

SORTING AND SEARCHING

Lab 22

In the very early days of Facebook, when it had fewer than 15 employees, a guy named Steve Chen decided after working there for only a few weeks that it just wasn't for him. He wanted to leave to found his own company, and his plan was to do a video startup.

Matt Cohler, the guy who had hired him in the first place, tried to convince him otherwise. "You're making a terrible mistake. Facebook is going to be huge! And there's already a ton of video sites. If you do this you're going to regret it for the rest of your life!"

Chen wasn't convinced, so he decided to do it anyway and left to start a company called YouTube.

Sorts

Quadratic (N^2)

The Bubble Sort

Bubble Sort

Bubble sort compares items that are adjacent and has the potential to swap a whole lot.

Bubble Sort is left in for historical purposes only!

Bubble Sort W/Objects

```
void bubbleSort( Comparable[] stuff ){  
    for(int i=0; i<stuff.length-1; i++){  
        for(int j=0; j<stuff.length-1; j++){  
            if(stuff[ j].compareTo(stuff[ j+1]) > 0 ){  
                Comparable temp = stuff[ j];  
                stuff[ j] = stuff [ j+1];  
                stuff [ j+1] = temp;  
            }  
        }  
    }  
}
```

Lots O Swaps!




Bubble Sort W/Objects

```
void bubbleSort(Comparable[] stuff) {  
    boolean changed = false;  
    do {  
        changed = false;  
        for (int a = 0; a < stuff.length - 1; a++) {  
            if (stuff[a].compareTo(stuff[a + 1]) > 0) {  
                Comparable tmp = stuff[a];  
                stuff[a] = stuff[a + 1];  
                stuff[a + 1] = tmp;  
                changed = true;  
            }  
        }  
    } while(changed);  
}
```

The Selection Sort

Selection Sort

The selection sort does not swap each time it finds elements out of position. Selection sort makes a complete pass while searching for the next item to swap. At the end of a pass once the item is located, one swap is made.



Selection Sort

```
void selectionSort( int[] ray )
{
    for(int i=0; i< ray.length-1; i++){
        int min = i;
        for(int j = i+1; j< ray.length; j++){
            {
                if(ray[j] < ray[min])
                    min = j;           //find location of smallest
            }
            if( min != i) {
                int temp = ray[min];
                ray[min] = ray[i];
                ray[i] = temp;         //put smallest in pos i
            }
        }
    }
}
```



Selection Sort

0 1 2 3 4

pass 0

9	2	8	5	1
---	---	---	---	---

pass 1

1	2	8	5	9
---	---	---	---	---

pass 2

1	2	8	5	9
---	---	---	---	---

pass 3

1	2	5	8	9
---	---	---	---	---

pass 4

1	2	5	8	9
---	---	---	---	---

```
public void selSort(Comparable[] ray){  
    for(int i=0; i<ray.length-1; i++){  
        {  
            int min=i;  
            for(int j=i; j<ray.length; j++){  
                if(ray[j].compareTo(ray[spot])<0)  
                    min=j;  
            }  
            if(min==i) continue;  
            Comparable save = ray[i];  
            ray[i]= ray[min];  
            ray[min]=save;  
        }  
    }  
}
```

**How many swaps
per pass?**

**Selection Sort
W/Objects**

Selection Sort in Action

Original List

Integer[] ray = {90,40,20,30,10,67};

pass 1 - 90 40 20 30 10 67

pass 2 - 90 67 20 30 10 40

pass 3 - 90 67 40 30 10 20

pass 4 - 90 67 40 30 10 20

pass 5 - 90 67 40 30 20 10

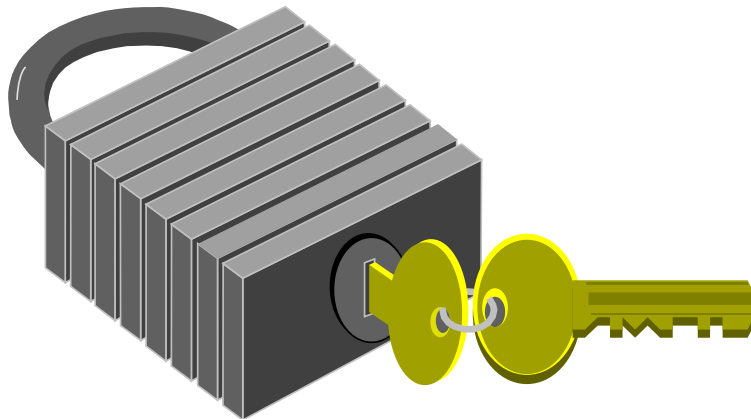
pass 6 - 90 67 40 30 20 10

open
selectionsort.java
selectionsorttester.java

People Sort Demo

Line up some students and selection sort them.

The Insertion Sort



Insertion Sort

The insertion sort first selects an item and moves items up or down based on the comparison to the selected item.

