Asymptotic Performance

• We compare algorithms based on large input size
• Predict upper and lower bounds on the speed of the algorithms rather than predicting their exact speed.
• To compare two algorithms, we consider the *asymptotic* behavior of the two functions for very large input sizes.
• Asymptotic Notations
  - Abstract away low order terms and constants
  - Describe the running time of an algorithm as \( n \) grows to \( \infty \).
Asymptotic Upper Bound – O (Big-Oh) Notation

• Definition
  - \( O(g(n)) = \{ f(n): \exists \text{ positive constants } c \text{ and } n_0 \text{ such that } 0 \leq f(n) \leq c \cdot g(n) \ \forall \ n \geq n_0 \} \)

• We write \( f(n) = O(g(n)) \) instead of \( f(n) \in O(g(n)) \)

*Intuitively*: Set of all functions whose *rate of growth* is the same as or lower than that of \( g(n) \).

\( g(n) \) is an *asymptotic upper bound* for \( f(n) \).
Example

• Consider the function \( f(n) = 4n + 32 \) - show that \( f(n) = O(n^2) \).

• This means that we need to find an integer \( n_0 \) and a constant \( c > 0 \) such that for all integers \( n \geq n_0 \), \( f(n) \leq cn^2 \).

• Suppose we choose \( c = 1 \). Then

\[
4n + 32 \leq n^2
\]

\[
0 \leq n^2 - 4n - 32
\]

\[
0 \leq (n - 8)(n + 4)
\]

For the above inequality to be satisfied \( (n_0 - 8) \geq 0 \) i.e. \( n_0 = 8 \). This means that for \( c = 1 \) and \( n_0 = 8 \), \( f(n) \leq cn^2 \) for all \( n \geq n_0 \). Hence, \( f(n) = O(n^2) \).
• Of course there are other choices for $c$ and $n_0$. The important thing is that there should be some choice for which it can be proven that $f(n) \leq cn^2$.

• Can you tell me what will be the value of $n_0$ if $c = 4$?
  - Ans:- $n_0 = 6$. 

Loose & Tight Bounds

• $f(n) = 3n^2 - 100n + 6 = O(n^2)$ because $3n^2 - 100n + 6 < 3n^2$
• $f(n) = 3n^2 - 100n + 6 = O(n^3)$ because $3n^2 - 100n + 6 < 0.0001n^3$
• $f(n) = 3n^2 - 100n + 6 \neq O(n)$ because $3n^2 - 100n + 6 > cn$ when $n > c$
• $O(n^2)$ is tight asymptotic upper bound whereas $O(n^3)$ is loose asymptotic upper bounds. It is also correct to say that $O(n^4), O(n^5)$ … are also loose asymptotic bounds for $f(n)$. 
Asymptotic Lower Bound – $\Omega$ (Big-Omega) Notation

- **Definition**
  - $\Omega(g(n)) = \{ f(n): \exists$ positive constants $c$ and $n_0$ such that $f(n) \geq c \cdot g(n) \geq 0 \ \forall \ n \geq n_0 \}$

- We write $f(n) = \Omega(g(n))$ instead of $f(n) \in \Omega(g(n))$

*Intuitively*: Set of all functions whose *rate of growth* is the same as or higher than that of $g(n)$.

$g(n)$ is an **asymptotic lower bound** for $f(n)$. 
Show that \( f(n) = 5n^2 - 64n + 256 = \Omega(n^2) \)
Asymptotic Tight Bound – \( \Theta \) (Theta)
Notation

For function \( g(n) \), we define \( \Theta(g(n)) \), big-Theta of \( n \), as the set:

\[
\Theta(g(n)) = \{ f(n) : \exists \text{ positive constants } c_1, c_2, \text{ and } n_0, \text{ such that } \forall n \geq n_0, \text{ we have } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \}
\]

Technically, \( f(n) \in \Theta(g(n)) \).
Usage, \( f(n) = \Theta(g(n)) \).

