

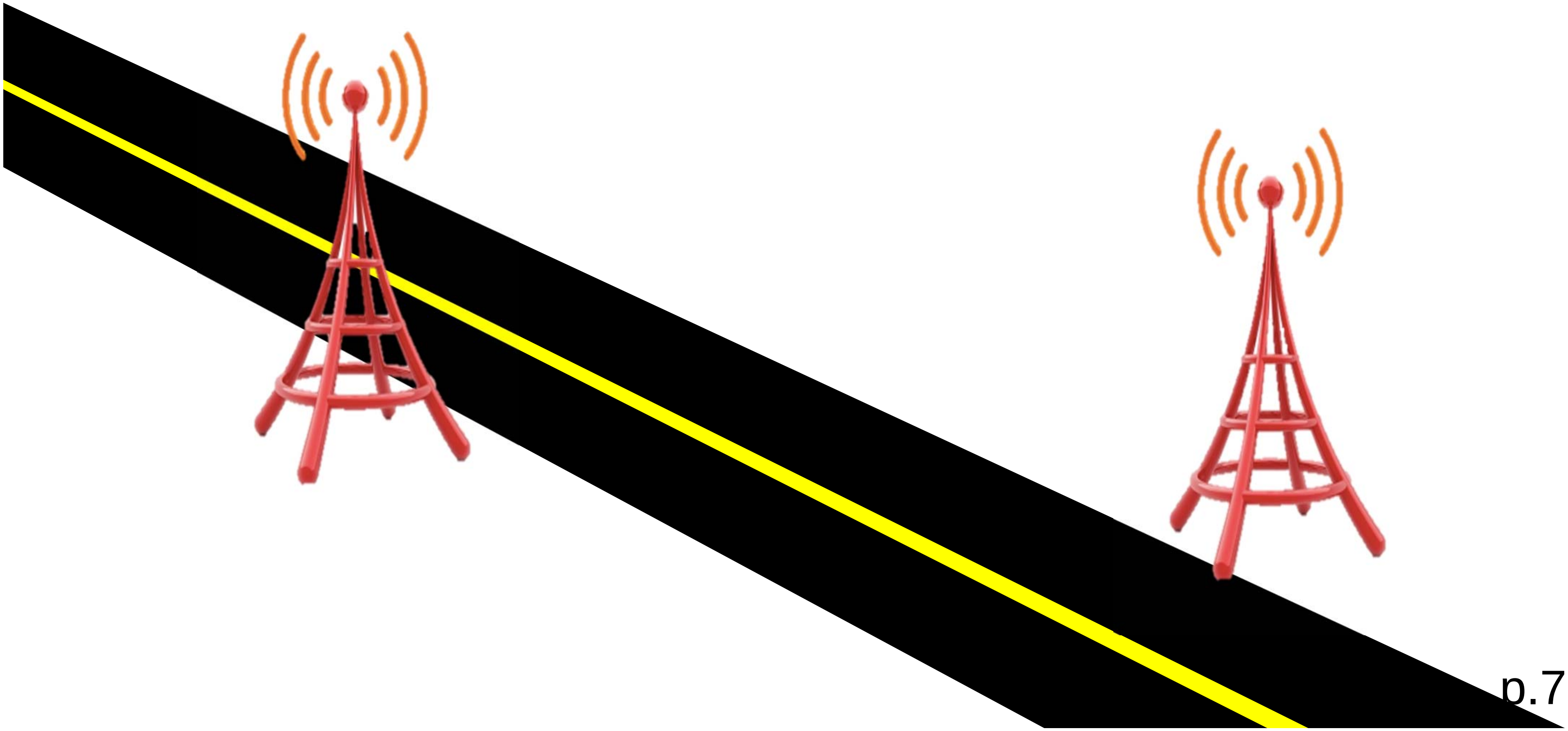
- Focus on the properties:
  - Asymptotic performance
  - Stability
  - Performance on special inputs
- Absolute speed is not a good reason…

(Material covered for this problem on Friday)

Problem Set 3

Problem 2: Cellular Coverage

- Build a data structure to help choose the best locations to deploy cellular towers.



Problem Set 3

Problem 2: Cellular Coverage

- Build a data structure to help choose the best locations to deploy cellular towers.
  - Given a set of towers, what is the coverage?
- Try small examples:
  - What about two towers that don't overlap?
  - What about two towers that do overlap?
- Hint (in the question):
  - Why should Tower implement Comparable?

Two Weeks of CS2020

Two Weeks of CS2020

Java

- Basic syntax
- Classes
- Interfaces
- Objects
- Access control (public/private/protected)
- Generics
- Etc.

Two Weeks of CS2020

Object-oriented programming

- Encapsulation
- Modularity
- Importance of interfaces
- Inheritance

Two Weeks of CS2020

Algorithms

- Theory:
  - Big-O notation
  - Sequential model of computation
  - Intro to algorithm analysis

- Binary search:
  - Basic approach
  - 1d Peak Finding
  - 2d Peak finding
  - Herbert

Why peak finding?

Why peak finding?

1. Common optimization technique: find a locally optimal solution.

Why peak finding?

1. Common optimization technique: find a locally optimal solution.

2. Binary search example: how can you apply techniques *similar* to binary search to other problems.

Why peak finding?

1. Common optimization technique: find a locally optimal solution.

2. Binary search example: how can you apply techniques *similar* to binary search to other problems.

3. Neat progression of algorithms: shows the steps you might go through to solve a difficult problem (e.g., d-dimensional peak finding).

Two Week Survey

[How are things going so far?]

Topics to cover

Generating effective test cases and debugging

Loop invariants

More searching

Proper documentation / Javadocs

Inner classes

Topics to cover

Generating effective test cases and debugging

Problem Set 3

Loop invariants

Today (and onwards)

More searching Yes

Proper documentation / Javadocs

Slowly over the semester.

Inner classes In 2 weeks.

Technical Questions

Important Question!

When should I use binary search?

- Look for monotonicity:
  - As I increase something, there is something else that always increases.
  - As I decrease something, there is something else that always decreases.
- Look for maximization/minimization/search problems.
- Look for problems where you can bound the thing you are searching for.

Java Questions

"Coming to Java from Python, I feel quite out of place. … I find myself constantly referring to javadocs or google for ways to do trivial things [that are easy in Python]."

"Working with arrays and strings because [what I expect to work always] fails in Java"

"Why is Java so painful to work with compared to Python?"

Problem Set Questions (from survey)

How long should I spend on them?

- Huge variance: 2-8 hours
- Talk to tutors for advice on how to go faster.

Will you go over problem set solutions?

- Go over problem sets in Discussion Groups. (Ask your tutor questions.)
- Discuss solutions on Nota Bene (after the due date).
- It's ok if you are still confused by some problems!

Technical Questions

Important Question!

How are classes and interfaces used differently in Java?

Classes

- Contain implementation
- Contains state
- Mixed public and private
- Objects instantiate classes.
- Classes implement interfaces.
- An object can inherit from only one parent class.

Interfaces

- **NO** implementation
- No state
- All public
- Cannot be instantiated.
- Defines how to interact with an object.
- An object may implement many interfaces.

Technical Questions

What exactly is a package or a project?

- A collection of classes / interfaces / etc. that are stored together in a common place.
- Put all your CS2020 classes in one package.
- In Eclipse, a project is an organizational construct that holds related code.
- An Eclipse project can have code from different packages.

Two Weeks of CS2020

And on to Week 3!

Today: Sorting

- Writing a sorting algorithm in Java
- Sorting algorithms
  - BubbleSort
  - SelectionSort
  - InsertionSort
  - MergeSort
- Properties
  - Running time
  - Space usage
  - Stability

Sorting

Problem definition:

Input: array A[1..n] of words / numbers

Output: array B[1..n] that is a permutation of A such that:

$B[1] \leq B[2] \leq \ldots \leq B[n]$

Example:

$A = [9, 3, 6, 6, 6, 4] \rightarrow [3, 4, 6, 6, 6, 9]$

Sorting

```java
public interface ISort{

    public void sort(int[] dataArray);

}
```

Sorting Widgets

```
public interface ISortWidgets{

    public void sort(Widget[] dataArray);

}
```

```
public void sort(Widget[] dataArray) {

   Widget x = dataArray[0];
   Widget y = dataArray[1];
   if (x < y) {
```

Generic Types

```
public interface ISort<TypeA>{

    public void sort(TypeA[] dataArray);

}
```

What goes wrong?

```
public void sort(TypeA[] dataArray) {

    TypeA x = dataArray[0];
    TypeA y = dataArray[1];

    if (x < y) {
        ...
        ...
    }
```

What goes wrong?

```
public void sort(TypeA[] dataArray) {

   TypeA x = dataArray[0];
   TypeA y = dataArray[1];

   if (x < y) {
       ...
       ...
   }
```

Illegal comparison!

What if: TypeA == Student?

```
class Student {
    double m_CAP;
    String m_name;
    Matric m_id;
}
```

Comparable Interface

```
class Student implements Comparable<Student> {
...
...
}
```

```
interface Comparable<TypeA> {

    int compareTo(TypeA other);

}
```

Comparable Interface

x.compareTo(y) :

-1: if (x<y)

0: if (x == y)

1: if (x>y)

Must define a total ordering
Must be transitive.

```
interface Comparable<TypeA> {

    int compareTo(TypeA other);

}
```

Sorting Students, again

```
class Student implements Comparable<Students> {
...
...
}
```

```
public void sort(Student[] dataArray) {

   Student x = dataArray[0];
   Student y = dataArray[1];

   if (x.compareTo(y) < 0) {  // if (x<y)
```

Implementing Comparable

```
class Student implements Comparable<Student> {
// compare students by CAP
int compareTo(Student other){
   if (this.getCAP() < other.getCAP())
        return -1;
   else if (this.getCAP() > other.getCAP())
        return 1;
   else // equal CAP
        return 0;
   }
}
```

Almost works…

```
public interface ISort{

    public void sort(Comparable[] dataArray);

}
```

Comparable to what?

```
public interface ISort{

    public void sort(Comparable<ZZZ>[] dataArray);

}
```

Generic Sorting

```
public interface ISort<TypeA extends Comparable<TypeA>>
{
    public void sort(TypeA[] dataArray);
}
```

Generic Sorting

```
public interface ISort<TypeA extends Comparable<TypeA>>
{
    public void sort(TypeA[] dataArray);
}
```

extends, not **implements**!!

weird… no good reason... a mystery…

Generic Sorting

```
public interface ISort<TypeA extends Comparable<TypeA>>
{
    public void sort(TypeA[] dataArray);
}
```

Generic Sorting

```
public interface ISort{

    public <TypeA extends Comparable<TypeA>>
    void sort(TypeA[] dataArray);

}
```

Sorting

```
public <TypeA extends Comparable<TypeA>>
void sort(TypeA[] dataArray) {
    for (int i=0; i<dataArray.length; i++){
        for (int j=0; j<dataArray.length-1; j++){
            TypeA first = dataArray[j];
            TypeA second = dataArray[j+1];
            if (first.compareTo(second) > 0)
                swap(dataArray, j, j+1);
        }
    }
}
```

Generic Sorting

```
Student[] dataArray = new Student[100];
sort(dataArray);
```

```
class Student implements Comparable<Student> {
    int compareTo(Student other) {
        …
    }
}
```

Generic Sorting

```
Emotion[] dataArray = new Emotion[100];
sort(dataArray);
```

Error!

```
class Emotion {
    int compareTo(Emotion other) {
        …
    }
}
```

Which of the following correctly implements:

boolean stringsEqual(String A, String B)

A. `return (A == B);`

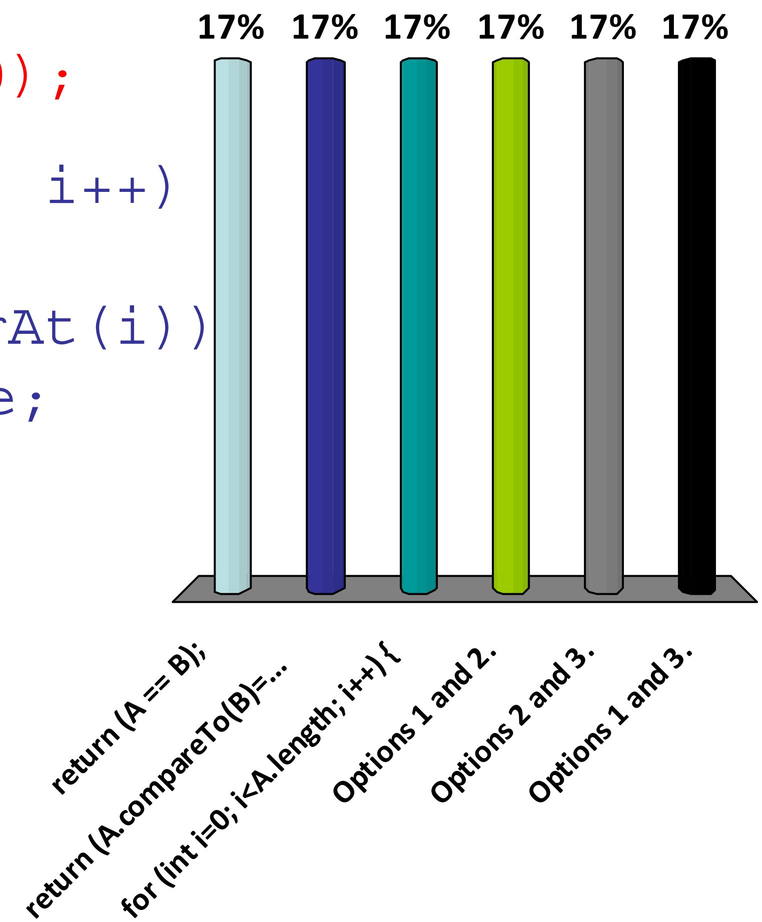
✔ B. `return (A.compareTo(B)==0);`

C.
```
for (int i=0; i<A.length; i++)
{
if (A.charAt(i) != B.charAt(i))
                return false;
}
return true;
```

D. Options 1 and 2.

E. Options 2 and 3.

F. Options 1 and 3.



Comparable Interface

Most Java classes support Comparable

- Integer, Float, etc.
- BigInteger
- String
- Date
- Time
- …

Generic Array

Problem:

```
class Widget<TypeA> {
    void buildArray(int size){
            TypeA[] array = new TypeA[size];
            ...
    }
}
```

Cannot instantiate generic arrays!

(How big should it be? Without knowing sizeof(TypeA), Java cannot decide.)

Generic Array

Solution: use ArrayList

```
class Widget<TypeA> {
    void buildArray(int size){
            ArrayList<TypeA> array = new ArrayList<TypeA>(size);
            …
    }
}
```

Comparing Students

```
class Student implements Comparable<Student> {
...
...
}
```

```
interface Comparable<TypeA> {

    int compareTo(TypeA other);

}
```

Generic Sorting

```
public interface ISort{

    public <TypeA extends Comparable<TypeA>>
    void sort(TypeA[] dataArray);

}
```

Today: Sorting

- Writing a sorting algorithm in Java
- Sorting algorithms
  - BubbleSort
  - SelectionSort
  - InsertionSort
  - MergeSort
- Properties
  - Running time
  - Space usage
  - Stability

Aside: BogoSort

```
BogoSort(A[1..n])
```

Repeat:

a) Choose a random permutation of the array A.

b) If A is sorted, return A.

What is the expected running time of BogoSort?

Aside: BogoSort

`QuantumBogoSort(A[1..n])`

a) Choose a random permutation of the array A.
b) If A is sorted, return A.
c) If A is not sorted, destroy the universe.

What is the expected running time of Quantum BogoSort?

(Remember QuantumBogoSort when you learn about non-deterministic Turing Machines.)