The idea is to get the selected item in proper position by shifting items around in the list.

```
void insertionSort( int[] ray)
{
    for (int i=1; i< ray.length; ++i)
    {
        int val = ray[i];
        int j=i;
        while(j>0 && val<ray[j-1]) {
            ray[j]=ray[j-1];
            j--;
        }
        ray[j]=val;
    }
}
```

**Insertion
w/primitives**

```
void insertionSort( Comparable[] ray)
{
    for (int i=1; i< ray.length; ++i)
    {
        int val = ray[i];
        int j=i;
        while(ray[mid].compareTo(ray[ i ])<0) {
            ray[j]=ray[j-1];
            j--;
        }
        ray[j]=val;
    }
}
```

**Insertion
w/Objects**

Insertion Sort in Action

Original List

Integer[] ray = {90,40,20,30,10,67};

pass 1 - 40 90 20 30 10 67

pass 2 - 20 40 90 30 10 67

pass 3 - 20 30 40 90 10 67

pass 4 - 10 20 30 40 90 67

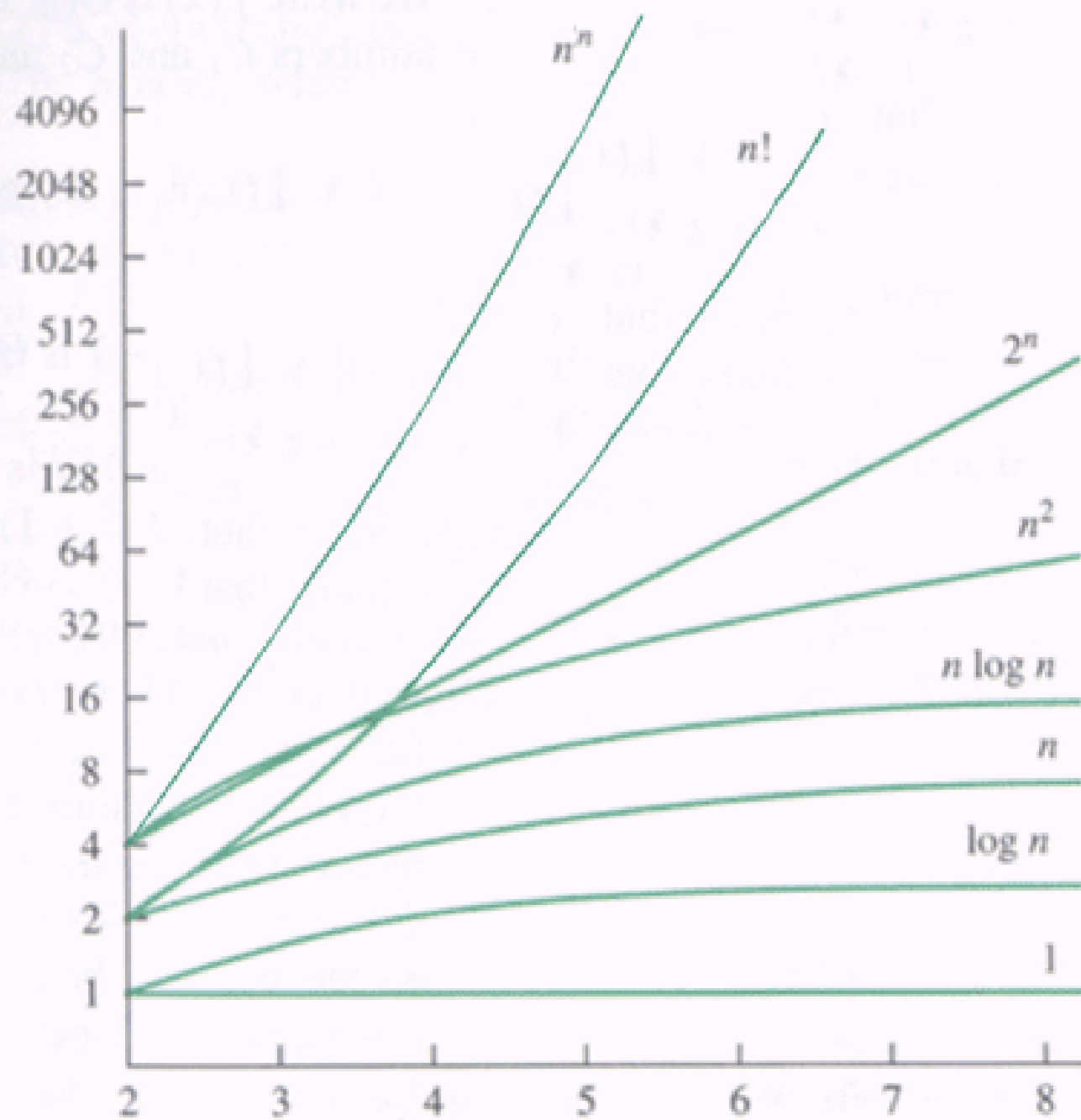
pass 5 - 10 20 30 40 67 90

```
void insertionSort( int[] ray){  
    for (int i=1; i< ray.length; ++i){  
        int bot=0, top=i-1;  
        while (bot<=top){  
            int mid=(bot+top)/2;  
            if (ray[mid] < ray[ i ])  
                bot=mid+1;  
            else top=mid-1;  
        }  
        Comparable temp = ray[i];  
        for (int j=i; j>bot; --j)  
            ray[ j]=ray[ j-1];  
        ray[bot]=temp;  
    }  
}
```

Insertion sort with a Binary search

People Sort Demo

Line up some students and insertion sort them.



