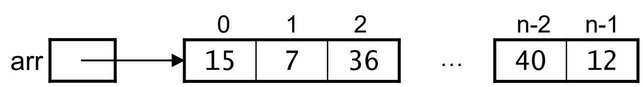
Unit 7

Sorting and Algorithm Analysis

Computer Science S-111
Harvard University

David G. Sullivan, Ph.D.

Sorting an Array of Integers



- Ground rules:
  - sort the values in increasing order
  - sort "in place," using only a small amount of additional storage
- Terminology:
  - position: one of the memory locations in the array
  - element: one of the data items stored in the array
  - element i: the element at position i
- Goal: minimize the number of **comparisons *C*** and the number of **moves *M*** needed to sort the array.
  - move = copying an element from one position to another
    example: `arr[3] = arr[5];`

Defining a Class for our Sort Methods

```
public class Sort {
    public static void bubbleSort(int[] arr) {
        ...
    }
    public static void insertionSort(int[] arr) {
        ...
    }
    ...
}
```

- Our `Sort` class is simply a collection of methods like Java's built-in `Math` class.
- Because we never create `Sort` objects, all of the methods in the class must be *static*.
  - outside the class, we invoke them using the class name: e.g., `Sort.bubbleSort(arr)`

Defining a Swap Method

- It would be helpful to have a method that swaps two elements of the array.
- Why won't the following work?

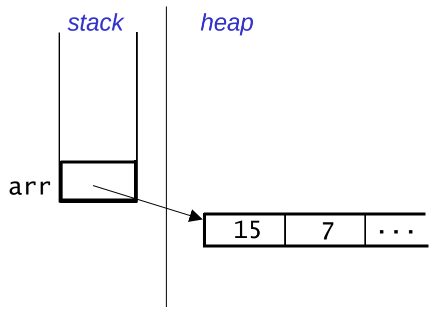
```
public static void swap(int a, int b) {
    int temp = a;
    a = b;
    b = temp;
}
```

An Incorrect Swap Method

```
public static void swap(int a, int b) {
    int temp = a;
    a = b;
    b = temp;
}
```

- Trace through the following lines to see the problem:

```
int[] arr = {15, 7, …};
swap(arr[0], arr[1]);
```



A Correct Swap Method

- This method works:

```
public static void swap(int[] arr, int a, int b) {
    int temp = arr[a];
    arr[a] = arr[b];
    arr[b] = temp;
}
```

- Trace through the following with a memory diagram to convince yourself that it works:

```
int[] arr = {15, 7, …};
swap(arr, 0, 1);
```

Selection Sort

- Basic idea:
  - consider the positions in the array from left to right
  - for each position, find the element that belongs there and put it in place by swapping it with the element that's currently there
- Example:

0	1	2	3	4
15	6	*2*	12	4

0	*1*	2	3	4
2	6	15	12	*4*

0	1	*2*	3	4
2	4	15	12	*6*

0	1	2	*3*	4
2	4	6	*12*	15

Why don't we need to consider position 4?

Selecting an Element

- When we consider position `i`, the elements in positions 0 through `i – 1` are already in their final positions.

example for `i = 3`:

0	1	2	*3*	4	5	6
2	4	7	21	25	10	17

- To select an element for position `i`:
  - consider elements `i`, `i+1,i+2,…,arr.length – 1`, and keep track of `indexMin`, the index of the smallest element seen thus far

`indexMin: 3, 5`

0	1	2	3	4	5	6
2	4	7	21	25	*10*	17

  - when we finish this pass, `indexMin` is the index of the element that belongs in position `i`.
  - swap `arr[i]` and `arr[indexMin]`:

0	1	2	3	4	5	6
2	4	7	10	25	21	17

Implementation of Selection Sort

- Use a helper method to find the index of the smallest element:

```
private static int indexSmallest(int[] arr, int start) {
    int indexMin = start;

    for (int i = start + 1; i < arr.length; i++) {
        if (arr[i] < arr[indexMin]) {
            indexMin = i;
        }
    }

    return indexMin;
}
```

- The actual sort method is very simple:

```
public static void selectionSort(int[] arr) {
    for (int i = 0; i < arr.length - 1; i++) {
        int j = indexSmallest(arr, i);
        swap(arr, i, j);
    }
}
```

Time Analysis

- Some algorithms are much more efficient than others.
- The *time efficiency* or *time complexity* of an algorithm is some measure of the number of operations that it performs.
  - for sorting, we'll focus on comparisons and moves
- We want to characterize how the number of operations depends on the size, n, of the input to the algorithm.
  - for sorting, n is the length of the array
  - how does the number of operations grow as n grows?
- We'll express the number of operations as functions of n
  - `C(n)` = number of comparisons for an array of length n
  - `M(n)` = number of moves for an array of length n

Counting Comparisons by Selection Sort

```
private static int indexSmallest(int[] arr, int start){
    int indexMin = start;

    for (int i = start + 1; i < arr.length; i++) {
        if (arr[i] < arr[indexMin]) {
            indexMin = i;
        }
    }

    return indexMin;
}
public static void selectionSort(int[] arr) {
    for (int i = 0; i < arr.length - 1; i++) {
        int j = indexSmallest(arr, i);
        swap(arr, i, j);
    }
}
```

- To sort n elements, selection sort performs n - 1 passes:

  on 1st pass, it performs ____ comparisons to find `indexSmallest`
  on 2nd pass, it performs ____ comparisons
  ...
  on the (n-1)st pass, it performs 1 comparison

- Adding them up: $C(n) = 1 + 2 + \ldots + (n - 2) + (n - 1)$

Counting Comparisons by Selection Sort (cont.)

- The resulting formula for $C(n)$ is the sum of an arithmetic sequence:

$$C(n) = 1 + 2 + \ldots + (n - 2) + (n - 1) = \sum_{i=1}^{n-1} i$$

- Formula for the sum of this type of arithmetic sequence:

$$\sum_{i=1}^{m} i = \frac{m(m+1)}{2}$$

- Thus, we can simplify our expression for C(n) as follows:

$$C(n) = \sum_{i=1}^{n-1} i$$

$$= \frac{(n-1)((n-1)+1)}{2}$$

$$= \frac{(n-1)n}{2}$$

$$\boxed{C(n) = n^2/2 - n/2}$$

Focusing on the Largest Term

- When n is large, mathematical expressions of n are dominated by their "largest" term — i.e., the term that grows fastest as a function of n.
  - example:

n	$n^2/2$	$n/2$	$n^2/2 - n/2$
10	50	5	45
100	5000	50	4950
10000	50,000,000	5000	49,995,000

- In characterizing the time complexity of an algorithm, we'll focus on the largest term in its operation-count expression.
  - for selection sort, $C(n) = n^2/2 - n/2 \approx n^2/2$
- In addition, we'll typically ignore the coefficient of the largest term (e.g., $n^2/2 \rightarrow n^2$).