Today: Sorting

- Writing a sorting algorithm in Java
- Sorting algorithms
  - BubbleSort
  - SelectionSort
  - InsertionSort
  - MergeSort
- Properties
  - Running time
  - Space usage
  - Stability

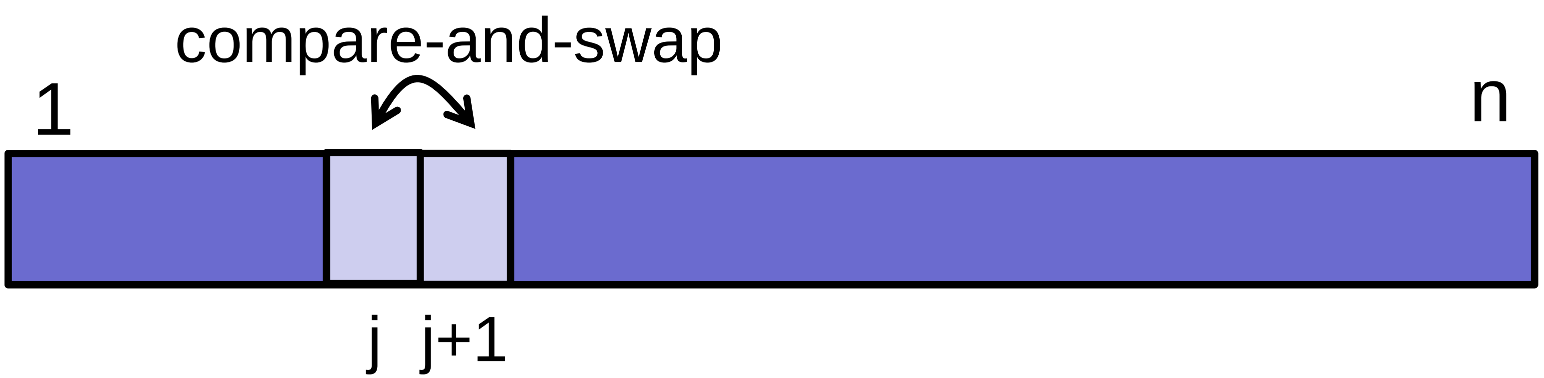
BubbleSort

BubbleSort(A, n)

repeat n **times:**

for j ← 1 **to** n-1

if A[j] > A[j+1] **then** swap(A[j], A[j+1])



BubbleSort

Example: 8 2 4 9 3 6

BubbleSort

Example: **8** **2** 4 9 3 6

2 **8** 4 9 3 6

BubbleSort

Example:

	8	2	4	9	3	6
	2	**8**	**4**	9	3	6
	2	**4**	**8**	9	3	6

BubbleSort

Example:

8	2	4	9	3	6
2	8	4	9	3	6
2	4	**8**	**9**	3	6
2	4	**8**	**9**	3	6

BubbleSort

Example:

8	2	4	9	3	6
2	8	4	9	3	6
2	4	8	9	3	6
2	4	8	**9**	**3**	6
2	4	8	**3**	**9**	6

BubbleSort

Example:

8	2	4	9	3	6
2	8	4	9	3	6
2	4	8	9	3	6
2	4	8	9	3	6
2	4	8	3	**9**	**6**
2	4	8	3	**6**	**9**

BubbleSort

Example:

8	2	4	9	3	6
2	8	4	9	3	6
2	4	8	9	3	6
2	4	8	9	3	6
2	4	8	3	9	6
2	**4**	**8**	**3**	**6**	**9**

BubbleSort

Example:

$\boxed{2 \quad 4}$		8	3	6	9
2	$\boxed{4 \quad 8}$		3	6	9
2	4	$\boxed{8 \quad 3}$		6	9
2	4	3	$\boxed{8 \quad 6}$		9
2	4	3	6	$\boxed{8 \quad 9}$	
2	**4**	**3**	**6**	**8**	**9**

BubbleSort

Example:

2	4	3	6	8	9
2	4	3	6	8	9
2	3	4	6	8	9
2	3	4	6	8	9
2	3	4	6	8	9
2	**3**	**4**	**6**	**8**	**9**

BubbleSort

Example:

2	3	4	6	8	9
2	3	4	6	8	9
2	3	4	6	8	9
2	3	4	6	8	9
2	3	4	6	8	9
2	**3**	**4**	**6**	**8**	**9**

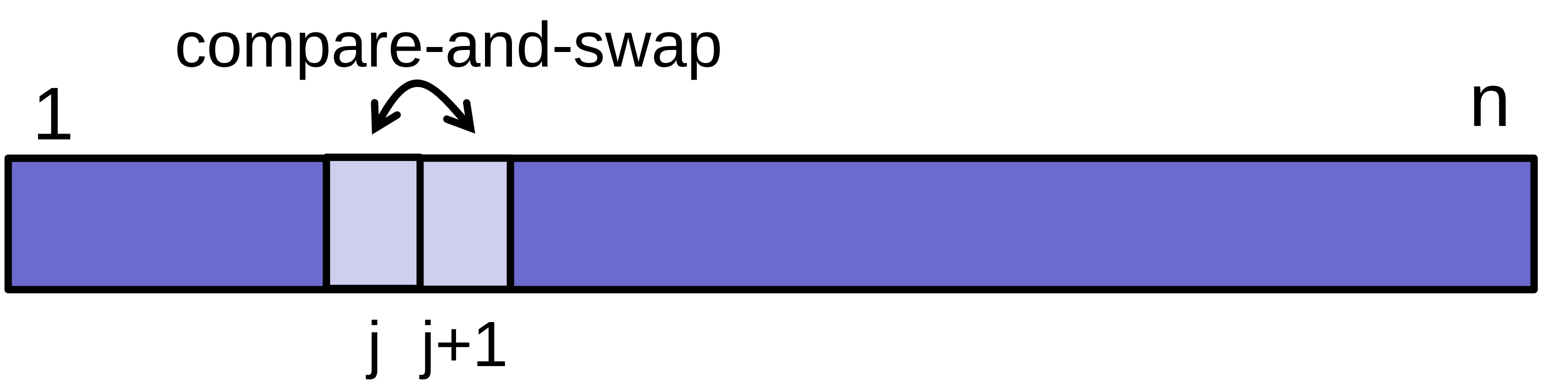
BubbleSort

BubbleSort(A, n)

repeat n **times:**

for j ← 1 **to** n-1

if A[j] > A[j+1] **then** swap(A[j], A[j+1])



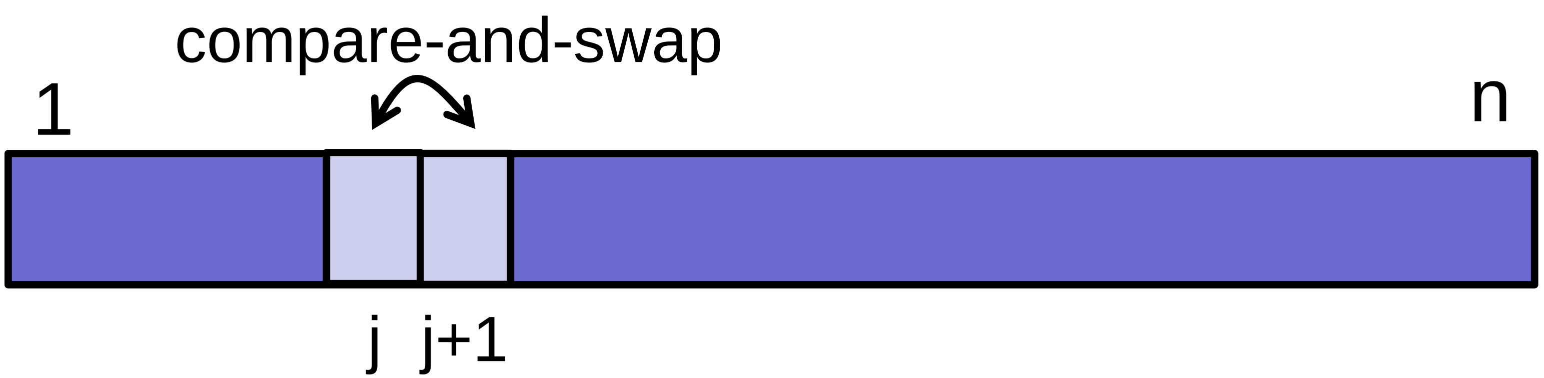
BubbleSort

BubbleSort(A, n)

repeat (until no swaps) **:**

for $j \leftarrow 1$ **to** $n-1$

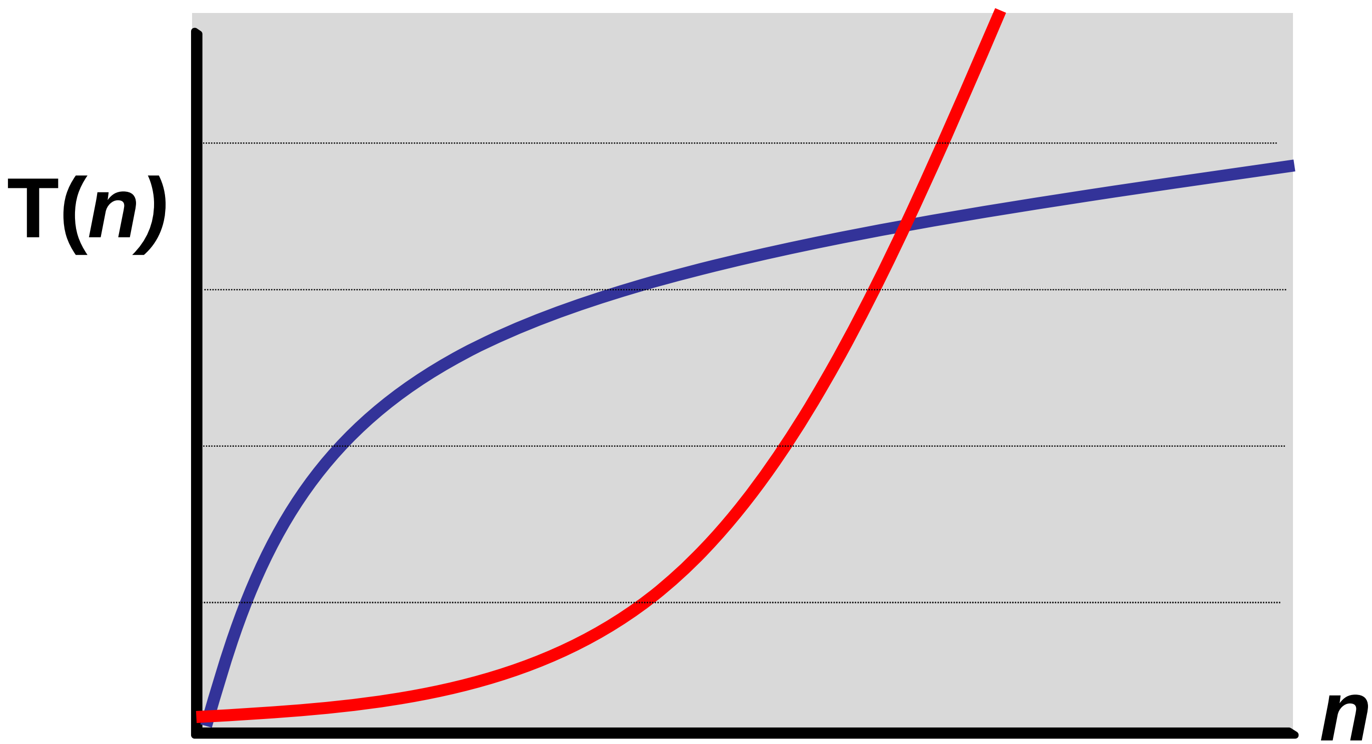
if $A[j] > A[j+1]$ **then** swap(A[j], A[j+1])



Big-O Notation

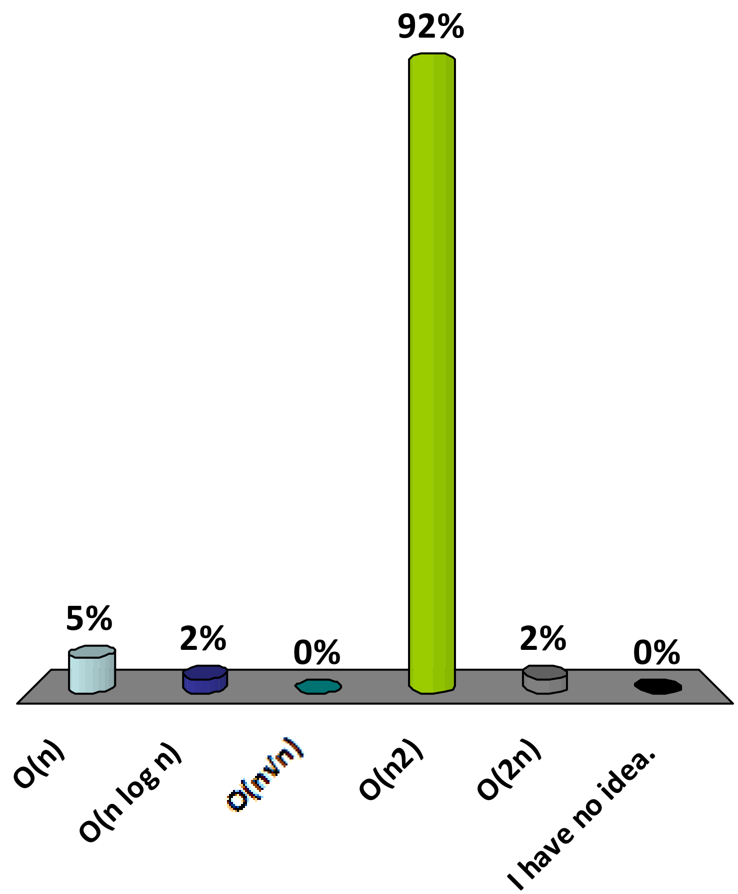
How does an algorithm scale?

- For large inputs, what is the running time?
- T(n) = running time on inputs of size n



What is the running time of BubbleSort?

A. O(n)
B. O(n log n)
C. O(n√n)
D. O(n^2)
E. O(2^n)
F. I have no idea.



BubbleSort

Running time:

- Depends on the input!

BubbleSort

Example:					
$\boxed{2 \quad 3}$		4	6	8	9
2	$\boxed{3 \quad 4}$		6	8	9
2	3	$\boxed{4 \quad 6}$		8	9
2	3	4	$\boxed{6 \quad 8}$		9
2	3	4	6	$\boxed{8 \quad 9}$	
2	**3**	**4**	**6**	**8**	**9**

BubbleSort

Running time:

- Depends on the input!

Best-case:

- Already sorted: O(n)

BubbleSort

Best-case:

- Already sorted: O(n)

Average-case:

- Assume inputs are chosen at random…

Worst-case:

- Bound how long it takes.

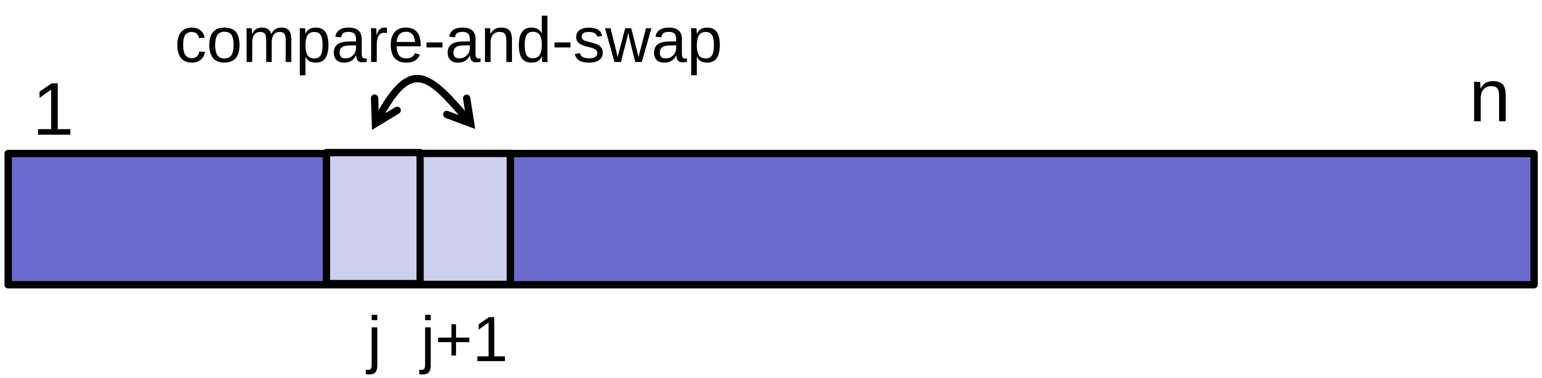
BubbleSort Analysis

BubbleSort(A, n)

repeat (until no swaps) **:**

for j ← 1 **to** n-1

if A[j] > A[j+1] **then** swap(A[j], A[j+1])



BubbleSort Analysis

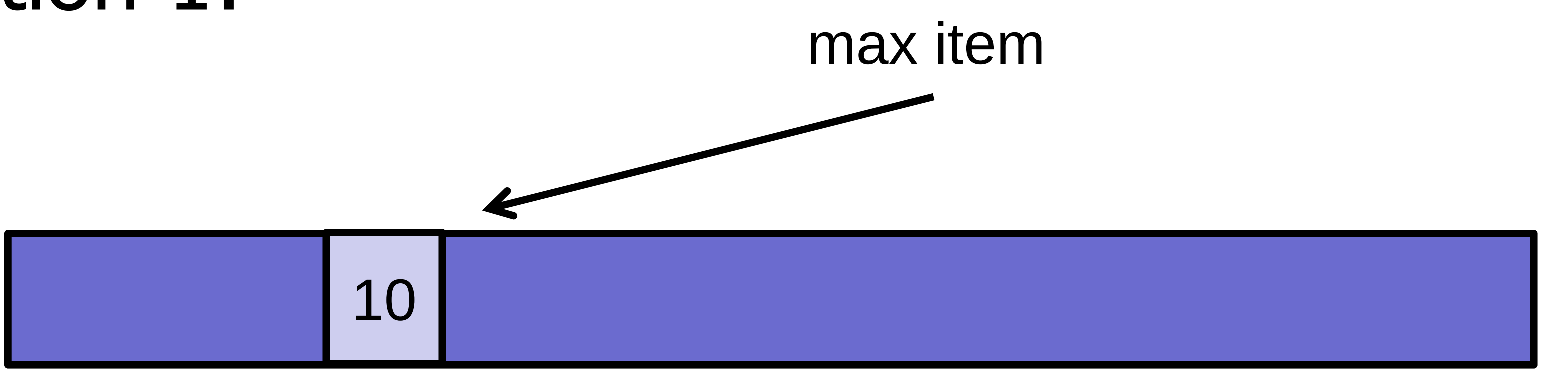
BubbleSort(A, n)

repeat (until no swaps) **:**

for j ← 1 **to** n-1

if A[j] > A[j+1] **then** swap(A[j], A[j+1])

