CS 4407
Algorithms

Lecture 3:
Non-recursive (Iterative) Algorithms

Prof. Gregory Provan
Department of Computer Science
University College Cork
Iterative Algorithms

- Technique for proving correctness and complexity of an algorithm
  - Loop invariant
- Steps for loop invariance approach
- Use of loop invariance
Today’s Learning Objectives

- Describe a framework that
  - leads to mathematically sound representations of an algorithm
  - Promotes quick implementation

- Understand Loop invariance technique
  - proving correctness and complexity of an algorithm
Two Key Types of Algorithms

- Iterative Algorithms
- Recursive Algorithms
Iterative Algorithms

Take one step at a time towards the final destination

loop (until done)
  take step
end loop

Isn’t it funny
How a bear likes honey?
Buzz! Buzz! Buzz!
I wonder why he does?
T(n), or the running time of a particular algorithm on input of size n, is taken to be the number of times the instructions in the algorithm are executed. Pseudo code algorithm illustrates the calculation of the mean (average) of a set of n numbers:

1. n = read input from user
2. sum = 0
3. i = 0
4. while i < n
5. number = read input from user
6. sum = sum + number
7. i = i + 1
8. mean = sum / n

The computing time for this algorithm in terms on input size n is: \( T(n) = 4n + 5 \).
Analysis of Simple Programs (no loops)

- Sum the “costs” of the lines of the program
- Compute the bounding function
  - e.g., O(*)
Example 1

Suppose \( f(n) = 5n \) and \( g(n) = n \).

- To show that \( f = O(g) \), we have to show the existence of a constant \( C \) as given earlier. Clearly 5 is such a constant so \( f(n) = 5 \times g(n) \).

- We could choose a larger \( C \) such as 6, because the definition states that \( f(n) \) must be less than or equal to \( C \times g(n) \), but we usually try and find the smallest one.

Therefore, a constant \( C \) exists (we only need one) and \( f = O(g) \).
Example 2

In the previous timing analysis, we ended up with $T(n) = 4n + 5$, and we concluded intuitively that $T(n) = O(n)$ because the running time grows linearly as $n$ grows. Now, however, we can prove it mathematically:

To show that $f(n) = 4n + 5 = O(n)$, we need to produce a constant $C$ such that:

$$f(n) \leq C \cdot n \text{ for all } n.$$ 

If we try $C = 4$, this doesn't work because $4n + 5$ is not less than $4n$. We need $C$ to be at least $9$ to cover all $n$. If $n = 1$, $C$ has to be $9$, but $C$ can be smaller for greater values of $n$ (if $n = 100$, $C$ can be $5$). Since the chosen $C$ must work for all $n$, we must use $9$:

$$4n + 5 \leq 4n + 5n = 9n$$

Since we have produced a constant $C$ that works for all $n$, we can conclude:

$$T(4n + 5) = O(n)$$
Example 3

Say $f(n) = n^2$: We will prove that $f(n) \neq O(n)$.

• To do this, we must show that there cannot exist a constant $C$ that satisfies the big-Oh definition. We will prove this by contradiction.

  Suppose there is a constant $C$ that works; then, by the definition of big-Oh: $n^2 \leq C \times n$ for all $n$.

• Suppose $n$ is any positive real number greater than $C$, then: $n \times n > C \times n$, or $n^2 > C \times n$.

So there exists a real number $n$ such that $n^2 > C \times n$.

This contradicts the supposition, so the supposition is false.

There is no $C$ that can work for all $n$:

\[ f(n) \neq O(n) \text{ when } f(n) = n^2 \]
Example 4

Suppose \( f(n) = n^2 + 3n - 1 \). We want to show that \( f(n) = O(n^2) \).

\[
\begin{align*}
f(n) &= n^2 + 3n - 1 \\
&< n^2 + 3n \quad \text{(subtraction makes things smaller so drop it)} \\
&\leq n^2 + 3n^2 \quad \text{(since } n \leq n^2 \text{ for all integers } n) \\
&= 4n^2
\end{align*}
\]

Therefore, if \( C = 4 \), we have shown that \( f(n) = O(n^2) \). Notice that all we are doing is finding a simple function that is an upper bound on the original function. Because of this, we could also say that

\[
f(n) = O(n^3) \text{ since } (n^3) \text{ is an upper bound on } n^2
\]

This would be a much weaker description, but it is still valid.
Big-Oh Operations

**Summation Rule**
Suppose $T_1(n) = O(f_1(n))$ and $T_2(n) = O(f_2(n))$. Further, suppose that $f_2$ grows no faster than $f_1$, i.e., $f_2(n) = O(f_1(n))$. Then, we can conclude that $T_1(n) + T_2(n) = O(f_1(n))$. More generally, the summation rule tells us $O(f_1(n) + f_2(n)) = O(\max(f_1(n), f_2(n)))$.

