Advanced Data Structures and Algorithms

CS 361 – Fall 2013

Lec. #02: Introduction to Algorithms

Tamer Nadeem
Dept. of Computer Science
Class Objective/Overview

• **Main Objective:** Start thinking about designing and analyzing algorithms.

• Understand *simple sorting and searching techniques* that will be used as examples.

• Understand *running-time complexity/analysis* of an algorithm using *Big-O notation*

• Apply *Big-O analysis* on simple sorting/searching algorithms

• Understand *function template* syntax

• Understand and use of *recursion*. 
Simple Sorting & Searching Techniques
The Problem of Sorting

**Input:** Sequence \( \langle a_1, a_2, \ldots, a_n \rangle \) of numbers.

**Output:** Permutation \( \langle a'_1, a'_2, \ldots, a'_n \rangle \) such that \( a'_1 \leq a'_2 \leq \ldots \leq a'_n \).

Example:

**Input:** 8 2 4 9 3 6

**Output:** 2 3 4 6 8 9
Insertion Sort


\[
\begin{array}{ccccccccc}
1 & 5 & 7 & 10 & 12 & 18 & 9 & 100 & 200 \\
1 & 5 & 7 & 10 & 12 & 18 & 100 & 200 \\
1 & 5 & 7 & 9 & 10 & 12 & 18 & 100 & 200 \\
\end{array}
\]
Example of Insertion Sort

8 2 4 9 3 6
Example of Insertion Sort

8 2 4 9 3 6
Example of Insertion Sort
Example of Insertion Sort

8 2 4 9 3 6

2 8 4 9 3 6
Example of Insertion Sort

2 8 4 9 3 6

2 8 4 9 3 6

2 4 8 9 3 6
Example of Insertion Sort

8 2 4 9 3 6
2 8 4 9 3 6
2 4 8 9 3 6
Example of Insertion Sort

\[
\begin{array}{cccccc}
8 & 2 & 4 & 9 & 3 & 6 \\
2 & 8 & 4 & 9 & 3 & 6 \\
2 & 4 & 8 & 9 & 3 & 6 \\
2 & 4 & 8 & 9 & 3 & 6 \\
\end{array}
\]
Example of Insertion Sort

8 2 4 9 3 6
2 8 4 9 3 6
2 4 8 9 3 6
2 4 8 9 3 6
Example of Insertion Sort

8  2  4  9  3  6
2  8  4  9  3  6
2  4  8  9  3  6
2  4  8  9  3  6
2  3  4  8  9  6
### Example of Insertion Sort

<table>
<thead>
<tr>
<th>8</th>
<th>2</th>
<th>4</th>
<th>9</th>
<th>3</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>8</td>
<td>4</td>
<td>9</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>8</td>
<td>9</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>8</td>
<td>9</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>4</td>
<td>8</td>
<td>9</td>
<td>6</td>
</tr>
</tbody>
</table>
Example of Insertion Sort

8 2 4 9 3 6
2 8 4 9 3 6
2 4 8 9 3 6
2 4 8 9 3 6
2 3 4 8 9 6
2 3 4 6 8 9 done
Algorithm of Insertion Sort

```
INSERTION-SORT (A, n) //input: A[1...n]
    for j ← 2 to n //outer loop
        do key ← A[j]
            i ← j - 1
        while i > 0 and A[i] > key //inner loop
            do { A[i+1] ← A[i]
                i ← i - 1
            } 
        A[i+1] = key
```

A:

```
  1
  i
  j
  n
```

sorted

key
Example of Insertion Sort

outer loop

inner loop

done
Code of Insertion Sort

```c
void insertionSort (int *a , int size)
{
    int i , j , n = size;
    int key;
    // place a[i] into the sublist
    // a[0] . . . a[i-1], 1 <= i < n,
    // so it is in the correct position
    for (int j=1; j<n; ++j)           // outer loop
    {
        key = a[j];
        i=j-1;
        while ((i > 0) && (key < a[i])) // inner loop
        {
            a[i+1] = a[i];
            i = i - 1;
        }
        a[i+1] = key;
    }
}
```
Comments on Insertion Sort

- If the key value is greater than all the sorted elements in the sublist, the algorithm does 0 iterations of the inner loop.