Iteration 1:



BubbleSort Analysis

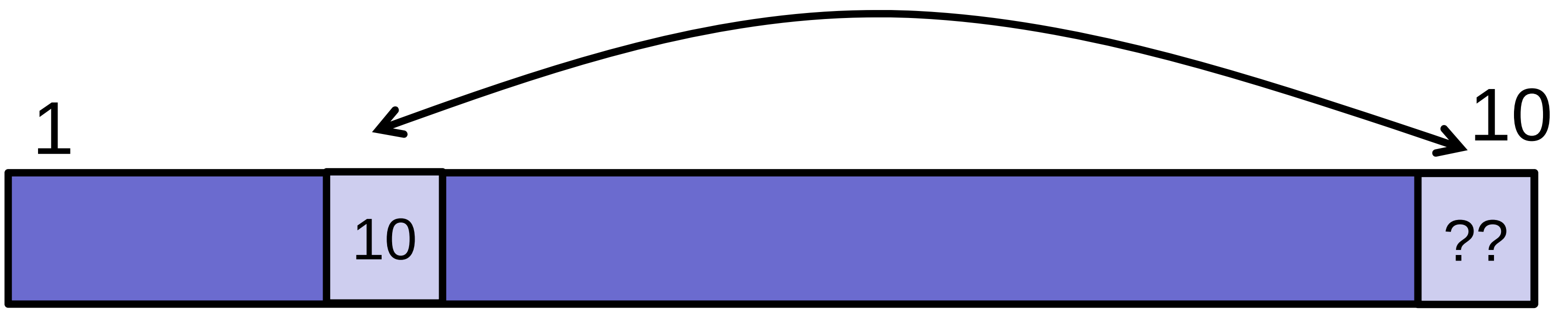
BubbleSort(A, n)

repeat (until no swaps) **:**

for j ← 1 **to** n-1

if A[j] > A[j+1] **then** swap(A[j], A[j+1])

Iteration 1:



BubbleSort Analysis

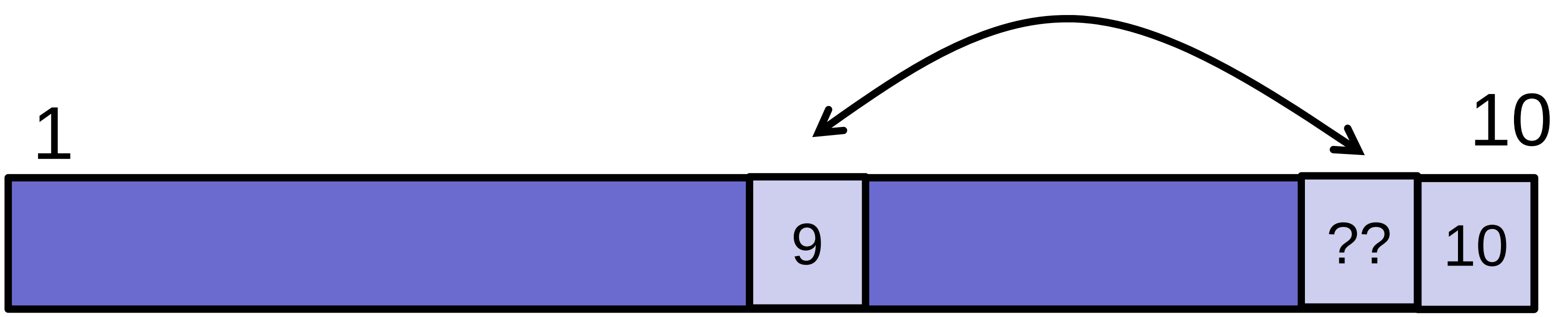
BubbleSort(A, n)

repeat (until no swaps) **:**

for j $\leftarrow$ 1 **to** n-1

if A[j] > A[j+1] **then** swap(A[j], A[j+1])

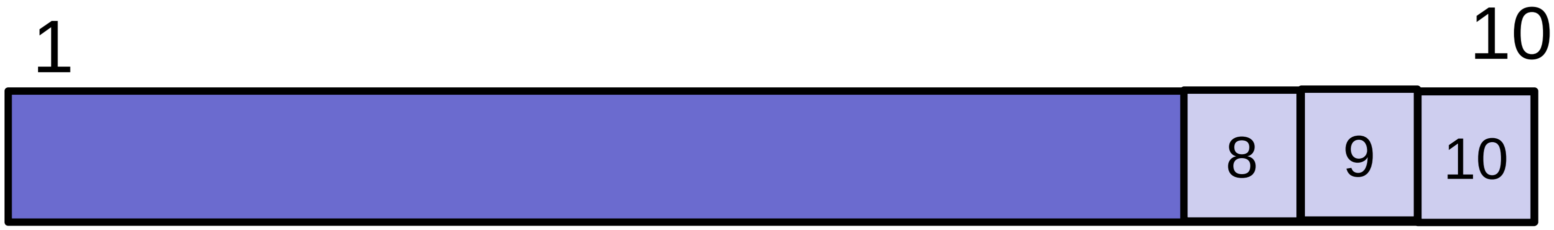
Iteration 2:



BubbleSort Analysis

Loop invariant:

At the end of iteration j: ??



BubbleSort Analysis

Loop invariant:

At the end of iteration j, the biggest j items are correctly sorted in the final j positions of the array.

BubbleSort Analysis

Loop invariant:

At the end of iteration j, the biggest j items are correctly sorted in the final j positions of the array.

Worst case: n iterations ➔ $O(n^2)$ time

BubbleSort

Best-case: $O(n)$

- Already sorted

Average-case: $O(n^2)$

- Assume inputs are chosen at random…

Worst-case: $O(n^2)$

- Bound on how long it takes.

Today: Sorting

- Writing a sorting algorithm in Java
- Sorting algorithms
  - BubbleSort
  - SelectionSort
  - InsertionSort
  - MergeSort
- Properties
  - Running time
  - Space usage
  - Stability

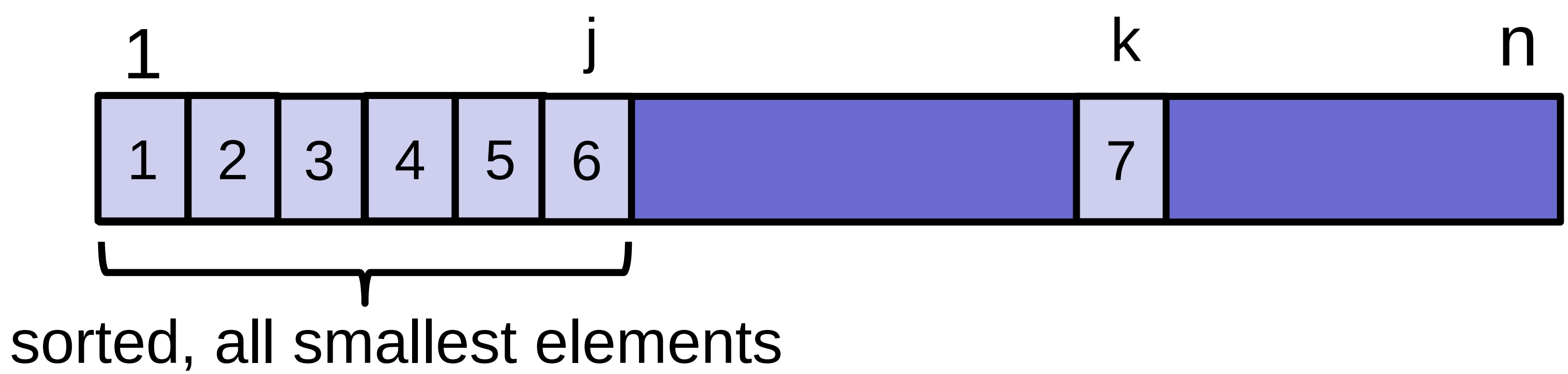
SelectionSort

SelectionSort(A, n)

for j← 1 **to** n-1**:**

find minimum element A[j] in A[j..n]

swap(A[j], A[k])



SelectionSort

Example: 8 2 4 9 3 6

SelectionSort

Example:	8	**2**	4	9	3	6
	2	8	4	9	3	6

SelectionSort

Example:	8	**2**	4	9	3	6
	2	8	4	9	**3**	6
	2	3	4	9	8	6

SelectionSort

Example:

Example:	8	**2**	4	9	3	6
	2	8	4	9	**3**	6
	2	3	**4**	9	8	6
	2	3	4	9	8	6

SelectionSort

Example:					
8	**2**	4	9	3	6
2	8	4	9	**3**	6
2	3	**4**	9	8	6
2	3	4	9	8	**6**
2	3	4	6	8	9

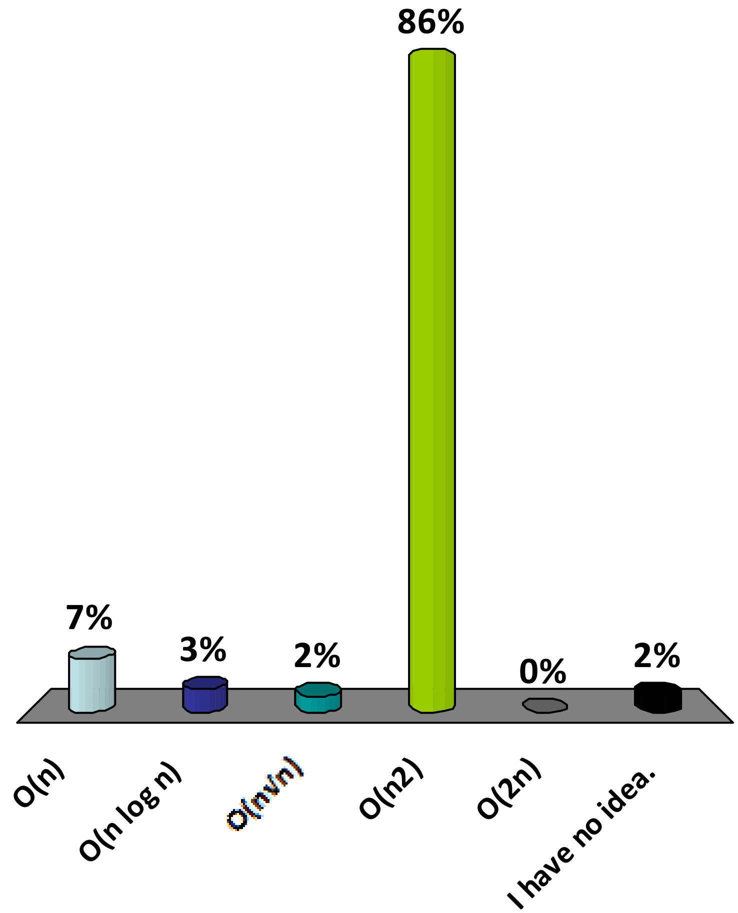
SelectionSort

Example:

Example:	8	**2**	4	9	3	6
	2	8	4	9	**3**	6
	2	3	**4**	9	8	6
	2	3	4	9	8	**6**
	2	3	4	6	**8**	9
	2	3	4	6	8	9

What is the (worst-case) running time of SelectionSort?

A. $O(n)$
B. $O(n \log n)$
C. $O(n\sqrt{n})$
✓ D. $O(n^2)$
E. $O(2^n)$
F. I have no idea.



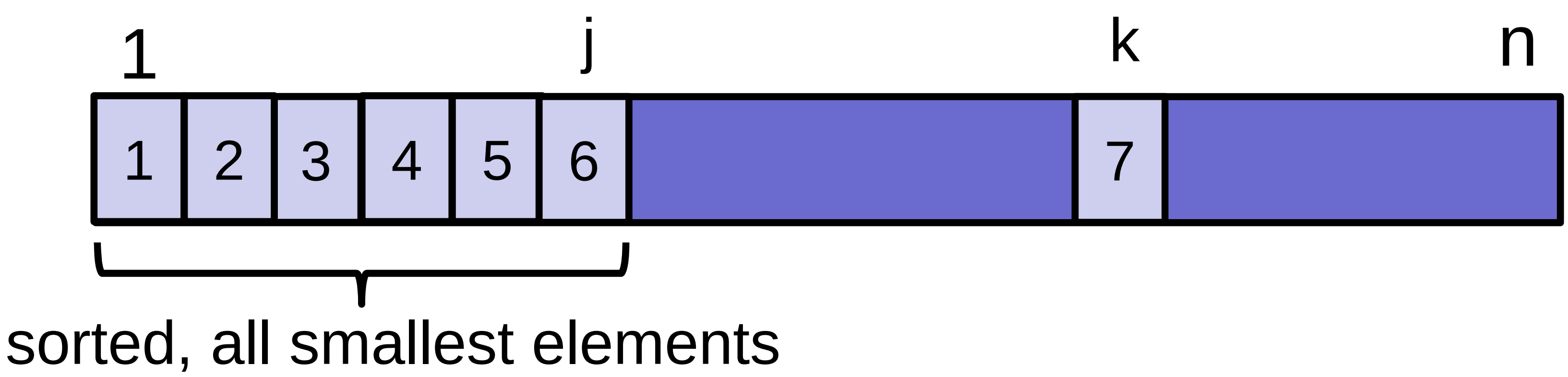
SelectionSort

SelectionSort(A, n)

for j← 1 **to** n-1**:**

find minimum element A[j] in A[j..n]

swap(A[j], A[k])



SelectionSort

Time: (n – j)

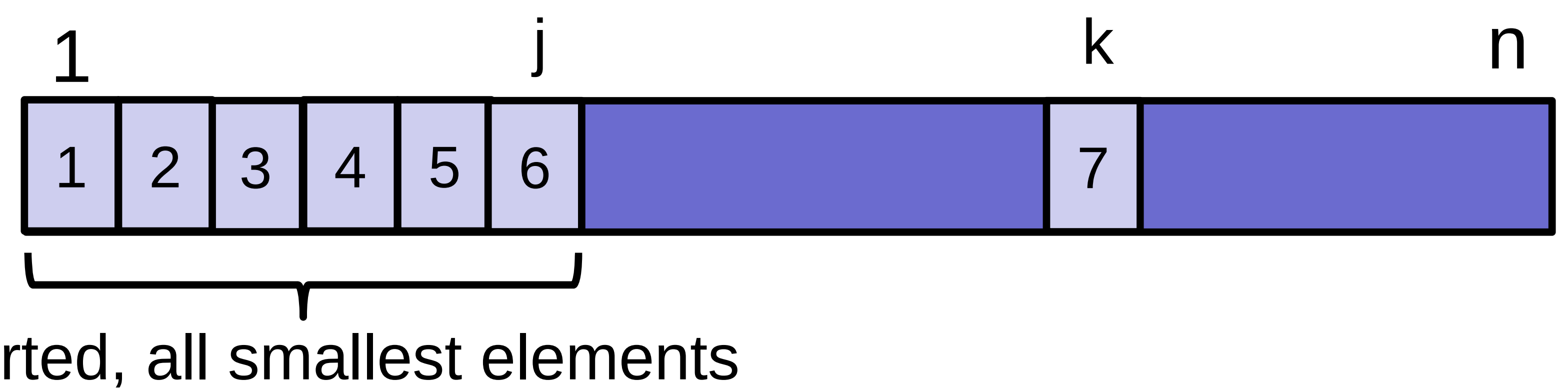
SelectionSort(A, n)

for j← 1 **to** n-1**:**

find minimum element A[j] in A[j..n]

swap(A[j], A[k])

Running time: n + (n-1) + (n-2) + (n-3) + ...



