Abstractions

In computer science, we're interested in devising efficient methods (i.e., algorithms) to solve problems.

- Abstraction vs. implementation.

Algorithm 1 Compute sum of integers in array

1: procedure ArraySum(A)
2:  \( sum = 0 \)
3:  for each integer \( i \) in \( A \) do
4:     \( sum = sum + i \)
5:  end for
6:  Return \( sum \)
7: end procedure

Spherical chickens in the vacuum

- “Well, I found a solution to your problem with the chickens. But I had to consider them as spherical objects in the vacuum and with a uniform mass distribution…”

[http://nossotradeoff.wordpress.com/2013/02/14/spherical-chickens-in-the-vacuum/](http://nossotradeoff.wordpress.com/2013/02/14/spherical-chickens-in-the-vacuum/)
Why computer scientists need to study algorithms?

- To know a standard set of important algorithms
- To be able to tell how efficient algorithms are
- To be able to recognize problems that can be solved with these algorithms
- To help understand problems and their solutions
- To help develop our analytical skills
An algorithm is a finite sequence of unambiguous instructions for solving a problem, i.e., for obtaining the required output for any legitimate input in a finite amount of time.

[Levitin 2003]
Types of algorithms

Some try to classify algorithms according the problem-solving approach they use:

- Iterative Algorithms
- Direct/Indirect Recursive Algorithms
- Divide-and-conquer Algorithms
- Dynamic-Programming Algorithms
- Randomized Algorithms
- Brute-Force Algorithms
Selecting an algorithm to solve a problem

Someone arriving at the airport and she needs to get to your house. What algorithm could she use?
- Get-a-Taxi Algorithm
- Call-You Algorithm
- Rent-a-Car Algorithm
- Bus Algorithm

How can we compare these approaches?
- What do they have in common?
- How do they differ?
Fundamental of Alg. Problem Solving

Algorithms are not answers to problems

- They are a set of specific instructions that produce the answer

---

Algorithm 1 Compute sum of integers in array

1: procedure ARRAYSUM(A)
2:   sum = 0
3:   for each integer i in A do
4:     sum = sum + i
5:   end for
6:   Return sum
7: end procedure
Typical steps to designing algorithms

1. Understand the problem
2. Assess the capabilities of a computational device
3. Choose between exact and approximate approach
4. Decide on appropriate data structures
   - Algorithms + Data Structures = Programs [Wirth 76]
5. Implement any solution (it helps fulfill Step 1)
6. Generate various improved solutions
7. Select the most efficient solution

Is the algorithm good enough?

Very important factors to consider:

- How often are you going to need the algorithm?
- What is the typical problem size you are trying to solve?

Less important factors that should also be considered:

- What language are you going to use to implement your algorithm?
- What is the cost-benefit of an efficient algorithm?
Practice

2.1-3
Consider the searching problem:

**Input:** A sequence of $n$ numbers $A = \langle a_1, a_2, \ldots, a_n \rangle$ and a value $\nu$.

**Output:** An index $i$ such that $\nu = A[i]$ or the special value NIL if $\nu$ does not appear in $A$.

Write pseudocode for linear search, which scans through the sequence, looking for $\nu$. 
Linear/sequential search

start here .....go through these, to the end ..... stop

a

toFind 25
**insertAt:** Insert an element into an array

![Diagram](http://www.algolist.net/Data_structures/Dynamic_array/Access_functions)
**removeAt**: Remove an element from an array

```
```

want to remove element at the position 3
Our first algorithm to analyze: Insertion sort

Input: A sequence of $n$ numbers $\langle a_1, a_2, \ldots, a_n \rangle$.

Output: A permutation (reordering) $\langle a'_1, a'_2, \ldots, a'_n \rangle$ of the input sequence such that $a'_1 \leq a'_2 \leq \cdots \leq a'_n$.