- If the initial array is sorted:
  - Then each new key will be greater than all the ones already inserted into the array.
  - For each outer loop iteration, the algorithm does 0 iterations of the inner loop.

- This special case is very common. Many practical problems require the construction of a sorted sequence of elements from a set of data that is already sorted or nearly so, with only a few items out of place.

- Note the "work from the back" is very efficient for such inputs.
Analysis of Algorithms
Algorithm Complexity

- A code of an algorithm is judged by its correctness, its ease of use, and its efficiency.
- This course focuses on the computational complexity (time efficiency) of algorithms that apply to container objects (data structures) that hold a large collection of data.
- We will learn how to develop measures of efficiency (criteria) that depend on \( n \), the number of data items in the container.
- The criteria of computational complexity often include the number of comparison tests and the number of assignment statements used by the algorithm.
Running Time

• The term *running time* is often used to refer to the computational complexity (*how fast the algorithm*).

• The running time *depends on the input* (e.g., an already sorted sequence is easier to sort).
  - *Parameterize* the running time by the *size of the input n*
  - *Seek upper bounds on the running time* $T(n)$ for the input size $n$, because everybody likes a guarantee.
  - $T(n)$ function *counts the frequency of the key operations* in terms of $n$. 
Brute Force Timing Analysis

The total run time, \( T(N) \), for this algorithm is:

\[
T(N) = (2N^2 + 3N + 1) * t_{\text{asst}} + (N^2 + 2N + 1) * t_{\text{comp}} + (4N^2 + 2N) * t_{\text{add}} + (3N^2 + N) * t_{\text{mult}}
\]

\[= c_1 N^2 + c_2 N + c_3\]
Analysis Rules for Program Statements

Individual Statements

1. Expressions and Assignments
   - The complexity of an expression is the sum of the complexity of all of the operations within it.
   - Assignment statements have a complexity equal to the sum of the complexities of the expressions to either side of the “=” operator, plus the complexity of the actual copy.

2. Function Calls
   - Function/Procedure calls are counted as the complexity of their bodies.

3. Loops
   - Compute the run time by adding up the time required for all the iterations.
Analysis Rules for Program Statements

Individual Statements

3. Loops

- Compute the run time by adding up the time required for all the iterations.
- The running time of a loop is at most:

\[
 t_{\text{loop}} \in O \left( t_{\text{init}} + \sum_{\text{iterations}} (t_{\text{condition}} + t_{\text{increment}} + t_{\text{body}}) + t_{\text{final condition}} \right)
\]

where \( t_{\text{init}} \) is the time required to do the loop initialization, \( t_{\text{condition}} \) is the time to evaluate the loop condition, with \( t_{\text{final condition}} \) being the time to evaluate the loop condition the final time (when we exit the loop), and \( t_{\text{body}} \) is the time required to do the loop body.

Example: A simple loop

```c
for ( int j = 0; j < n; ++j )
a[ i + 10* j ] = i + j ;
```

\[
 t_{\text{loop}} = O(c_{\text{init}} + \sum (c_{\text{condition}} + c_{\text{increment}} + c_{\text{body}}) + c_{\text{condition}})
 = O(c_0 + n \cdot c_1)
 = O(n)
\]
3. **Conditional Statement**

- The worst case time for the if is the slower of the two possibilities.
- The running time of an if-then-else is at most:

  \[ t_{if} \in O(t_{condition} + \max(t_{then}, t_{else})) \]

  where \( t_{condition} \) is the time to evaluate the if condition, with \( t_{then} \) is the time to do the ‘then’ body, and \( t_{else} \) is the time required to do the ‘else’ body.

- A missing else clause (or, for that matter, any “empty” statement list) is \( O(1) \).
Analysis Rules for Program Statements

Combining Statements

1. Sequences of Statements
   - The time for consecutive (straight-line) statements is the sum of the individual statements.

