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

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

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

• Trace through the following lines to see the problem:

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

```
stack
   arr

heap
   15  7  ...
```

A Correct Swap Method

• This method works:

```java
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:

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

```
<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>2</td>
<td>6</td>
<td>15</td>
<td>12</td>
<td>4</td>
</tr>
</tbody>
</table>
```

```
<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>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:

```
<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2</td>
<td>4</td>
<td>7</td>
<td>21</td>
<td>25</td>
<td>10</td>
<td>17</td>
</tr>
</tbody>
</table>
```

- 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

```
<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2</td>
<td>4</td>
<td>7</td>
<td>21</td>
<td>25</td>
<td>10</td>
<td>17</td>
</tr>
</tbody>
</table>
```

- when we finish this pass, indexMin is the index of the element that belongs in position i.

- swap arr[i] and arr[indexMin]:

```
<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2</td>
<td>4</td>
<td>7</td>
<td>10</td>
<td>25</td>
<td>21</td>
<td>17</td>
</tr>
</tbody>
</table>
```
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 algorithms, we'll focus on two types of operations: comparisons and moves

- The number of operations that an algorithm performs typically depends on the size, n, of its input.
  - for sorting algorithms, n is the # of elements in the array
  - \( C(n) \) = number of comparisons
  - \( M(n) \) = number of moves