\( f(n) \) and \( g(n) \) are nonnegative, for large \( n \).
Linked List Representation

- Naïve implementation: use a linked list to represent each set:
  - First element is the set Representative
    - MakeSet(): ??? time
    - FindSet(): ??? time
    - Union(A,B): “copy” elements of A into B: ??? time
Disjoint Set Union

- Naïve implementation: use a linked list to represent each set
  - First element is the set Representative

- MakeSet(): O(1) time
- FindSet(): O(1) time
- Union(A,B): “copy” elements of A into B: O(A) time
Disjoint Set Union: Analysis

- **Worst-case analysis:** $O(n^2)$ time for $n$ Union’s

  - Make-Set($x_1$) 1 element updated
  - Make-Set($x_2$) 1 element updated
  - …
  - Make-Set($x_n$) 1 element updated
  - Union($x_1, x_2$) “copy” 1 element
  - Union($x_2, x_3$) “copy” 2 elements
  - …
  - Union($x_{n-1}, x_n$) “copy” n-1 elements
  - $O(n^2)$
Implementation of FIND and UNION

- **FIND(x)** → follow the path from x until the root is reached, then return *root(x)*.
  - *Time complexity is O(n)*
  - Find(x) = Find(y), when x and y are in the same set

- **UNION(x,y)** → UNION(FIND(x) , FIND(y) ) → UNION(root(x) , root(y) ) → UNION(u,v) then let v be the parent of u. Assume u is root(x), v is root(y)
  - *Time complexity is O(n)*
  - Union(x, y) Combine the set that contains x with the set that contains y
Suppose we start with the single element sets \{1\}, \{2\}, ..., \{n\}, then, execute the following sequence of unions and finds:

- UNION(1, 2), UNION(2, 3), ..., UNION(n – 1, n)
- FIND(1), FIND(2), ..., FIND(n).

In this case, the total cost of the \(n\) find operations is proportional to:

\[
n + (n - 1) + \ldots + 1 = n(n+1)/2 = O(n^2)
\]

**Improvement:**

- Union by rank
- Path compression.
The UNION Algorithm

Input: Two elements $x$ and $y$
Output: The union of the two trees containing $x$ and $y$. The original trees are destroyed.

1. $u \leftarrow \text{FIND}(x)$; $v \leftarrow \text{FIND}(y)$
2. if $\text{rank}(u) \leq \text{rank}(v)$ then
3. \hspace{1em} $p(u) \leftarrow v$
4. \hspace{1em} if $\text{rank}(u) = \text{rank}(v)$ then $\text{rank}(v) \leftarrow \text{rank}(v) + 1$
5. \hspace{1em} else $p(v) \leftarrow u$
6. end if
Algorithm FIND

Input: A node $x$
Output: $root(x)$, the root of the tree containing $x$.

1. $y \leftarrow x$
2. while $p(y) \neq \text{null}$  \{Find the root of the tree containing $x$\}
3. \hspace{1em} $y \leftarrow p(y)$
4. end while
5. $root \leftarrow y$; \hspace{0.5em} $y \leftarrow x$
6. while $p(y) \neq \text{null}$  \{Do path compression\}
7. \hspace{1em} $w \leftarrow p(y)$
8. \hspace{1.5em} $p(y) \leftarrow root$
9. \hspace{2em} $y \leftarrow w$
10. end while
11. return $root$
a) Let \( S = \{1,2,\ldots,9\} \) and consider applying the following sequence of \textit{unions} and \textit{finds}:

b) \text{UNION}(1,2), \text{UNION}(3,4), \text{UNION}(5,6), \text{UNION}(7,8)

c) \text{UNION}(2,4), \text{UNION}(8,9), \text{UNION}(6,8)

d) \text{FIND}(5)

e) \text{UNION}(4,8)

f) \text{FIND}(1)
More variations:

- A *weighted graph* associates weights with either the edges or the vertices
  - E.g., a road map: edges might be weighted with distance
- A *multigraph* allows multiple edges between the same vertices
  - E.g., the call graph in a program (a function can get called from multiple points in another function)
Graphs

- We will typically express running times in terms of $|E|$ and $|V|$ (often dropping the ’s)
  - If $|E| \approx |V|^2$ the graph is *dense*
  - If $|E| \approx |V|$ the graph is *sparse*

- If you know you are dealing with dense or sparse graphs, different data structures may make sense
Representing Graphs

- Assume $V = \{1, 2, \ldots, n\}$
- An adjacency matrix represents the graph as a $n \times n$ matrix $A$:
  - $A[i, j] = 1$ if edge $(i, j) \in E$ (or weight of edge)
  - $= 0$ if edge $(i, j) \notin E$
Graphs: Adjacency Matrix

- Example:
Graphs: Adjacency Matrix

Example:

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Graphs: Adjacency Matrix

- How much storage does the adjacency matrix require?
- A: $O(V^2)$
- What is the minimum amount of storage needed by an adjacency matrix representation of an undirected graph with 4 vertices?
- A: 6 bits
  - Undirected graph $\rightarrow$ matrix is symmetric
  - No self-loops $\rightarrow$ don’t need diagonal
The adjacency matrix is a dense representation.