2. Nested Statements
   - In general, nested statements are evaluated by applying the other rules from the inside out.

Example: A simple nested loop

```c
for ( int i = 0; i < n; ++i )
{
    for ( int j = i; j < n; ++j )
    {
        a[i][j] = a[j][i];
    }
}
```

Analysis:

$$t_{\text{inner loop}} \in O \left( c_{\text{init}} + \sum_{\text{iterations}} \left( c_{\text{condition}} + c_{\text{increment}} + c_{\text{body}} \right) + c_{\text{condition}} \right)$$

\[
= O \left( c_0 + \sum_{j=t}^{n-1} (c_1) \right) \\
= O(c_0 + (n - i)c_1) \\
= O(n - i)
\]
Analysis Rules for Program Statements

Combining Statements

2. Nested Statements (cont’d)

• In general, nested statements are evaluated by applying the other rules from the inside out.

Example: A simple nested loop

```c
for ( int i = 0; i < n; ++i )
{
    for ( int j = i; j < n; ++j )
    {
        a[i][j] = a[j][i];
    }
}
```

\[
\begin{align*}
t_{outerloop} & \in O\left(c_{init} + \sum_{i=0}^{n-1} (c_{condition} + c_{increment} + t_{innerloop}) + c_{condition}\right) \\
& = O\left(c_2 + \sum_{i=0}^{n-1} (c_3 + c_4(n-i))\right) \\
& = O\left(c_2 + \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} c_4(n-i)\right) \\
& = O\left(c_5 + c_4 \sum_{i=0}^{n-1} (n-i)\right) \\
& = O\left(c_5 + c_4 \sum_{i=0}^{n-1} n - c_4 \sum_{i=0}^{n-1} i\right) \\
& = O\left(c_5 + c_4 n^2 - c_4 \frac{n(n-1)}{2}\right) \\
& = O(c_5 + c_4 n^2 - 0.5c_4 n^2 + 0.5c_4 n) \\
& = O(c_5 + 0.5c_4 n^2 + 0.5c_4 n) \\
& = O(c_5 + c_6 n^2 + c_6 n) \\
& = O(c_6 n^2) \\
& = O(n^2)
\end{align*}
\]
Machine-independent time

• What is insertion sort’s worst-case time?
  • It depends on the speed of our computer:
    • relative speed (on the same machine),
    • absolute speed (on different machines).

• BIG IDEA:
  • Ignore machine-dependent constants.
  • Look at growth of $T(n)$ as $n \to \infty$

“Asymptotic Analysis”
Worst Case Analysis

Definition:
We say that an algorithm requires time proportional to \( f(n) \) if there are constants \( c \) and \( n_0 \) such that the algorithm requires no more than \( c \cdot f(n) \) time units to process an input set of size \( n \) whenever \( n \geq n_0 \).

- The \( f(n) \) here describes the rate at which the time required by this algorithm goes up as you change the size of the input for particular program or algorithm.

- The multiplier \( c \) is used so that we can talk about the algorithm requiring no more than “\( c \cdot f(n) \) time units”.

- \( n_0 \) is used to place a lower limit on how small the inputs are we really worried about.
**O-notation**

**Math:**

- We say that $T(n)= O(g(n))$ iff there exists positive constants $c_1$, and $n_0$ such that
  
  $0 \leq T(n) \leq c_1 g(n)$ for all $n \geq n_0$

- Usually $T(n)$ is running time, and $n$ is size of input

**Engineering:**

- Drop low-order terms; ignore leading constants.
- Example: $3n^3 + 90n^2 – 5n + 6046 = O(n^3)$
**Ω-notation**

**Math:**

- We say that $T(n) = \Omega(g(n))$ iff there exists positive constants $c_2$, and $n_0$ such that $0 \leq c_2 g(n) \leq T(n)$ for all $n \geq n_0$.

**Engineering:**

- Drop low-order terms; ignore leading constants.
- Example: $3n^3 + 90n^2 - 5n + 6046 = \Omega(n^3)$
**O-notation (contd’)**

- So if \( T(n) = O(n^2) \) then we are also sure that
  \[
  T(n) = O(n^3) \quad \text{and that} \\
  T(n) = O(n^{3.5}) \quad \text{and} \\
  T(n) = O(2^n)
  \]

- But it might or might not be true that \( T(n) = O(n^{1.5}) \).

