Central Algorithmic Techniques

Iterative Algorithms
class InsertionSortAlgorithm extends SortAlgorithm {
    void sort(int a[]) throws Exception {
        for (int i = 1; i < a.length; i++) {
            int j = i;
            int B = a[i];
            while ((j > 0) && (a[j-1] > B)) {
                a[j] = a[j-1];
                j--; }
            a[j] = B;
        }
    }
}

Problems and Cons?
Code
Representation of an Algorithm

**Pros:**
- Runs on computers
- Precise and succinct

**Cons:**
- I am not a computer
- I need a higher level of intuition.
- Prone to bugs
- Language dependent

We will focus on a more natural, intuitive and powerful method for developing, analyzing and proving correctness of iterative algorithms. This method will be based on **loop invariants**.
Iterative Algorithms

Take one step at a time
towards the final destination

loop (done)
take step
end loop
Loop Invariants

A good way to structure many programs:

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

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

- Pre-condition: location of home and school
- Post-condition: Traveled from home to school
General Principle

• Do not worry about the entire computation.

• Take one step at a time!
A Measure of Progress

79 km to school

75 km to school
Safe Locations

- Algorithm specifies from which locations it knows how to step.
Loop Invariant

- “The computation is presently in a safe location.”
- May or may not be true.
Defining Algorithm

• From every safe location, define one step towards school.
Take a step

• What is required of this step?
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?
Establishing Loop Invariant

From the Pre-Conditions on the input instance we must establish the loop invariant.
Maintain Loop Invariant

• Suppose that
  – We start in a safe location (pre-condition)
  – If we are in a safe location, we always step to another safe location (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" alt="Diagram" /></td>
<td><img src="image2" alt="Diagram" /></td>
<td><img src="image3" alt="Diagram" /></td>
</tr>
<tr>
<td>Define Step</td>
<td>Define Exit Condition</td>
<td>Maintain Loop Inv</td>
</tr>
<tr>
<td><img src="image4" alt="Diagram" /></td>
<td><img src="image5" alt="Diagram" /></td>
<td><img src="image6" alt="Diagram" /></td>
</tr>
<tr>
<td>Make Progress</td>
<td>Initial Conditions</td>
<td>Ending</td>
</tr>
<tr>
<td><img src="image7" alt="Diagram" /></td>
<td><img src="image8" alt="Diagram" /></td>
<td><img src="image9" alt="Diagram" /></td>
</tr>
</tbody>
</table>

- **Define Problem**
  - Define Problem
  - Define Loop Invariants
  - Define Measure of Progress

- **Define Step**
  - Define Step
  - Define Exit Condition
  - Maintain Loop Inv

- **Make Progress**
  - Make Progress
  - Initial Conditions
  - Ending

- **Define Measure of Progress**
  - Define Measure of Progress
  - Exit

- **Initial Conditions**
  - Initial Conditions

- **Ending**
  - Ending

- **Exit**
  - Exit

- **79 km to school**
  - 79 km to school

- **79 km**
  - 79 km

- **75 km**
  - 75 km

- **0 km**
  - 0 km
Simple Example

Insertion Sort Algorithm
class InsertionSortAlgorithm extends SortAlgorithm {
    void sort(int a[]) throws Exception {
        for (int i = 1; i < a.length; i++) {
            int j = i;
            int B = a[i];
            while ((j > 0) && (a[j-1] > B)) {
                a[j] = a[j-1];
                j--; }
            a[j] = B;
        }
    }
}
Higher Level Abstract View
Representation of an Algorithm
Problem Specification

• Pre-condition: The input is a list of n values with the same value possibly repeated.

• Post-condition: The output is a list consisting of the same n values in non-decreasing order.
Define Loop Invariant

• Some subset of the elements are sorted
• The remaining elements are off to the side.
Defining Measure of Progress

6 elements to school
Define Step

- Select arbitrary element from side.
- Insert it where it belongs.
Making progress while maintaining the loop invariant

Sorted sublist

COSC 3101, PROF. J. ELDER
Beginning & Ending

\[ n \text{ elements to school} \]

\[ 14, 23, 25, 30, 31, 52, 62, 79, 88, 98 \]

\[ 0 \text{ elements to school} \]

\[ 14, 23, 25, 30, 31, 52, 62, 79, 88, 98 \]
Running Time

Inserting an element into a list of size $i$ takes $\Theta(i)$ time.