- To express the time complexity of an algorithm, we'll express the number of operations performed as a function of n.
  - examples:  \( C(n) = n^2 + 3n \)
    \( M(n) = 2n^2 - 1 \)
Counting Comparisons by Selection Sort

```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;
}

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:
  \[
  C(n) = \sum_{i=1}^{n-1} i = \frac{(n - 1)(n - 1) + 1}{2} = \frac{(n - 1)n}{2}
  \]

\[
C(n) = \frac{n^2}{2} - \frac{n}{2}
\]
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 \( n \).

<table>
<thead>
<tr>
<th>( n )</th>
<th>( n^2/2 )</th>
<th>( n/2 )</th>
<th>( n^2/2 - n/2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>50</td>
<td>5</td>
<td>45</td>
</tr>
<tr>
<td>100</td>
<td>5000</td>
<td>50</td>
<td>4950</td>
</tr>
<tr>
<td>10000</td>
<td>50,000,000</td>
<td>5000</td>
<td>49,995,000</td>
</tr>
</tbody>
</table>

• In characterizing the time complexity of an algorithm, we’ll focus on the largest term in its operation-count expression.
  • for selection sort, \( C(n) = \frac{n^2}{2} - \frac{n}{2} \approx \frac{n^2}{2} \)

• In addition, we’ll typically ignore the coefficient of the largest term (e.g., \( \frac{n^2}{2} \to n^2 \)).

Big-O Notation

• We specify the largest term using big-O notation.
  • e.g., we say that \( C(n) = \frac{n^2}{2} - \frac{n}{2} \) is \( O(n^2) \)

<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>( 3\log_{10}n, \log_2n + 5 )</td>
<td>( O(\log n) )</td>
</tr>
<tr>
<td>linear time</td>
<td>( 5n, 10n - 2\log_2n )</td>
<td>( O(n) )</td>
</tr>
<tr>
<td>( n\log n ) time</td>
<td>( 4n\log_2n, n\log_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>exponential time</td>
<td>( 2^n, 5e^n + 2n^2 )</td>
<td>( O(c^n) )</td>
</tr>
</tbody>
</table>

• 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 \( n \geq 0 \).

- \( c = 1 \), \( n_0 = 0 \)

- \( g(n) = n^2 \)

- \( f(n) = \frac{n^2}{2} - \frac{n}{2} \)

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

- 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, we generally try to use big-O notation to characterize a function as closely as possible – i.e., as if we were using it to specify a tight bound.
  - For our example, we would say that $3n - 3$ is $O(n)$

Big-Theta Notation

- In theoretical computer science, *big-theta* notation ($\Theta$) is used to specify a tight bound.

- $f(n) = \Theta(g(n))$ if there exist constants $c_1$, $c_2$, and $n_0$ such that $c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n > n_0$

- Example: $f(n) = \frac{n^2}{2} - \frac{n}{2}$ is $\Theta(n^2)$, because $(1/4)*n^2 \leq \frac{n^2}{2} - \frac{n}{2} \leq n^2$ for all $n \geq 2$

- $c_1 = \frac{1}{4}$
- $c_2 = 1$
- $n_0 = 2$

![Graph showing examples of big-O and big-theta notations](image)
## Sorting by Insertion I: Insertion Sort

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

- **Example:**

  
  \[
  \begin{array}{cccccc}
  0 & 1 & 2 & 3 & 4 \\
  15 & 4 & 2 & 12 & 6 \\
  4 & 15 & 2 & 12 & 6 \\
  2 & 4 & 15 & 12 & 6 \\
  2 & 4 & 12 & 15 & 6 \\
  2 & 4 & 6 & 12 & 15 \\
  \end{array}
  \]

## 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:

  
  \[
  \begin{array}{ccccccc}
  0 & 1 & 2 & 3 & 4 & 5 & 6 \\
  18 & 12 & 15 & 9 & 25 & 2 & 17 \\
  \end{array}
  \]

- 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: 6 14 19 9 ...

- 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:**

  - The number of operations is minimized when the input array is already sorted.
  - Complexity: $O(n)$

- **worst case:**

  - The number of operations is maximized when the input array is sorted in reverse order.
  - Complexity: $O(n^2)$

- **average case:**

  - Complexity: $O(n^2)$
Sorting by Insertion II: Shell Sort

- Developed by Donald Shell in 1959
- Improves on insertion sort
- Takes advantage of the fact that insertion sort is fast when an array is almost sorted.
- Seeks to eliminate a disadvantage of insertion sort: if an element is far from its final location, many “small” moves are required to put it 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.

Sorting Subarrays

- Basic idea:
  - use insertion sort on subarrays that contain elements separated by some increment
    - 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 consider the subarrays one at a time.
- We consider elements \( arr[incr] \) through \( arr[arr.length-1] \), inserting each element into its proper place with respect to the elements from its subarray that are to the left of the element.

The same example (\( incr = 3 \)):

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>36</td>
<td>18</td>
<td>10</td>
<td>27</td>
<td>3</td>
<td>20</td>
<td>9</td>
<td>8</td>
</tr>
<tr>
<td>27</td>
<td>18</td>
<td>10</td>
<td>36</td>
<td>3</td>
<td>20</td>
<td>9</td>
<td>8</td>
</tr>
<tr>
<td>27</td>
<td>3</td>
<td>10</td>
<td>36</td>
<td>18</td>
<td>20</td>
<td>9</td>
<td>8</td>
</tr>
<tr>
<td>27</td>
<td>3</td>
<td>10</td>
<td>36</td>
<td>18</td>
<td>20</td>
<td>9</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>3</td>
<td>10</td>
<td>27</td>
<td>18</td>
<td>20</td>
<td>36</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>3</td>
<td>10</td>
<td>27</td>
<td>8</td>
<td>20</td>
<td>36</td>
<td>18</td>
</tr>
</tbody>
</table>

- When we consider element \( i \), the other elements in its subarray are already sorted with respect to each other.
- To insert element \( i \):
  - make a copy of element \( i \), storing it in the variable \( toInsert \):

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>27</td>
<td>3</td>
<td>10</td>
<td>36</td>
<td>18</td>
<td>20</td>
<td>9</td>
<td>8</td>
</tr>
</tbody>
</table>

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

- 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 <= \( toInsert \)
- copy \( toInsert \) into the “hole”:
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 \( n \)?

  value of \( i \) number of times statement 2 is executed
Sorting by Exchange I: Bubble Sort

- Perform a sequence of passes through the array.
- On each pass: proceed from left to right, swapping 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, so 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>
</tr>
</thead>
<tbody>
<tr>
<td>28</td>
<td>24</td>
<td>27</td>
<td>18</td>
<td></td>
</tr>
</tbody>
</table>

- after the first pass: 24 27 18 28
- after the second: 24 18 27 28
- after the third: 18 24 27 28

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);
                }
            }
        }
    }
}
```

