Sorting and Algorithm Analysis

Computer Science E-22
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

```java
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?
  ```java
  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:

  
<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>6</td>
<td>2</td>
<td>12</td>
<td>4</td>
</tr>
</tbody>
</table>

  
<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>6</td>
<td>15</td>
<td>12</td>
<td>4</td>
</tr>
</tbody>
</table>

  
<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>4</td>
<td>15</td>
<td>12</td>
<td>6</td>
</tr>
</tbody>
</table>

  
  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 \( i, i+1, i+2, \ldots, \text{arr.length} - 1 \), and keep track of \( \text{indexMin} \), the index of the smallest element seen thus far

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

  - when we finish this pass, \( \text{indexMin} \) is the index of the element that belongs in position \( i \).
  - swap \( \text{arr}[i] \) and \( \text{arr}[\text{indexMin}] \):

    
    | 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:
  
  ```java
  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:
  
  ```java
  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 \( \text{indexSmallest} \)
  - on 2nd pass, it performs ____ comparisons ...
  - on the \((n-1)\)st pass, it performs 1 comparison

• Adding them up: \( C(n) = 1 + 2 + ... + (n - 2) + (n - 1) \)


Counting Comparisons by Selection Sort (cont.)

• The resulting formula for \( C(n) \) is the sum of an arithmetic sequence:
  \[
  C(n) = 1 + 2 + ... + (n - 2) + (n - 1) = \sum_{i=1}^{n-1} i
  \]

• 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:
  \[
  C(n) = \sum_{i=1}^{n-1} i
  = \frac{(n - 1)((n - 1) + 1)}{2}
  = \frac{(n - 1)n}{2}
  \]
  \[
  C(n) = \frac{n^2}{2} - \frac{n}{2}
  \]
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:
    
    \[
    \begin{array}{c|cccc}
      n & n^2/2 & n/2 & n^2/2 - n/2 \\
      \hline
      10 & 50 & 5 & 45 \\
      100 & 5000 & 50 & 4950 \\
      10000 & 50,000,000 & 5000 & 49,995,000 \\
    \end{array}
    \]

- 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:

  \[
  \begin{array}{l|lll}
    \text{name} & \text{example expressions} & \text{big-O notation} \\
    \hline
    \text{constant time} & 1, 7, 10 & O(1) \\
    \text{logarithmic time} & 3\log_{10}n, \log_2n + 5 & O(\log n) \\
    \text{linear time} & 5n, 10n - 2\log_2n & O(n) \\
    \text{nlogn time} & 4n\log_2n, n\log_2n + n & O(n\log n) \\
    \text{quadratic time} & 2n^2 + 3n, n^2 - 1 & O(n^2) \\
    \text{exponential time} & 2^n, 5e^n + 2n^2 & O(c^n) \\
  \end{array}
  \]

- 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:
  - $n^2 \geq n$ for all $n \geq 1$
  - $n \log_2 n \geq n$ for all $n \geq 2$
  - $n > \log_2 n$ for all $n \geq 1$
Big-O Time Analysis of Selection Sort

- **Comparisons:** we showed that $C(n) = \frac{n^2}{2} - \frac{n}{2}$
  - 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 $f(n) \leq cg(n)$ for all $n \geq n_0$

- Example: $f(n) = \frac{n^2}{2} - \frac{n}{2}$ is $O(n^2)$, because $\frac{n^2}{2} - \frac{n}{2} \leq n^2$ for all $n \geq 0$
  - $c = 1$
  - $n_0 = 0$

- 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 $3n - 3 \leq n^2$ for all $n \geq 1$
  - $3n - 3$ is also $O(2^n)$ because $3n - 3 \leq 2^n$ for all $n \geq 1$

- 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:

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

4 15 2 12 6

2 4 15 12 6

2 4 12 15 6

2 4 6 12 15
```
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.