Big-*O* Notation

- We specify the largest term using big-*O* notation.
  - e.g., we say that $C(n) = n^2/2 - n/2$ is $O(n^2)$
- Common classes of algorithms:

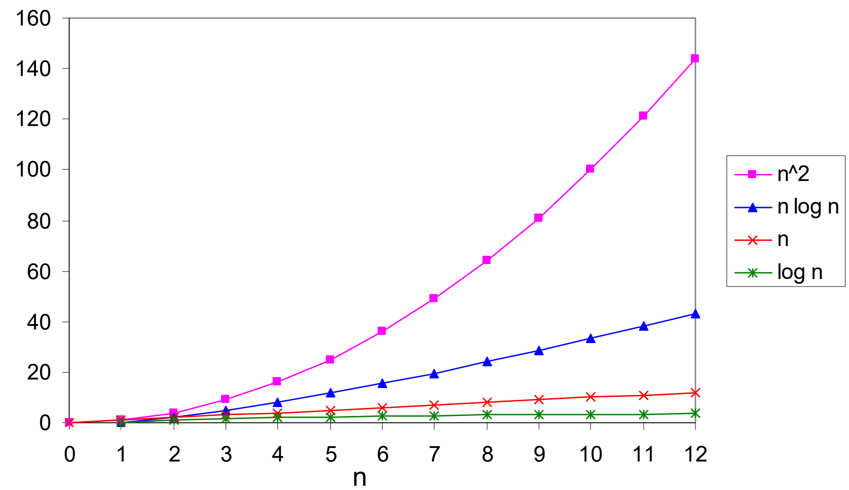
name	example expressions	big-O notation
constant time	1, 7, 10	$O(1)$
logarithmic time	$3\log_{10}n$, $\log_2 n + 5$	$O(\log n)$
linear time	$5n$, $10n - 2\log_2 n$	$O(n)$
$n\log n$ time	$4n\log_2 n$, $n\log_2 n + n$	$O(n \log n)$
quadratic time	$2n^2 + 3n$, $n^2 - 1$	$O(n^2)$
exponential time	2^n, $5e^n + 2n^2$	$O(c^n)$

slower

- For large inputs, efficiency matters more than CPU speed.
  - e.g., an $O(\log n)$ algorithm on a slow machine will outperform an $O(n)$ algorithm on a fast machine

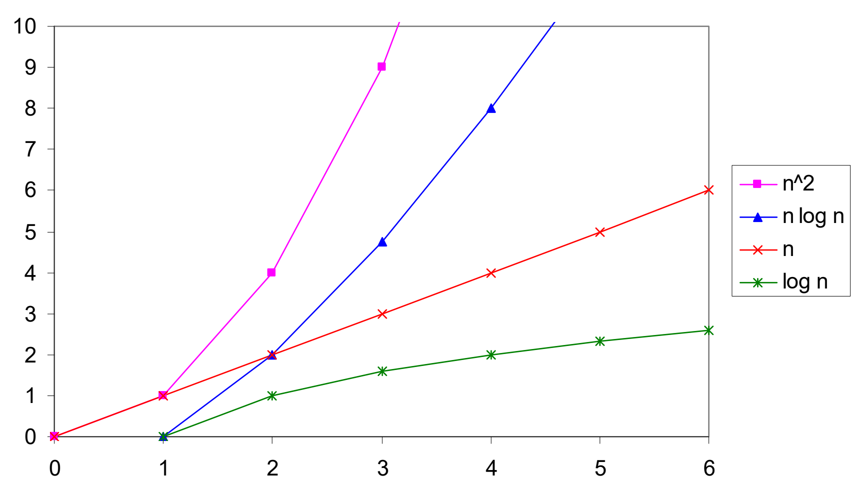
Ordering of Functions

- We can see below that: n^2 grows faster than $n\log_2 n$
  $n\log_2 n$ grows faster than n
  n grows faster than $\log_2 n$



Ordering of Functions (cont.)

- Zooming in, we see that: $n^2 >= n$ for all $n >= 1$
  $n\log_2 n >= n$ for all $n >= 2$
  $n > \log_2 n$ for all $n >= 1$

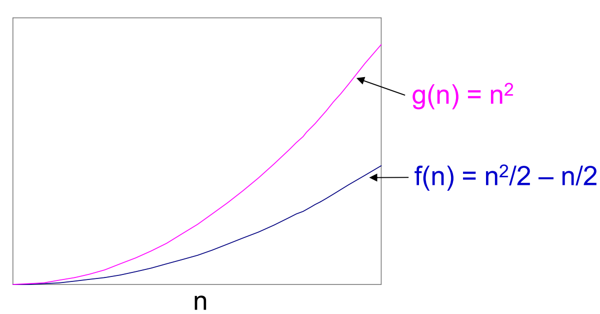


Big-*O* Time Analysis of Selection Sort

- Comparisons: we showed that $C(n) = n^2/2 - n/2$
  - selection sort performs $O(n^2)$ comparisons
- Moves: after each of the `n-1` passes, the algorithm does one swap.
  - `n-1` swaps, 3 moves per swap
  - `M(n) = 3(n-1) = 3n-3`
  - selection sort performs $O(n)$ moves.
- Running time (i.e., total operations): ?

Mathematical Definition of Big-*O* Notation

- $f(n) = O(g(n))$ if there exist positive constants c and n_0 such that $f(n) <= c\,g(n)$ for all $n >= n_0$
- Example: $f(n) = n^2/2 - n/2$ is $O(n^2)$, because
  $n^2/2 - n/2 <= n^2$ for all $n >= 0$.
  $c = 1$ $\qquad$ $n_0 = 0$



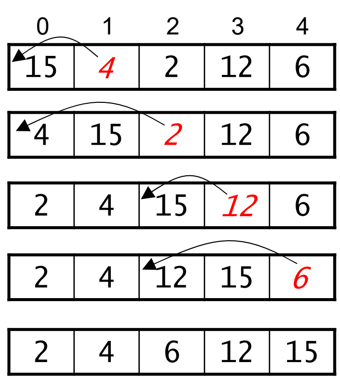
- Big-*O* notation specifies an *upper bound* on a function f(n) as n grows large.

Big-*O* Notation and Tight Bounds

- Strictly speaking, big-O notation provides an upper bound, *not* a tight bound (upper and lower).
- Example:
  - $3n - 3$ is $O(n^2)$ because $3n - 3 <= n^2$ for all $n >= 1$
  - $3n - 3$ is also $O(2^n)$ because $3n - 3 <= 2^n$ for all $n >= 1$
- However, it is common to use big-O notation to characterize a function as closely as possible – as if it specified a tight bound.
  - for our example, we would say that $3n - 3$ is $O(n)$
  - this is how you should use big-O in this class!

Insertion Sort

- Basic idea:
  - going from left to right, “insert” each element into its proper place with respect to the elements to its left
  - “slide over” other elements to make room
- Example:



Comparing Selection and Insertion Strategies

- In selection sort, we start with the *positions* in the array and *select* the correct elements to fill them.
- In insertion sort, we start with the *elements* and determine where to *insert* them in the array.
- Here's an example that illustrates the difference:

0	1	2	3	4	5	6
18	12	15	9	25	2	17

- Sorting by selection:
  - consider position 0: find the element (2) that belongs there
  - consider position 1: find the element (9) that belongs there
  - …
- Sorting by insertion:
  - consider the 12: determine where to insert it
  - consider the 15; determine where to insert it
  - …

Inserting an Element

- When we consider element `i`, elements 0 through `i – 1` are already sorted with respect to each other.

example for `i = 3`:

0	1	2	3	4
6	14	19	9	...