- Usually too much storage for large graphs.
- But can be very efficient for small graphs.

Most large interesting graphs are sparse.

- For this reason the *adjacency list* is often a more appropriate representation.
Graphs: Adjacency List

- Adjacency list: for each vertex $v \in V$, store a list of vertices adjacent to $v$

- Example:
  - $\text{Adj}[1] = \{2,3\}$
  - $\text{Adj}[2] = \{3\}$
  - $\text{Adj}[3] = \{\}$
  - $\text{Adj}[4] = \{3\}$

- Variation: can also keep a list of edges coming into vertex

![Graph Diagram]
Graphs: Adjacency List

- How much storage is required?
  - The *degree* of a vertex \( v \) = # incident edges
    - Directed graphs have in-degree, out-degree
  - For directed graphs, # of items in adjacency lists is
    \[ \sum \text{out-degree}(v) = |E| \]
    takes \( \Theta(V + E) \) storage  
    *Why?*
  - For undirected graphs, # items in adj lists is
    \[ \sum \text{degree}(v) = 2 |E| \]
    also \( \Theta(V + E) \) storage
- So: Adjacency lists take \( O(V+E) \) storage
Graph Searching

- Given: a graph $G = (V, E)$, directed or undirected
- Goal: methodically explore every vertex and every edge
- Ultimately: build a tree on the graph
  - Pick a vertex as the root
  - Choose certain edges to produce a tree
  - Note: might also build a forest if graph is not connected
Minimum Spanning Trees
Minimum Spanning Tree

- Problem: given a connected, undirected, weighted graph:
Minimum Spanning Tree

Problem: given a connected, undirected, weighted graph, find a spanning tree using edges that minimize the total weight.
Minimum Spanning Tree

- Which edges form the minimum spanning tree (MST) of the below graph?
Minimum Spanning Tree

- Answer:

H B C
G E D
F
A

14 10 3 6 4 5 2 9 15 8
Kruskal’s Algorithm

Kruskal() {
    T = ∅;
    for each v ∈ V
        MakeSet(v);
    sort E by increasing edge weight w
    for each (u,v) ∈ E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            T = T U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
Kruskal’s Algorithm

Kruskal()
{
    T = \emptyset;
    for each v ∈ V
        MakeSet(v);
    sort E by increasing edge weight w
    for each (u,v) ∈ E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            T = T U \{\{u,v\}\};
            Union(FindSet(u), FindSet(v));
}
Kruskal’s Algorithm

Kruskal()
{
    T = Ø;
    for each v ∈ V
        MakeSet(v);
    sort E by increasing edge weight w
    for each (u,v) ∈ E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            T = T U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
Correctness Of Kruskal’s Algorithm

- Sketch of a proof that this algorithm produces an MST for $T$:
  - Assume algorithm is wrong: result is not an MST
  - Then algorithm adds a wrong edge at some point
  - If it adds a wrong edge, there must be a lower weight edge (cut and paste argument)
  - But algorithm chooses lowest weight edge at each step. Contradiction
Kruskal’s Algorithm: Running Time

- To summarize:
  - Sort edges: $O(E \lg E)$
  - $O(V)$ MakeSet()’s
  - $O(E)$ FindSet()’s
  - $O(E)$ Union()’s

- Upshot:
  - Best disjoint-set union algorithm makes above 3 operations take $O((V+E)\alpha(V))$, $\alpha$ almost constant
  - Overall thus $O(E \lg V)$, since for a connected graph $|E| \geq |V| - 1$ and $\alpha(V) = O(\lg V)$
Prim’s Algorithm