Total $= 1 + 2 + 3 + \ldots + n = \Theta(n^2)$
Ok: I know you knew Insertion Sort

But hopefully you are beginning to appreciate the value of loop invariants for describing algorithms

The loop invariant is an example of a more general thing called an assertion.
Assertions

in Algorithms
Purpose of Assertions

Useful for

– thinking about algorithms
– developing
– describing
– proving correctness
Definition of Assertions

An assertion is a statement about the current state of the data structure that is either true or false.

e.g., the amount in your bank account is not negative.
Definition of Assertions

It is made at some particular point during the execution of an algorithm.

If it is false, then something has gone wrong in the logic of the algorithm.
Definition of Assertions

An assertion is not a task for the algorithm to perform.

It is only a comment that is added for the benefit of the reader.
Specifying a Computational Problem

Example of Assertions

- **Preconditions**: Any assumptions that must be true about the input instance.

- **Postconditions**: The statement of what must be true when the algorithm/program returns.
Definition of Correctness

\(<\text{PreCond}> \& \ <\text{code}> \Rightarrow \ <\text{PostCond}>\)

If the input meets the preconditions, then the output must meet the postconditions.

If the input does not meet the preconditions, then nothing is required.
An Example:
A Sequence of Assertions

<assertion_0>
if( <condition_1> ) then
    code_{1,true}
else
    code_{1,false}
end if
<assertion_1>

... any <conditions>
code

<assertion_r>
if( <condition_r> ) then
    code_{r,true}
else
    code_{r,false}
end if
<assertion_r>

Definition of Correctness

How is this proved?

Must check $2^r$ different
• settings of <conditions>
• paths through the code.

Is there a faster way?
An Example:
A Sequence of Assertions

<assertion₀>
if( <condition₁> ) then
    code<₁,true>
else
    code<₁,false>
end if
<assertion₁>

...  

<assertion₀>
<condition₁>
code<₁,true>
<assertion₁>
<assertion₀> ¬<condition₁>
code<₁,false>
<assertion₁>

Step 1
An Example:  
A Sequence of Assertions

<assertion_0>
if( <condition_1> ) then
    code_{1,true}
else
    code_{1,false}
end if
<assertion_1>

…

Step 2

<assertion_1>
<condition_2>
code_{2,true}
<assertion_1>

¬<condition_2>  
code_{2,false}
<assertion_2>

<assertion_r-1>
if( <condition_r> ) then
    code_{r,true}
else
    code_{r,false}
end if
<assertion_r>
An Example: A Sequence of Assertions

<assertion_0>
if( <condition_1> ) then
  code_{1, true}
else
  code_{1, false}
end if
<assertion_1>

<assertion_{r-1}>
if( <condition_r> ) then
  code_{r, true}
else
  code_{r, false}
end if
<assertion_r>

...
A Sequence of Assertions

<assertion₀>
if( <condition₁> ) then
  code<1,true>
else
  code<1,false>
end if
<assertion₁>

<assertionᵣ⁻¹>
if( <conditionᵣ> ) then
  code<r,true>
else
  code<r,false>
end if
<assertionᵣ>

\[ \vdots \]
<assertionᵣ⁻¹>

We have reduced the number of cases that must be considered from \(2^r\) to \(r\).

Step r

<assertionᵣ⁻¹>
<conditionᵣ>
\[\rightarrow\]
<assertionᵣ>

\[ \vdots \]
<assertionᵣ⁻¹>
¬<conditionᵣ>
\[\rightarrow\]
<assertionᵣ>
Another Example:
A Loop

Type of Algorithm:
• Iterative

Type of Assertion:
• Loop Invariants
Iterative Algorithms
Loop Invariants

<preCond>
codeA
loop
  <loop-invariant>
  exit when <exit Cond>
  codeB
endloop
codeC
<postCond>

Definition of Correctness?
Iterative Algorithms
Loop Invariants

Definition of Correctness

How is this proved?
Iterative Algorithms

Loop Invariants

The computation may go around the loop an arbitrary number of times.

Is there a faster way?