- However, if \( T(n) = \Omega(n^2) \) then it is not true that
  \[
  T(n) = O(n^{1.5})
  \]
**Θ-notation**

**Math:**

- We say that $T(n) = \Theta(g(n))$ iff
  
  there exist positive constants $c_1$, $c_2$, and $n_0$ such that
  
  $$0 \leq c_1 g(n) \leq T(n) \leq c_2 g(n) \text{ for all } n \geq n_0$$

  In other words
  
  $$T(n) = \Theta(g(n)) \text{ iff } T(n) = O(g(n)) \text{ and } T(n) = \Omega(g(n))$$

**Engineering:**

- Drop low-order terms; ignore leading constants.
- Example: $3n^3 + 90n^2 - 5n + 6046 = \Theta(n^3)$
**Asymptotic Performance**

- When $n$ gets large enough, a $\Theta(n^2)$ algorithm always beats a $\Theta(n^3)$ algorithm.

- We shouldn’t ignore asymptotically slower algorithms, however.

- Real-world design situations often call for a careful balancing of engineering objectives.

- Asymptotic analysis is a useful tool to help to structure our thinking.
Detailed Timing Example Revisited

1. The total run time, \( T(N) = c_1 N^2 + c_2 N + c_3 \)

2. For \( n_0 = 1 \), it is clear that
   \[
   c_1 N^2 + c_2 N + c_3 \leq c_1 N^2 + c_2 N^2 + c_3 N^2
   \]

3. Then, \( T(N) \leq c_1 N^2 + c_2 N^2 + c_3 N^2 \)

4. Let \( c = c_1 + c_2 + c_3 \), then \( T(N) \leq cN^2 \)

5. This means that \( T(N) \) is in order of \( O(N^2) \)
**Special Cases**

**Constant Time Algorithms:** An algorithm is $O(1)$ when its running time is independent of the number of data items. The algorithm runs in constant time.

- e.g., direct insert at rear of array involves a simple assignment statement and thus has efficiency $O(1)$

**Linear Time Algorithms:** An algorithm is $O(n)$ when its running time is proportional to the size of the list.

- e.g., sequential search. When the number of elements doubles, the number of operations doubles.
Special Cases

Exponential Algorithms:
• Algorithms with running time $O(n^2)$ are quadratic.
  • practical only for relatively small values of $n$.
  • Whenever $n$ doubles, the running time of the algorithm increases by a factor of 4.
• Algorithms with running time $O(n^3)$ are cubic.
  • efficiency is generally poor; doubling the size of $n$ increases the running time eight-fold.

Logarithmic Time Algorithms: The logarithm of $n$, base 2, is commonly used when analyzing computer algorithms
• Ex. $\log_2(2) = 1$, $\log_2(75) = 6.2288$
• When compared to the functions $n$ and $n^2$, the function $\log_2 n$ grows very slowly
Comparison of different $O()$

<table>
<thead>
<tr>
<th>$n$</th>
<th>$\log_2 n$</th>
<th>$n \log_2 n$</th>
<th>$n^2$</th>
<th>$n^3$</th>
<th>$2^n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>8</td>
<td>16</td>
<td>64</td>
<td>16</td>
</tr>
<tr>
<td>8</td>
<td>3</td>
<td>24</td>
<td>64</td>
<td>512</td>
<td>256</td>
</tr>
<tr>
<td>16</td>
<td>4</td>
<td>64</td>
<td>256</td>
<td>4096</td>
<td>65536</td>
</tr>
<tr>
<td>32</td>
<td>5</td>
<td>160</td>
<td>1024</td>
<td>32768</td>
<td>4294967296</td>
</tr>
<tr>
<td>128</td>
<td>7</td>
<td>896</td>
<td>16384</td>
<td>2097152</td>
<td>3.4 x 10^{38}</td>
</tr>
<tr>
<td>1024</td>
<td>10</td>
<td>10240</td>
<td>1048576</td>
<td>1073741824</td>
<td>1.8 x 10^{308}</td>
</tr>
<tr>
<td>65536</td>
<td>16</td>
<td>1048576</td>
<td>4294967296</td>
<td>2.8 x 10^{14}</td>
<td>Forget it!</td>
</tr>
</tbody>
</table>
The Algebra of Big-O