- To insert element `i`:
  - make a copy of element `i`, storing it in the variable `toInsert`:

`toInsert` 9

0	1	2	3
6	14	19	9

  - consider elements `i-1`, `i-2`, …
    - if an element `>` `toInsert`, slide it over to the right
    - stop at the first element `<=` `toInsert`

`toInsert` 9

0	1	2	3
6		14	19

  - copy `toInsert` into the resulting "hole":

0	1	2	3
6	9	14	19

Insertion Sort Example (done together)

description of steps

12	5	2	13	18	4

Implementation of Insertion Sort

```
public class Sort {
  ...
  public static void insertionSort(int[] arr) {
      for (int i = 1; i < arr.length; i++) {
          if (arr[i] < arr[i-1]) {
              int toInsert = arr[i];

              int j = i;
              do {
                  arr[j] = arr[j-1];
                  j = j - 1;
              } while (j > 0  &&  toInsert < arr[j-1]);

              arr[j] = toInsert;
          }
      }
  }
}
```

Time Analysis of Insertion Sort

- The number of operations depends on the contents of the array.
- *best case:* array is sorted
  - each element is only compared to the element to its left
  - we never execute the do-while loop!
  - `C(n)` = ________, `M(n)` = ________, running time = ______

  also true if array is almost sorted
- *worst case:* array is in reverse order
  - each element is compared to *all* of the elements to its left:

    `arr[1]` is compared to 1 element (`arr[0]`)
    `arr[2]` is compared to 2 elements (`arr[0]` and `arr[1]`)
    ...
    `arr[n-1]` is compared to `n-1` elements
  - `C(n)` = 1 + 2 + … + `(n - 1)` = ________
  - similarly, `M(n)` = ________, running time = __________
- *average case:* elements are randomly arranged
  - on average, each element is compared to *half* of the elements to its left
  - still get `C(n)` = `M(n)` = ________, running time = __________

Shell Sort

- Developed by Donald Shell
- Improves on insertion sort
  - takes advantage of the fact that it's fast for almost-sorted arrays
  - eliminates a key disadvantage: an element may need to move many times to get to where it belongs.
- Example: if the largest element starts out at the beginning of the array, it moves one place to the right on *every* insertion!

0	1	2	3	4	5	…	1000
999	42	56	30	18	23	…	11

- Shell sort uses larger moves that allow elements to quickly get close to where they belong in the sorted array.

Sorting Subarrays

- Basic idea:
  - use insertion sort on subarrays that contain elements separated by some increment `incr`
    - increments allow the data items to make larger “jumps”
  - repeat using a decreasing sequence of increments
- Example for an initial increment of 3:

0	1	2	3	4	5	6	7
36	18	10	27	3	20	9	8

  - three subarrays:
    1) elements 0, 3, 6 2) elements 1, 4, 7 3) elements 2 and 5
- Sort the subarrays using insertion sort to get the following:

0	1	2	3	4	5	6	7
9	3	10	27	8	20	36	18

- Next, we complete the process using an increment of 1.

Shell Sort: A Single Pass

- We *don’t* actually consider the subarrays one at a time.
- For each element from position `incr` to the end of the array, we insert the element into its proper place with respect to the elements *from its subarray* that come before it.
- The same example (`incr` = 3):

0	1	2	3	4	5	6	7
36	18	10	*27*	3	20	9	8
27	*18*	10	36	*3*	20	9	8
27	3	*10*	36	18	*20*	9	8
27	3	10	*36*	18	20	*9*	8
9	*3*	10	27	*18*	20	36	*8*
9	3	10	27	8	20	36	18

Inserting an Element in a Subarray

- When we consider element `i`, the other elements in its subarray are already sorted with respect to each other.

example for `i = 6`:
`(incr = 3)`

0	1	2	3	4	5	6	7
27	3	10	*36*	18	20	*9*	8

the other element's in 9's subarray (the 27 and 36)
are already sorted with respect to each other

- To insert element `i`:
  - make a copy of element `i`, storing it in the variable `toInsert`:

`toInsert` | *9* |

0	1	2	3	4	5	6	7
27	3	10	*36*	18	20	*9*	8

  - consider elements `i-incr`, `i-(2*incr)`, `i-(3*incr)`,…
    - if an element `>` `toInsert`, slide it right *within the subarray*
    - stop at the first element `<=` `toInsert`

`toInsert` | *9* |

0	1	2	3	4	5	6	7
	3	10	*27*	18	20	*36*	8

  - copy `toInsert` into the "hole":

0	1	2	3	4	
9	3	10	*27*	18	…

The Sequence of Increments

- Different sequences of decreasing increments can be used.
- Our version uses values that are one less than a power of two.
  - $2^k - 1$ for some k
  - … 63, 31, 15, 7, 3, 1
  - can get to the next lower increment using integer division:

```
incr = incr/2;
```

- Should avoid numbers that are multiples of each other.
  - otherwise, elements that are sorted with respect to each other in one pass are grouped together again in subsequent passes
    - repeat comparisons unnecessarily
    - get fewer of the large jumps that speed up later passes
  - example of a bad sequence: 64, 32, 16, 8, 4, 2, 1
    - what happens if the largest values are all in odd positions?

Implementation of Shell Sort

```
public static void shellSort(int[] arr) {
    int incr = 1;
    while (2 * incr <= arr.length) {
        incr = 2 * incr;
    }
    incr = incr - 1;

    while (incr >= 1) {
        for (int i = incr; i < arr.length; i++) {
            if (arr[i] < arr[i-incr]) {
                int toInsert = arr[i];

                int j = i;
                do {
                    arr[j] = arr[j-incr];
                    j = j - incr;
                } while (j > incr-1 &&
                   toInsert < arr[j-incr]);

                arr[j] = toInsert;
            }
        }
        incr = incr/2;
    }
}
```

(If you replace `incr` *with* `1` *in the for-loop, you get the code for insertion sort.)*

Time Analysis of Shell Sort

- Difficult to analyze precisely
  - typically use experiments to measure its efficiency
- With a bad interval sequence, it's $O(n^2)$ in the worst case.
- With a good interval sequence, it's better than $O(n^2)$.
  - at least $O(n^{1.5})$ in the average and worst case
  - some experiments have shown average-case running times of $O(n^{1.25})$ or even $O(n^{7/6})$
- Significantly better than insertion or selection for large n:

n	n^2	$n^{1.5}$	$n^{1.25}$
10	100	31.6	17.8
100	10,000	1000	316
10,000	100,000,000	1,000,000	100,000
10^6	10^{12}	10^9	3.16×10^7

- We've wrapped insertion sort in another loop and increased its efficiency! The key is in the larger jumps that Shell sort allows.