MST-Prim(G, w, r)
Q = V[G];
for each u ∈ Q
    key[u] = ∞;
key[r] = 0;
p[r] = NULL;
while (Q not empty)
    u = ExtractMin(Q);
    for each v ∈ Adj[u]
        if (v ∈ Q and w(u,v) < key[v])
            p[v] = u;
            key[v] = w(u,v);
Prim’s Algorithm

MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
Example – Find MST using Prim’s Algorithm \((r = D)\)
Analysis

- ExtractMin called $\Theta(V)$ times
- DecreaseKey called $\Theta(E)$ times
- Running time = $\Theta(V) \cdot T_{\text{ExtractMin}} + \Theta(E) \cdot T_{\text{DecreaseKey}}$
- Binary Heap
  - Time = $\Theta(V \cdot \log V) + \Theta(E \cdot \log V) = O(E \cdot \log V)$ (since for a connected graph $|E| \geq |V| - 1$)
- Fibonacci Heap
  - Time = $O(V \cdot \log V + E)$
Single-Source Shortest Path

Dijkstra’s Algorithm (Greedy Algorithm)
Bellman-Ford (Dynamic Algorithm)
Shortest Paths Problems

Given a weighted directed graph,

\(<u,v,t,x,z>\) is a path of weight 29 from \(u\) to \(z\).

\(<u,v,w,x,y,z>\) is another path from \(u\) to \(z\); it has weight 16 and is the shortest path from \(u\) to \(z\).
Variants of Shortest Paths Problems

A. Single pair shortest path problem
   ○ Given s and d, find shortest path from s to d.

B. Single source shortest paths problem
   ○ Given s, for each d find shortest path from s to d.

C. All-pairs shortest paths problem
   ○ For each ordered pair s,d, find shortest path.

- (A) and (B) seem to have same asymptotic complexity.
- (C) takes longer, but not as long as repeating (B) for each s.
More Shortest Paths Variants

1. All weights are non-negative.
2. Weights may be negative, but no negative cycles
   (A cycle is a path from a vertex to itself.)
3. Negative cycles allowed.
   Algorithm reports “∞” if there is a negative cycle on path from
   source to destination

- (2) and (3) seem to be harder than (1).
Single-Source Shortest Path

- Problem: given a weighted directed graph $G$, find the minimum-weight path from a given source vertex $s$ to another vertex $v$
  - “Shortest-path” = minimum weight
  - Weight of path is sum of edges
  - E.g., a road map: what is the shortest path from Chapel Hill to Charlottesville?
Dijkstra's Algorithm

Dijkstra(G)
   for each v \in V
      d[v] = \infty;
   d[s] = 0; S = \emptyset; Q = V;
   while (Q \neq \emptyset)
      u = ExtractMin(Q);
      S = S \cup \{u\};
      for each v \in u->Adj[]
         if (d[v] > d[u]+w(u,v))
            d[v] = d[u]+w(u,v);

\textbf{Relaxation Step}
Example

\[ s \rightarrow 0 \rightarrow x \rightarrow y \rightarrow v \rightarrow u \rightarrow s \]

Weights:
- \( s \rightarrow 0 \): 5
- \( 0 \rightarrow x \): 2
- \( x \rightarrow y \): 2
- \( y \rightarrow v \): 4
- \( v \rightarrow u \): 9
- \( u \rightarrow s \): 10
- \( s \rightarrow x \): 7
- \( x \rightarrow y \): 3
- \( y \rightarrow v \): 6
Dijkstra’s Algorithm

- If no negative edge weights, we can beat BF
- Similar to breadth-first search
  - Grow a tree gradually, advancing from vertices taken from a queue
- Also similar to Prim’s algorithm for MST
  - Use a priority queue keyed on $d[v]$
- Greedy Algorithm
Bellman-Ford Algorithm

BellmanFord()
   for each $v \in V$
      $d[v] = \infty$;
   $d[s] = 0$;
   for $i=1$ to $|V|-1$
      for each edge $(u,v) \in E$
         Relax($u,v$, w($u,v$));
      for each edge $(u,v) \in E$
         if ($d[v] > d[u] + w(u,v)$)
            return “no solution”;

Initialize $d[]$, which will converge to shortest-path value $\delta$

Relaxation:
Make $|V|-1$ passes, relaxing each edge

Test for solution
Under what condition do we get a solution?