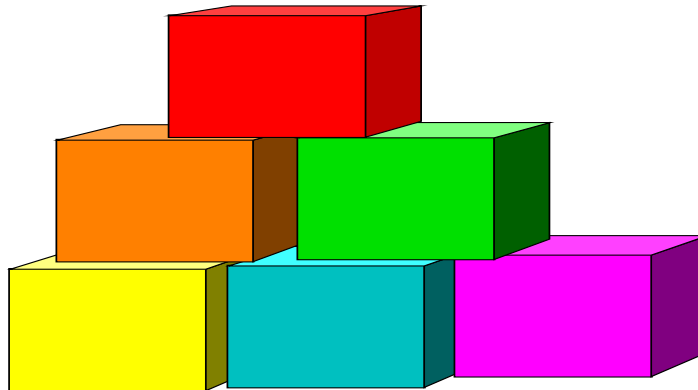
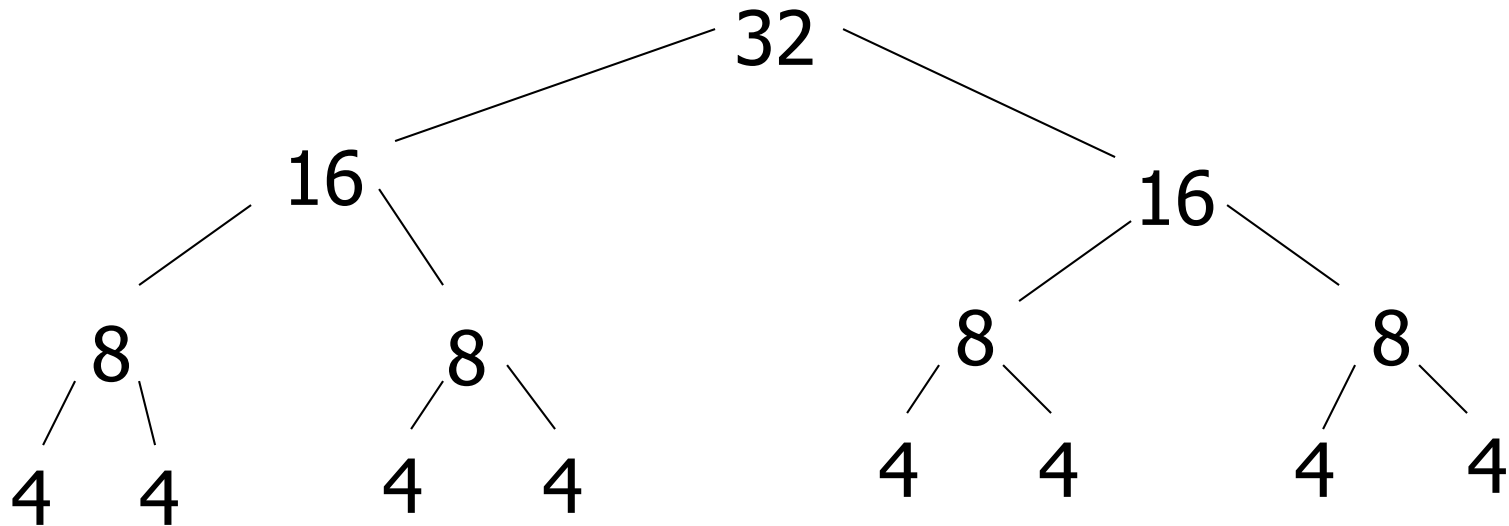
open