Practicing Time Analysis

- Consider the following static method:

```
public static int mystery(int n) {
    int x = 0;
    for (int i = 0; i < n; i++) {
        x += i;          // statement 1
        for (int j = 0; j < i; j++) {
            x += j;
        }
    }
    return x;
}
```

- What is the big-O expression for the number of times that statement 1 is executed as a function of the input **n**?

What about now?

- Consider the following static method:

```
public static int mystery(int n) {
    int x = 0;
    for (int i = 0; i < 3*n + 4; i++) {
        x += i;          // statement 1
        for (int j = 0; j < i; j++) {
            x += j;
        }
    }
    return x;
}
```

- What is the big-O expression for the number of times that statement 1 is executed as a function of the input **n**?

Practicing Time Analysis

- Consider the following static method:

```
public static int mystery(int n) {
    int x = 0;
    for (int i = 0; i < n; i++) {
        x += i;          // statement 1
        for (int j = 0; j < i; j++) {
            x += j;      // statement 2
        }
    }
    return x;
}
```

- What is the big-O expression for the number of times that statement 2 is executed as a function of the input n?

value of i	number of times statement 2 is executed

Bubble Sort

- Perform a sequence of passes from left to right
  - each pass swaps adjacent elements if they are out of order
  - larger elements “bubble up” to the end of the array
- At the end of the kth pass:
  - the k rightmost elements are in their final positions
  - we don’t need to consider them in subsequent passes.
- Example:

	0	1	2	3	4
	28	24	37	15	5
after the first pass:	24	28	15	5	*37*
after the second:	24	15	5	*28*	*37*
after the third:	15	5	*24*	*28*	*37*
after the fourth:	5	*15*	*24*	*28*	*37*

Implementation of Bubble Sort

```
public class Sort {
    ...
    public static void bubbleSort(int[] arr) {
        for (int i = arr.length - 1; i > 0; i--) {
            for (int j = 0; j < i; j++) {
                if (arr[j] > arr[j+1]) {
                    swap(arr, j, j+1);
                }
            }
        }
    }
}
```

- Nested loops:
  - the inner loop performs a single pass
  - the outer loop governs:
    - the number of passes (`arr.length - 1`)
    - the ending point of each pass (the current value of `i`)

Time Analysis of Bubble Sort

- Comparisons (n = length of array):
  - they are performed in the inner loop
  - *how many repetitions does each execution of the inner loop perform?*

value of i	number of comparisons
n – 1	n – 1
n – 2	n – 2
…	…
2	2
1	1

1 + 2 + … + n – 1 =

```
public static void bubbleSort(int[] arr) {
    for (int i = arr.length - 1; i > 0; i--) {
        for (int j = 0; j < i; j++) {
            if (arr[j] > arr[j+1]) {
                swap(arr, j, j+1);
            }
        }
    }
}
```

Time Analysis of Bubble Sort

- Comparisons: the kth pass performs n - k comparisons, so we get $C(n) = \sum_{i=1}^{n-1} i = n^2/2 - n/2 = O(n^2)$
- Moves: depends on the contents of the array
  - in the worst case:
    - M(n) =
  - in the best case:
- Running time:
  - $C(n)$ is always $O(n^2)$, $M(n)$ is never worse than $O(n^2)$
  - therefore, the largest term of $C(n) + M(n)$ is $O(n^2)$
- Bubble sort is a quadratic-time or $O(n^2)$ algorithm.
  - can't do much worse than bubble!

Quicksort

- Like bubble sort, quicksort uses an approach based on swapping out-of-order elements, but it's more efficient.
- A recursive, divide-and-conquer algorithm:
  - *divide:* rearrange the elements so that we end up with two subarrays that meet the following criterion:

    each element in left array <= each element in right array

    example:

12	8	14	4	6	13

    →

6	8	4	14	12	13

  - *conquer:* apply quicksort recursively to the subarrays, stopping when a subarray has a single element
  - *combine:* nothing needs to be done, because of the way we formed the subarrays

Partitioning an Array Using a Pivot

- The process that quicksort uses to rearrange the elements is known as *partitioning* the array.
- It uses one of the values in the array as a *pivot,* rearranging the elements to produce two subarrays:
  - left subarray: all values <= pivot
  - right subarray: all values >= pivot

equivalent to the criterion on the previous page.

7	15	4	9	6	18	9	12

partition using a pivot of 9

7	9	4	6	9	18	15	12

all values <= 9 | all values >= 9

- The subarrays will *not* always have the same length.
- This approach to partitioning is one of several variants.

Possible Pivot Values

- First element or last element
  - risky, can lead to terrible worst-case behavior
  - especially poor if the array is almost sorted

4	8	14	12	6	18

pivot = 18

4	8	14	12	6	18

- Middle element (what we will use)
- Randomly chosen element
- Median of three elements
  - left, center, and right elements
  - three randomly selected elements
  - taking the median of three decreases the probability of getting a poor pivot

Partitioning an Array: An Example

arr →

first							last
7	15	4	*9*	6	18	9	12

pivot = 9

- Maintain indices `i` and `j`, starting them "outside" the array:

  `i = first - 1`
  `j = last + 1`

i									j
	7	15	4	9	6	18	9	12	

- *Find* "out of place" elements:
  - increment `i` until `arr[i] >= pivot`
  - decrement `j` until `arr[j] <= pivot`

	i					j	
7	15	4	9	6	18	9	12

- *Swap* `arr[i]` and `arr[j]`:

	i					j	
7	9	4	9	6	18	15	12

Partitioning Example (cont.)

from prev. page:

	i					j	
7	9	4	9	6	18	15	12

- Find:

			i	j			
7	9	4	9	6	18	15	12

- Swap:

			i	j			
7	9	4	6	9	18	15	12

- Find:

			j	i			
7	9	4	6	9	18	15	12

  and now the indices have crossed, so we return `j`.

- Subarrays: left = from `first` to `j`, right = from `j+1` to `last`

first			j	i			last
7	9	4	6	9	18	15	12

Partitioning Example 2

- Start (pivot = 13):

i									j
	24	5	2	*13*	18	4	20	19	

- Find:

i					j		
24	5	2	13	18	4	20	19

- Swap:

i					j		
4	5	2	13	18	24	20	19

- Find:

			i j				
4	5	2	13	18	24	20	19

and now the indices are equal, so we return `j`.