Definition of Correctness

COSC 3101, PROF. J. ELDER
Iterative Algorithms
Loop Invariants

<preCond>
codeA
loop
  <loop-invariant>
  exit when <exit Cond>
  codeB
endloop
codeC
<postCond>

<preCond>  
<loop-invariant>
<exit Cond>  

Step 0
Iterative Algorithms
Loop Invariants

Step 1

preCond

loop

loop-invariant

exit when

postCond

codeA

loop

loop-invariant

exit when

endloop

codeB

codeC

postCond

step

loop-invariant

exit Cond

loop-invariant

exit Cond

loop-invariant

exit Cond

codeB

codeB
Iterative Algorithms
Loop Invariants

Step 2

<loop-invariant>
¬<exit Cond>
codeB
<loop-invariant>
<exit Cond>
codeB
<loop-invariant>
<postCond>

<preCond>
codeA
loop
<loop-invariant>
exit when <exit Cond>
codeB
endloop
codeC
Iterative Algorithms
Loop Invariants

Step 3

<loop-invariant> exit when <exit Cond> codeB

<loop-invariant> ¬<exit Cond> codeB

<loop-invariant> <exit Cond> codeB

<postCond>

<preCond>
Iterative Algorithms
Loop Invariants

<preCond>
codeA

loop

<loop-invariant>
exit when <exit Cond>
codeB

endloop
codeC

<postCond>

All these steps are the same and therefore only need be done once!
Iterative Algorithms

Loop Invariants

<preCond>
codeA
loop
<loop-invariant>
exit when <exit Cond>
codeB
endloop
codeC
<postCond>

Last Step

<loop-invariant>
<exit Cond>
<postCond>
<exit Cond>
codeC
Partial Correctness

Establishing Loop Invariant

<preCond> codeA

Maintaining Loop Invariant

<loop-invariant> ¬<exit Cond> codeB

Clean up loose ends

<loop-invariant> <exit Cond> codeC

Proves that IF the program terminates then it works

<PreCond> & <code> ⇒ <PostCond>
Algorithm Termination

Measure of progress

0 km

79 km

75 km

Exit
Algorithm Correctness

Partial Correctness + Termination → Correctness
Designing Loop Invariants

Coming up with the loop invariant is the hardest part of designing an algorithm.

It requires practice, perseverance, and insight.

Yet from it the rest of the algorithm follows easily
Don’t start coding

You must design a working algorithm first.
Exemplification:
Try solving the problem on small input examples.
Start with Small Steps

What basic steps might you follow to make some kind of progress towards the answer?

Describe or draw a picture of what the data structure might look like after a number of these steps.
Picture from the Middle

Leap into the middle of the algorithm.

What would you like your data structure to look like when you are half done?
Ask for 100%

Pretend that a genie has granted your wish.

- You are now in the middle of your computation and your dream loop invariant is true.
Ask for 100%

Maintain the Loop Invariant:

– From here, are you able to take some computational steps that will make progress while maintaining the loop invariant?
Ask for 100%

• If you can maintain the loop invariant, great.
• If not,
  – Too Weak: If your loop invariant is too weak, then the genie has not provided you with everything you need to move on.
  – Too Strong: If your loop invariant is too strong, then you will not be able to establish it initially or maintain it.
Differentiating between Iterations

\( x = x + 2 \)
- Meaningful as code
- False as a mathematical statement

\( x' = x_i = \text{value at the beginning of the iteration} \)
\( x'' = x_{i+1} = \text{new value after going around the loop one more time.} \)
\( x'' = x' + 2 \)
- Meaningful as a mathematical statement
Loop Invariants for Iterative Algorithms

Three Search Examples
Define Problem: Binary Search

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

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

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

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

- Make Progress
- Maintain Loop Invariant

key 25
Define Step

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

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

If \( key > \text{mid} \), 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.
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

key 25

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

key 24

- If the key is contained in the original list, then the key is at this location.
- Conclusion still solves the problem. Simply check this one location for the key.
Running Time

The sublist is of size $n$, $n/2$, $n/4$, $n/8$, ..., 1

Each step $\Theta(1)$ time.