Algebraic Rule 1: $f(n) + g(n) \in O(f(n) + g(n))$

Algebraic Rule 2: $f(n) \times g(n) \in O(f(n) \times g(n))$

Algebraic Rule 3: $O(c \times f(n)) = O(f(n))$
Proof of Algebraic Rule 3: \( O(c \ast f(n)) = O(f(n)) \)

1. Given running time \( T(n) = O(c \ast f(n)) \)

2. By \( O() \) definition, \( \exists c_1, n_0 | n > n_0 \Rightarrow T(n) \leq c_1(c \ast f(n)) \)

3. \( \Rightarrow \exists c_1, n_0 | n > n_0 \Rightarrow T(n) \leq (c_1 \ast c) f(n) \)

4. Let \( c_2 = c_1 \ast c \Rightarrow \exists c_1, n_0 | n > n_0 \Rightarrow T(n) \leq c_2 \ast f(n) \)

5. \( \Rightarrow T(n) = O(f(n)) \)

6. \( \Rightarrow O(c \ast f(n)) = O(f(n)) \)

Similarly, \( O(c_1 \ast f(n) + c_2 \ast g(n)) = O(f(n) + g(n)) \)
Algebraic Rule 4: Larger Terms Dominate a Sum

\[
\text{if } \exists n_0 \mid \forall n > n_0, f(n) \geq g(n),
\]

\[
\text{then } O(f(n) + g(n)) = O(f(n))
\]
The Algebra of Big-O

Proof of Algebraic Rule 4: \( \forall n > n_0, \ f(n) \geq g(n), \)

then \( O(f(n) + g(n)) = O(f(n)) \)

1. Given running time \( T(n) = O(f(n) + g(n)) \)

2. By \( O() \) definition, \( \exists c, n_1 | n > n_1 \rightarrow T(n) \leq c(f(n) + g(n)) \)

3. Assume \( \forall n > 0,\ f(n) \geq g(n) \) then

\[
\begin{align*}
    n > \max(n_0, n_1) & \rightarrow T(n) \leq c \ (f(n) + g(n)) & \quad \ldots \ (1) \\
    n > \max(n_0, n_1) & \rightarrow f(n) > g(n) & \quad \ldots \ (2)
\end{align*}
\]

4. Using (2) to replacing \( g(n) \) by \( f(n) \) in (1) then

\[
\begin{align*}
    n > \max(n_0, n_1) & \rightarrow T(n) \leq c \ (f(n) + f(n)) & \quad \ldots \ (3) \\
    n > \max(n_0, n_1) & \rightarrow T(n) \leq 2c \ * \ f(n) & \quad \ldots \ (4)
\end{align*}
\]

5. \( \rightarrow \)

\( T(n) = O(f(n)) \)
Algebraic Rule 5: Logarithms are Fast

\[ \forall k \geq 0, \; O(\log^k(n)) \subseteq O(n) \]
Analysis of Sorting/Searching Algorithms
Algorithm of Insertion Sort

```
Algorithm: Insertion-Sort

\[ A_1, A_2, \ldots, A_n \]  //input:  \[ A[1 \ldots n] \]

\[ \text{for } j \leftarrow 2 \text{ to } n \]  //outer loop

\[ \text{do } key \leftarrow A[j] \]

\[ i \leftarrow j - 1 \]

\[ \text{while } i > 0 \text{ and } A[i] > key \]  //inner loop

\[ \text{do } \{ A[i+1] \leftarrow A[i] \]

\[ i \leftarrow i - 1 \}

\[ A[i+1] = key \]
```

**Pseudocode**

```
"pseudocode"
```

```
A: 1 i j n

sorted

key
```
Insertion sort analysis

• So if $T(n) = O(n^2)$ then we are also sure that

$$T(n) = O(n^3)$$ and that

$$T(n) = O(n^{3.5})$$ and

$$T(n) = O(2^n)$$

• But it might or might not be true that $T(n) = O(n^{1.5})$.