```
element: 6 14 19 9 ...
```

example for *i* = 3:

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

```
toInsert: 9 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 6 14 19
```

• copy `toInsert` into the resulting “hole”:

```
6 9 14 19
```
Insertion Sort Example (done together)

*description of steps*

12 5 2 13 18 4

Implementation of Insertion Sort

```java
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!
  - \( C(n) = \_, M(n) = \_, \text{running time} = \_ \)
- **worst case:** array is in reverse order
  - each element is compared to all of the elements to its left:
    - \( \text{arr[1]} \) is compared to 1 element (\( \text{arr[0]} \))
    - \( \text{arr[2]} \) is compared to 2 elements (\( \text{arr[0]} \) and \( \text{arr[1]} \))
    - \( \ldots \)
    - \( \text{arr[n-1]} \) is compared to \( n-1 \) elements
  - \( C(n) = 1 + 2 + \ldots + (n - 1) = \_ \)
  - 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 \( \text{incr} \)
    - 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 \( \text{incr} \) 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 (\( \text{incr} = 3 \)):

  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 \( i = 6 \):
  \[
  \begin{array}{cccccccc}
  0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
  27 & 3 & 10 & 36 & 18 & 20 & 9 & 8 \\
  \end{array}
  \]
  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 \( \text{toInsert} \):
    
    \[
    \begin{array}{cccccccc}
    0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
    \text{toInsert} & 9 & 27 & 3 & 10 & 36 & 18 & 20 & 9 & 8 \\
    \end{array}
    \]
  - consider elements \( i - \text{incr}, i - (2 \times \text{incr}), i - (3 \times \text{incr}), \ldots \)
    - if an element > \( \text{toInsert} \), slide it right within the subarray
    - stop at the first element \( \leq \) \( \text{toInsert} \)
  - copy \( \text{toInsert} \) into the “hole”:
    
    \[
    \begin{array}{cccccccc}
    0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
    9 & 3 & 10 & 27 & 18 & 20 & 36 & 8 \\
    \end{array}
    \]

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:
    \[
    \text{incr} = \text{incr}/2;
    \]
- 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

```java
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 $n$:

<table>
<thead>
<tr>
<th>$n$</th>
<th>$n^2$</th>
<th>$n^{1.5}$</th>
<th>$n^{1.25}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>100</td>
<td>31.6</td>
<td>17.8</td>
</tr>
<tr>
<td>100</td>
<td>10,000</td>
<td>1000</td>
<td>316</td>
</tr>
<tr>
<td>10,000</td>
<td>100,000,000</td>
<td>1,000,000</td>
<td>100,000</td>
</tr>
<tr>
<td>$10^6$</td>
<td>$10^{12}$</td>
<td>$10^9$</td>
<td>$3.16 \times 10^7$</td>
</tr>
</tbody>
</table>

- 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:
  
  ```java
  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:
  
  ```java
  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:

```java
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 \( n \)?

<table>
<thead>
<tr>
<th>value of ( i )</th>
<th>number of times statement 2 is executed</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>6</td>
</tr>
</tbody>
</table>

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 \( k \)th pass:
  • the \( k \) rightmost elements are in their final positions
  • we don’t need to consider them in subsequent passes.

• Example:

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>28</td>
<td>24</td>
<td>37</td>
<td>15</td>
<td>5</td>
</tr>
<tr>
<td>after the first pass:</td>
<td>24</td>
<td>28</td>
<td>15</td>
<td>5</td>
<td>37</td>
</tr>
<tr>
<td>after the second:</td>
<td>24</td>
<td>15</td>
<td>5</td>
<td>28</td>
<td>37</td>
</tr>
<tr>
<td>after the third:</td>
<td>15</td>
<td>5</td>
<td>28</td>
<td>24</td>
<td>37</td>
</tr>
<tr>
<td>after the fourth:</td>
<td>5</td>
<td>15</td>
<td>24</td>
<td>28</td>
<td>37</td>
</tr>
</tbody>
</table>
Implementation of Bubble Sort

```java
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?

<table>
<thead>
<tr>
<th>value of i</th>
<th>number of comparisons</th>
</tr>
</thead>
<tbody>
<tr>
<td>n – 1</td>
<td>n – 1</td>
</tr>
<tr>
<td>n – 2</td>
<td>n – 2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

\[
1 + 2 + \ldots + n - 1 = \]
Time Analysis of Bubble Sort

- **Comparisons**: the kth pass performs n – k comparisons, so we get \( C(n) = \sum_{i=1}^{n-k} i = \frac{n^2}{2} - \frac{n}{2} = O(n^2) \)
- **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:
  
  \[
  \begin{array}{cccccc}
  12 & 8 & 14 & 4 & 6 & 13 \\
  \end{array}
  \]
  \[
  \begin{array}{cccccc}
  6 & 8 & 4 & 14 & 12 & 13 \\
  \end{array}
  \]
  
  - **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

  \[
  \begin{array}{cccccccc}
  7 & 15 & 4 & 9 & 6 & 18 & 9 & 12 \\
  \end{array}
  \]

  partition using a pivot of 9

  \[
  \begin{array}{cccccccc}
  7 & 9 & 4 & 6 & 9 & 18 & 15 & 12 \\
  \end{array}
  \]

  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

  \[
  \begin{array}{cccccc}
  4 & 8 & 14 & 12 & 6 & 18 \\
  \end{array}
  \]

  \[
  \begin{array}{cccccc}
  4 & 8 & 14 & 12 & 6 & 18 \\
  \end{array}
  \]

  pivot = 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

- Maintain indices \( i \) and \( j \), starting them “outside” the array:
  \( i = \text{first} - 1 \)
  \( j = \text{last} + 1 \)

- Find “out of place” elements:
  - increment \( i \) until \( \text{arr}[i] \geq \text{pivot} \)
  - decrement \( j \) until \( \text{arr}[j] \leq \text{pivot} \)

- Swap \( \text{arr}[i] \) and \( \text{arr}[j] \):

```
7 15 4 9 6 18 9 12
```

\( i \) and now the indices have crossed, so we return \( j \).

- Subarrays: left = from \( \text{first} \) to \( j \), right = from \( j+1 \) to \( \text{last} \)
Partitioning Example 2
• Start (pivot = 13):
  \[
  \begin{array}{cccccccc}
  24 & 5 & 2 & \text{13} & 18 & 4 & 20 & 19 \\
  \end{array}
  \]
• Find:
  \[
  \begin{array}{cccccccc}
  24 & 5 & 2 & \text{13} & 18 & 4 & 20 & 19 \\
  \end{array}
  \]
• Swap:
  \[
  \begin{array}{cccccccc}
  4 & 5 & 2 & \text{13} & 18 & 24 & 20 & 19 \\
  \end{array}
  \]
• Find:
  \[
  \begin{array}{cccccccc}
  4 & 5 & 2 & \text{13} & 18 & 24 & 20 & 19 \\
  \end{array}
  \]
and now the indices are equal, so we return \( j \).
• Subarrays:
  \[
  \begin{array}{cccccccc}
  4 & 5 & 2 & \text{13} & 18 & 24 & 20 & 19 \\
  \end{array}
  \]

Partitioning Example 3 (done together)
• Start (pivot = 5):
  \[
  \begin{array}{cccccccc}
  4 & 14 & 7 & \text{5} & 2 & 19 & 26 & 6 \\
  \end{array}
  \]
• Find:
  \[
  \begin{array}{cccccccc}
  4 & 14 & 7 & \text{5} & 2 & 19 & 26 & 6 \\
  \end{array}
  \]
Partitioning Example 4

- Start (pivot = 15):

```
8 10 7 15 20 9 6 18
```

- Find:

```
8 10 7 15 20 9 6 18
```

**partition() Helper Method**

```java
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
        }
    }
}
```