- One for-loop nested in another:
  - the inner loop performs a single pass
  - the outer loop governs the number of passes, and the ending point of each pass
Time Analysis of Bubble Sort

- **Comparisons**: the kth pass performs _____ comparisons, so we get \( C(n) = \)
- **Moves**: depends on the contents of the array
  - in the worst case:
  - in the best case:
- **Running time**

---

Sorting by Exchange II: Quicksort

- Like bubble sort, quicksort uses an approach based on exchanging 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 the left array <= each element in the right array*

    example:

    ![Example Image](image)

    - **conquer**: apply quicksort recursively to the subarrays, stopping when a subarray has a single element
    - **combine**: nothing needs to be done, because of the criterion used in forming the subarrays
Partitioning an Array Using a Pivot

- The process that quicksort uses to rearrange the elements is known as **partitioning** the array.
- Partitioning is done using a value known as the **pivot**.
- We rearrange the elements to produce two subarrays:
  - **left subarray**: all values <= pivot
  - **right subarray**: all values >= pivot

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

\[\text{partition using a pivot of 9}\]

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

- Our approach to partitioning is one of several variants.
- Partitioning is useful in its own right. ex: find all students with a GPA > 3.0.

Possible Pivot Values

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

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

\[\text{pivot = 18}\]

- **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 $\text{arr}[j]$:

Partitioning Example (cont.)

- Find:

- Swap:

- Find:

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

- Subarrays: $\text{left} = \text{arr}[\text{first} : j]$, $\text{right} = \text{arr}[j+1 : \text{last}]$
Partitioning Example 2

• Start (pivot = 13):
  
  \[
  \begin{array}{cccccccc}
  24 & 5 & 2 & 13 & 18 & 4 & 20 & 19 \\
  \end{array}
  \]

• Find:
  
  \[
  \begin{array}{cccccccc}
  24 & 5 & 2 & 13 & 18 & 4 & 20 & 19 \\
  \end{array}
  \]

• Swap:
  
  \[
  \begin{array}{cccccccc}
  4 & 5 & 2 & 13 & 18 & 24 & 20 & 19 \\
  \end{array}
  \]

• Find:
  
  \[
  \begin{array}{cccccccc}
  4 & 5 & 2 & 13 & 18 & 24 & 20 & 19 \\
  \end{array}
  \]

and now the indices are equal, so we return \( j \).

• Subarrays:
  
  \[
  \begin{array}{cccccccc}
  4 & 5 & 2 & 13 & 18 & 24 & 20 & 19 \\
  \end{array}
  \]

Partitioning Example 3 (done together)

• Start (pivot = 5):
  
  \[
  \begin{array}{cccccccc}
  4 & 14 & 7 & 5 & 2 & 19 & 26 & 6 \\
  \end{array}
  \]

• Find:
  
  \[
  \begin{array}{cccccccc}
  4 & 14 & 7 & 5 & 2 & 19 & 26 & 6 \\
  \end{array}
  \]
Partitioning Example 4

- Start (pivot = 15):
  
  - Find:

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

... 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
    $18/2 = 9$  $9/2 = 4$  $4/2 = 2$  $2/2 = 1$
A Quick Review of Logs (cont.)