insertionsort.java

insertionsttester.java

Divide and Conquer Algorithms $O(N \log_2 N)$

Divide and Conquer



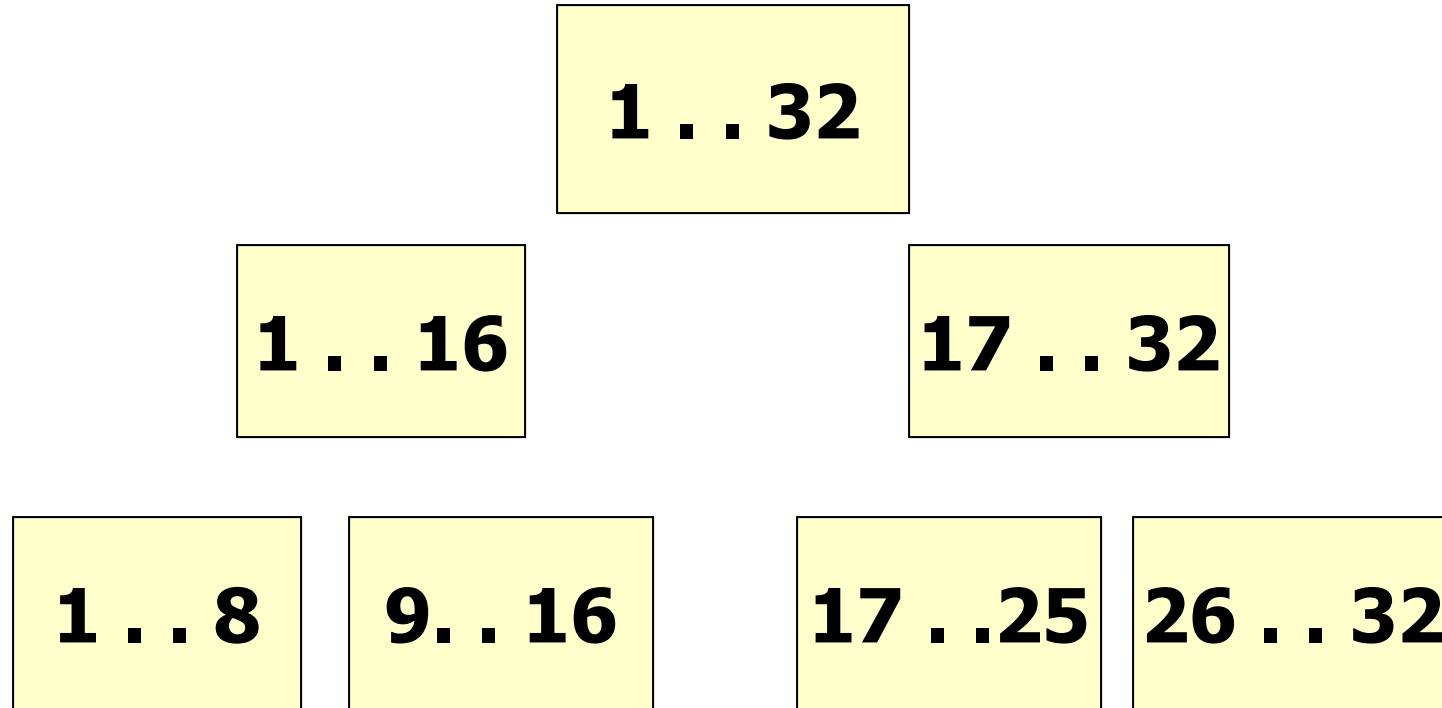
The Merge Sort



Merge Sort

Merge sort splits the list into smaller sections working its way down to groups of two or one. Once the smallest groups are reached, the merge method is called to organize the smaller lists. Merge copies from the sub list to a temp array. The items are put in the temp array in sorted order.

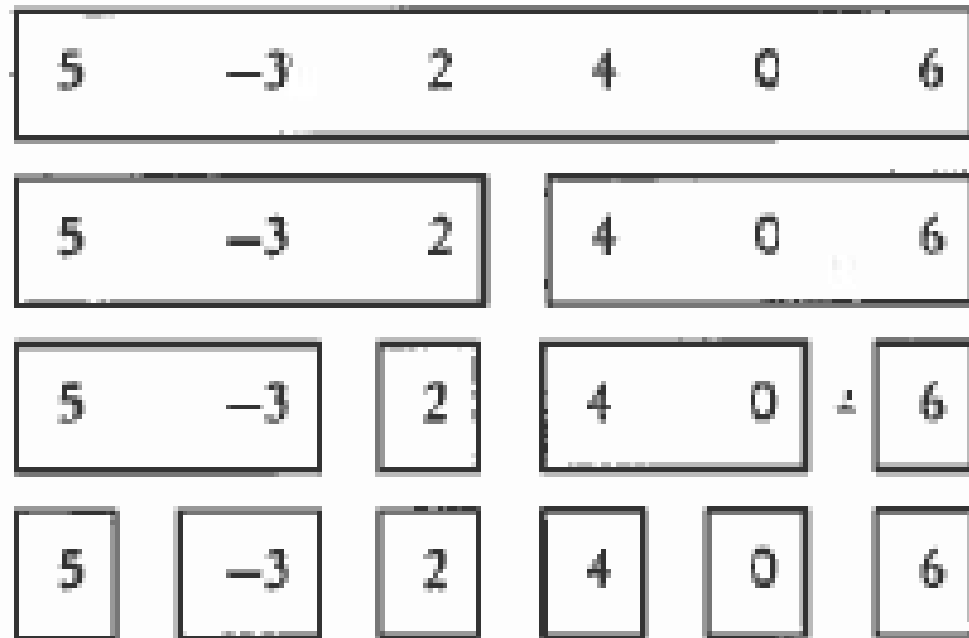
Merge Sort



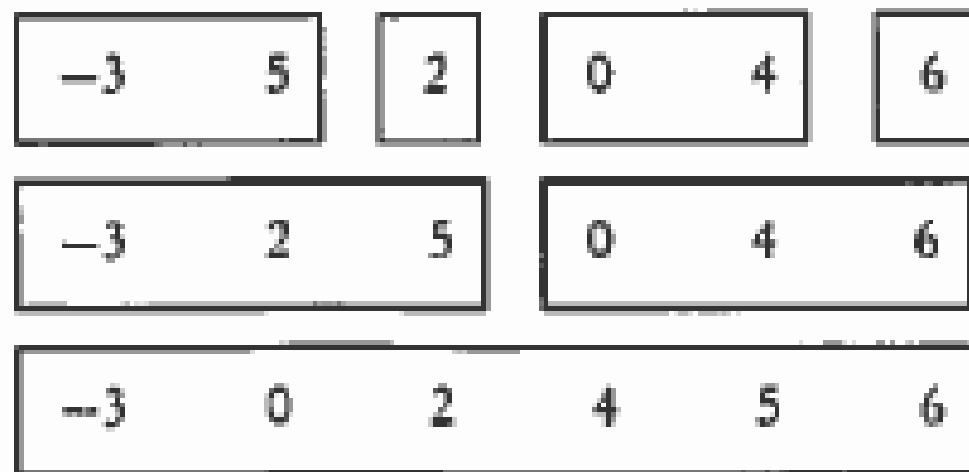
Merge sort chops in half repeatedly to avoid processing the whole list at once.