• However, if $T(n) = \Omega(n^2)$ then it is not true that

$$T(n) = O(n^{1.5})$$
Sequential Search

• Search algorithm start with a target value and employ sequential visit to the elements looking for a match.
  • If target is found, the index of the matching element becomes the return value.

```cpp
int seqSearch (const int arr[ ], int first, int last, int target)
// Look for target within a sorted array arr [first . . last -1].
// Return the position where found, or last if not found .
{
    int i = first;
    while (i < last && arr[i] != target )
        i++;
    return i ;
}
```

Index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8
--- | --- | --- | --- | --- | --- | --- | --- | --- | ---
6 | 4 | 2 | 9 | 5 | 3 | 10 | 7 | match at index = 5

match at index = 5
return index 5
Sequential Search for a Sorted Array

- Previous code for any general array (i.e., non sorted).

- When arrays have been sorted, they could be searched using a slightly faster variant of the sequential sort.

- If array is sorted, then encountering a value greater than the target would imply that the target value is not in the array.
  - Replacing the `!=` operator by a `<` causes the algorithm to exit the loop as soon as we see a value that is equal to or greater than the target.

```c
int seqSearch (const int arr[ ], int first, int last, int target)
// Look for target within a sorted array arr [first . . last -1].
// Return the position where found, or last if not found.
{
    int i = first;
    while (i < last && arr[i] < target )
        i++;
    return i ;
}
```
Binary Search

• An alternative method of searching sorted arrays is the binary search

Example: Search for number 85 in the 63-entry list

<table>
<thead>
<tr>
<th>First</th>
<th>Last</th>
<th>Middle</th>
<th>Entry</th>
<th>Outcome</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>63</td>
<td>32</td>
<td>71</td>
<td>&gt;</td>
</tr>
<tr>
<td>33</td>
<td>63</td>
<td>48</td>
<td>102</td>
<td>&lt;</td>
</tr>
<tr>
<td>33</td>
<td>47</td>
<td>40</td>
<td>87</td>
<td>&lt;</td>
</tr>
<tr>
<td>33</td>
<td>39</td>
<td>36</td>
<td>80</td>
<td>&gt;</td>
</tr>
<tr>
<td>37</td>
<td>39</td>
<td>38</td>
<td>83</td>
<td>&gt;</td>
</tr>
<tr>
<td>37</td>
<td>37</td>
<td>37</td>
<td>85</td>
<td>=</td>
</tr>
</tbody>
</table>

Six probes are needed with a 63-entry list, in the worst case
Code of Binary Search

```c
int binSearch (const int arr[ ], int first, int last, int target)
// Look for target within a sorted array arr [first . . last -1].
// Return the position where found, or , or last if not found
{
    int mid;   // index of the midpoint
    int midValue ; // object that is assigned arr[mid]
    int origLast = last ;  // save original value of last
    // repeatedly reduce the area of search
    // until it is just one element
    while (first < last) {
        // test for nonempty sublist
        mid = (first + last) / 2;
        midValue = arr[mid] ;
        if (target == midValue)
            return mid;
        else if (target < midValue)
            last = mid;  // search lower sublist , reset last
        else
            first = mid+1;  // search upper sublist , reset first
    }
    return origLast;
}
```
Binary Search Tree

Example 1:
Find 47

Example 2:
Find 112

63-item list

1 2 49 91
5 47 93 99
6 57 99 100
8 59 101
12 64 102
16 66 105
17 67 107
18 69 110
21 70 111
23 71 116
24 74 117
30 75 118
32 77 120
33 80 122
35 81 125
38 83 126
40 85 128
44 87 130
45 88 131
47 90 133
54 91
66 93
71 99
110
128
131
133