- Subarrays:

			i j				
4	5	2	13	18	24	20	19

Partitioning Example 3 (done together)

- Start (pivot = 5):

i									j
	4	14	7	*5*	2	19	26	6	

- Find:

4	14	7	5	2	19	26	6

Partitioning Example 4

- Start (pivot = 15):

i								j
	8	10	7	*15*	20	9	6	18

- Find:

8	10	7	15	20	9	6	18

partition() Helper Method

```
private static int partition(int[] arr, int first, int last)
{
    int pivot = arr[(first + last)/2];
    int i = first - 1;  // index going left to right
    int j = last + 1;   // index going right to left
    while (true) {
        do {
            i++;
        } while (arr[i] < pivot);
        do {
            j--;
        } while (arr[j] > pivot);
        if (i < j) {
            swap(arr, i, j);
        } else {
            return j;   // arr[j] = end of left array
        }
    }
}
```

	first						last	
…	7	15	4	9	6	18	9	12	…

Implementation of Quicksort

```
public static void quickSort(int[] arr) { // "wrapper" method
    qSort(arr, 0, arr.length - 1);
}

private static void qSort(int[] arr, int first, int last) {
    int split = partition(arr, first, last);

    if (first < split) {  // if left subarray has 2+ values
        qSort(arr, first, split);  // sort it recursively!
    }
    if (last > split + 1) {        // if right has 2+ values
        qSort(arr, split + 1, last);  // sort it!
    }
}   // note: base case is when neither call is made,
    // because both subarrays have only one element!
```

	first			split (j)				last	
…	7	9	4	6	9	18	15	12	…

A Quick Review of Logarithms

- $\log_b n$ = the exponent to which b must be raised to get n
  - $\log_b n = p$ if $b^p = n$
  - examples: $\log_2 8 = 3$ because $2^3 = 8$
    $\log_{10} 10000 = 4$ because $10^4 = 10000$

- Another way of looking at logs:
  - let's say that you repeatedly divide n by b (using integer division)
  - $\log_b n$ is an upper bound on the number of divisions needed to reach 1
  - example: $\log_2 18$ is approx. 4.17
    $18/2 = 9 \quad 9/2 = 4 \quad 4/2 = 2 \quad 2/2 = 1$

A Quick Review of Logs (cont.)

- $O($`log n`$)$ algorithm – one in which the number of operations is proportional to $\log_b n$ for any base `b`
- $\log_b n$ grows much more slowly than `n`

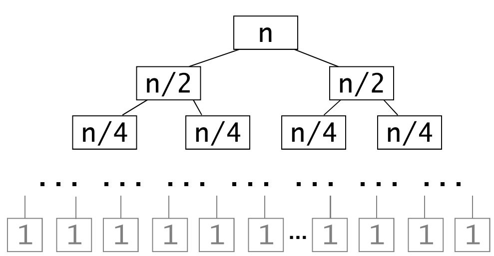
n	$\log_2 n$
2	1
1024 (1K)	10
1024*1024 (1M)	20
1024*1024*1024 (1G)	30

- Thus, for large values of `n`:
  - a $O($`log n`$)$ algorithm is much faster than a $O($`n`$)$ algorithm
    - `log n  <<  n`
  - a $O($`n log n`$)$ algorithm is much faster than a $O(n^2)$ algorithm
    - `n * log n  <<  n * n`
      $n \log n \ll n^2$

it's also faster than a $O(n^{1.5})$ algorithm like Shell sort

Time Analysis of Quicksort

- Partitioning an array requires approx. n comparisons.
  - most elements are compared with the pivot once; a few twice
- *best case:* partitioning always divides the array in half
  - repeated recursive calls give:

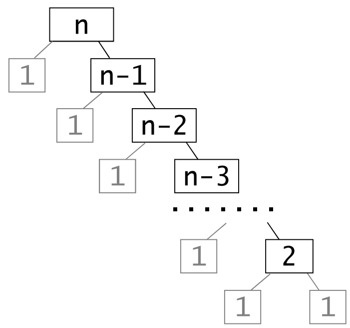


<u>*comparisons*</u>
n
`2*(n/2) = n`
`4*(n/4) = n`
...
0

- at each "row" except the bottom, we perform `n` comparisons
- there are ________ rows that include comparisons
- `C(n) = ?`
- Similarly, `M(n)` and running time are both ____________

Time Analysis of Quicksort (cont.)

- *worst case:* pivot is always the smallest or largest element
  - one subarray has `1` element, the other has `n - 1`
  - repeated recursive calls give:



comparisons
n
n-1
n-2
n-3
...
2

  - $C(n) = \sum_{i=2}^{n} i = O(n^2)$. $M(n)$ and run time are also $O(n^2)$.

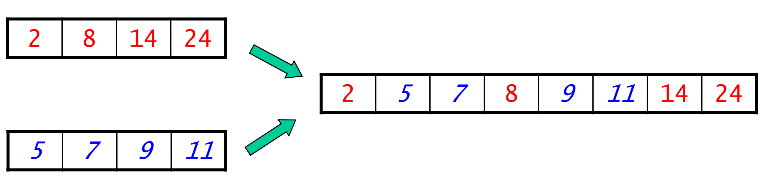
- *average case* is harder to analyze
  - $C(n) > n \log_2 n$, but it's still $O(n \log n)$

Mergesort

- The algorithms we've seen so far have sorted the array in place.
  - use only a small amount of additional memory

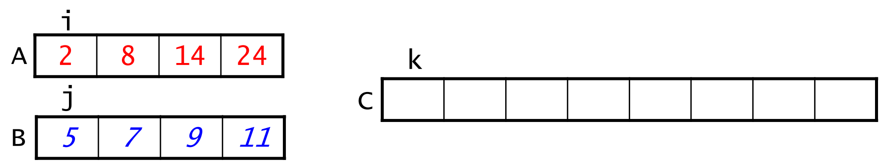
- Mergesort requires an additional temporary array of the same size as the original one.
  - it needs $O(n)$ additional space, where n is the array size

- It is based on the process of *merging* two sorted arrays.
  - example:

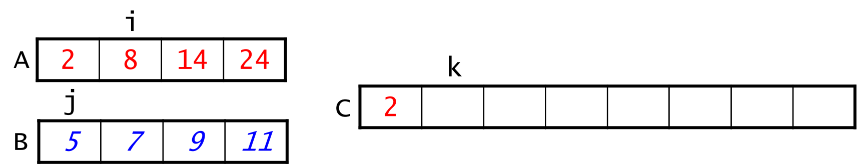


Merging Sorted Arrays

- To merge sorted arrays A and B into an array C, we maintain three indices, which start out on the first elements of the arrays:

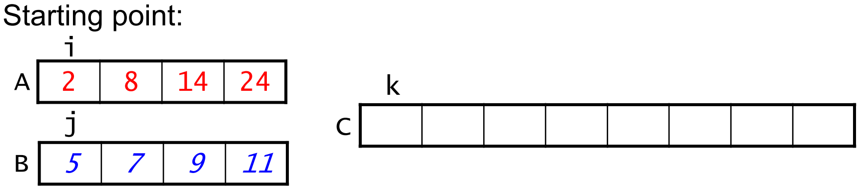


- We repeatedly do the following:
  - compare A[i] and B[j]
  - copy the smaller of the two to C[k]
  - increment the index of the array whose element was copied
  - increment k

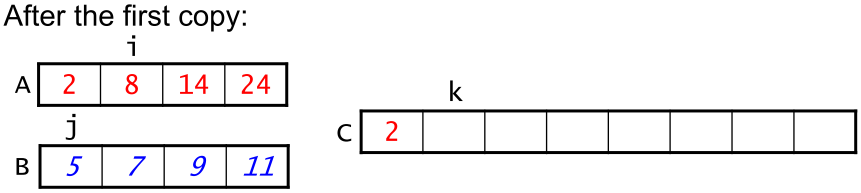


Merging Sorted Arrays (cont.)