Relax($u,v,w$): if ($d[v] > d[u]+w$) then $d[v]=d[u]+w$
Each pass relaxes edges in the order:

- $u,v$
- $u,y$
- $u,x$
- $v,u$
- $x,v$
- $x,y$
- $y,v$
- $y,z$
- $z,u$
- $z,x$
**Another Look**

**Note:** This is essentially **dynamic programming**.

Let \( d(i, j) \) = cost of the shortest path from \( s \) to \( i \) that is at most \( j \) hops.

\[
d(i, j) = \begin{cases} 
0 & \text{if } i = s \land j = 0 \\
\infty & \text{if } i \neq s \land j = 0 \\
\min\{d(k, j-1) + w(k, i) : i \in \text{Adj}(k)\} & \text{if } j > 0 
\end{cases}
\]
Example

```
S
A
  4
  -1
B
  3
  2
  2
  1
C
  5
D
  3
E
  -3
```
Quicksort

• Another divide-and-conquer algorithm
  ■ The array $A[p..r]$ is \textit{partitioned} into two possibly empty subarrays $A[p..q-1]$ and $A[q+1..r]$
    ◆ Invariant: All elements in $A[p..q]$ are less than all elements in $A[q+1..r]$
  ■ The subarrays are recursively sorted by calls to quicksort
  ■ Unlike merge sort, no combining step: two subarrays form an already-sorted array
Quicksort

Quicksort(A, p, r)
{
    if (p < r)
    {
        q = Partition(A, p, r);
        Quicksort(A, p, q-1);
        Quicksort(A, q+1, r);
    }
}

Initial call is quicksort(A, 1, length[A])
Analyzing Quicksort

- **What will be the worst case for the algorithm?**
  - Partition is always unbalanced
- **What will be the best case for the algorithm?**
  - Partition is perfectly balanced
- **Which is more likely?**
  - The latter, by far
- **Will any particular input elicit the worst case?**
  - Yes: Already-sorted input
Improving Quicksort

- The real liability of quicksort is that it runs in $O(n^2)$ on already-sorted input
- Book discusses two solutions:
  - Randomize the input array, OR
  - *Pick a random pivot element*
- *How will these solve the problem?*
  - By insuring that no particular input can be chosen to make quicksort run in $O(n^2)$ time
Strassen discovered a different recursive approach that requires only 7 recursive multiplications of \( n/2 \times n/2 \) matrices and 18 \( \Theta(n^2) \) scalar additions and subtractions, yielding the recurrence

\[
T(n) = 7 T(n/2) + n^2
\]

\[
= \Theta(n \lg 7)
\]

\[
= O(n^{2.81})
\]
Strassens’s Matrix Multiplication

\[
\begin{pmatrix}
C_{11} & C_{12} \\
C_{21} & C_{22}
\end{pmatrix} = \begin{pmatrix}
A_{11} & A_{12} \\
A_{21} & A_{22}
\end{pmatrix} \begin{pmatrix}
B_{11} & B_{12} \\
B_{21} & B_{22}
\end{pmatrix}
\]

P_1 = (A_{11} + A_{22})(B_{11} + B_{22})
P_2 = (A_{21} + A_{22}) \cdot B_{11}
P_3 = A_{11} \cdot (B_{12} - B_{22})
P_4 = A_{22} \cdot (B_{21} - B_{11})
P_5 = (A_{11} + A_{12}) \cdot B_{22}
P_6 = (A_{21} - A_{11}) \cdot (B_{11} + B_{12})
P_7 = (A_{12} - A_{22}) \cdot (B_{21} + B_{22})

C_{11} = P_1 + P_4 - P_5 + P_7
C_{12} = P_3 + P_5
C_{21} = P_2 + P_4
C_{22} = P_1 + P_3 - P_2 + P_6
\[ C_{11} = P_1 + P_4 - P_5 + P_7 \]
\[ = (A_{11} + A_{22})(B_{11} + B_{22}) + A_{22} \times (B_{21} - B_{11}) - (A_{11} + A_{12}) \times B_{22} + 
(A_{12} - A_{22}) \times (B_{21} + B_{22}) \]
\[ = A_{11} B_{11} + A_{11} B_{22} + A_{22} B_{11} + A_{22} B_{22} + A_{22} B_{21} - A_{22} B_{11} - 
A_{11} B_{22} - A_{12} B_{22} + A_{12} B_{21} + A_{12} B_{22} - A_{22} B_{21} - A_{22} B_{22} \]
\[ = A_{11} B_{11} + A_{12} B_{21} \]
Simple Sorting Algorithms
Outline