a[0] a[1] a[2] a[3] a[4] a[5]

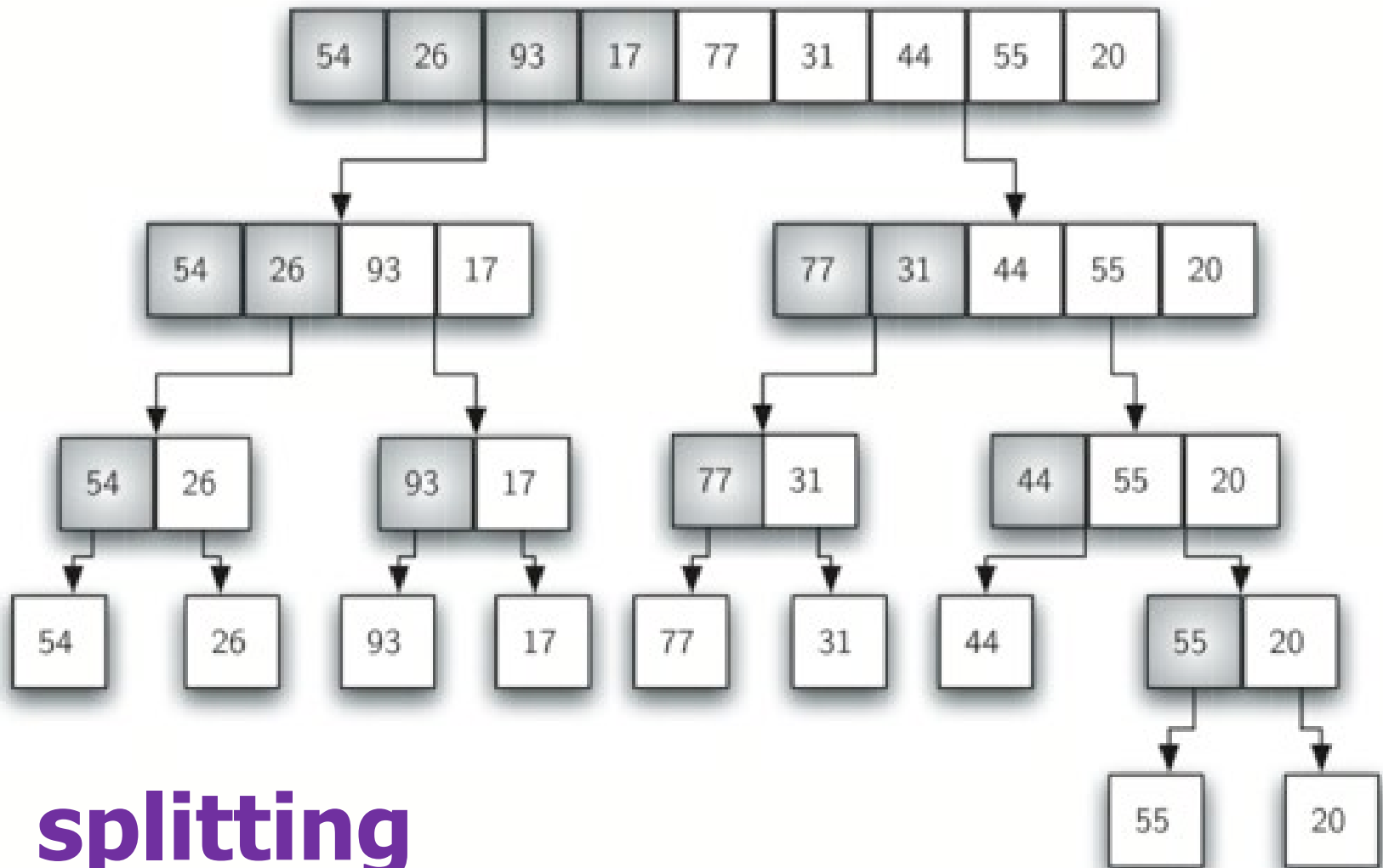
Break list into
 n sublists of
length 1

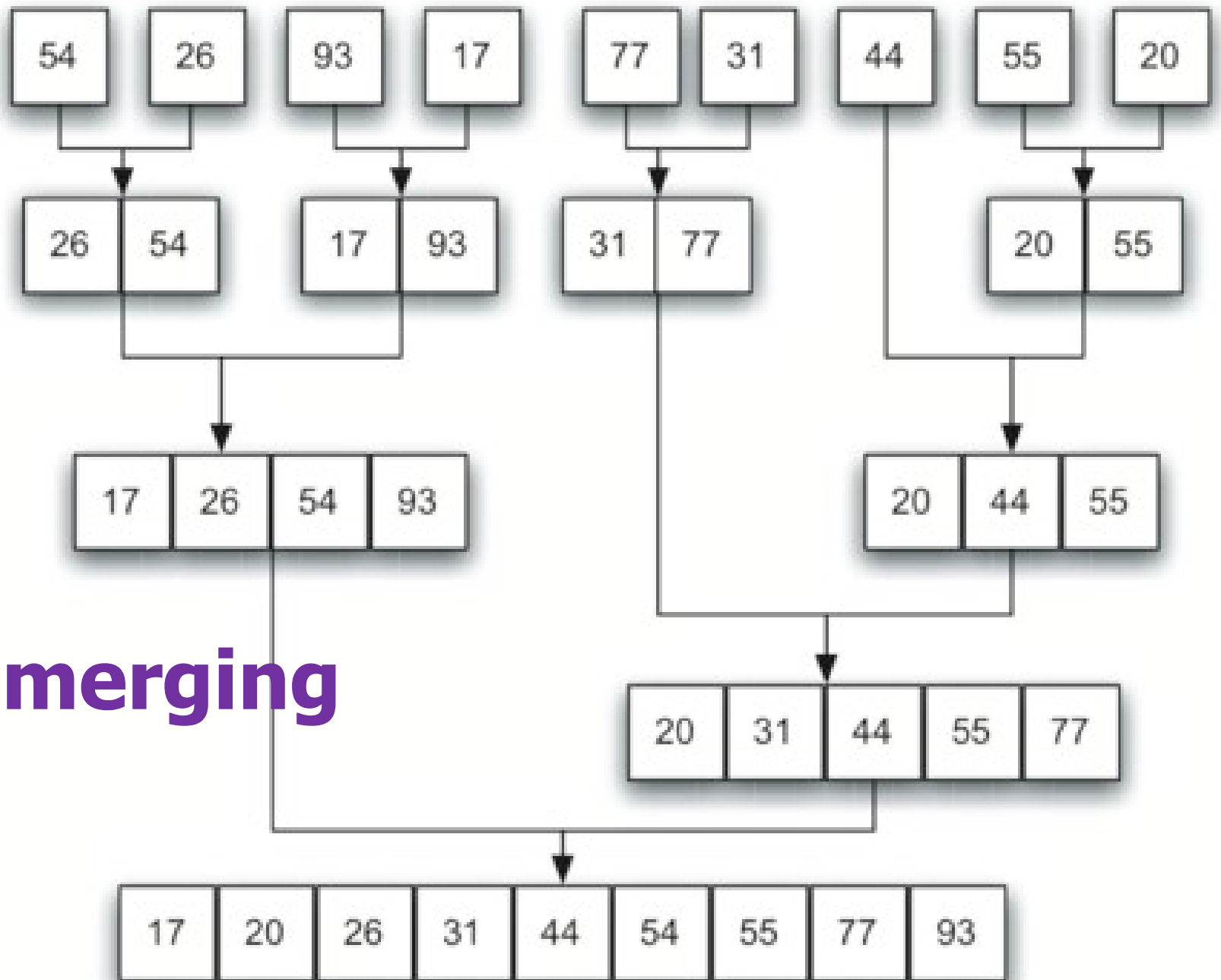


Merge adjacent
pairs of lists



mergeSort, but with a right-hand split





mergeSort Algorithm

```
void mergeSort(Comparable[] stuff, int front, int back)
{
    int mid = (front+back)/2;
    if(mid==front) return;
    mergeSort(stuff, front, mid);
    mergeSort(stuff, mid, back);
    merge(stuff, front, back);
}
```

Collections.sort() uses the mergeSort.

Arrays.sort() uses mergeSort for objects.

```

void merge(Comparable[] stuff, int front, int back)
{
    Comparable[] temp = new Comparable[back-front];
    int i = front, j = (front+back)/2, k = 0, mid = j;
    while( i < mid && j < back) {
        if(stuff[i].compareTo(stuff[j]) < 0)
            temp[k++] = stuff[i++];
        else
            temp[k++] = stuff[j++];
    }

    while(i < mid)
        temp[k++] = stuff[i++];
    while(j < back)
        temp[k++] = stuff[j++];
    for(i = 0; i < back-front; ++i)
        stuff[front+i] = temp[i];
}

```

Merge
W/Objects

Merge Sort in Action

Original List

Integer[] stuff = {90,40,20,30,67,10};

pass 0	-	40	90	20	30	67	10
pass 1	-	20	40	90	30	67	10
pass 2	-	20	40	90	30	67	10
pass 3	-	20	40	90	10	30	67
pass 4	-	10	20	30	40	67	90

The mergeSort has a $N * \log_2 N$ BigO.

mergeSort

The mergeSort method alone has a $\log_2 N$ run time, but cannot be run without the merge method.

Merge

The merge method alone has an N run time and can be run without the mergeSort method.