- If the number of operations performed by an algorithm is proportional to \( \log_b n \) for any base \( b \), we say it is a \( O(\log n) \) algorithm – dropping the base.
- \( \log_b n \) 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 \times 1024 (1M)</td>
<td>20</td>
</tr>
</tbody>
</table>

- Thus, for large values of \( n \):
  - a \( O(\log n) \) algorithm is much faster than a \( O(n) \) algorithm
  - a \( O(n \log n) \) algorithm is much faster than a \( O(n^2) \) algorithm
- We can also show that an \( O(n \log n) \) algorithm is faster than a \( O(n^{1.5}) \) algorithm like Shell sort.

Time Analysis of Quicksort

- Partitioning an array requires \( n \) comparisons, because each element is compared with the pivot.
- \textit{best case:} partitioning always divides the array in half
  - repeated recursive calls give:

\[
\begin{align*}
&n \\
&\frac{n}{2} \quad \frac{n}{2} \\
&\frac{n}{4} \quad \frac{n}{4} \quad \frac{n}{4} \quad \frac{n}{4} \\
&\ldots \quad \ldots \quad \ldots \quad \ldots \\
&1 \quad 1 \quad 1 \quad 1 \quad \ldots \quad 1 \quad 1 \quad 1 \quad 1 \\
&\text{comparisons} \\
&n \\
&2^{2}(n/2) = n \\
&4^{2}(n/4) = n \\
&\ldots \quad \ldots \quad \ldots \quad \ldots \\
&0
\end{align*}
\]

- 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:
    \[
    \begin{align*}
    C(n) &= \frac{n}{n} \frac{n-1}{n-1} \frac{n-2}{n-2} \frac{n-3}{n-3} \ldots \frac{2}{2} \\
    &\sum_{i=2}^{n} i = O(n^2).
    \end{align*}
    \]
  - \( C(n) = O(n^2) \) 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

- All of the comparison-based sorting algorithms that we've seen thus far have sorted the array in place.
  - used only a small amount of additional memory

- Mergesort is a sorting algorithm that 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 into a single sorted array.
  - example:
    \[
    \begin{array}{cccc}
    2 & 8 & 14 & 24 \\
    \end{array}
    \quad
    \begin{array}{cccccccc}
    2 & 5 & 7 & 8 & 9 & 11 & 14 & 24 \\
    \end{array}
    \quad
    \begin{array}{cccc}
    5 & 7 & 9 & 11 \\
    \end{array}
    \]
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:

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

- 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 index k

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

Merging Sorted Arrays (cont.)

- Starting point:

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

- After the first copy:

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

- After the second copy:

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

• After the third copy:

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

  i
  j
  k

• After the fourth copy:

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

  i
  j
  k

• After the fifth copy:

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

  i
  j
  k

• After the sixth copy:

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

  i
  j
  k

• 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

  i
  j
  k
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:

```
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:

12 8 14 4 6 33 2 27
12 8

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

12 8

make a recursive call to sort its right subarray:

12 8 14 4 6 33 2 27
12 8
12

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

12 8
Tracing the Calls to Mergesort

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

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

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

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

Tracing the Calls to Mergesort

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

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

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

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

base case…
Tracing the Calls to Mergesort

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

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

\[
\begin{array}{cccc}
8 & 12 & 14 & 4 \\
\end{array}
\]

\[
\begin{array}{cc}
14 & 4 \\
\end{array}
\]

make a recursive call to sort its right subarray:

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

\[
\begin{array}{cccc}
8 & 12 & 14 & 4 \\
\end{array}
\]

\[
\begin{array}{cc}
14 & 4 \\
\end{array}
\]

\[
\begin{array}{c}
4 \\
\end{array}
\]  
base case…

Tracing the Calls to Mergesort

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

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

\[
\begin{array}{cccc}
8 & 12 & 14 & 4 \\
\end{array}
\]

\[
\begin{array}{cc}
14 & 4 \\
\end{array}
\]

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

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

\[
\begin{array}{cccc}
8 & 12 & 14 & 4 \\
\end{array}
\]

\[
\begin{array}{cccc}
14 & 4 \Rightarrow 4 & 14 \\
\end{array}
\]
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 => 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

- One approach is to create new arrays for each new set of subarrays, and to 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:

```plaintext
| 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:

```plaintext
| 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
 /   \
/     \
1/     \
1/     \
1/     \
1/     \
1/     \
```

- at all but the last level of the call tree, there are $2n$ moves
- how many levels are there?
  - $M(n) = ?$
  - $C(n) = ?$
Comparison-Based 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</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Insertion</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 extra memory – and moves to and from the temp array.
- Quicksort is comparable to mergesort in the average case. With a reasonable pivot choice, its worst case is seldom seen.
- Use `SortCount.java` to experiment.

Comparison-Based vs. Distributive Sorting

- Until now, all of the sorting algorithms we have 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.

Exampless:

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

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

33 41 12 24 31 14 13 42 34

gotten:

41 31 12 42 33 13 24 14 34

Repeat, moving back one digit each time:

gotten:


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:

<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>$3\log_{10}n$, $\log_2n + 5$</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>linear time</td>
<td>$5n$, $10n - 2\log_2n$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>nlogn time</td>
<td>$4n\log_2n$, $n\log_2n + n$</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>$n^3 + 3n^3$, $5n^3 - 5$</td>
<td>$O(n^3)$</td>
</tr>
<tr>
<td>exponential time</td>
<td>$2^n$, $5e^n + 2n^2$</td>
<td>$O(e^n)$</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
  - $O(\log_2n)$-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>(n)</th>
<th>(n^2)</th>
<th>(n^5)</th>
<th>(2^n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>.00001 s</td>
<td>.0001 s</td>
<td>.1 s</td>
<td>.001 s</td>
</tr>
<tr>
<td>20</td>
<td>.00002 s</td>
<td>.0004 s</td>
<td>3.2 s</td>
<td>1.0 s</td>
</tr>
<tr>
<td>30</td>
<td>.00003 s</td>
<td>.0008 s</td>
<td>24.3 s</td>
<td>17.9 min</td>
</tr>
<tr>
<td>40</td>
<td>.00004 s</td>
<td>.0016 s</td>
<td>1.7 min</td>
<td>12.7 days</td>
</tr>
<tr>
<td>50</td>
<td>.00005 s</td>
<td>.0025 s</td>
<td>5.2 min</td>
<td>35.7 yrs</td>
</tr>
<tr>
<td>60</td>
<td>.00006 s</td>
<td>.0036 s</td>
<td>13.0 min</td>
<td>36,600 yrs</td>
</tr>
</tbody>
</table>

• sample computations:
  • when \(n = 10\), an \(n^2\) algorithm performs \(10^2\) operations.
    \(10^2 \times (1 \times 10^{-6} \text{ sec}) = .0001 \text{ sec}\)
  • when \(n = 30\), a \(2^n\) 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>(n)</th>
<th>(n^2)</th>
<th>(n^5)</th>
<th>(2^n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 min</td>
<td>60,000,000</td>
<td>7745</td>
<td>35</td>
<td>25</td>
</tr>
<tr>
<td>1 hour</td>
<td>3.6 \times 10^9</td>
<td>60,000</td>
<td>81</td>
<td>31</td>
</tr>
<tr>
<td>1 week</td>
<td>6.0 \times 10^{11}</td>
<td>777,688</td>
<td>227</td>
<td>39</td>
</tr>
<tr>
<td>1 year</td>
<td>3.1 \times 10^{13}</td>
<td>5,615,692</td>
<td>500</td>
<td>44</td>
</tr>
</tbody>
</table>

• sample computations:
  • 1 hour = 3600 sec
    that's enough time for \(3600/(1 \times 10^{-6}) = 3.6 \times 10^6\) 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:
    \(2^n = 3.6 \times 10^9\) \(\Rightarrow\) \(n = \log_2(3.6 \times 10^9) \approx 31\)