**Proof:**

Suppose that $C$ and $C'$ are constants such that $T_1(n) \leq C \cdot f_1(n)$ and $T_2(n) \leq C' \cdot f_2(n)$. Let $D = \text{the larger of } C \text{ and } C'$. Then,

$$T_1(n) + T_2(n) \leq C \cdot f_1(n) + C' \cdot f_2(n) \leq D \cdot f_1(n) + D \cdot f_2(n) \leq D \cdot (f_1(n) + f_2(n)) \leq O(f_1(n) + f_2(n))$$
**Product Rule**
Suppose $T_1(n) = O(f_1(n))$ and $T_2(n) = O(f_2(n))$. Then, we can conclude that $T_1(n) \cdot T_2(n) = O(f_1(n) \cdot f_2(n))$. The Product Rule can be proven using a similar strategy as the Summation Rule proof.

**Analyzing Some Simple Programs (with No Sub-program Calls)**

General Rules:

- All basic statements (assignments, reads, writes, conditional testing, library calls) run in constant time: $O(1)$.

- The time to execute a loop is the sum, over all times around the loop, of the time to execute all the statements in the loop, plus the time to evaluate the condition for termination. Evaluation of basic termination conditions is $O(1)$ in each iteration of the loop.

- The complexity of an algorithm is determined by the complexity of the most frequently executed statements. If one set of statements have a running time of $O(n^3)$ and the rest are $O(n)$, then the complexity of the algorithm is $O(n^3)$. This is a result of the Summation Rule.
Analysing Loops (Iterative)

- Many possible methods
- **Approach**
  - Define recursion
  - Use loop invariants to define recursion
Loop Invariants

A good way to structure many programs:

– Store the key information you currently know in some data structure.

– In the main loop,
  • take a step forward towards destination
  • by making a simple change to this data.
The Getting to School Problem
Algorithm

- The algorithm defines the computation route.
Complexity

- There are an infinite number of input instances.
- Algorithm must work for each.
Complexity

- Difficult to predict where computation might be in the middle of the computation.
A Measure of Progress

79 km to school

75 km to school
Loop Invariant

• “The computation is presently in a safe location.”

• Maybe true and maybe not.
Maintain Loop Invariant

- If the computation is in a safe location, it does not step into an unsafe one.
Maintain Loop Invariant

- If the computation is in a safe location, it does not step into an unsafe one.

- Can we be assured that the computation will always be in a safe location?
Maintain Loop Invariant

- If the computation is in a safe location, it does not step into an unsafe one.

- Can we be assured that the computation will always be in a safe location?

No. What if it is not initially true?
From the Pre-Conditions on the input instance we must establish the loop invariant.
Maintain Loop Invariant

- Can we be assured that the computation will always be in a safe location?
- By what principle?
Maintain Loop Invariant

- By **Induction** the computation will always be in a safe location.

\[
\Rightarrow S(0) \\
\Rightarrow \forall i S(i) \Rightarrow S(i+1)
\]
Ending The Algorithm

● Define Exit Condition

● Termination: With sufficient progress, the exit condition will be met.