Basic facts

$$n + (n - 1) + (n - 2) + (n - 3) + \ldots + 1 \qquad = (n)(n+1)/2$$

$$= \Theta(n^2)$$

SelectionSort

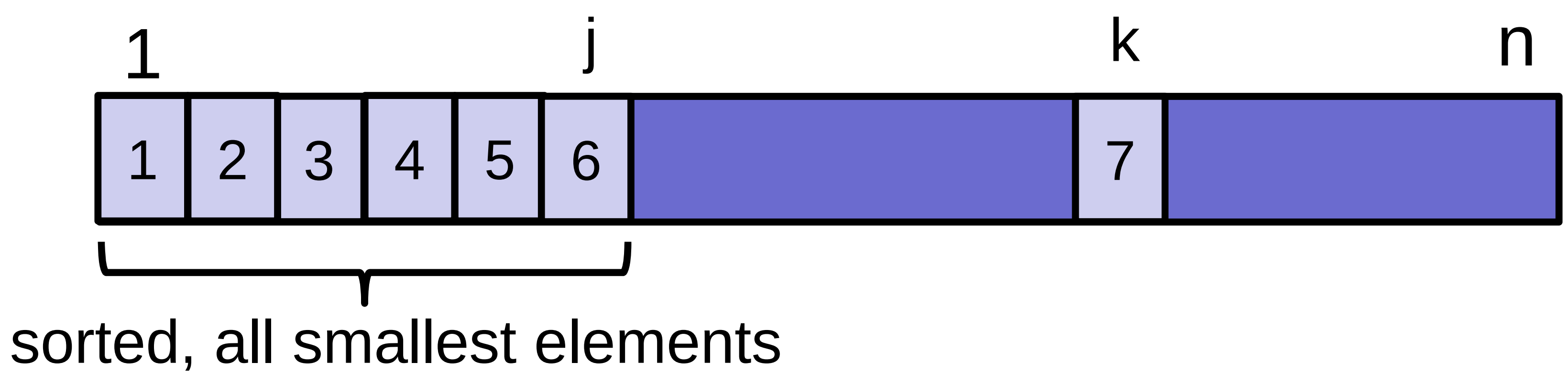
SelectionSort(A, n)

for j← 1 **to** n-1**:**

find minimum element A[j] in A[j..n]

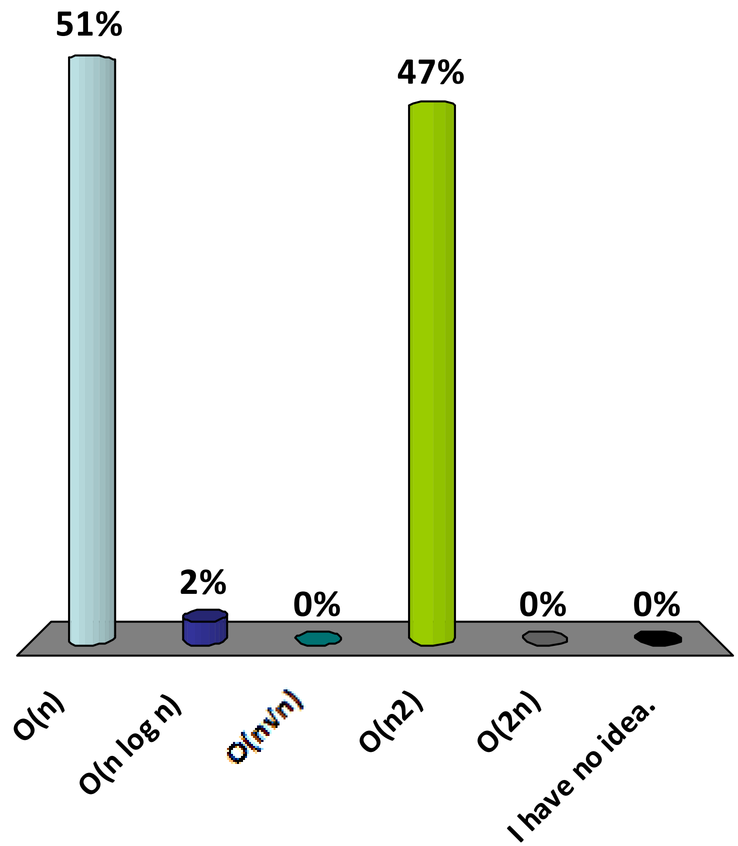
swap(A[j], A[k])

Running time: $O(n^2)$



What is the BEST CASE running time of SelectionSort?

A. $O(n)$
B. $O(n \log n)$
C. $O(n\sqrt{n})$
D. $O(n^2)$
E. $O(2^n)$
F. I have no idea.



SelectionSort

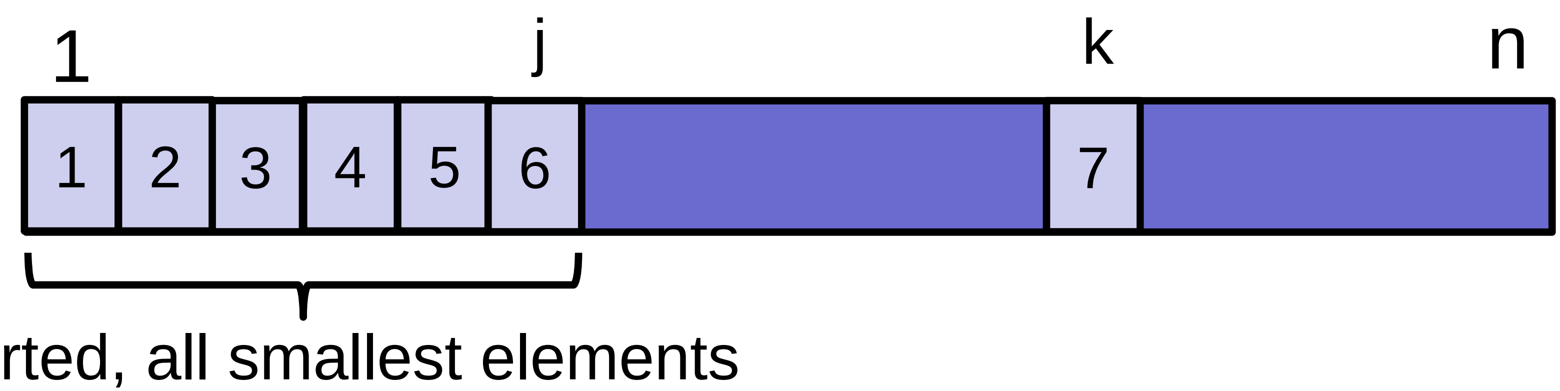
SelectionSort(A, n)

for j← 1 **to** n-1**:**

find minimum element A[j] in A[j..n]

swap(A[j], A[k])

Running time: $O(n^2)$ and $\Omega(n^2)$



SelectionSort Analysis

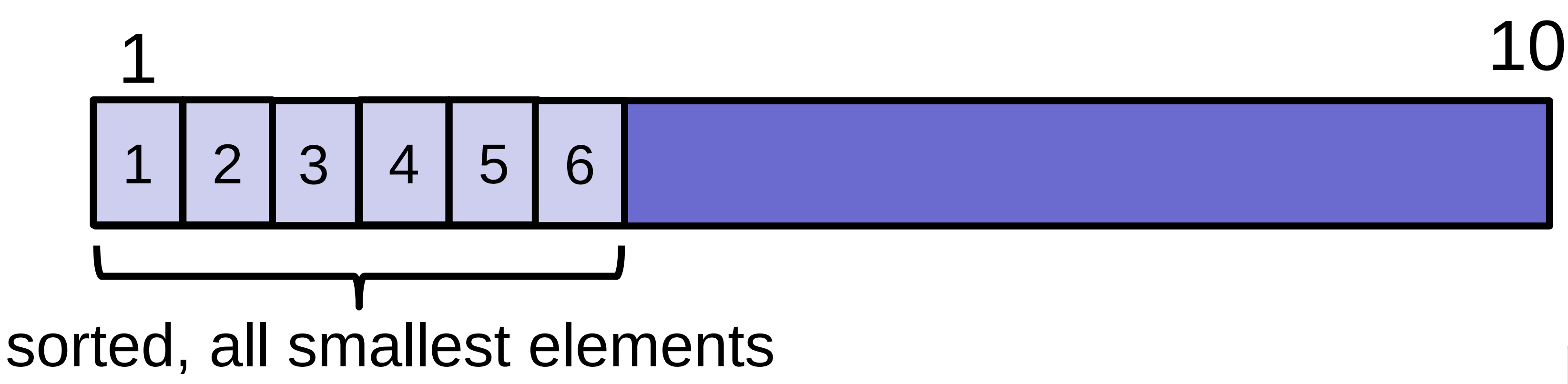
Loop invariant:

At the end of iteration j: ??

SelectionSort Analysis

Loop invariant:

At the end of iteration j: the smallest j items are correctly sorted in the first j positions of the array.



Today: Sorting

- Writing a sorting algorithm in Java
- Sorting algorithms
  - BubbleSort
  - SelectionSort
  - InsertionSort
  - MergeSort
- Properties
  - Running time
  - Space usage
  - Stability

Insertion Sort

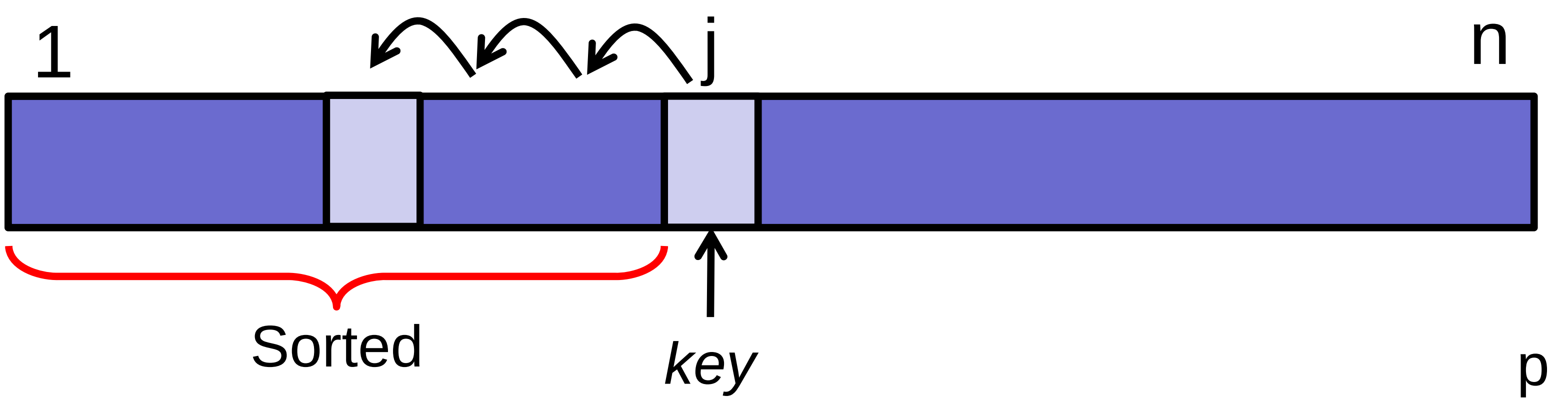
InsertionSort(A, n)

for j ← 2 **to** n | **Invariant:** A[1..j-1] is sorted

key ← A[j]

Insert key into the sorted array A[1..j-1]

Illustration:



Insertion Sort

InsertionSort(A, n)

for j ← 2 **to** n

Invariant: A[1..j-1] is sorted

key ← A[j]

i ← j-1

while (i > 0) **and** (A[i] > *key*)

A[i+1] ← A[i]

i ← i-1

A[i+1] ← *key*

Insertion Sort

Example: **8 2 4 9 3 6**

Insertion Sort

Example: 8 2 4 9 3 6

2 8 4 9 3 6

Insertion Sort

Example:

8	2	4	9	3	6
2	8	4	9	3	6
2	**4**	**8**	**9**	**3**	**6**

Insertion Sort

Example:

8	2	4	9	3	6
2	8	4	9	3	6
2	4	8	9	3	6
2	**4**	**8**	**9**	**3**	**6**

Insertion Sort

Example:

8	2	4	9	3	6
2	8	4	9	3	6
2	4	8	9	3	6
2	4	8	9	3	6
2	**3**	**4**	**8**	**9**	**6**

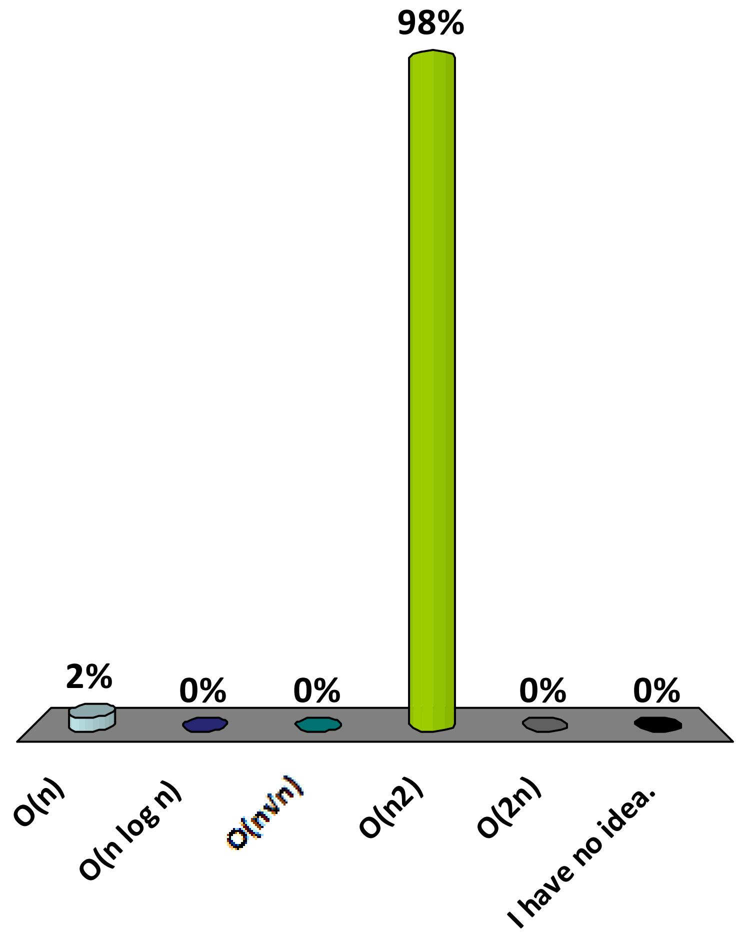
Insertion Sort

Example:

8	2	4	9	3	6
2	8	4	9	3	6
2	4	8	9	3	6
2	4	8	9	3	6
2	3	4	8	9	6
2	**3**	**4**	**6**	**8**	**9**

What is the (worst-case) running time of InsertionSort?

A. $O(n)$
B. $O(n \log n)$
C. $O(n\sqrt{n})$
D. $O(n^2)$
E. $O(2^n)$
F. I have no idea.



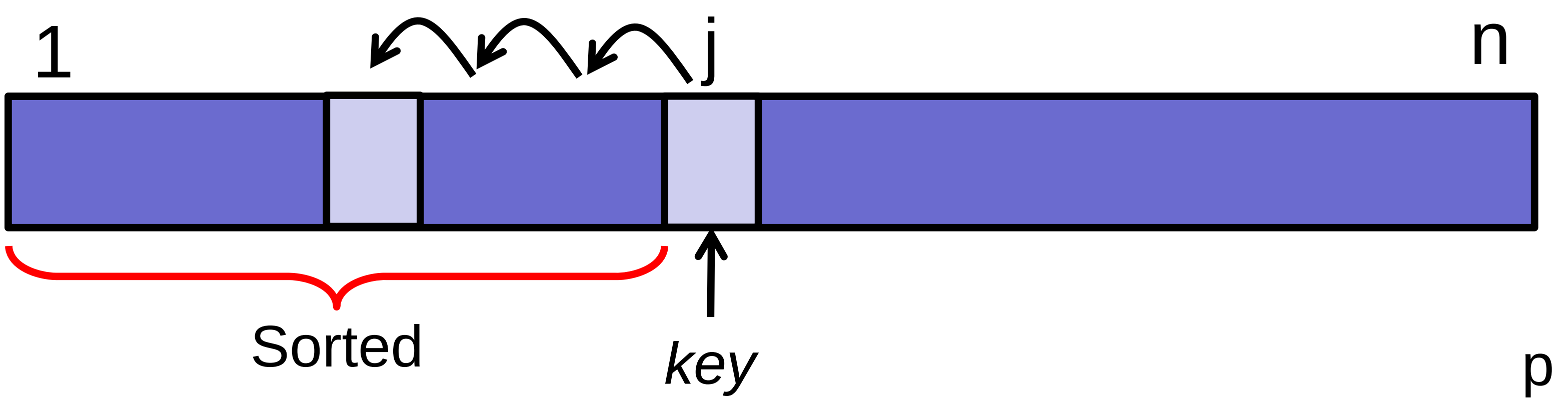
Insertion Sort

Insertion-Sort(A, n)

for j ← 2 **to** n

key ← A[j]

Insert key into the sorted array A[1..j-1]



Insertion Sort Analysis

Insertion-Sort(A, n)

```
for  j ← 2 to n
        key ← A[j]
        i ← j–1
        while (i > 0) and (A[i] > key)
                A[i+1] ← A[i]
                i ← i–1
        A[i+1] ← key
```

Repeat at most j times.