Total = $\Theta(\log n)$
BinarySearch(A[1..n], key)
<precondition>: A[1..n] is sorted in non-decreasing order
<postcondition>: If key is in A[1..n], algorithm returns its location
p = 1, q = n
while q > p
    <loop-invariant>: If key is in A[1..n], then key is in A[p..q]
    
    mid = \left\lfloor \frac{p + q}{2} \right\rfloor
    
    if key ≤ A[mid]
        q = mid
    else
        p = mid + 1
    end
end
if key = A[p]
    return(p)
else
    return("Key not in list")
end
<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>

**COSC 3101, PROF. J. ELDER**

- 79 km to school
- Initial Conditions: 0 km
Simple, right?

- Although the concept is simple, binary search is notoriously easy to get wrong.
- Why is this?
The Devil in the Details

- The basic idea behind binary search is easy to grasp.
- It is then easy to write pseudocode that works for a ‘typical’ case.
- Unfortunately, it is equally easy to write pseudocode that fails on the boundary conditions.
The Devil in the Details

What condition will break the loop invariant?

```plaintext
if key ≤ A[mid]
    q = mid
else
    p = mid + 1
end
```

or

```plaintext
if key < A[mid]
    q = mid
else
    p = mid + 1
end
```
The Devil in the Details

Code: \textit{key} \geq A[\text{mid}] \rightarrow \text{select right half}

Bug!!
The Devil in the Details

if \( key \leq A[mid] \)
\[
q = mid
\]
else
\[
p = mid + 1
\]
end

if \( key < A[mid] \)
\[
q = mid - 1
\]
else
\[
p = mid
\]
end

if \( key < A[mid] \)
\[
q = mid
\]
else
\[
p = mid + 1
\]
end

OK

OK

Not OK!!
The Devil in the Details

\[ \text{mid} = \left\lfloor \frac{p+q}{2} \right\rfloor \quad \text{or} \quad \text{mid} = \left\lceil \frac{p+q}{2} \right\rceil \]

Shouldn’t matter, right? Select \( \text{mid} = \left\lceil \frac{p+q}{2} \right\rceil \)
The Devil in the Details

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

If \( \text{key} > \text{mid} \), then key is in right half.
The Devil in the Details

If $key \leq mid$, then $key$ is in the left half.

If $key > mid$, then $key$ is in the right half.
The Devil in the Details

• Another bug!

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

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

No progress toward goal: Loops Forever!
The Devil in the Details

\[
\text{mid} = \left\lfloor \frac{p + q}{2} \right\rfloor
\]

if \( \text{key} \leq A[\text{mid}] \)
\[
q = \text{mid}
\]
else
\[
p = \text{mid} + 1
\]
end

OK

\[
\text{mid} = \left\lfloor \frac{p + q}{2} \right\rfloor
\]

if \( \text{key} < A[\text{mid}] \)
\[
q = \text{mid} - 1
\]
else
\[
p = \text{mid}
\]
end

OK

\[
\text{mid} = \left\lfloor \frac{p + q}{2} \right\rfloor
\]

if \( \text{key} \leq A[\text{mid}] \)
\[
q = \text{mid}
\]
else
\[
p = \text{mid} + 1
\]
end

Not OK!!
End of Lecture 4

March 16, 2009
Announcements

• Typo on Assignment 1: The last problem is worth 16 marks, not 30 marks.

• Check out the York programming contests. The first contest of the Winter term will take place this Friday March 20, 15:00-17:00 in CSEB 1006.
How Many Possible Algorithms?

\[ \text{mid} = \left\lfloor \frac{p + q}{2} \right\rfloor \]

if \( \text{key} \leq A[\text{mid}] \) or if \( \text{key} < A[\text{mid}] \)?

\[ q = \text{mid} \]

else

\[ p = \text{mid} + 1 \]

end

or \[ q = \text{mid} - 1 \]

end
Alternative Algorithm: Less Efficient but More Clear

BinarySearch(A[1..n], key)

<precondition>: A[1..n] is sorted in non-decreasing order

<postcondition>: If key is in A[1..n], algorithm returns its location

p = 1, q = n

while q > p

<loop-invariant>: If key is in A[1..n], then key is in A[p..q]

mid = \left\lfloor \frac{p+q}{2} \right\rfloor

if key = A[mid]

return(mid)

elseif key < A[mid]

q = mid − 1

else

p = mid + 1

end

end

if key = A[p]

return(p)

else

return("Key not in list")

end

Still \( \Theta(\log n) \), but with slightly larger constant.
Moral

• Use the loop invariant method to think about algorithms.
• Be careful with your definitions.
• Be sure that the loop invariant is always maintained.
• Be sure progress is always made.
• Having checked the ‘typical’ cases, pay particular attention to boundary conditions and the end game.
• Sometimes it is worth paying a little in efficiency for clear, correct code.
Loop Invariants for Iterative Algorithms

A Second Search Example: The Binary Search Tree
Define Problem: Binary Search Tree

• PreConditions
  – Key 25
  – A binary search tree.