● When we exit, we know
  – exit condition is true
  – loop invariant is true

from these we must establish the post conditions.
Let’s Recap
### Designing an Algorithm

<table>
<thead>
<tr>
<th>Define Problem</th>
<th>Define Loop Invariants</th>
<th>Define Measure of Progress</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1.png" alt="School" /></td>
<td><img src="image2.png" alt="House" /></td>
<td><img src="image3.png" alt="79 km to school" /></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Define Step</th>
<th>Define Exit Condition</th>
<th>Maintain Loop Inv</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image4.png" alt="People" /></td>
<td><img src="image5.png" alt="Exit" /></td>
<td><img src="image6.png" alt="Exit" /></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Make Progress</th>
<th>Initial Conditions</th>
<th>Ending</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image7.png" alt="People" /></td>
<td><img src="image8.png" alt="People" /></td>
<td><img src="image9.png" alt="People" /></td>
</tr>
</tbody>
</table>
Explaining Insertion Sort

We maintain a subset of elements sorted within a list. The remaining elements are off to the side somewhere. Initially, think of the first element in the array as a sorted list of length one. One at a time, we take one of the elements that is off to the side and we insert it into the sorted list where it belongs. This gives a sorted list that is one element longer than it was before. When the last element has been inserted, the array is completely sorted.
Explaining Selection Sort

We maintain that the k smallest of the elements are sorted in a list. The larger elements are in a set on the side. Initially, with k=0, all the elements are in this set. Progress is made by finding the smallest element in the remaining set of large elements and adding this selected element at the end of the sorted list of elements. This increases k by one. Stop with k=n. At this point, all the elements have been selected and the list is sorted.
Typical Loop Invariant

If the input consists of an array of objects

I have a solution for the first $i$ objects.

Extend the solution into a solution for the first $i+1$.

Done when solution for $n$
Typical Loop Invariant

If the output consists of an array of objects

I have an output produced for the first $i$ objects.

Produce the $i+1$-st output object.

Done when output $n$ objects.
Problem Specification

- Pre condition: location of home and school
- Post condition: Traveled from home to school
Example of Approach

- Binary search
  - Standard algorithm
Define Problem: Binary Search

- **PreConditions**
  - Key 25
  - Sorted List

- **PostConditions**
  - Find key in list (if there).
Define Loop Invariant

- Maintain a sublist
- Such that

| 3 | 5 | 6 | 13 | 18 | 21 | 21 | 25 | 36 | 43 | 49 | 51 | 53 | 60 | 72 | 74 | 83 | 88 | 91 | 95 |

key 25
Define Loop Invariant

- Maintain a sublist.
- If the key is contained in the original list, then the key is contained in the sublist.

key 25

3 5 6 13 18 21 21 25 36 43 49 51 53 60 72 74 83 88 91 95
Define Step

- Make Progress
- Maintain Loop Invariant

key 25

| 3 | 5 | 6 | 13 | 18 | 21 | 21 | 25 | 36 | 43 | 49 | 51 | 53 | 60 | 72 | 74 | 83 | 88 | 91 | 95 |
Define Step

- Cut sublist in half.
- Determine which half key would be in.
- Keep that half.

key 25

3 5 6 13 18 21 21 25 36 43 49 51 53 60 72 74 83 88 91 95
Define Step

- Cut sublist in half.
- Determine which half the key would be in.
- Keep that half.

If \( \text{key} \leq \text{mid} \), then key is in left half.
If \( \text{key} > \text{mid} \), then key is in right half.
Define Step

- It is faster not to check if the middle element is the key.
- Simply continue.

If $\text{key} \leq \text{mid}$, then key is in left half.
If $\text{key} > \text{mid}$, then key is in right half.
Make Progress

- The size of the list becomes smaller.

| 3 | 5 | 6 | 13 | 18 | 21 | 21 | 25 | 36 | 43 | 49 | 51 | 53 | 60 | 72 | 74 | 83 | 88 | 91 | 95 |

79 km

75 km
Initial Conditions

key 25

- The sublist is the entire original list.
- If the key is contained in the original list, then the key is contained in the sublist.
**Ending Algorithm**

| 3 | 5 | 6 | 13 | 18 | 21 | 21 | 25 | 36 | 43 | 49 | 51 | 53 | 60 | 72 | 74 | 83 | 88 | 91 | 95 |

- If the key is contained in the original list, then the key is contained in the sublist.
- Sublist contains one element.

If the key is contained in the original list, then the key is at this location.
If key not in original list

- If the key is contained in the original list, then the key is contained in the sublist.

| 3 | 5 | 6 | 13 | 18 | 21 | 21 | 25 | 36 | 43 | 49 | 51 | 53 | 60 | 72 | 74 | 83 | 88 | 91 | 95 |

- If the key is contained in the original list, then the key is at this location.

- Loop invariant true, even if the key is not in the list.