Not found

Found
Making Algorithms Generic
Generalize the Element Type

- Let’s go back to some of the array manipulation functions we developed earlier.
- Our `binSearch` routine operates on arrays of `int`. What to do if we want to have a binary search over arrays of double, or of string?

```cpp
int binSearch (const int arr[ ], int first, int last, int target)  
// Look for target within a sorted array arr [first . . last -1].  
// Return the position where found, or , or last if not found
{
    int mid;   // index of the midpoint
    int midValue;   // object that is assigned arr[mid]
    int origLast = last;   // save original value of last
    . . .
    return origLast;
}
```

```cpp
typedef int T;
#include "arrayops .h"  // T == int
importantNumbers [ 100 ] zn;
.. orderedInsert ( importantNumbers , 0 , n , 42 );
```
Generalize the Element Type

• One day, discover that we need to manipulate an array of int \textit{and} an array of some other type (e.g., int)

\texttt{Mian.cc}

\begin{verbatim}
typedef int T;
#include "arrayops.h" // T == int
typedef std::string T;
#include "arrayops.h" // T == std::string
int importantNumbers[100];
std::string favoriteNames[100];
in
...  
sequential Insert (importantNumbers, n1, 42);
sequential Insert (favoriteNames, n2, "Arthur Dent");
\end{verbatim}

• We will get compilation errors, this time at the second `typedef`, because \textit{we can’t define T to mean two different things at the same time}. 

Page 56 Fall 2013 CS 361 - Advanced Data Structures and Algorithms
Generalize the Element Type

• we could simply make a distinct copy of binSearch for each kind of array, using an ordinary text editor to replace T by a “real” type name.

• Doing this in a large program does get to be a bit of a project management headache.

• Since it is a common situation, and it does involve a fair amount of work, some programming language designers (e.g., C++) eventually found a way to do the same thing automatically; use of Templates.
Templates in C++

• *Templates* describe common “patterns” for similar classes and functions that differ only in a few names.

• Templates come in two varieties: Class templates, and Function templates

• *Class templates* are patterns for similar classes. *Function templates* are patterns for similar function

• A C++ *template* is a kind of pattern for code in which certain names that, like T in the earlier example, we intend to replace by something “real” when we use the pattern. These names are called *template parameters*.

• The *compiler instantiates* (creates an instance of) a template by *filling in the appropriate replacement* for these template parameter names, thereby *generating the actual code* for a function or class, and compiling that instantiated code.
Writing Function Templates

• We define a *template header* for each function to tell the compiler which *names* in the “pattern” are to be replaced.

• The template header indicates that this is a “pattern” for a function in which certain *type names* and *constants* are left to be *filled in later*.

• The header begins with the key word “*template*”. After that, inside the “< >”, is a list of all type (class) names to be replaced when we instantiate the template.

• Bodies of function templates are defined in header files (.h)
Inside the header file (binsearch.h file):

```cpp
/* * * * * * * binsearch.h * * * * * */
#ifndef BINSEARCH_H
#define BINSEARCH_H

template <typename T>            //or template <class T>
int binSearch (const T arr[ ], int first, int last, T target)
// Look for target within a sorted array arr[first .. last -1].
// Return the position where found, or , or last if not found
{
    int mid;   // index of the midpoint
    T midValue ;  // object that is assigned arr[mid]
    int origLast = last ;  // save original value of last
    // repeatedly reduce the area of search
    // until it is just one element
    while (first < last) {
        // test for nonempty sublist
        .. . . .
        .. . . .
    }
    return origLast;
}
#endif
```
Using Function Templates

• We instantiate a function template when we try to use it:

  ```
  #include ”binsearch.h”
  int importantNumbers [100] ;
  std::string favoriteNames [100] ;
  int n1, n2 ;
  int found;
  ...  ...
  n1  = binSearch(importantNumbers, 0, 100, 42) ;
  n2  = binSearch(favoriteNames , 0, 100, ”Arthur Den”);
  ...
  ```

• The compiler deduces from the calls that it must use the binSearch template to produce the functions array elements).

  ```
  int binSearch (int arr[], int first, int last, int target); and
  int binSearch (std::string arr[], int first, int last, std::string target);
  ```

• It does this simply by replacing T in the template by int and std::string, respectively.

• We can do this for any T that supports assignment and < operators.
Function Templates and the C++ std Library

• The C++ std library has a number of useful templates for small, common programming idioms.