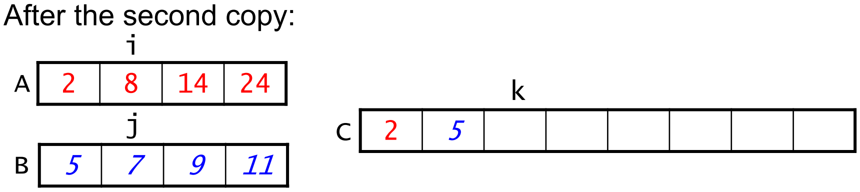
- Starting point:



- After the first copy:

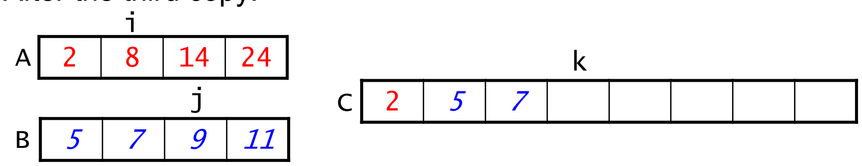


- After the second copy:

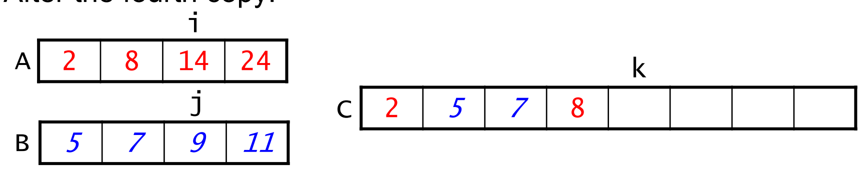


Merging Sorted Arrays (cont.)

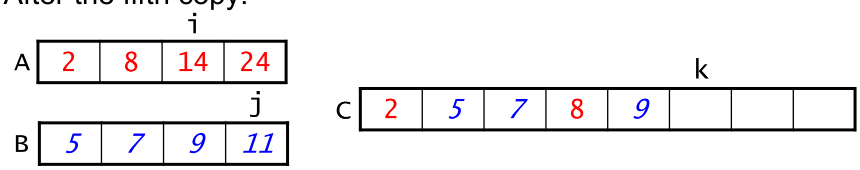
- After the third copy:



- After the fourth copy:

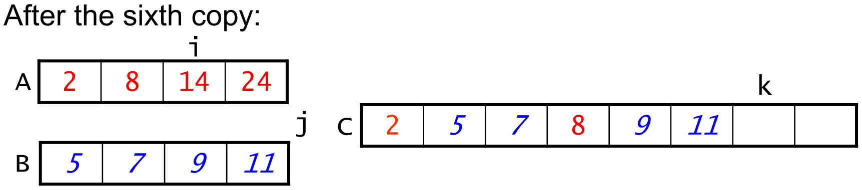


- After the fifth copy:

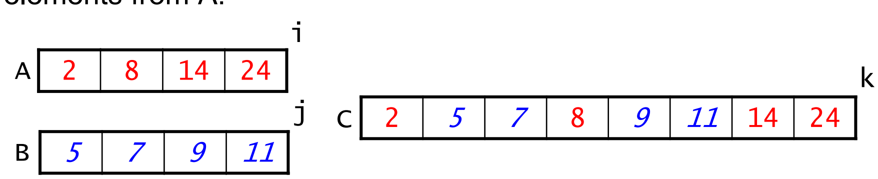


Merging Sorted Arrays (cont.)

- After the sixth copy:

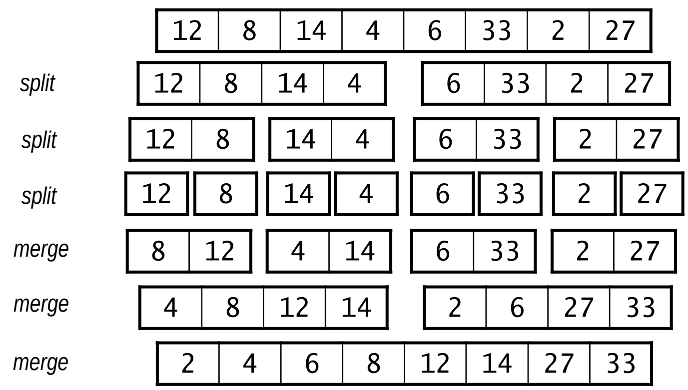


- There's nothing left in B, so we simply copy the remaining elements from A:



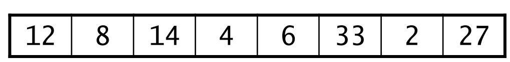
Divide and Conquer

- Like quicksort, mergesort is a divide-and-conquer algorithm.
  - *divide:* split the array in half, forming two subarrays
  - *conquer:* apply mergesort recursively to the subarrays, stopping when a subarray has a single element
  - *combine:* merge the sorted subarrays

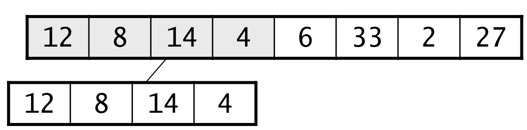


Tracing the Calls to Mergesort

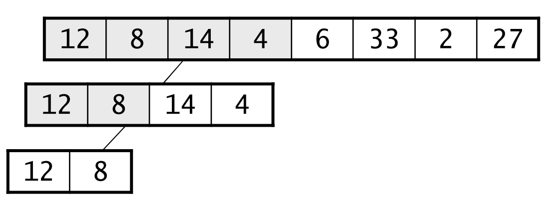
the initial call is made to sort the entire array:



split into two 4-element subarrays, and make a recursive call to sort the left subarray:

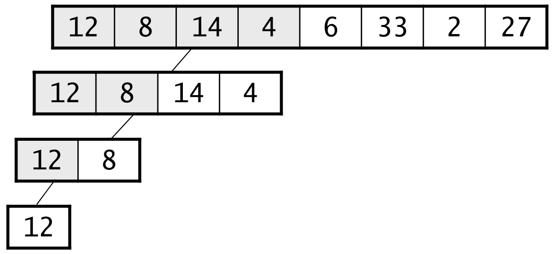


split into two 2-element subarrays, and make a recursive call to sort the left subarray:

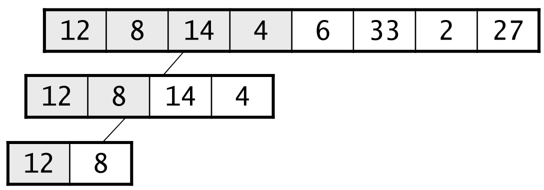


Tracing the Calls to Mergesort

split into two 1-element subarrays, and make a recursive call to sort the left subarray:

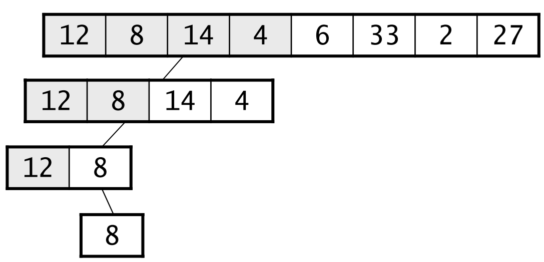


base case, so return to the call for the subarray {12, 8}:

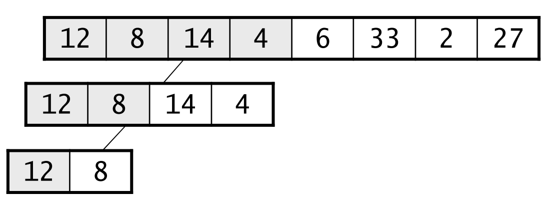


Tracing the Calls to Mergesort

make a recursive call to sort its right subarray:

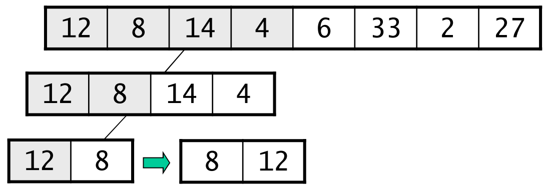


base case, so return to the call for the subarray {12, 8}:

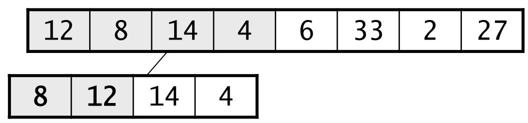


Tracing the Calls to Mergesort

merge the sorted halves of {12, 8}:

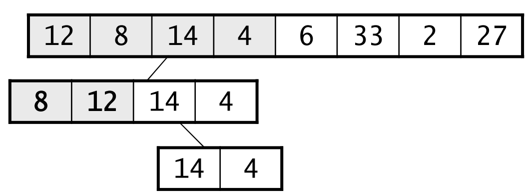


end of the method, so return to the call for the 4-element subarray, which now has a sorted left subarray:

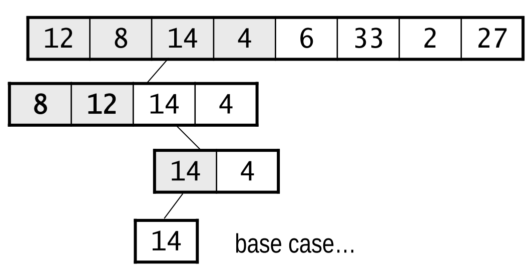


Tracing the Calls to Mergesort

make a recursive call to sort the right subarray of the 4-element subarray

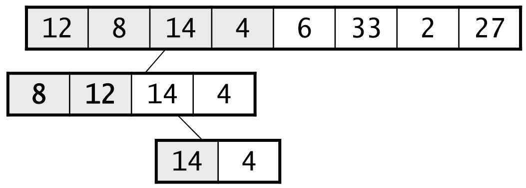


split it into two 1-element subarrays, and make a recursive call to sort the left subarray:

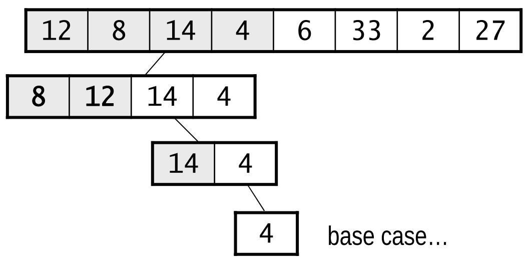


Tracing the Calls to Mergesort

return to the call for the subarray {14, 4}:



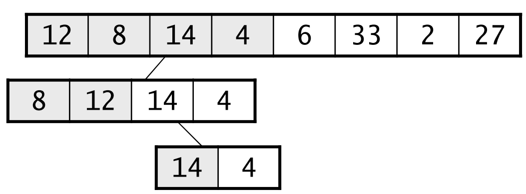
make a recursive call to sort its right subarray:



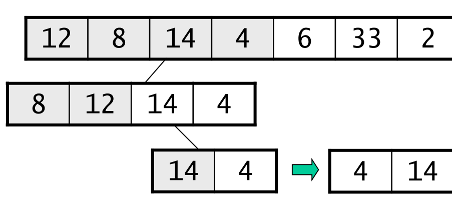
base case…

Tracing the Calls to Mergesort

return to the call for the subarray {14, 4}:



merge the sorted halves of {14, 4}:



Tracing the Calls to Mergesort

end of the method, so return to the call for the 4-element subarray, which now has two sorted 2-element subarrays:

12	8	14	4	6	33	2	27

8	12	4	14

merge the 2-element subarrays:

12	8	14	4	6	33	2	27

8	12	4	14

4	8	12	14

Tracing the Calls to Mergesort

end of the method, so return to the call for the original array, which now has a sorted left subarray:

4	8	12	14	6	33	2	27

perform a similar set of recursive calls to sort the right subarray. here's the result:

4	8	12	14	2	6	27	33

finally, merge the sorted 4-element subarrays to get a fully sorted 8-element array:

4	8	12	14	2	6	27	33

2	4	6	8	12	14	27	33

Implementing Mergesort

- In theory, we could create new arrays for each new pair of subarrays, and merge them back into the array that was split.
- Instead, we'll create a temp. array of the same size as the original.
  - pass it to each call of the recursive mergesort method
  - use it when merging subarrays of the original array:

arr	8	12	4	14	6	33	2	27
temp	4	8	12	14				

  - after each merge, copy the result back into the original array:

arr	4	8	12	14	6	33	2	27
temp	4	8	12	14				

A Method for Merging Subarrays

```
private static void merge(int[] arr, int[] temp,
  int leftStart, int leftEnd, int rightStart, int rightEnd) {
     int i = leftStart;    // index into left subarray
     int j = rightStart;   // index into right subarray
     int k = leftStart;    // index into temp

     while (i <= leftEnd && j <= rightEnd) {
         if (arr[i] < arr[j]) {
             temp[k] = arr[i];
             i++; k++;
         } else {
             temp[k] = arr[j];
             j++; k++;
         }
     }

     while (i <= leftEnd) {
         temp[k] = arr[i];
         i++; k++;
     }
     while (j <= rightEnd) {
         temp[k] = arr[j];
         j++; k++;
     }

     for (i = leftStart; i <= rightEnd; i++) {
         arr[i] = temp[i];
     }
}
```

A Method for Merging Subarrays

```
private static merge(int[] arr, int[] temp,
  int leftStart, int leftEnd, int rightStart, int rightEnd) {
     int i = leftStart;    // index into left subarray
     int j = rightStart;   // index into right subarray
     int k = leftStart;    // index into temp

     while (i <= leftEnd && j <= rightEnd) { // both subarrays still have values
         if (arr[i] < arr[j]) {
             temp[k] = arr[i];
             i++; k++;
         } else {
             temp[k] = arr[j];
             j++; k++;
         }
     }

     ...
}
```

		leftStart			leftEnd	rightStart			rightEnd	
arr:	...	4	8	12	14	2	6	27	33	...
temp:	...									...

Methods for Mergesort

- Here's the key recursive method:

```
private static void mSort(int[] arr, int[] temp, int start, int end){
    if (start >= end) {  // base case: subarray of length 0 or 1
        return;
    } else {
        int middle = (start + end)/2;

        mSort(arr, temp, start, middle);
        mSort(arr, temp, middle + 1, end);

        merge(arr, temp, start, middle, middle + 1, end);
    }
}
```

		start							end	
arr:	...	12	8	14	4	6	33	2	27	...
temp:	...									...