Basic facts

$$1 + 2 + 3 + \ldots + (n-2) + (n-1) + n \quad = (n)(n+1)/2$$

$$= \Theta(n^2)$$

Insertion Sort

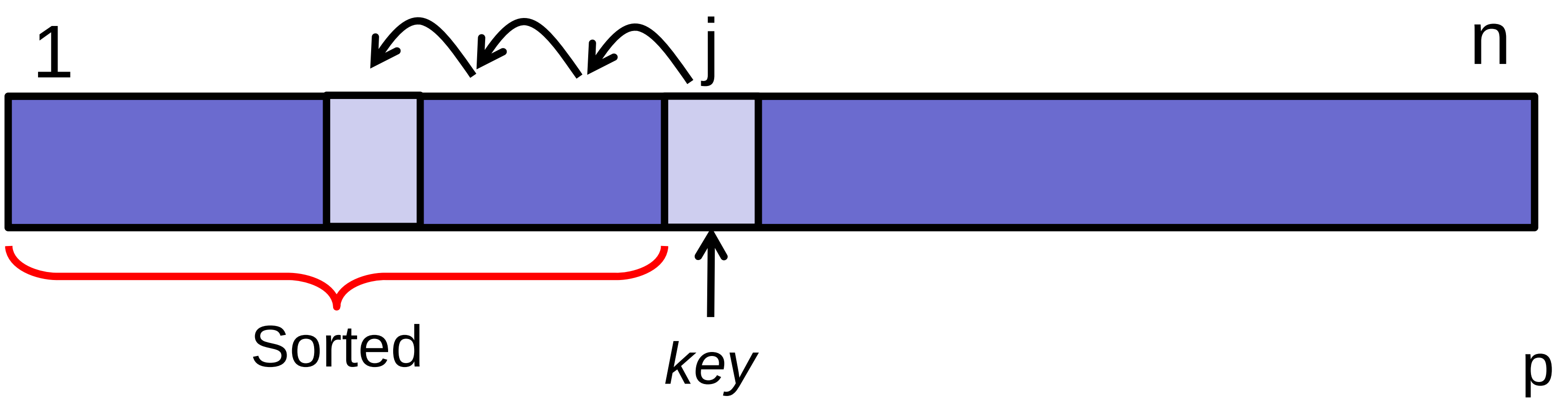
Insertion-Sort(A, n)

for j $\leftarrow$ 2 **to** n

key $\leftarrow$ A[j]

Insert key into the sorted array A[1..j-1]

Running time: $O(n^2)$



Insertion Sort

Best-case:

Average-case:

- Random permutation

Worst-case:

Insertion Sort

Best-case:

- Already sorted: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

Average-case:

- Random permutation?

Worst-case:

- Inverse sorted: [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]

Insertion Sort

Very fast!

Best-case: $O(n)$

– Already sorted: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

Average-case:

– Random permutation?

Worst-case: $O(n^2)$

– Inverse sorted: [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]

Performance Profiling, V2

(*Dracula* vs. *Lewis & Clark*)

Step	Function	Running Time
Create vectors:	Read each file	1.09s
	Parse each file	3.68s
	Sort words in each file	332.13s
	Count word frequencies	0.30s
Dot product:		6.06s
Norm:		3.80s
Angle:		6.06s
Total:		**11minutes ≈ 680.49s**

p.118

Today: Sorting

- Writing a sorting algorithm in Java
- Sorting algorithms
  - BubbleSort
  - SelectionSort
  - InsertionSort
  - MergeSort
- Properties
  - Running time
  - Space usage
  - Stability

Properties of Sorting Algorithms

Time complexity

- Worst case: $O(n^2)$
- Sorted list: BubbleSort

  SelectionSort

  InsertionSort

- Almost sorted list?

How expensive is it to sort:

[1, 2, 3, 4, 5**, 7, 6**, 8, 9, 10]

How expensive is it to sort:

[1, 2, 3, 4, 5**, 7, 6**, 8, 9, 10]

BubbleSort and InsertionSort are fast.

SelectionSort is slow.

NB Challenge of the Day:

Find a permutation of [1..n] where:

- BubbleSort is slow.
- InsertionSort is fast.

Or explain why no such sequence exists.

Properties of Sorting Algorithms

Space complexity

- Worst case: $O(n)$

- In-place sorting algorithm:
  - Only $O(1)$ extra space needed.
  - All manipulation happens within the array.

So far:

All sorting algorithms we have seen are in-place.

Properties of Sorting Algorithms

Stability

What happens with repeated elements?

Key	1	2	**5**	3	4	**5**	6	7	8	9
Data	a	b	**C**	g	h	**D**	j	k	l	m

Properties of Sorting Algorithms

Stability

What happens with repeated elements?

Key	1	2	**5**	3	4	**5**	6	7	8	9
Data	a	b	**C**	g	h	**D**	j	k	l	m

UNSTABLE

Key	1	2	3	4	**5**	**5**	6	7	8	9
Data	a	b	g	h	**D**	**C**	j	k	l	m

Properties of Sorting Algorithms

Stability: preserves order of equal elements

What happens with repeated elements?

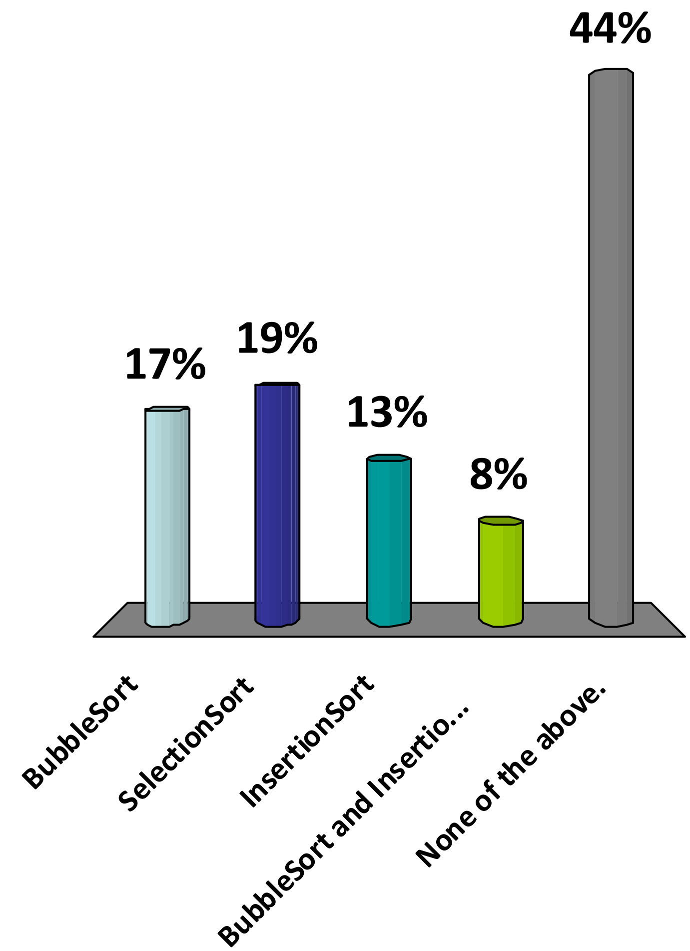
Key	1	2	**5**	3	4	**5**	6	7	8	9
Data	a	b	**C**	g	h	**D**	j	k	l	m

⇩ STABLE

Key	1	2	3	4	**5**	**5**	6	7	8	9
Data	a	b	g	h	**C**	**D**	j	k	l	m

Which are NOT stable?

A. BubbleSort
B. SelectionSort
C. InsertionSort
D. BubbleSort and InsertionSort
E. None of the above.



InsertionSort

```
Insertion-Sort(A, n)
    for  j ← 2 to n
         key ← A[j]
         i ← j-1
         while(i > 0) and(A[i] > key)
               A[i+1] ← A[i]
               i ← i-1
               A[i+1] ← key
```

SelectionSort

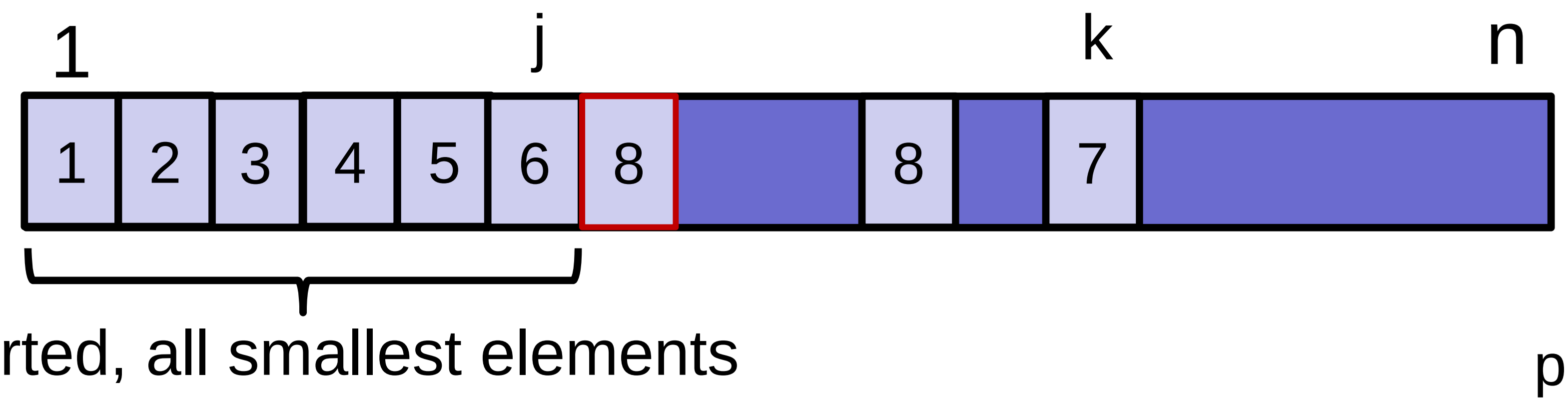
SelectionSort(A, n)

for j ← 1 **to** n-1**:**

find minimum element A[j] in A[j..n]

swap(A[j], A[k])

Stable: No: swap changes order



Today: Sorting

- Writing a sorting algorithm in Java
- Sorting algorithms
    - BubbleSort
    - SelectionSort
    - InsertionSort
    - MergeSort
- Properties
    - Running time
    - Space usage
    - Stability

MergeSort

MergeSort(A, n)

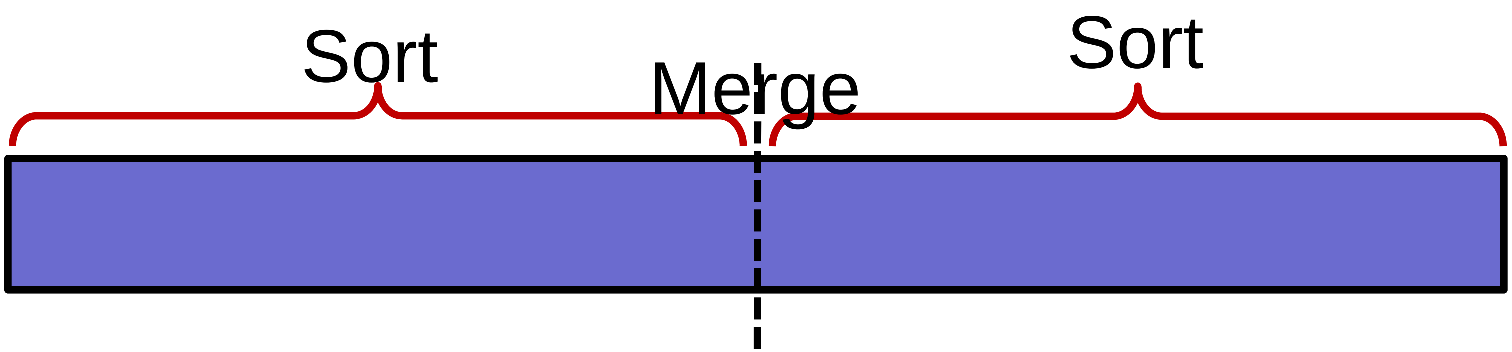
if (n=1) **then return;**

else: **recurse**

X ← MergeSort**(**A[1..n/2], n/2**);**

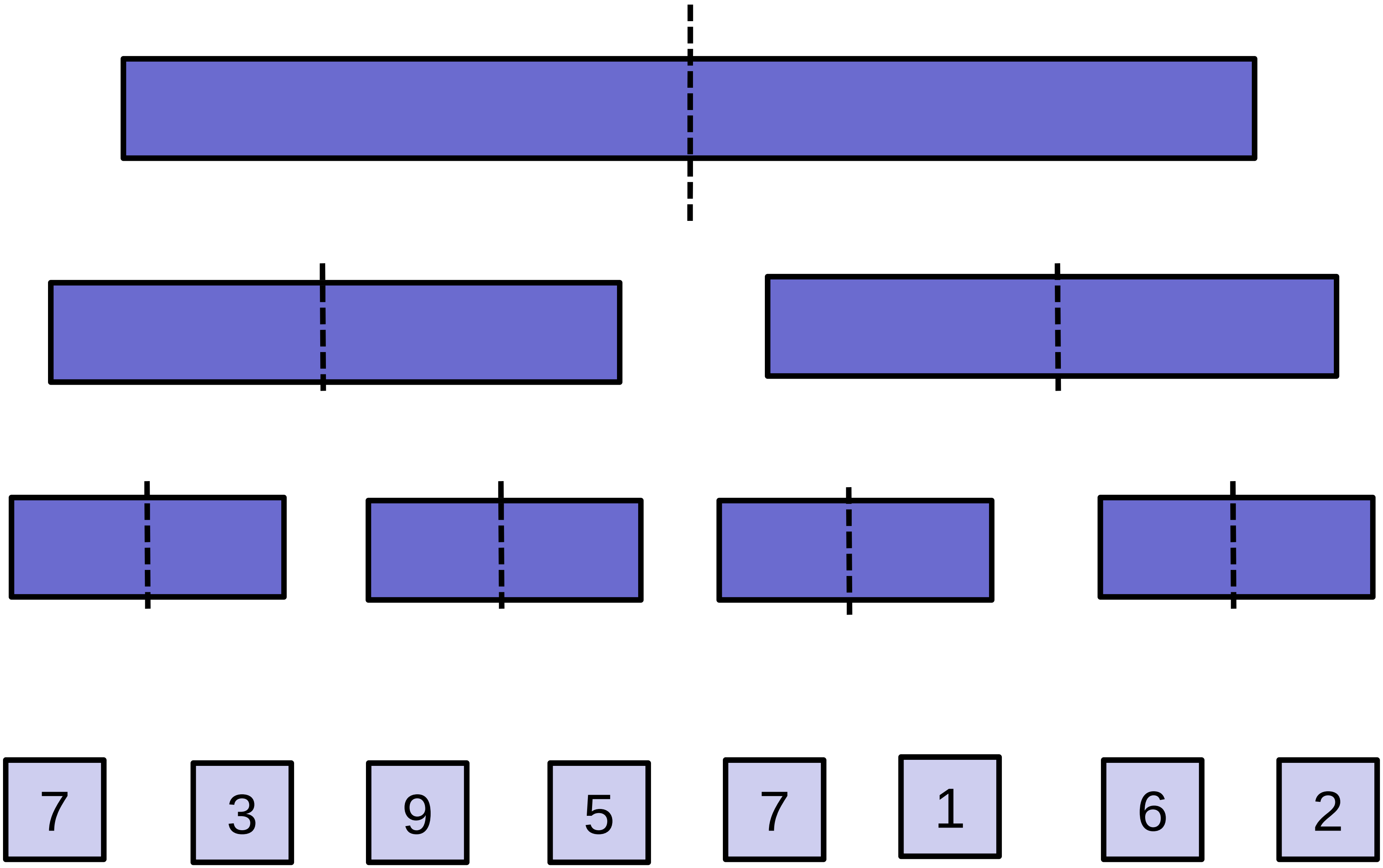
Y ← MergeSort**(**A[n/2+1, n], n/2**);**