  – PostConditions
  – Find key in BST (if there).
Binary Search Tree

All nodes in left subtree ≤ Any node ≤ All nodes in right subtree
Define Loop Invariant

- Maintain a sub-tree.
- If the key is contained in the original tree, then the key is contained in the sub-tree.
Define Step

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

If key < root, then key is in left half.
If key = root, then key is found
If key > root, then key is in right half.
Card Trick

- A volunteer, please.
Loop Invariants for Iterative Algorithms

A Third Search Example: A Card Trick
Pick a Card

Done
Loop Invariant:
The selected card is one of these.
Which column?

left
Loop Invariant:
The selected card is one of these.
Selected column is placed in the middle
I will rearrange the cards
Relax Loop Invariant: I will remember the same about each column.
Which column?

right
Loop Invariant:
The selected card is one of these.
Selected column is placed in the middle
I will rearrange the cards
Which column?

left
Loop Invariant: The selected card is one of these.
Selected column is placed in the middle
Here is your card.

Wow!
Ternary Search

• Loop Invariant: selected card in central subset of cards

\[
\text{Size of subset} = \left\lceil \frac{n}{3^{i-1}} \right\rceil
\]

where

\( n = \text{total number of cards} \)
\( i = \text{iteration index} \)

• How many iterations are required to guarantee success?
Loop Invariants
for
Iterative Algorithms

A Fourth Example:
Partitioning
(Not a search problem:
can be used for sorting, e.g., Quicksort)
The “Partitioning” Problem


Problem: Partition a list into a set of small values and a set of large values.
Precise Specification

Precondition: $A[p...r]$ is an arbitrary list of values. $x = A[r]$ is the pivot.

Postcondition: $A$ is rearranged such that $A[p...q-1] \leq A[q] = x < A[q+1...r]$ for some $q$. 
Loop Invariant

- 3 subsets are maintained
  - One containing values less than or equal to the pivot
  - One containing values greater than the pivot
  - One containing values yet to be processed

Loop invariant:

1. All entries in $A[p \ldots i]$ are $\leq$ pivot.
2. All entries in $A[i + 1 \ldots j - 1]$ are $> \text{pivot}$.
Maintaining Loop Invariant

- Consider element at location j
  - If greater than pivot, incorporate into ‘> set’ by incrementing j.
  - If less than or equal to pivot, incorporate into ‘≤ set’ by swapping with element at location i+1 and incrementing both i and j.
  - Measure of progress: size of unprocessed set.
Maintaining Loop Invariant

\textbf{PARTITION}(A, p, r)

1. \(x \leftarrow A[r]\)
2. \(i \leftarrow p - 1\)
3. \textbf{for} \(j \leftarrow p \textbf{ to } r - 1\)
4. \textbf{do if} \(A[j] \leq x\)
5. \textbf{then} \(i \leftarrow i + 1\)
6. exchange \(A[i] \leftrightarrow A[j]\)
7. exchange \(A[i + 1] \leftrightarrow A[r]\)
8. \textbf{return} \(i + 1\)

\textbf{Loop invariant:}

1. All entries in \(A[p \ldots i]\) are \(\leq\) pivot.
2. All entries in \(A[i + 1 \ldots j - 1]\) are \(>\) pivot.
Establishing Loop Invariant

**Loop invariant:**

1. All entries in $A[p..i]$ are $\leq$ pivot.
2. All entries in $A[i+1..j-1]$ are $> \text{pivot}$.  

<table>
<thead>
<tr>
<th></th>
<th>8</th>
<th>1</th>
<th>6</th>
<th>4</th>
<th>0</th>
<th>3</th>
<th>9</th>
<th>5</th>
</tr>
</thead>
</table>
Establishing Postcondition