- Conclusion still solves the problem. Simply check this one location for the key.

key 24
Running Time

The sublist is of size $n, n/2, n/4, n/8, \ldots, 1$
Each step $\Theta(1)$ time.
Total = $\Theta(\log n)$

If $\text{key} \leq \text{mid}$, then key is in left half.

If $\text{key} > \text{mid}$, then key is in right half.
algorithm BinarySearch ((L(1..n), key))

⟨pre-cond⟩: ⟨L(1..n), key⟩ is a sorted list and key is an element.

⟨post-cond⟩: If the key is in the list, then the output consists of an index i such that L(i) = key.

begin
    i = 1, j = n
loop

    ⟨loop-invariant⟩: If the key is contained in L(1..n), then the key is contained in the sublist L(i..j).

    exit when j ≤ i
    mid = ⌊(i+j)/2⌋
    if(key ≤ L(mid)) then
        j = mid
        % Sublist changed from L(i, j) to L(i..mid)
    else
        i = mid + 1
        % Sublist changed from L(i, j) to L(mid+1, j)
    end if
end loop
if(key = L(i)) then
    return( i )
else
    return( “key is not in list” )
end if
end algorithm
<table>
<thead>
<tr>
<th>Define Problem</th>
<th>Define Loop Invariants</th>
<th>Define Measure of Progress</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1.png" alt="Image" /></td>
<td><img src="image2.png" alt="Image" /></td>
<td><img src="image3.png" alt="Image" /></td>
</tr>
<tr>
<td>Define Step</td>
<td>Define Exit Condition</td>
<td>Maintain Loop Inv</td>
</tr>
<tr>
<td><img src="image4.png" alt="Image" /></td>
<td><img src="image5.png" alt="Image" /></td>
<td><img src="image6.png" alt="Image" /></td>
</tr>
<tr>
<td>Make Progress</td>
<td>Initial Conditions</td>
<td>Ending</td>
</tr>
<tr>
<td><img src="image7.png" alt="Image" /></td>
<td><img src="image8.png" alt="Image" /></td>
<td><img src="image9.png" alt="Image" /></td>
</tr>
</tbody>
</table>

CS 4407, Algorithms, University College Cork, Gregory M. Provan
Example 5

Compute the big-Oh running time of the following C++ code segment:

```cpp
for (i = 2; i < n; i++) {
    sum += i;
}
```

The number of iterations of a for-loop is equal to the top index of the loop minus the bottom index, plus one more instruction to account for the final conditional test.

Note: if the for loop terminating condition is \( i \leq n \), rather than \( i < n \), then the number of times the conditional test is performed is:

\[
((\text{top\_index} + 1) - \text{bottom\_index}) + 1
\]

In this case, we have \( n - 2 + 1 = n - 1 \). The assignment in the loop is executed \( n - 2 \) times. So, we have \( (n - 1) + (n - 2) = (2n - 3) \) instructions executed = \( O(n) \).
• **While/repeat**: add \( f(n) \) to the running time for each iteration. We then multiply that time by the number of iterations. For a *while* loop, we must add one additional \( f(n) \) for the final loop test.

• **For** loop: if the function call is in the initialization of a *for* loop, add \( f(n) \) to the total running time of the loop. If the function call is the termination condition of the *for* loop, add \( f(n) \) for each iteration.

• **If** statement: add \( f(n) \) to the running time of the statement.

```c
int a, n, x;
int bar(int x, int n) {
    int i;
    1) for (i = 1; i < n; i++)
       2) x = x + i;
    3) return x;
}

int foo(int x, int n) {
    int i;
    4) for (i = 1; i <= n; i++)
       5) x = x + bar(i, n);
    6) return x;
}

void main(void) {
    n = GetInteger();
    7) a = 0;
    8) x = foo(a, n)
    9) printf("%d", bar(a, n))
```
Here is the body of a function:

```c
sum = 0;
for (i = 1; i <= f(n); i++)
    sum += i;
```

where \( f(n) \) is a function call. Give a big-oh upper bound on this function if the running time of \( f(n) \) is \( O(n) \), and the \textit{value} of \( f(n) \) is \( n! \):
Study:

- Many experienced programmers were asked to code up binary search.

80% got it wrong

Good thing is was not for a nuclear power plant.