The numbers that we wish to sort are also known as the keys. Although conceptually we are sorting a sequence, the input comes to us in the form of an array with $n$ elements.
Figure 2.2 shows how the Insertion Sort algorithm works for \( A = 5, 2, 4, 6, 1, 3 \). The index \( j \) indicates the "current card" being inserted into the hand. At the beginning of each iteration of the for loop, which is indexed by \( j \), the subarray consisting of elements \( A[1:\ldots j] \) constitutes the currently sorted hand, and the remaining subarray \( A[j+1:\ldots n] \) corresponds to the pile of cards still on the table. In fact, elements \( A[1:\ldots j] \) are the elements originally in positions 1 through \( j \), but now in sorted order. We state these properties of \( A[1:\ldots j] \) formally as a loop invariant:

At the start of each iteration of the for loop of lines 1–8, the subarray \( A[1:\ldots j] \) consists of the elements originally in \( A[1:\ldots j] \), but in sorted order.

We use loop invariants to help us understand why an algorithm is correct. We must show three things about a loop invariant:
Insertion sort

**Insertion-Sort**$(A)$

1. **for** $j = 2$ **to** $A.length$
2. $key = A[j]$
4. $i = j − 1$
5. **while** $i > 0$ and $A[i] > key$
7. $i = i − 1$
8. $A[i + 1] = key$
2.1-2
Rewrite the INSERTION-SORT procedure to sort into nonincreasing instead of non-decreasing order.
Analysis of insertion sort

- The sorting time depends on the size of the input sequence.
- If two equal-size sequences are partially sorted, then sorting time will be different for each sequence.
- In general, the longer the sequence, the longer the sorting time.
- For running time, we can simplify the analysis by defining a cost $c_i$ to represent the time it takes to complete the instruction $i$. 

\[ \text{(a) } \begin{array}{ccccccc} 5 & 2 & 4 & 6 & 1 & 3 & \end{array} \]

\[ \text{(b) } \begin{array}{ccccccc} 2 & 5 & 4 & 6 & 1 & 3 & \end{array} \]

\[ \text{(c) } \begin{array}{ccccccc} 2 & 4 & 5 & 6 & 1 & 3 & \end{array} \]

\[ \text{(d) } \begin{array}{ccccccc} 2 & 4 & 5 & 6 & 1 & 3 & \end{array} \]

\[ \text{(e) } \begin{array}{ccccccc} 1 & 2 & 4 & 5 & 6 & 3 & \end{array} \]

\[ \text{(f) } \begin{array}{ccccccc} 1 & 2 & 3 & 4 & 5 & 6 & \end{array} \]
## Analysis of insertion sort

**Insertion-Sort**(*A*)

<table>
<thead>
<tr>
<th>Line</th>
<th>Description</th>
<th>Cost</th>
<th>Times</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><code>for j = 2 to A.length</code></td>
<td>$c_1$</td>
<td>$n$</td>
</tr>
<tr>
<td>2</td>
<td><code>key = A[j]</code></td>
<td>$c_2$</td>
<td>$n - 1$</td>
</tr>
<tr>
<td>3</td>
<td><code>// Insert A[j] into the sorted sequence A[1..j - 1].</code></td>
<td>0</td>
<td>$n - 1$</td>
</tr>
<tr>
<td>4</td>
<td><code>i = j - 1</code></td>
<td>$c_4$</td>
<td>$n - 1$</td>
</tr>
<tr>
<td>5</td>
<td><code>while i &gt; 0 and A[i] &gt; key</code></td>
<td>$c_5$</td>
<td>$\sum_{j=2}^{n} t_j$</td>
</tr>
<tr>
<td>6</td>
<td><code>A[i + 1] = A[i]</code></td>
<td>$c_6$</td>
<td>$\sum_{j=2}^{n} (t_j - 1)$</td>
</tr>
<tr>
<td>7</td>
<td><code>i = i - 1</code></td>
<td>$c_7$</td>
<td>$\sum_{j=2}^{n} (t_j - 1)$</td>
</tr>
<tr>
<td>8</td>
<td><code>A[i + 1] = key</code></td>
<td>$c_8$</td>
<td>$n - 1$</td>
</tr>
</tbody>
</table>
**Insertion sort**