Methods for Mergesort

- Here's the key recursive method:

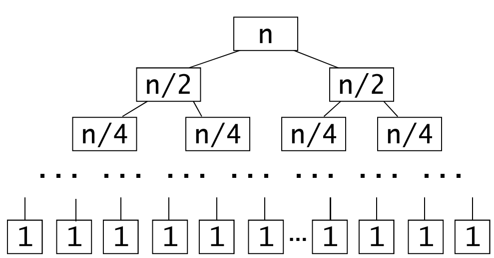
```
private static void mSort(int[] arr, int[] temp, int start, int end){
    if (start >= end) {  // base case: subarray of length 0 or 1
        return;
    } else {
        int middle = (start + end)/2;

        mSort(arr, temp, start, middle);
        mSort(arr, temp, middle + 1, end);

        merge(arr, temp, start, middle, middle + 1, end);
    }
}
```

- We use a "wrapper" method to create the `temp` array, and to make the initial call to the recursive method:

```
public static void mergeSort(int[] arr) {
    int[] temp = new int[arr.length];
    mSort(arr, temp, 0, arr.length - 1);
}
```

Time Analysis of Mergesort

- Merging two halves of an array of size n requires 2n moves. Why?
- Mergesort repeatedly divides the array in half, so we have the following call tree (showing the sizes of the arrays):



moves
2n
2*2*(n/2) = 2n
4*2*(n/4) = 2n
...

  - at all but the last level of the call tree, there are 2n moves
  - how many levels are there?
  - `M(n)` = ?
  - `C(n)` = ?

Summary: Sorting Algorithms

algorithm	best case	avg case	worst case	extra memory
selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
insertion sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Shell sort	$O(n \log n)$	$O(n^{1.5})$	$O(n^{1.5})$	$O(1)$
bubble sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
quicksort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	*best/avg:* $O(\log n)$ *worst:* $O(n)$
mergesort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$

- Insertion sort is best for nearly sorted arrays.
- Mergesort has the best worst-case complexity, but requires O(n) extra memory – and moves to and from the temp. array.
- Quicksort is comparable to mergesort in the best/average case.
  - efficiency is also $O(n \log n)$, but less memory and fewer moves
  - its extra memory is from…
  - with a reasonable pivot choice, its worst case is seldom seen

Comparison-Based vs. Distributive Sorting

- All of the sorting algorithms we've considered have been *comparison-based:*
  - treat the keys as wholes (comparing them)
  - don't "take them apart" in any way
  - all that matters is the relative order of the keys, not their actual values
- No comparison-based sorting algorithm can do better than $O(n \log_2 n)$ on an array of length n.
  - $O(n \log_2 n)$ is a *lower bound* for such algorithms.
- *Distributive* sorting algorithms do more than compare keys; they perform calculations on the values of individual keys.
- Moving beyond comparisons allows us to overcome the lower bound.
  - tradeoff: use more memory.

Distributive Sorting Example: Radix Sort

- Relies on the representation of the data as a sequence of **m** quantities with **k** possible values.
- Examples:

Examples	**m**	**k**
integer in range 0 ... 999	3	10
string of 15 upper-case letters	15	26
32-bit integer	32	2 (in binary)
	4	256 (as bytes)

- Strategy: Distribute according to the last element in the sequence, then concatenate the results:

  33 41 12 24 31 14 13 42 34

  get: 41 31 | 12 42 | 33 13 | 24 14 34

- Repeat, moving back one digit each time:

  get: | | |

Analysis of Radix Sort

- Recall that we treat the values as a sequence of `m` quantities with `k` possible values.
- Number of operations is O(`n*m`) for an array with `n` elements
  - better than O(`n log n`) when `m < log n`
- Memory usage increases as `k` increases.
  - `k` tends to increase as `m` decreases
  - tradeoff: increased speed requires increased memory usage

Big-*O* Notation Revisited

- We've seen that we can group functions into classes by focusing on the fastest-growing term in the expression for the number of operations that they perform.
  - e.g., an algorithm that performs $n^2/2 - n/2$ operations is a $O(n^2)$-time or quadratic-time algorithm
- Common classes of algorithms:

slower

name	example expressions	big-O notation
constant time	1, 7, 10	$O(1)$
logarithmic time	$3\log_{10}n$, $\log_2 n + 5$	$O(\log n)$
linear time	$5n$, $10n - 2\log_2 n$	$O(n)$
$n\log n$ time	$4n\log_2 n$, $n\log_2 n + n$	$O(n\log n)$
quadratic time	$2n^2 + 3n$, $n^2 - 1$	$O(n^2)$
cubic time	$n^2 + 3n^3$, $5n^3 - 5$	$O(n^3)$
exponential time	2^n, $5e^n + 2n^2$	$O(c^n)$
factorial time	$3n!$, $5n + n!$	$O(n!)$

How Does the Number of Operations Scale?

- Let's say that we have a problem size of 1000, and we measure the number of operations performed by a given algorithm.
- If we double the problem size to 2000, how would the number of operations performed by an algorithm increase if it is:
  - $O(n)$-time
  - $O(n^2)$-time
  - $O(n^3)$-time
  - $O(\log_2 n)$-time
  - $O(2^n)$-time

How Does the Actual Running Time Scale?

- How much time is required to solve a problem of size n?
  - assume that each operation requires 1 μsec (1×10^{-6} sec)

time function	problem size (n) 10	20	30	40	50	60
n	.00001 s	.00002 s	.00003 s	.00004 s	.00005 s	.00006 s
n^2	.0001 s	.0004 s	.0009 s	.0016 s	.0025 s	.0036 s
n^5	.1 s	3.2 s	24.3 s	1.7 min	5.2 min	13.0 min
2^n	.001 s	1.0 s	17.9 min	12.7 days	35.7 yrs	36,600 yrs

- sample computations:
  - when n = 10, an n^2 algorithm performs 10^2 operations.
    $10^2 * (1 \times 10^{-6}$ sec) = .0001 sec
  - when n = 30, a 2^n algorithm performs 2^{30} operations.
    $2^{30} * (1 \times 10^{-6}$ sec) = 1073 sec = 17.9 min

What's the Largest Problem That Can Be Solved?

- What's the largest problem size n that can be solved in a given time T? (again assume 1 μsec per operation)

time function	time available (T) 1 min	1 hour	1 week	1 year
n	60,000,000	3.6×10^9	6.0×10^{11}	3.1×10^{13}
n^2	7745	60,000	777,688	5,615,692
n^5	35	81	227	500
2^n	25	31	39	44

- sample computations:
  - 1 hour = 3600 sec
    that's enough time for $3600/(1 \times 10^{-6}) = 3.6 \times 10^9$ operations
    - n^2 algorithm:
      $n^2 = 3.6 \times 10^9 \rightarrow n = (3.6 \times 10^9)^{1/2} = 60{,}000$
    - 2^n algorithm:
      $2^n = 3.6 \times 10^9 \rightarrow n = \log_2(3.6 \times 10^9) \sim= 31$