return Merge **(**X,Y, n/2**);**

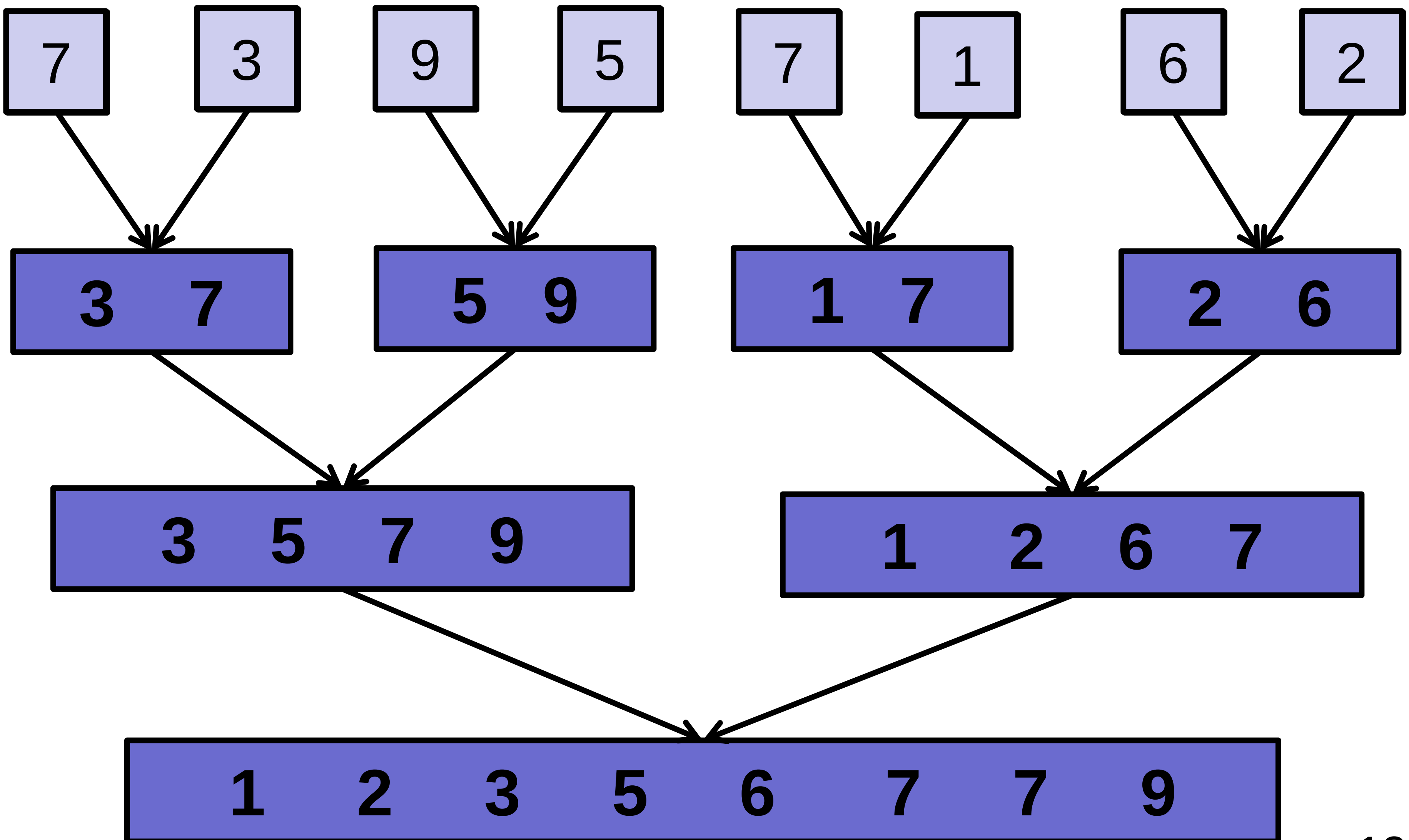


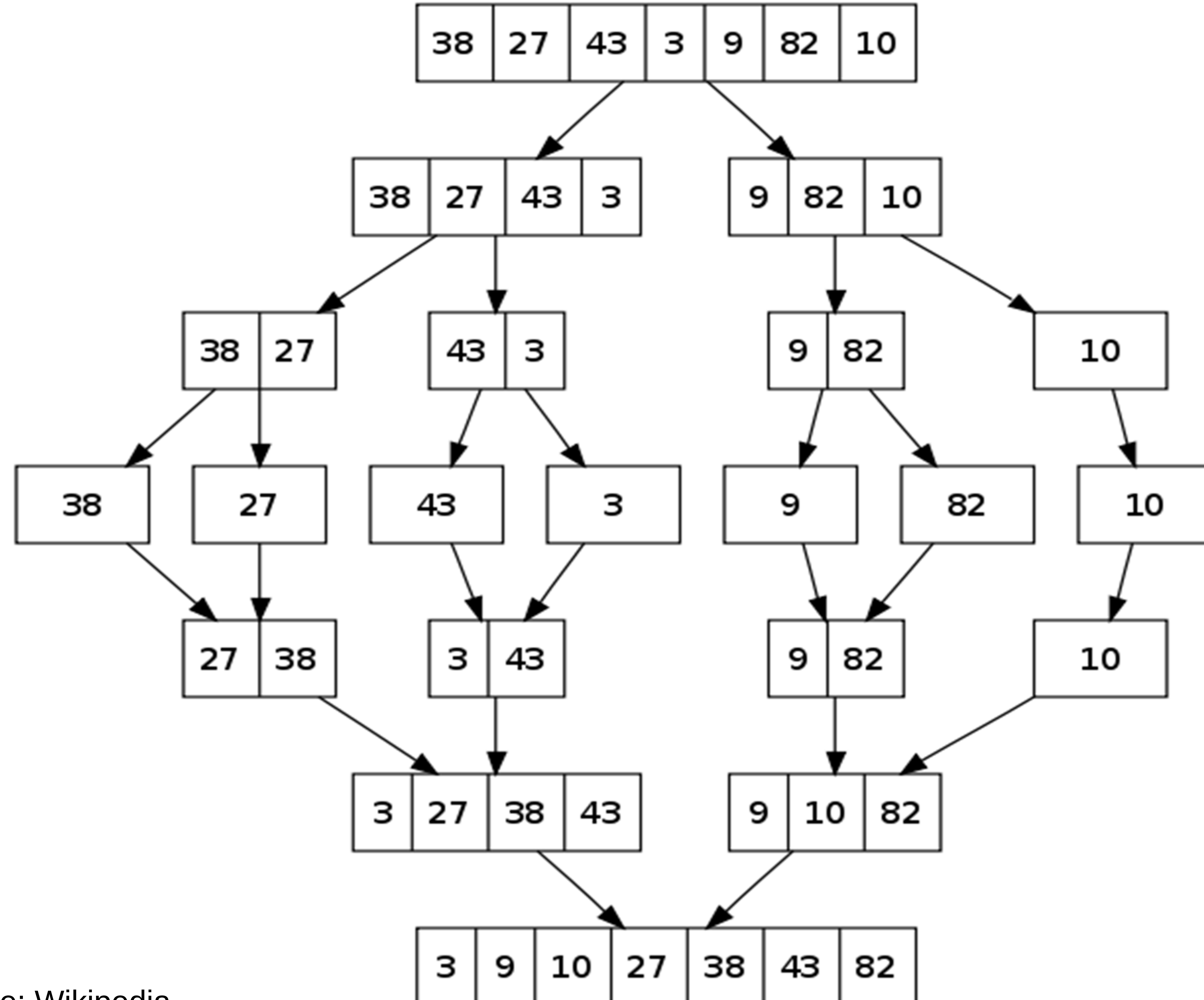
p.134

Divide-and-Conquer



Merging





Source: Wikipedia

Merging Two Sorted Lists

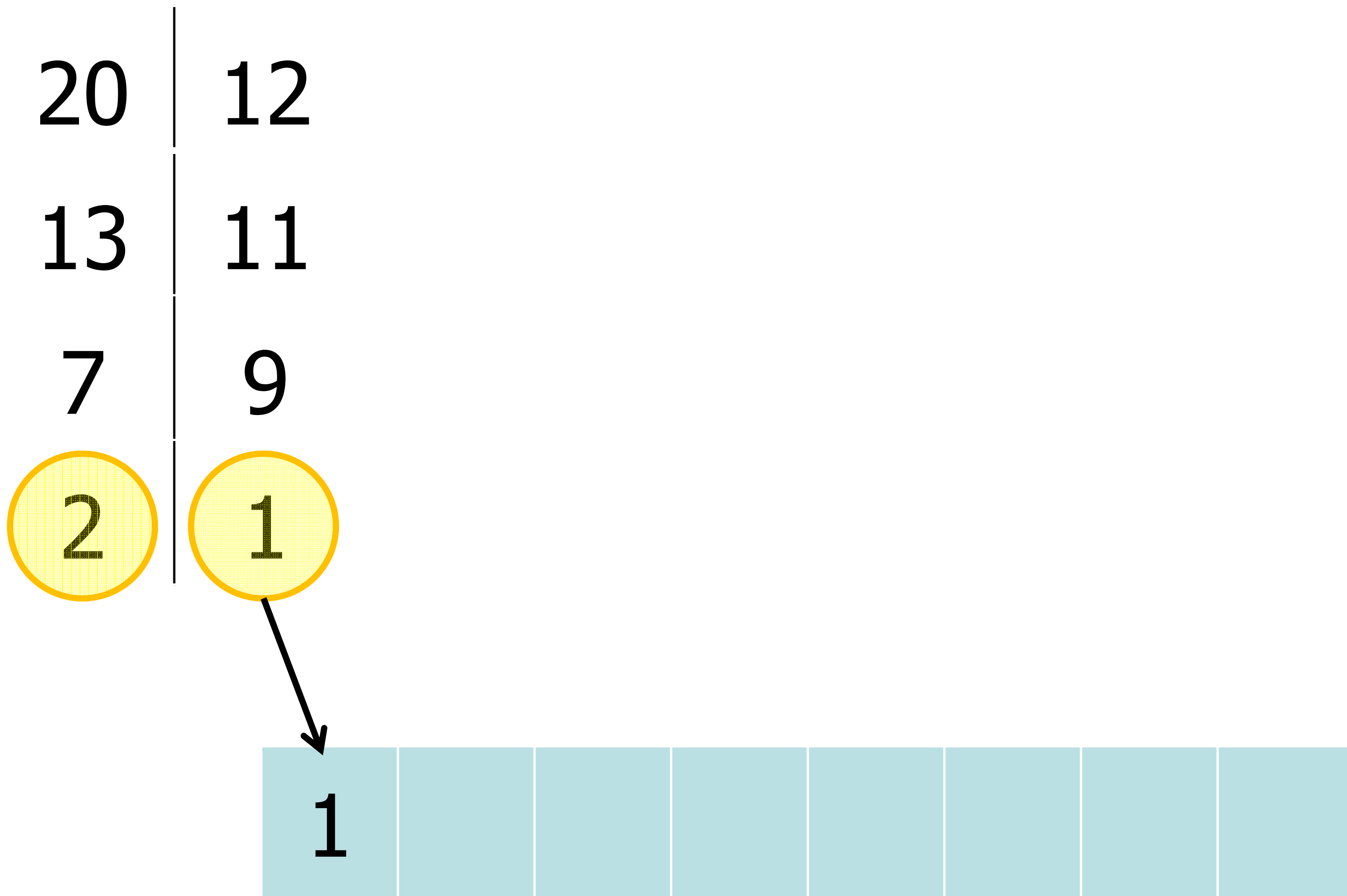
Key subroutine: Merge

- How?
- How fast??