```java
... 7 15 4 9 6 18 9 12 ...
```
Implementation of Quicksort

```java
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!
```

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 $n$

<table>
<thead>
<tr>
<th>$n$</th>
<th>$\log_2 n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>1024 (1K)</td>
<td>10</td>
</tr>
<tr>
<td>1024*1024 (1M)</td>
<td>20</td>
</tr>
<tr>
<td>1024<em>1024</em>1024 (1G)</td>
<td>30</td>
</tr>
</tbody>
</table>

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

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

Time Analysis of Quicksort

- Partitioning an array of length $n$ 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:
    - 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:
    \[
    \sum_{i=2}^{n} i = O(n^2).
    \]
    \[M(n)\] 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:

\[
\begin{array}{c}
| i | j | k |
\end{array}
\]

\[
\begin{array}{c|c|c|c|c|c}
A & 2 & 8 & 14 & 24 & \\
B & 5 & 7 & 9 & 11 & \\
C & \_ & \_ & \_ & \_ & \\
\end{array}
\]

- 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

\[
\begin{array}{c|c|c|c|c|c}
A & 2 & 8 & 14 & 24 & \\
B & 5 & 7 & 9 & 11 & \\
C & 2 & \_ & \_ & \_ & \\
\end{array}
\]

Merging Sorted Arrays (cont.)

- Starting point:

\[
\begin{array}{c|c|c|c|c|c}
A & 2 & 8 & 14 & 24 & \\
B & 5 & 7 & 9 & 11 & \\
C & \_ & \_ & \_ & \_ & \\
\end{array}
\]

- After the first copy:

\[
\begin{array}{c|c|c|c|c|c}
A & 2 & 8 & 14 & 24 & \\
B & 5 & 7 & 9 & 11 & \\
C & 2 & \_ & \_ & \_ & \\
\end{array}
\]

- After the second copy:

\[
\begin{array}{c|c|c|c|c|c}
A & 2 & 8 & 14 & 24 & \\
B & 5 & 7 & 9 & 11 & \\
C & 2 & 5 & \_ & \_ & \\
\end{array}
\]
Merging Sorted Arrays (cont.)

- After the third copy:

  A: 2 8 14 24  
  B: 5 7 9 11  
  C: 2 5 7  

- After the fourth copy:

  A: 2 8 14 24  
  B: 5 7 9 11  
  C: 2 5 7 8  

- After the fifth copy:

  A: 2 8 14 24  
  B: 5 7 9 11  
  C: 2 5 7 8 9  

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

  A: 2 8 14 24  
  B: 5 7 9 11  
  C: 2 5 7 8 9 11 14 24
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

```
12 8 14 4 6 33 2 27

split
split
split
merge
merge
merge
```

Tracing the Calls to Mergesort

the initial call is made to sort the entire array:

```
12 8 14 4 6 33 2 27
```

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

```
12 8 14 4 6 33 2 27

12 8 14 4
```

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

```
12 8 14 4 6 33 2 27

12 8 14 4

12 8
```
Tracing the Calls to Mergesort

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

<table>
<thead>
<tr>
<th>12</th>
<th>8</th>
<th>14</th>
<th>4</th>
<th>6</th>
<th>33</th>
<th>2</th>
<th>27</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>8</td>
<td>14</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>12</th>
<th>8</th>
<th>14</th>
<th>4</th>
<th>6</th>
<th>33</th>
<th>2</th>
<th>27</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>8</td>
<td>14</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Tracing the Calls to Mergesort

make a recursive call to sort its right subarray:

<table>
<thead>
<tr>
<th>12</th>
<th>8</th>
<th>14</th>
<th>4</th>
<th>6</th>
<th>33</th>
<th>2</th>
<th>27</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>8</td>
<td>14</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>12</th>
<th>8</th>
<th>14</th>
<th>4</th>
<th>6</th>
<th>33</th>
<th>2</th>
<th>27</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>8</td>
<td>14</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Tracing the Calls to Mergesort

merge the sorted halves of \{12, 8\}:

```
12  8  14  4  6  33  2  27
```
```
12  8  14  4
```
```
12  8  ￢  8  12
```

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

```
12  8  14  4  6  33  2  27
```
```
8  12  14  4
```

Tracing the Calls to Mergesort

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

```
12  8  14  4  6  33  2  27
```
```
8  12  14  4
```
```
14  4
```

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

```
12  8  14  4  6  33  2  27
```
```
8  12  14  4
```
```
14  4
```
```
14
```

base case…
Tracing the Calls to Mergesort

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

\[
\begin{array}{cccccccc}
12 & 8 & 14 & 4 & 6 & 33 & 2 & 27 \\
8 & 12 & 14 & 4 \\
8 & 12 & 14 & 4 \\
14 & 4 \\
\end{array}
\]

make a recursive call to sort its right subarray:

\[
\begin{array}{cccccccc}
12 & 8 & 14 & 4 & 6 & 33 & 2 & 27 \\
8 & 12 & 14 & 4 \\
14 & 4 \\
4 \\
\end{array}
\]

base case…

Tracing the Calls to Mergesort

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

\[
\begin{array}{cccccccc}
12 & 8 & 14 & 4 & 6 & 33 & 2 & 27 \\
8 & 12 & 14 & 4 \\
8 & 12 & 14 & 4 \\
14 & 4 \\
\end{array}
\]

merge the sorted halves of \{14, 4\}:

\[
\begin{array}{cccccccc}
12 & 8 & 14 & 4 & 6 & 33 & 2 & 27 \\
8 & 12 & 14 & 4 \\
14 & 4 \\
4 & 14 \\
\end{array}
\]
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
```

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

```java
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

```java
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) { // both subarrays still have values
        if (arr[i] < arr[j]) {
            temp[k] = arr[i];
            i++; k++;
        } else {
            temp[k] = arr[j];
            j++; k++;
        }
    }
}
```

Methods for Mergesort

- Here's the key recursive method:

```java
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);
    }
}
```
Methods for Mergesort

• Here’s the key recursive method:

```java
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:

```java
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):