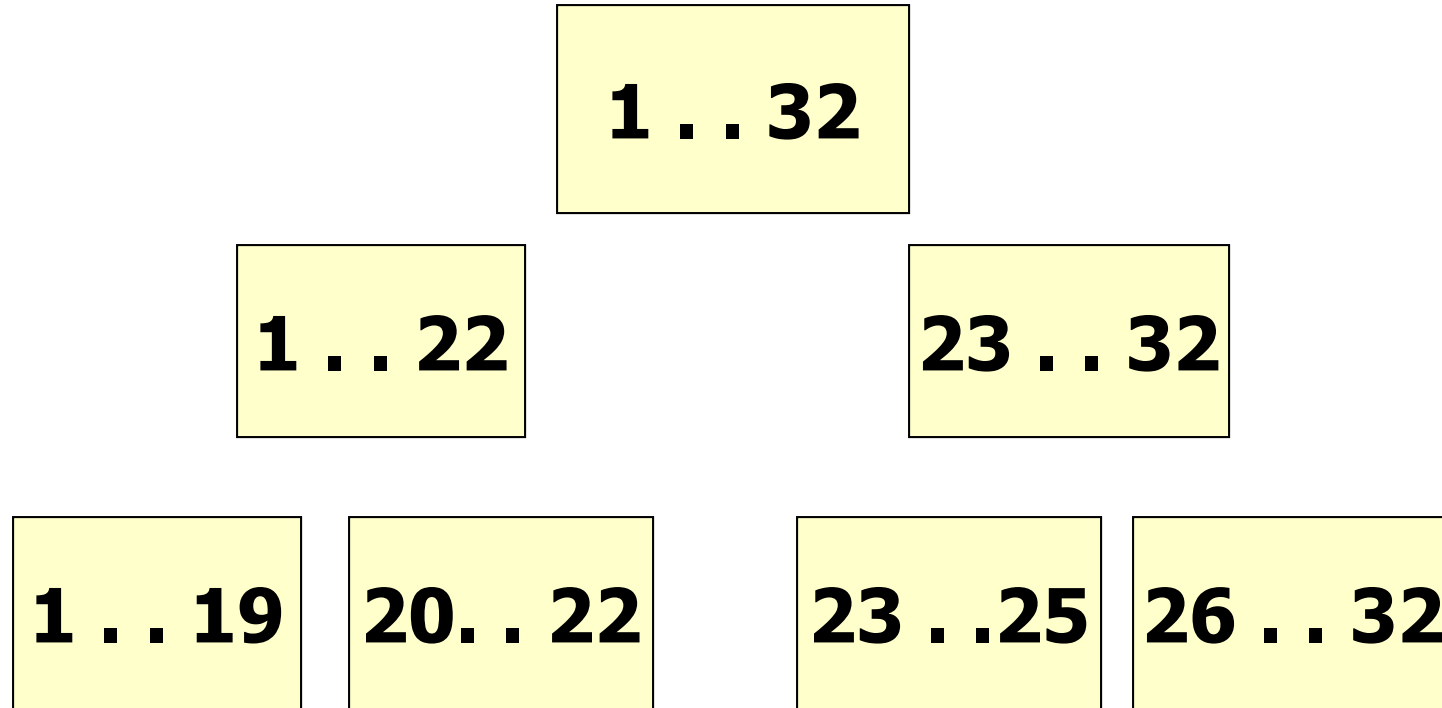
The Quick Sort

Quick Sort

Quick sort finds a pivot value. All numbers greater than the pivot move to the right and all numbers less move to the left.

This list is then chopped in two and the process above is repeated on the smaller sections.

Quick Sort



Quick sort chops up the list into smaller pieces as to avoid processing the whole list at once.