Merging Two Sorted Lists

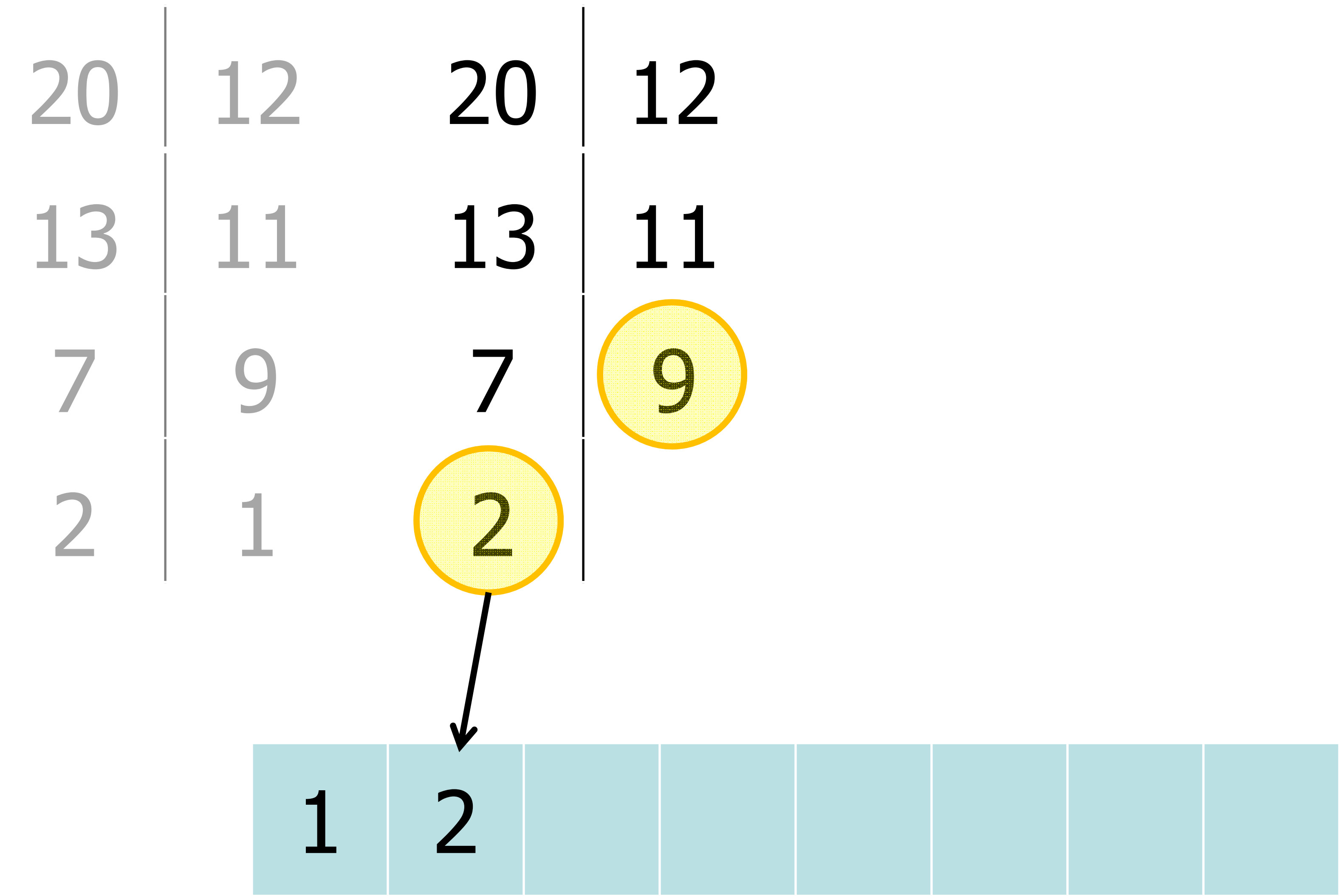
20	12
13	11
7	9
2	1

Merging Two Sorted Lists



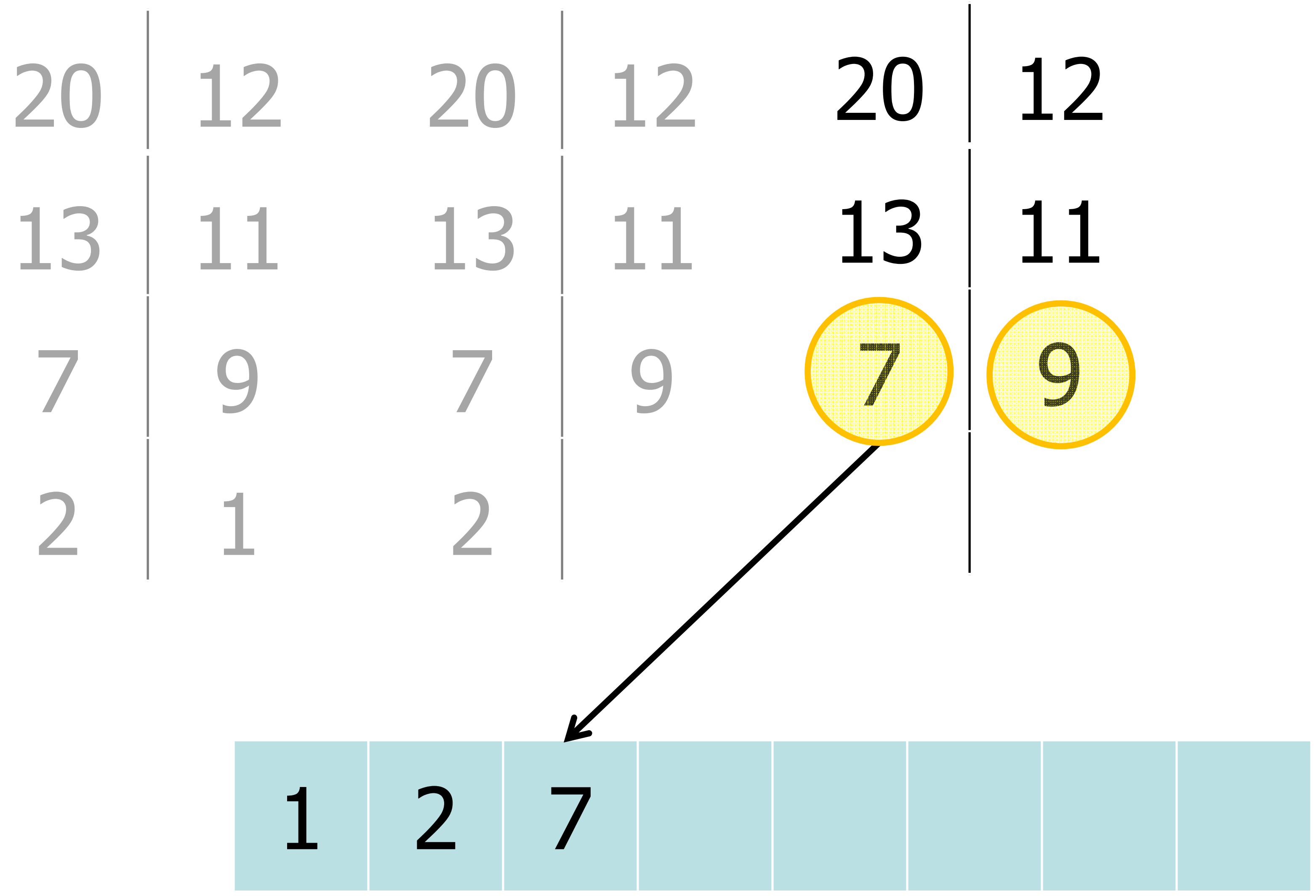
p.140

Merging Two Sorted Lists

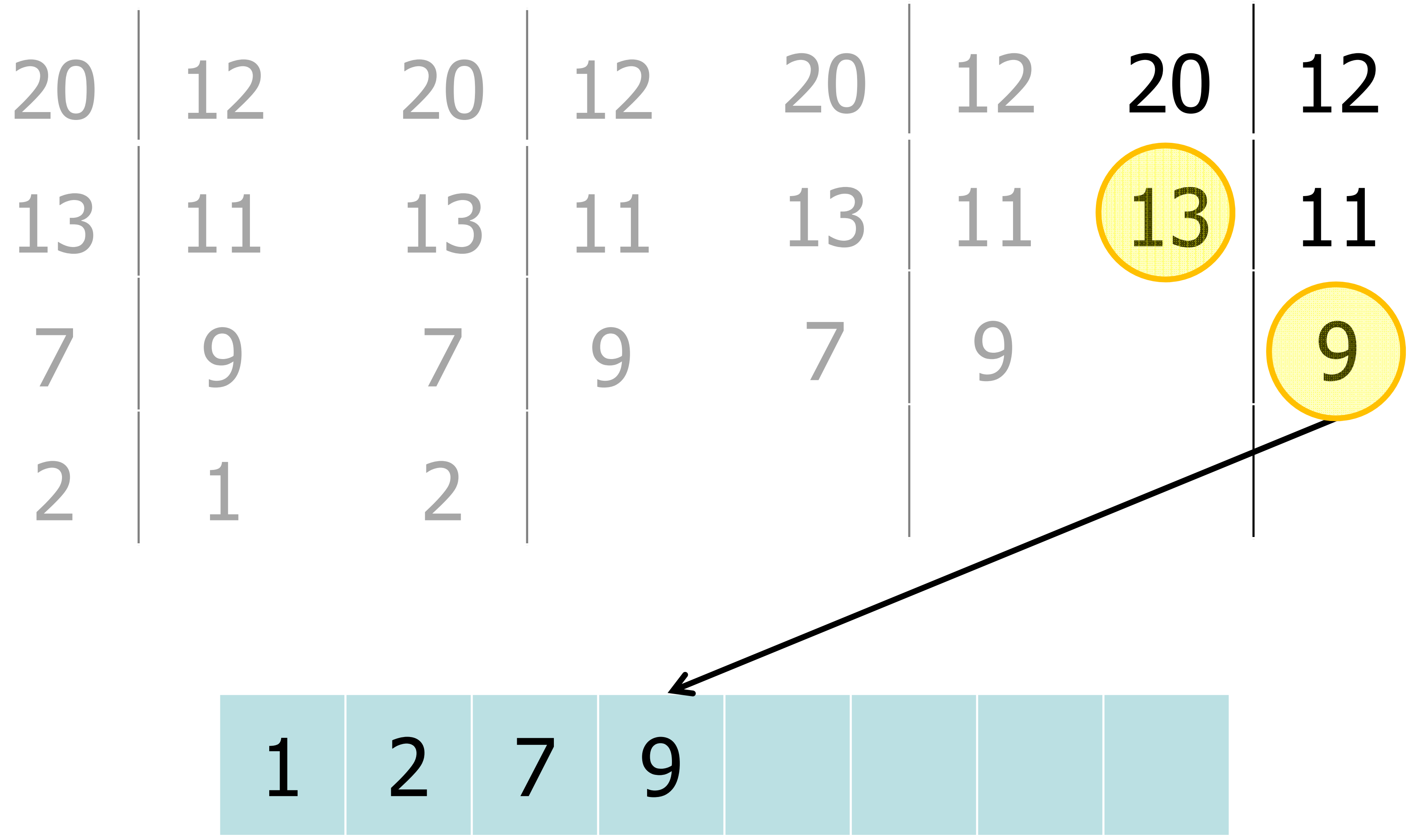


p.141

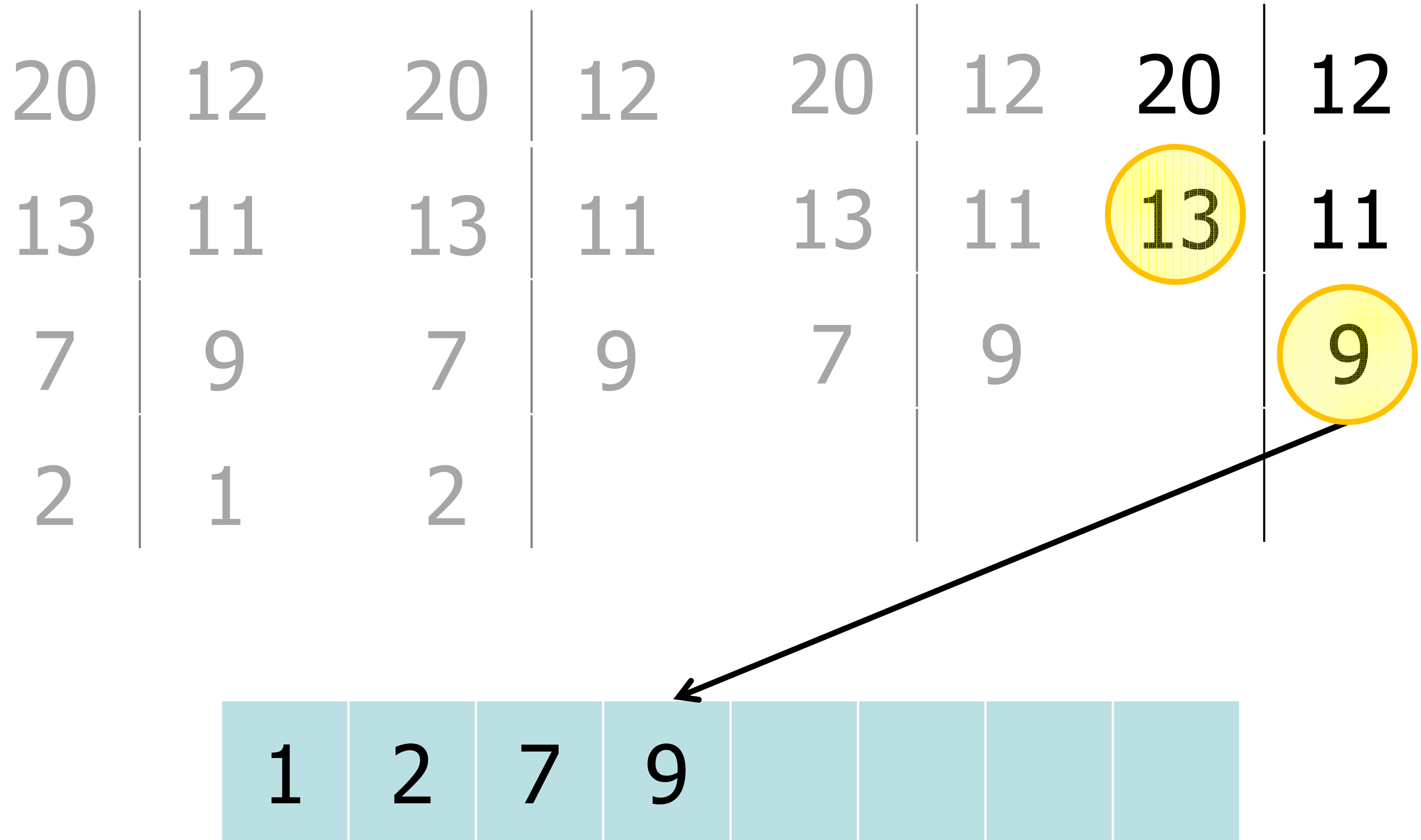
Merging Two Sorted Lists



Merging Two Sorted Lists



Merging Two Sorted Lists



p.144

Merging Two Sorted Lists

20	12	20	12	20	12	20	12
13	11	13	11	13	11	13	11
7	9	7	9	7	9		
2	1	2					

1	2	7	9	11	12	13	20

Merge: Running Time

Given two lists:

- A of size n/2
- B of size n/2

Total running time:??

Merge: Running Time

Given two lists:

- A of size n/2
- B of size n/2

Total running time: $O(n) = cn$

- In each iteration, move one element to final list

Merge-Sort Analysis

Let $T(n)$ be the worst-case running time for an array of n elements.

MergeSort(A, n)

if $(n=1)$ **then return;** ←------------------- $\theta(1)$

else:

$X \leftarrow$ Merge-Sort**(**…**);** ←------------ $T(n/2)$

$Y \leftarrow$ Merge-Sort**(**…**);** ←------------ $T(n/2)$

return Merge **(**$X, Y, n/2$**);** ←------------ $\theta(n)$

MergeSortAnalysis

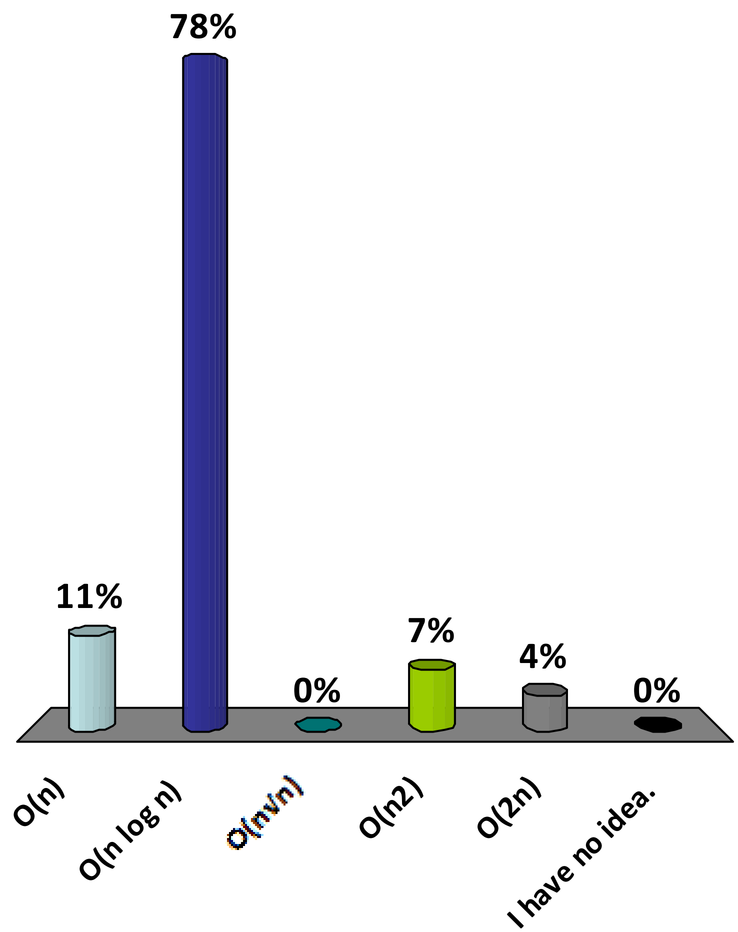
Let $T(n)$ be the worst-case running time for an array of n elements.

$$T(n) = \theta(1) \quad \textbf{if } (n=1)$$

$$= 2T(n/2) + cn \quad \textbf{if } (n>1)$$

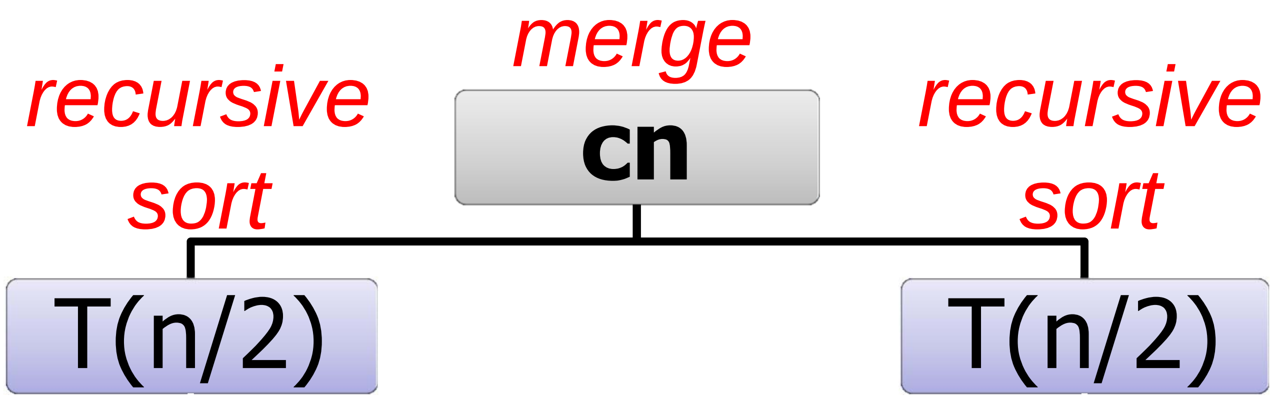
What is the (worst-case) running time of MergeSort?