- We are going to look at three simple sorting techniques: Bubble Sort, Selection Sort, and Insertion Sort
- We are going to develop the notion of a loop invariant
- We will write code for Bubble Sort and Selection Sort, then derive their loop invariants
- We will start with a loop invariant for Insertion Sort, and derive the code for it
- We will analyze the running time for each of the above
Bubble Sort

- Compare each element (except the last one) with its neighbor to the right
  - If they are out of order, swap them
  - This puts the largest element at the very end
  - The last element is now in the correct and final place

- Compare each element (except the last two) with its neighbor to the right
  - If they are out of order, swap them
  - This puts the second largest element next to last
  - The last two elements are now in their correct and final places

- Compare each element (except the last three) with its neighbor to the right
  - Continue as above until you have no unsorted elements on the left
Example of Bubble Sort

\[
\begin{array}{cccc}
7 & 2 & 8 & 5 & 4 \\
2 & 7 & 8 & 5 & 4 \\
2 & 7 & 8 & 5 & 4 \\
2 & 7 & 5 & 8 & 4 \\
2 & 7 & 5 & 4 & 8 \\
\end{array}
\]
Analysis of Bubble Sort

- for (outer = a.length - 1; outer > 0; outer--) {
  for (inner = 0; inner < outer; inner++) {
    if (a[inner] > a[inner + 1]) {
      // code for swap omitted
  }
}

- Let \( n = a.length \) = size of the array
- The outer loop is executed \( n-1 \) times (call it \( n \), that’s close enough)
- Each time the outer loop is executed, the inner loop is executed
  - Inner loop executes \( n-1 \) times at first, linearly dropping to just once
  - On average, inner loop executes about \( n/2 \) times for each execution of the outer loop
  - In the inner loop, the comparison is always done (constant time), the swap might be done (also constant time)
- Result is \( n \times n/2 \times k \), that is, \( O(n^2/2 + k) = O(n^2) \)
Another sort: Selection Sort

- Given an array of length \( n \),
  - Search elements 0 through \( n-1 \) and select the smallest
    - Swap it with the element in location 0
  - Search elements 1 through \( n-1 \) and select the smallest
    - Swap it with the element in location 1
  - Search elements 2 through \( n-1 \) and select the smallest
    - Swap it with the element in location 2
  - Search elements 3 through \( n-1 \) and select the smallest
    - Swap it with the element in location 3
  - Continue in this fashion until there’s nothing left to search
Example and analysis of Selection Sort

- The Selection Sort might swap an array element with itself--this is harmless, and not worth checking for.

- **Analysis:**
  - The outer loop executes $n-1$ times.
  - The inner loop executes about $n/2$ times on average (from $n$ to 2 times).
  - Work done in the inner loop is constant (swap two array elements).
  - Time required is roughly $(n-1)*(n/2)$.
  - You should recognize this as $O(n^2)$. 

```
<table>
<thead>
<tr>
<th>7</th>
<th>2</th>
<th>8</th>
<th>5</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>7</td>
<td>8</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>8</td>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>5</td>
<td>8</td>
<td>7</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>5</td>
<td>7</td>
<td>8</td>
</tr>
</tbody>
</table>
```
One step of insertion sort

```
3 4 7 12 14 14 20 21 33 38 10 55 9 23 28 16
```

sorted

```
3 4 7 10 12 14 14 20 21 33 38 55 9 23 28 16
```

sorted

next to be inserted

```
3 4 7 12 14 14 20 21 33 38
```

less than 10

```
3 4 7 10 12 14 14 20 21 33 38 55 9 23 28 16
```

temp 10
Analysis of insertion sort

- We run once through the outer loop, inserting each of $n$ elements; this is a factor of $n$
- On average, there are $n/2$ elements already sorted
  - The inner loop looks at (and moves) half of these
  - This gives a second factor of $n/4$
- Hence, the time required for an insertion sort of an array of $n$ elements is proportional to $n^2/4$
- Discarding constants, we find that insertion sort is $O(n^2)$