• swap Function

```cpp
template <typename T>
inline void swap(T& a, T& b) {
    T tmp = a;
    a = b;
    b = tmp;
}
```

• min & max Functions

```cpp
template <typename T>
inline const T& min(const T& a, const T& b) {
    return b < a ? b : a ;
}

template <typename T>
inline const T& max(const T& a, const T& b) {
    return b < a ? a : b;
}
```

• relops Function

```cpp
namespace relops {

    template <typename T>
    inline bool operator!= (const T& a, const T& b){
        return !(a == b);
    }

template <typename T>
inline bool operator> (const T& a, const T& b){
    return b < a;
}

template <typename T>
inline bool operator<= (const T& a, const T& b){
    return !(b < a);
}

template <typename T>
inline bool operator>= (const T& a, const T& b){
    return !(a < b) ;
}
}
```
Recurssion
The Concept of Recursion

• Let’s consider the problem of evaluating the power \( x^n \) where \( x \) is a real number and \( n \) is a nonnegative integer.

• Another approach, since \( x^n = x^m \times x^{(n-m)} \), we can split the problem into smaller problems. For example, \( 2^{15} \) could be computed as \( 2^5 \times 2^{10} = 32 \times 1024 \).

• Recursion, in simple, is solving a problem by solving smaller problems of “same” form and using their results to compute the final result.

• Recursion is the process a function goes through when one of the steps of the function involves invoking the function itself. A procedure that goes through recursion is said to be 'recursive'.

Iterative Approach

```c
Double power(double x, int n){
    double product = 1;
    int i;
    for (i = 1; i <= n; i++)
        product *= x;  //\( x^n = x \times \ldots \times x \) (n times)
    return product;
}
```
The Concept of Recursion

- A function exhibits recursive behavior when it can be defined by two properties:
  - A simple base case (or cases) (stopping condition)
  - A set of rules that reduce all other cases toward the base case

- Example: Computing \( x^n \)
  - Base case: \( x^0 = 1 \)
  - Rule: \( x^n = x^{n-1} \times x \)

Recursive Approach

```java
Double power(double x, int n) {
    if (n == 0)
        return 1;
    else
        return x * power(x, n-1);
}
```

- Example: Computing \( x! \)
  - Base case: \( 0! = 1 \)
  - Rule: \( x! = x \times (x-1)! \)

Recursive Approach

```java
Double factorial(int x) {
    if (x == 0)
        return 1;
    else
        return x * factorial(x-1);
}
```
Fibonacci Numbers

- Fibonacci numbers are the sequence of integers 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...

- Example: Computing fib(n) //Fibonacci element with index n
  - Base case: fib(0) = 0
    fib(1) = 1
  - Rule: fib(n) = fib(n-1) + fib(n-2)

Recursive Approach

```c
int fib(int n) {
    if (n <= 1)
        return n;
    else
        return fib(n-1) + fib(n-2);
}
```

Iterative Approach

```c
int fibiter(int n) {
    int oneback = 1, twoback = 0, current, i;
    if (n <= 1)
        return n;
    else
        for (i=2; i <= n; i++){
            current = oneback + twoback;
            twoback = oneback;
            oneback = current;
        }
    return current;
}
```
Evaluating Fibonacci Numbers

```cpp
#include <iostream>
#include "d_timer.h"

int fibiter(int n) { // from previous slide
    // ... 
}

int fib(int n) { // from previous slide
    // ... 
}

int main() {
    timer t1, t2;
    t1.start();
    fibiter(45);
    t1.stop();
    t2.start();
    fib(45);
    t2.stop();

    cout << Time required by Iterative version is " << t1.time() << " sec" << endl;
    cout << Time required by Recursive version is " << t2.time() << " sec" << endl;
    return 0;
}
```
Tower of Hanoi
Tower of Hanoi

1. Needle A
   Needle B
   Needle C

2. Needle A
   Needle B
   Needle C

3. Needle A
   Needle B
   Needle C

4. Needle A
   Needle B
   Needle C

5. Needle A
   Needle B
   Needle C

6. Needle A
   Needle B
   Needle C

7. Needle A
   Needle B
   Needle C
Questions?