A. O(n)
B. O(n log n)
C. O(n√n)
D. O(n^2)
E. O(2^n)
F. I have no idea.



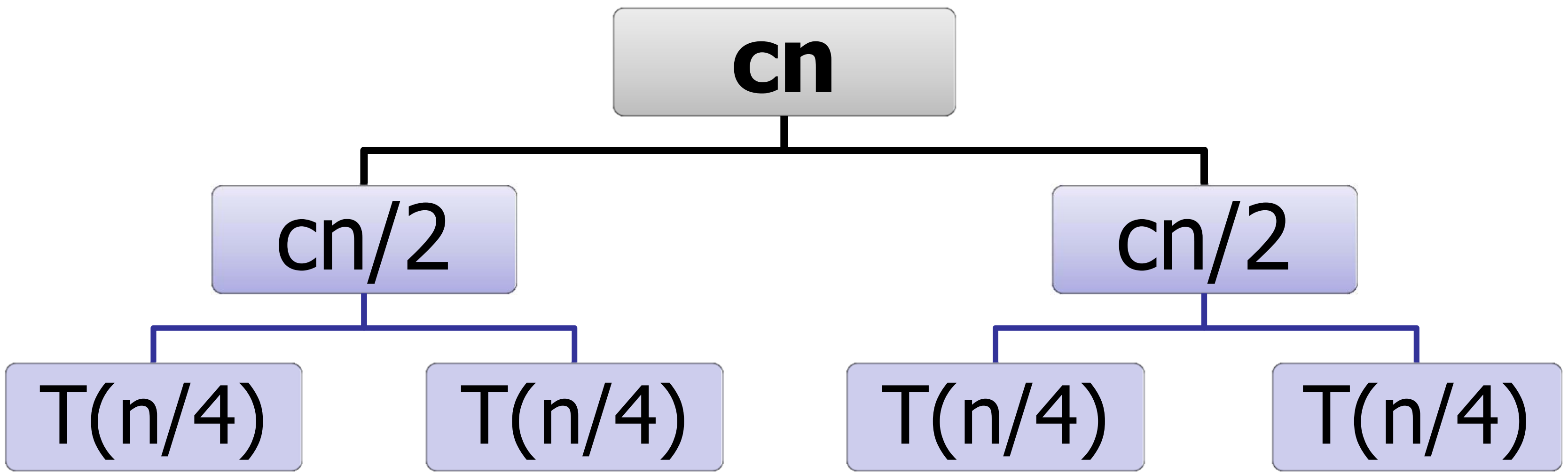
MergeSortAnalysis

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



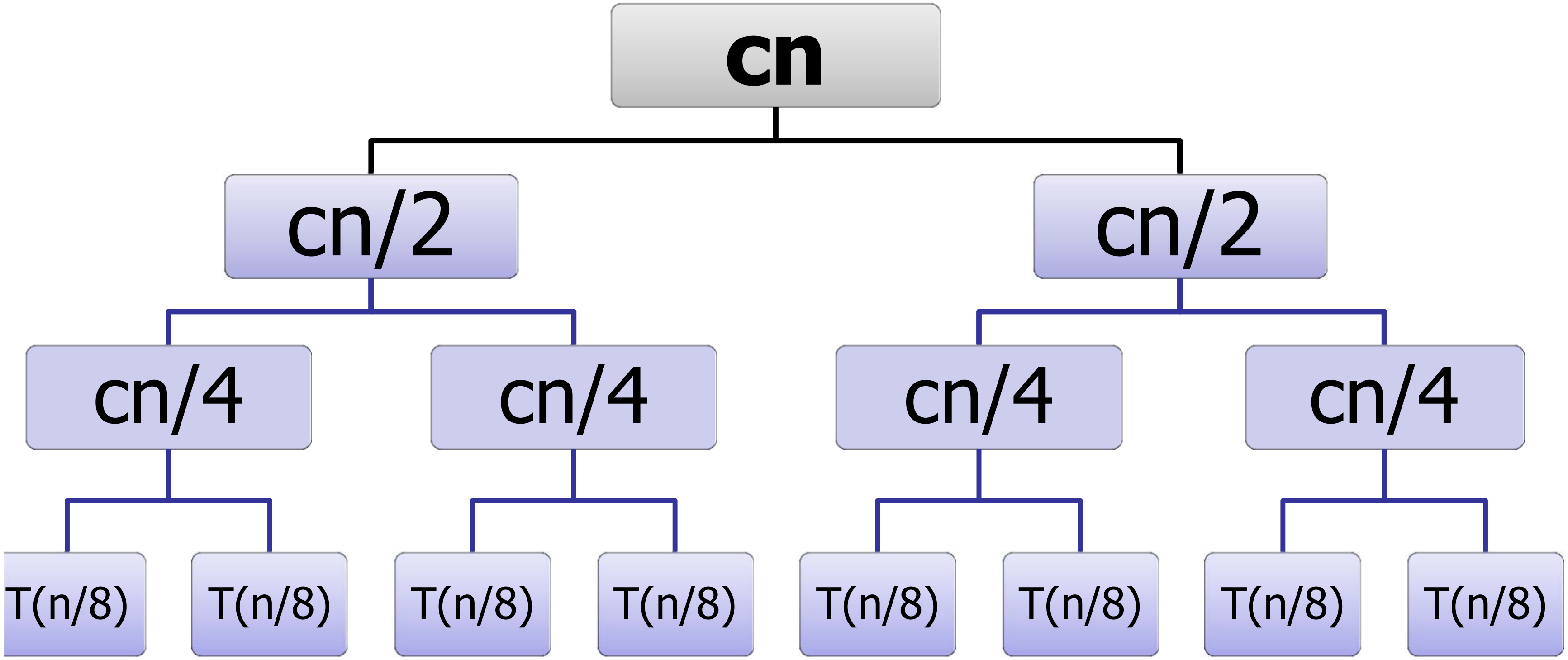
MergeSortAnalysis

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



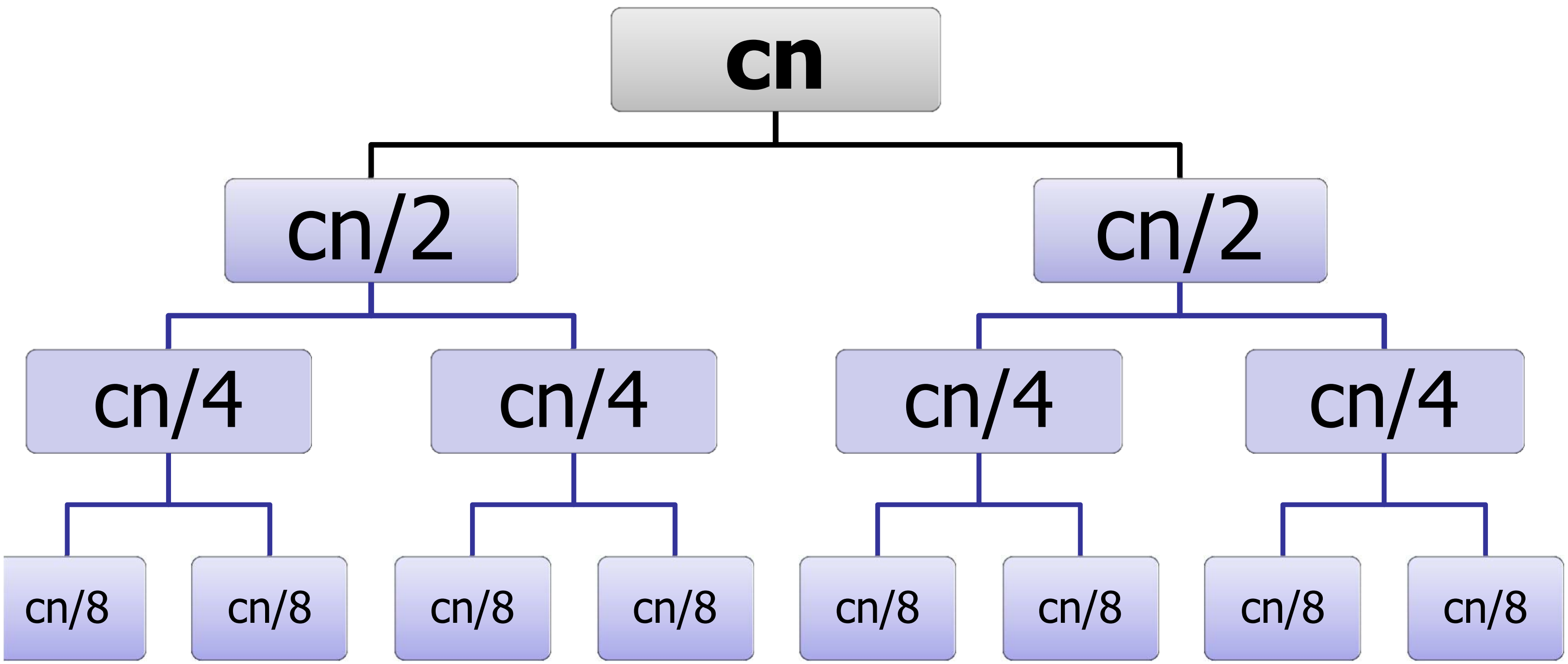
MergeSort Analysis

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



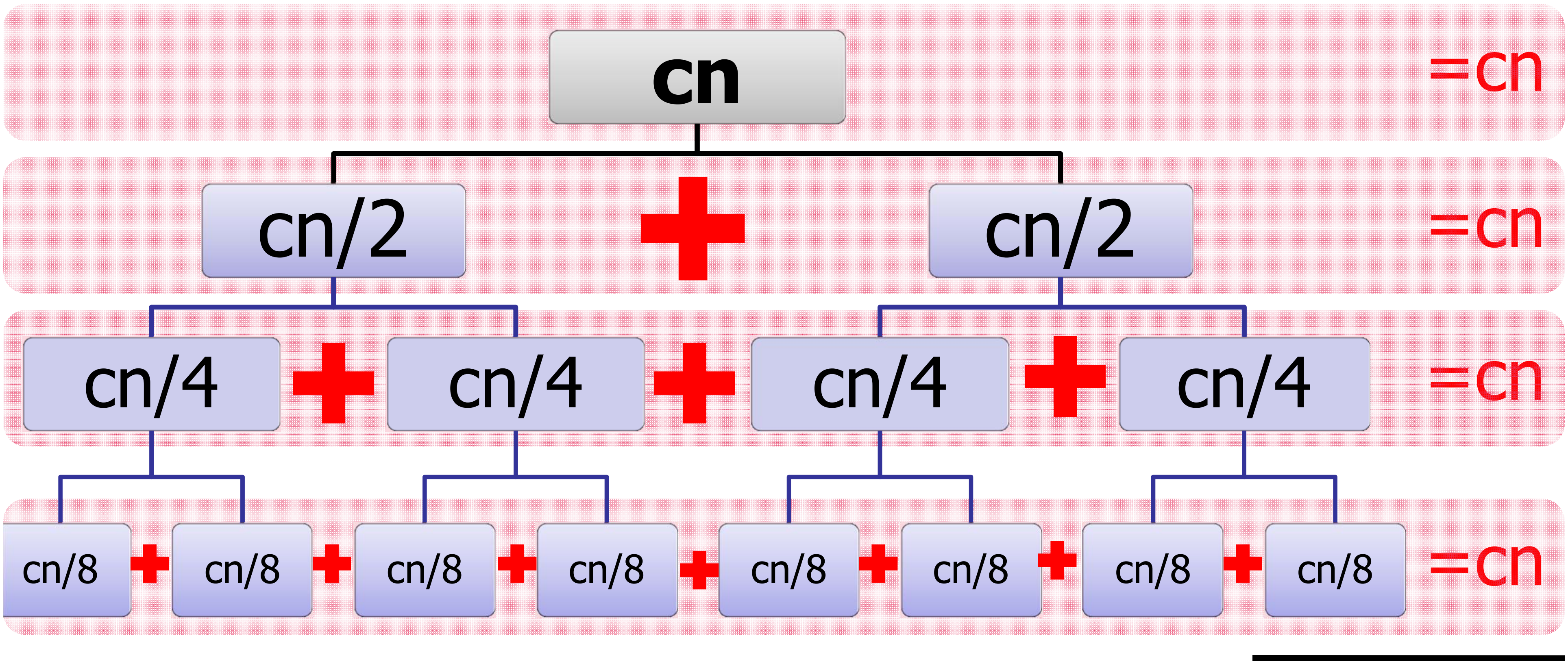
MergeSort Analysis

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



MergeSort Analysis

T(n) = 2T(n/2) + cn



MergeSort Analysis

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

Level	Number
0	1
1	2
2	4
3	8
4	16
…	…
h	??

$$\text{Number} = 2^{\text{Level}}$$

MergeSort Analysis

T(n) = 2T(n/2) + cn

Level	Number
0	1
1	2
2	4
3	8
4	16
…	…
h	n

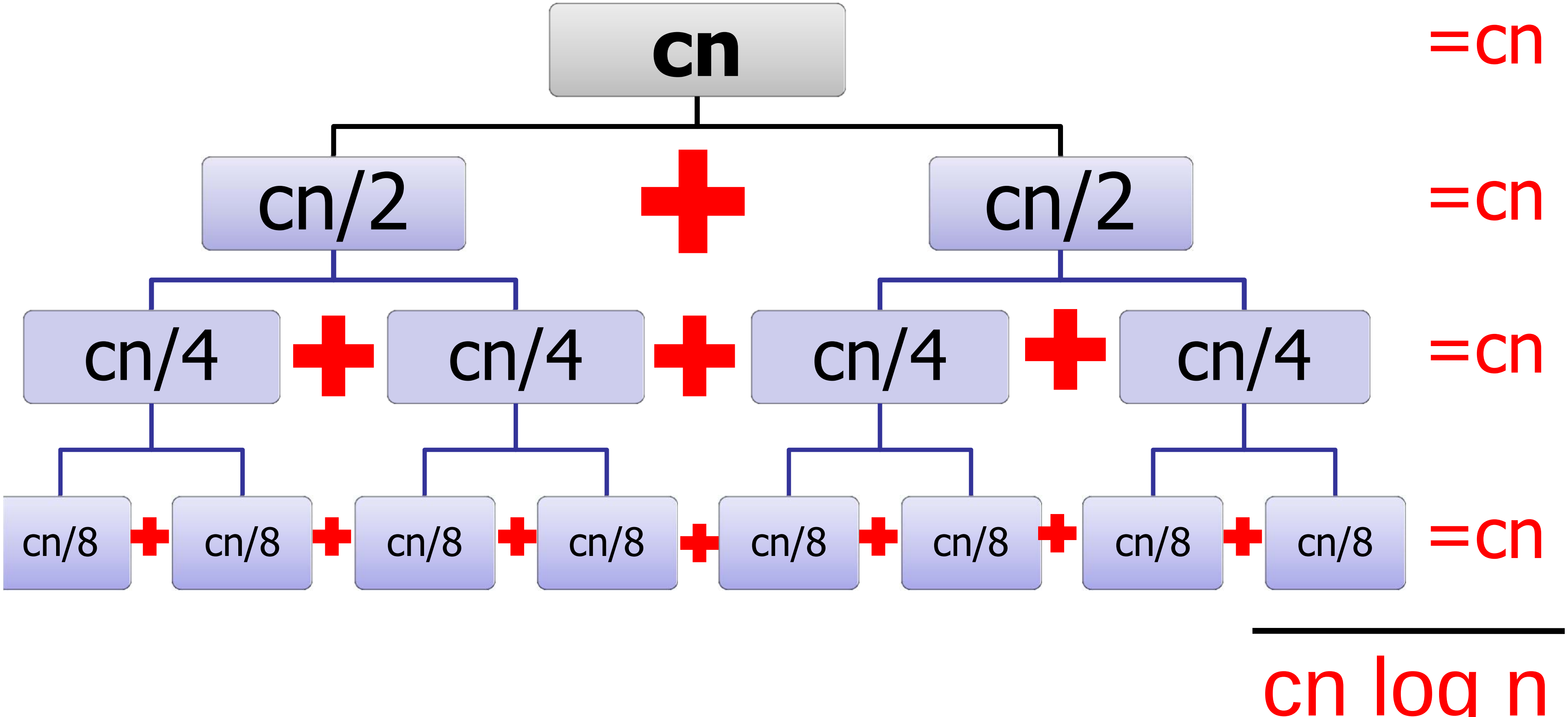
$$\text{Number} = 2^{\text{Level}}$$

$$n = 2^h$$

$$\log n = h$$

MergeSort Analysis

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



p.158

MergeSortAnalysis

$T(n) = O(n \log n)$

MergeSort(A, n)

if (n=1) **then return;**

else:

X ← MergeSort**(**…**);**

Y ← MergeSort**(**…**);**

return Merge **(**X,Y, n/2**);**

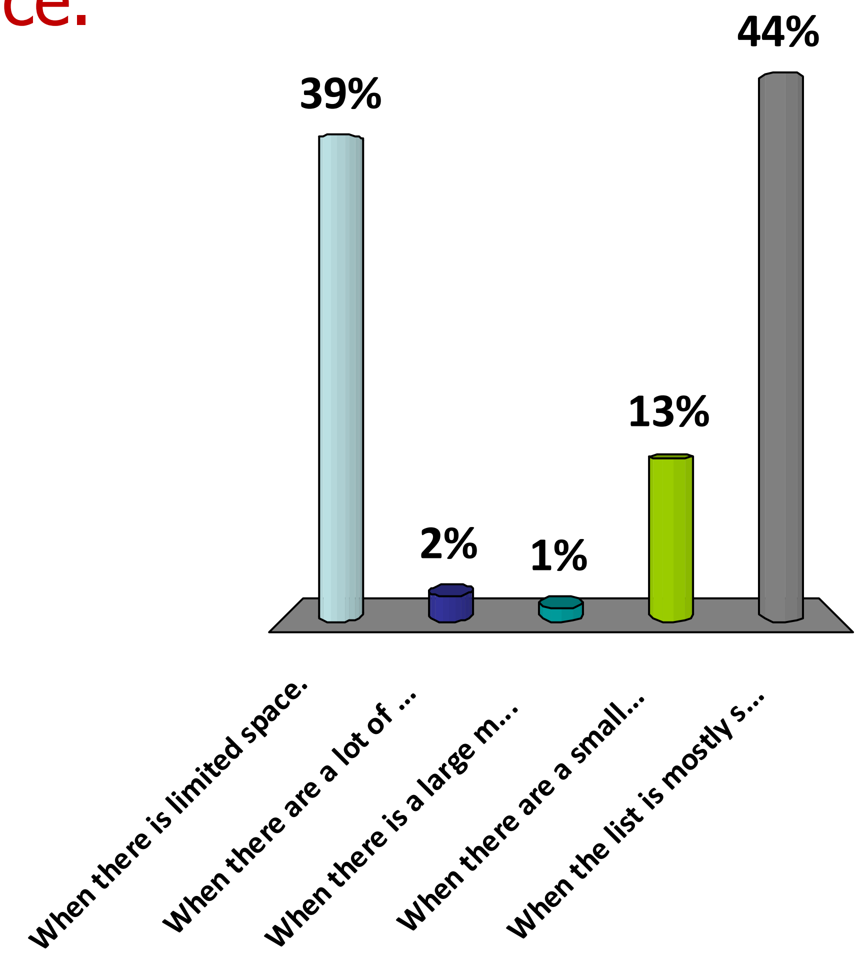
Performance Profiling

(*Dracula* vs. *Lewis & Clark*)

Version	Change	Running Time
Version 1		4,311.00s
Version 2	Better file handling	676.50s
Version 3	Faster sorting	6.59s
Version 4	No sorting!	2.35s

When is it better to use InsertionSort instead of MergeSort?

A. When there is limited space.
B. When there are a lot of items to sort.
C. When there is a large memory cache.
D. When there are a small number of items.
E. When the list is mostly sorted.



MergeSort

When the list is mostly sorted:

- InsertionSort is fast!
- MergeSort is O(n log n)

How "close to sorted" should a list be for InsertionSort to be faster?

MergeSort

Small number of items to sort:

- MergeSort is slow!
- Caching performance, branch prediction, etc.
- User InsertionSort for $n < 1024$, say.

Base case of recursion:

- Use slower sort.

MergeSort

Space usage:

- Need extra space to do merge.
- Merge copies data to new array.
- How much extra space??

NB Challenge of the Day 2:

How much space does MergeSort need to sort n items?

MergeSort

Stability:

- MergeSort is stable if “merge” is stable.
- Merge is stable if carefully implemented.

Sorting Analysis

Summary:

- BubbleSort: $O(n^2)$
- SelectionSort: $O(n^2)$
- InsertionSort: $O(n^2)$
- MergeSort: $O(n \log n)$

Properties: time, space, stability

For next time…

Friday lecture:

– More sorting

Problem Set 3:

– Released today. Due next week.