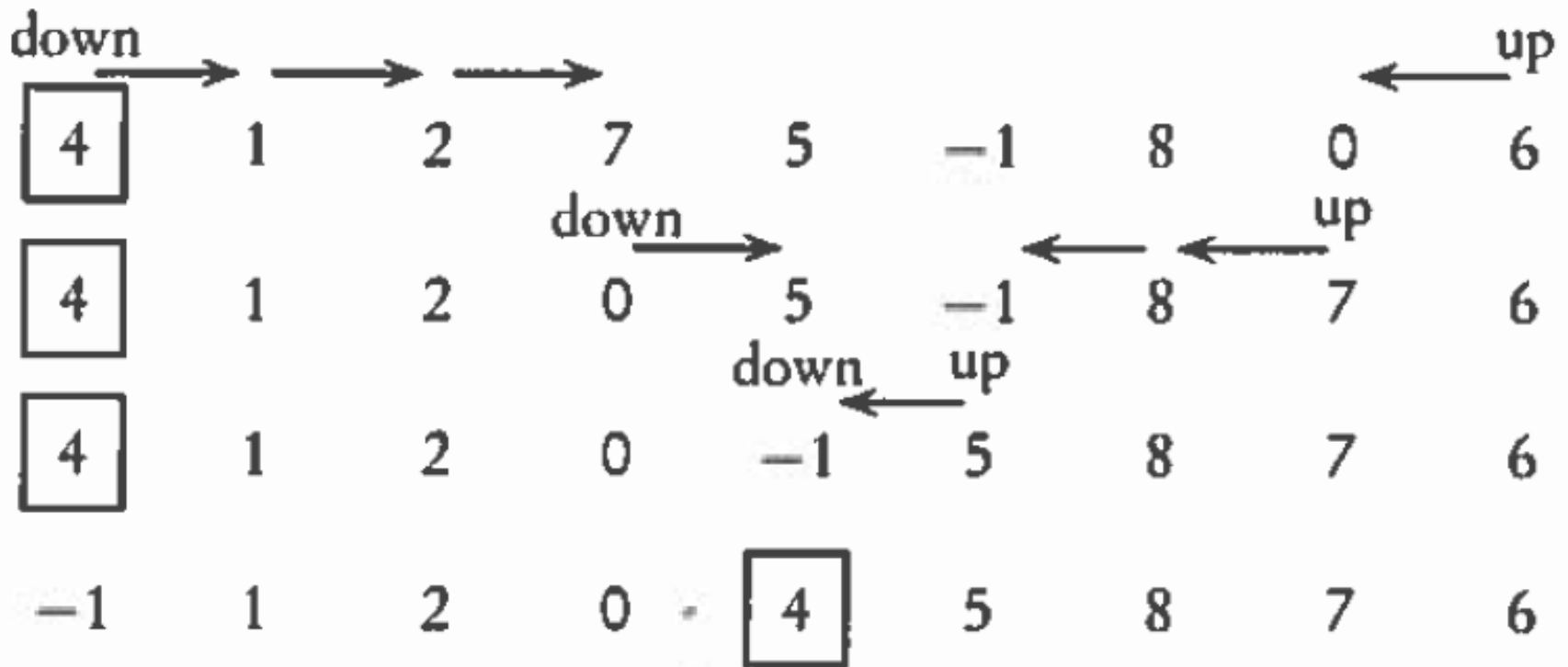
quickSort Algorithm

```
void quickSort(Comparable[] stuff, int low, int high)
{
    if (low < high)
    {
        int spot = partition(stuff, low, high);
        quickSort(stuff, low, spot);
        quickSort(stuff, spot+1, high);
    }
}
```

**Arrays.sort() uses the quickSort
if sorting primitives.**

partition Algorithm

4, 1, 2, 7, 5, -1, 8, 0, 6



partition Algorithm

```
int partition(Comparable[] stuff, int low, int high)
{
    Comparable pivot = stuff[low];
    int bot = low-1;
    int top = high+1;
    while(bot<top) {
        while (stuff[--top].compareTo(pivot) > 0);
        while (stuff[++bot].compareTo(pivot) < 0);
        if(bot >= top)
            return top;
        Comparable temp = stuff[bot];
        stuff[bot] = stuff[top];
        stuff[top] = temp;
    }
}
```

Quick Sort in Action

Original List

Integer[] ray = {90,40,20,30,10,67};

pass 1 - 67 40 20 30 10 90
pass 2 - 10 40 20 30 67 90
pass 3 - 10 40 20 30 67 90
pass 4 - 10 30 20 40 67 90
pass 5 - 10 20 30 40 67 90

The quickSort has a $N * \log_2 N$ BigO.

quickSort

The quickSort method alone has a $\log_2 N$ run time, but cannot be run without the partition method.

Partition

The partition method alone has an N run time and can be run without the quickSort method.

Speed

Runtime Analysis

```
for( int i=0; i<20; i++)  
    System.out.println(i);
```

```
for( int j=0; j<20; j++)  
    for( int k=0; k<20; k++)  
        System.out.println(j*k);
```

**Which section of
code would execute
the fastest?**

Runtime Analysis

```
ArrayList<Integer> iRay;  
iRay = new ArrayList<Integer>();  
for( int i=0; i<20; i++)  
    iRay.add(i);
```

**Which section of
code would execute
the fastest?**

```
ArrayList<Double> dRay;  
dRay = new ArrayList<Double>();  
for( int j=0; j<20; j++)  
    dRay.add(0,j);
```

General Big O Chart for N^2 Sorts

Name	Best Case	Avg. Case	Worst
Selection Sort	$O(N^2)$	$O(N^2)$	$O(N^2)$
Bubble Sort	$O(N^2)$	$O(N^2)$	$O(N^2)$
Insertion Sort	$O(N)$ *	$O(N^2)$	$O(N^2)$

* If the data is sorted, Insertion sort should only make one pass through the list. If this case is present, Insertion sort would have a best case of $O(n)$.

General Big O Chart for NLogN Sorts

Name	Best Case	Avg. Case	Worst
Merge Sort	$O(N \log_2 N)$	$O(N \log_2 N)$	$O(N \log_2 N)$
QuickSort	$O(N \log_2 N)$	$O(N \log_2 N)$	$O(N^2)$ *

* QuickSort can degenerate to N^2 . It typically will degenerate on sorted data if using a left or right pivot. Using a median pivot will help tremendously, but QuickSort can still degenerate on certain sets of data. The split position determines how QuickSort behaves.

**Continue work
on Lab22**