\[
\text{INSERTION-SORT}(A) \\
\text{1 for } j = 2 \text{ to } A\text{.length} \\
\text{2 key} = A[j] \\
\text{3 // Insert } A[j] \text{ into the sorted sequence } A[1 \ldots j - 1]. \\
\text{4 } i = j - 1 \\
\text{5 while } i > 0 \text{ and } A[i] > key \\
\text{6 } A[i + 1] = A[i] \\
\text{7 } i = i - 1 \\
\text{8 } A[i + 1] = key
\]

\[
T(n) = c_1n + c_2(n - 1) + c_4(n - 1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) \\
+ c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n - 1).
\]
Insertion sort

Best case

**Insertion-Sort**(A)

1. for *j* = 2 to *A*.length
2.  
3. // Insert *A*[j] into the sorted sequence *A*[1..*j* − 1].
4.  
5. while *i* > 0 and *A*[i] > key
6.  
7.  
8. *A*[i + 1] = key

<table>
<thead>
<tr>
<th>cost</th>
<th>times</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>c</em>₁</td>
<td><em>n</em></td>
</tr>
<tr>
<td><em>c</em>₂</td>
<td><em>n</em> − 1</td>
</tr>
<tr>
<td><em>c</em>₄</td>
<td><em>n</em> − 1</td>
</tr>
<tr>
<td><em>c</em>₅</td>
<td>( \sum_{j=2}^{n} t_j )</td>
</tr>
<tr>
<td><em>c</em>₆</td>
<td>( \sum_{j=2}^{n} (t_j - 1) )</td>
</tr>
<tr>
<td><em>c</em>₇</td>
<td>( \sum_{j=2}^{n} (t_j - 1) )</td>
</tr>
<tr>
<td><em>c</em>₈</td>
<td><em>n</em> − 1</td>
</tr>
</tbody>
</table>

\[
T(n) = c_1 n + c_2 (n - 1) + c_4 (n - 1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8 (n - 1)
\]
Insertion sort

Best case
Insertion sort

**Best case**

The best case for the Insertion Sort algorithm occurs when the input array is already sorted. In this case, the algorithm will perform a minimum number of comparisons and moves, resulting in the best possible time complexity.

### Insertion-Sort(A)

1. for $j = 2$ to $A.length$
2.   key = $A[j]$
4.   $i = j - 1$
5.   while $i > 0$ and $A[i] > key$
7.     $i = i - 1$
8.   $A[i + 1] = key$

### Time Complexity

The time complexity $T(n)$ for the best case can be expressed as:

$$T(n) = c_1n + c_2(n - 1) + c_4(n - 1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) + \sum_{j=2}^{n} (t_j - 1) + c_8(n - 1)$$

To simplify, we can use the best case assumption and set $t_j = 1$ for all $j$.

Thus, the simplified time complexity is:

$$T(n) = c_1n + c_2(n - 1) + c_4(n - 1) + c_5n + c_6(n - 1)(n - 1) + c_7(n - 1) + c_8(n - 1)$$

Simplifying further:

$$T(n) = (c_1 + c_2 - c_3 + c_4 + c_5 + c_6 + c_7 + c_8)n - c_2 - c_4 - c_5 - c_6 - c_7 - c_8$$

For the best case, $T(n)$ simplifies to:

$$T(n) = (c_1 + c_2 - c_3 + c_4 + c_5 + c_6 + c_7 + c_8)n - (c_2 + c_4 + c_5 + c_6 + c_7 + c_8)$$

This expression shows that the best case time complexity is linear, with a constant term that depends on the coefficients $c_i$.
Insertion sort