```
      n
    /   \
   /     \
  n/2   n/2
    /     /   \
  n/4  n/4  n/4  n/4
    /     /     /     /     /     \
  1  1  1  1  1  1  1  1  1  1  1  1
```

- 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

<table>
<thead>
<tr>
<th>algorithm</th>
<th>best case</th>
<th>avg case</th>
<th>worst case</th>
<th>extra memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>selection sort</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>insertion sort</td>
<td>$O(n)$</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Shell sort</td>
<td>$O(n \log n)$</td>
<td>$O(n^{1.5})$</td>
<td>$O(n^{1.5})$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>bubble sort</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>quicksort</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
<td>$O(n^2)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>mergesort</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
<td>$O(n)$</td>
</tr>
</tbody>
</table>

- 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 values being sorted as wholes (comparing them)
  - don’t “take them apart” in any way
  - all that matters is the relative order of the 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 values; they perform calculations on the values being sorted.

- Moving beyond comparisons allows us to overcome the lower bound.
  - tradeoff: use more memory.
Distributive Sorting Example: Radix Sort

- Breaks each value into a sequence of \( m \) components, each of which has \( k \) possible values.
- Examples:
  - integer in range 0 ... 999
  - string of 15 upper-case letters
  - 32-bit integer
    - 4 bytes
    - 256 (as bytes)
- Strategy: Distribute the values into "bins" according to their last component, then concatenate the results:
  - \( 33, 41, 12, 24, 31, 14, 13, 42, 34 \)
  - get: \( 41, 31 \mid 12, 42 \mid 33, 13 \mid 24, 14, 34 \)
- Repeat, moving back one component each time:
  - get: \( \mid \mid \mid \)

Analysis of Radix Sort

- \( m \) = number of components
- \( k \) = number of possible values for each component
- \( n \) = length of the array
- Time efficiency: \( O(m \cdot n) \)
  - perform \( m \) distributions, each of which processes all \( n \) values
  - \( O(m \cdot n) < O(n \log n) \) when \( m < \log n \)
    - so we want \( m \) to be small
- However, there is a tradeoff:
  - as \( m \) decreases, \( k \) increases
    - fewer components \( \Rightarrow \) more possible values per component
  - as \( k \) increases, so does memory usage
    - need more bins for the results of each distribution
  - 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 \( \frac{n^2}{2} - \frac{n}{2} \) operations is a \( O(n^2) \)-time or quadratic-time algorithm

- Common classes of algorithms:

<table>
<thead>
<tr>
<th>name</th>
<th>example expressions</th>
<th>big-O notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>constant time</td>
<td>1, 7, 10</td>
<td>( O(1) )</td>
</tr>
<tr>
<td>logarithmic time</td>
<td>3log_{10}n, log_2n + 5</td>
<td>( O(\log n) )</td>
</tr>
<tr>
<td>linear time</td>
<td>5n, 10n – 2log_2n</td>
<td>( O(n) )</td>
</tr>
<tr>
<td>nlogn time</td>
<td>4nlog_2n, nlog_2n + n</td>
<td>( O(n\log n) )</td>
</tr>
<tr>
<td>quadratic time</td>
<td>2n^2 + 3n, n^2 – 1</td>
<td>( O(n^2) )</td>
</tr>
<tr>
<td>cubic time</td>
<td>n^2 + 3n^3, 5n^3 – 5</td>
<td>( O(n^3) )</td>
</tr>
<tr>
<td>exponential time</td>
<td>2^n, 5e^n + 2n^2</td>
<td>( O(c^n) )</td>
</tr>
<tr>
<td>factorial time</td>
<td>3n!, 5n + n!</td>
<td>( O(n!) )</td>
</tr>
</tbody>
</table>

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 x 10^{-6} sec)

<table>
<thead>
<tr>
<th>Problem Size (n)</th>
<th>Time Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>.00001 s</td>
</tr>
<tr>
<td>20</td>
<td>.00002 s</td>
</tr>
<tr>
<td>30</td>
<td>.00003 s</td>
</tr>
<tr>
<td>40</td>
<td>.00004 s</td>
</tr>
<tr>
<td>50</td>
<td>.00005 s</td>
</tr>
<tr>
<td>60</td>
<td>.00006 s</td>
</tr>
</tbody>
</table>

- sample computations:
  - when n = 10, an n² algorithm performs 10² operations.
    \[10^2 \times (1 \times 10^{-6} \text{ sec}) = .0001 \text{ sec}\]
  - when n = 30, a 2ⁿ algorithm performs 2^{30} operations.
    \[2^{30} \times (1 \times 10^{-6} \text{ sec}) = 1073 \text{ sec} = 17.9 \text{ min}\]

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)

<table>
<thead>
<tr>
<th>Time Available (T)</th>
<th>Time Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 min</td>
<td>60,000,000</td>
</tr>
<tr>
<td>1 hour</td>
<td>3.6 x 10⁹</td>
</tr>
<tr>
<td>1 week</td>
<td>6.0 x 10¹¹</td>
</tr>
<tr>
<td>1 year</td>
<td>3.1 x 10¹³</td>
</tr>
</tbody>
</table>

- sample computations:
  - 1 hour = 3600 sec
    - that's enough time for 3600/(1 x 10^{-6}) = 3.6 x 10⁶ operations
  - n² algorithm:
    \[n^2 = 3.6 \times 10^9 \Rightarrow n = (3.6 \times 10^9)^{1/2} = 60,000\]
  - 2ⁿ algorithm:
    \[2^n = 3.6 \times 10^9 \Rightarrow n = \log_2(3.6 \times 10^9) \approx 31\]