**Best case**

\[
T(n) = c_1 n + c_2 (n - 1) + c_4 (n - 1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) \\
+ c_7 \sum_{j=2}^{n} (t_j - 1) + c_8 (n - 1). 
\]

\[
T(n) = c_1 n + c_2 (n - 1) + c_4 (n - 1) + c_5 (n - 1) + c_8 (n - 1) \\
= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8). 
\]

We can express this running time as \( an + b \) for constants \( a \) and \( b \) that depend on the statement costs \( c_i \); it is thus a **linear function** of \( n \).
Insertion sort

Best case

\[
T(n) = c_1 n + c_2 (n - 1) + c_4 (n - 1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) \\
+ c_7 \sum_{j=2}^{n} (t_j - 1) + c_8 (n - 1).
\]

\[
T(n) = c_1 n + c_2 (n - 1) + c_4 (n - 1) + c_5 (n - 1) + c_8 (n - 1)
\]
\[
= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8).
\]

We can express this running time as \( an + b \) for constants \( a \) and \( b \) that depend on the statement costs \( c_i \); it is thus a \textit{linear function} of \( n \).
Insertion sort

Best case

\[
T(n) = c_1n + c_2(n - 1) + c_4(n - 1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n - 1).
\]

We can express this running time as \( an + b \) for constants \( a \) and \( b \) that depend on the statement costs \( c_i \); it is thus a \textit{linear function} of \( n \).
Insertion sort

Worst case

\[ T(n) = c_1 n + c_2 (n - 1) + c_4 (n - 1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) \]

\[ + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8 (n - 1) \]
Insertion sort

Worst case

\[ T(n) = c_1 n + c_2 (n - 1) + c_4 (n - 1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) \]

\[ + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8 (n - 1). \]

Useful identities:

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

\[ \sum_{j=2}^{n} (j - 1) = \frac{n(n - 1)}{2} \]
Worst case
\[
T(n) = c_1 n + c_2 (n - 1) + c_4 (n - 1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8 (n - 1).
\]

Useful identities:
\[
\sum_{j=2}^{n} j = \frac{n(n + 1)}{2} - 1 \quad \sum_{j=2}^{n} (j - 1) = \frac{n(n - 1)}{2}
\]
Worst-case and average-case analysis

even for a fixed input.

The worst-case running time of an algorithm gives us an upper bound on the worst-case running time will often occur when the information is not present in the database.

For some algorithms, the worst case occurs fairly often. For example, in search:

In some applications, searches for absent information may be frequent.

We can express this worst-case running time as

\[
T(n) = c_1 n + c_2 (n-1) + c_4 (n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8 (n-1)
\]

Useful identities:

\[
\sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1 \quad \sum_{j=2}^{n} (j-1) = \frac{n(n-1)}{2}
\]
Insertion sort

Worst case

\[ T(n) = c_1 n + c_2 (n - 1) + c_4 (n - 1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8 (n - 1) \]

\[ \left( \frac{n(n + 1)}{2} \right) - 1 \]

\[ \left( \frac{n(n - 1)}{2} \right) \]

Useful identities:

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

\[ \sum_{j=2}^{n} (j - 1) = \frac{n(n - 1)}{2} \]
We can express this worst-case running time as \( an^2 + bn + c \) for constants \( a, b, \) and \( c \) that again depend on the statement costs \( c_i \); it is thus a **quadratic function** of \( n \).
Worst case vs. Average case

- Average case is often as bad as the worst case.
- When analyzing algorithms, we will mostly focus on the worst case.
Rate of growth: Example functions that often appear in algorithm analysis

Constant $\approx 1$
Logarithmic $\approx \log n$
Linear $\approx n$
N-Log-N $\approx n \log n$
Quadratic $\approx n^2$
Cubic $\approx n^3$
Exponential $\approx 2^n$
**Rate of growth:** consider only the leading term

Running time of \( an^2 + bn + c \)

running time of \( \Theta(n^2) \) (pronounced “theta of \( n \)-squared”)

for large enough inputs, a \( \Theta(n^2) \) algorithm, for example, will run more quickly in the worst case than a \( \Theta(n^3) \) algorithm.