**Partition**\((A, p, r)\)

1. \(x \leftarrow A[r]\)
2. \(i \leftarrow p - 1\)
3. \(\text{for } j \leftarrow p \text{ to } r - 1\)
4. \(\text{do if } A[j] \leq x\)
5. \(\text{then } i \leftarrow i + 1\)
6. \(\text{exchange } A[i] \leftrightarrow A[j]\)
7. \(\text{exchange } A[i + 1] \leftrightarrow A[r]\)
8. \(\text{return } i + 1\)

**Loop invariant:**

1. All entries in \(A[p \ldots i]\) are \(\leq\) pivot.
2. All entries in \(A[i + 1 \ldots j - 1]\) are \(>\) pivot.
3. \(A[r] = \text{pivot.}\)
Establishing Postcondition

\textsc{Partition}(A, p, r)
1 \ x \leftarrow A[r] \\
2 \ i \leftarrow p - 1 \\
3 \textbf{for} \ j \leftarrow p \textbf{ to } r - 1 \\
4 \quad \textbf{do if } A[j] \leq x \\
5 \quad \quad \textbf{then } i \leftarrow i + 1 \\
6 \quad \text{exchange } A[i] \leftrightarrow A[j] \\
7 \text{exchange } A[i + 1] \leftrightarrow A[r] \\
8 \textbf{return } i + 1 

\begin{tabular}{c|c|c|c|c|c|c|c|c}
\hline
p & i & 1 & 4 & 0 & 3 & 6 & 8 & 9 & 5 \\
\hline
\end{tabular}

\begin{tabular}{c|c|c|c|c|c|c|c|c}
\hline
p & i & 1 & 4 & 0 & 3 & 5 & 8 & 9 & 6 \\
\hline
\end{tabular}
An Example
Running Time

Each iteration takes $\Theta(1)$ time $\rightarrow$ Total $= \Theta(n)$
Algorithm Definition Completed

- Define Problem
- Define Loop Invariants
- Define Measure of Progress
- Define Step
- Define Exit Condition
- Maintain Loop Inv
- Make Progress
- Initial Conditions
- Ending

- COSC 3101, PROF. J. ELDER

- 79 km to school

- 128

- 5 km

- 5 km

- Exit

- Exit

- Exit

- Exit

- Exit
More Examples of Iterative Algorithms

Using Constraints on Input to Achieve Linear-Time Sorting
Recall: InsertionSort

**Insertion-Sort**

1. **for** $j \leftarrow 2$ **to** length$[A]$
2. **do** key $\leftarrow A[j]$
   
4. $i \leftarrow j - 1$
5. **while** $i > 0$ and $A[i] > key$
6. **do** $A[i + 1] \leftarrow A[i]$
7. $i \leftarrow i - 1$
8. $A[i + 1] \leftarrow key$

| $c_1$ | $n$ |
| $c_2$ | $n - 1$ |
| $0$ | $n - 1$ |
| $c_4$ | $n - 1$ |
| $\sum_{j=2}^{n} t_j$ | $T(n) \in \theta(n^2)$ |
| $\sum_{j=2}^{n} (t_j - 1)$ |

Worst case (reverse order): $t_j = j$ :  
\[ \sum_{j=2}^{n} j = \frac{n(n + 1)}{2} - 1 \rightarrow T(n) \in \theta(n^2) \]
Recall: MergeSort

\[ T(n) \in \theta(n \log n) \]
Comparison Sorts

• InsertionSort and MergeSort are examples of (stable) Comparison Sort algorithms.

• QuickSort is another example we will study shortly.

• Comparison Sort algorithms sort the input by successive comparison of pairs of input elements.

• Comparison Sort algorithms are very general: they make no assumptions about the values of the input elements.
Comparison Sorts

InsertionSort is $\theta(n^2)$.  
MergeSort is $\theta(n \log n)$.  

Can we do better?
Comparison Sort: Decision Trees

- Example: Sorting a 3-element array $A[1..3]$

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```
Comparison Sort

- Worst-case time is equal to the height of the binary decision tree.
- The height of the tree is the log of the number of leaves.
- The leaves of the tree represent all possible permutations of the input. How many are there?

\[ \log(n!) \in \Omega(n \log n) \]

Thus MergeSort is asymptotically optimal.
Linear Sorts?

Comparison sorts are very general, but are $\Omega(n \log n)$

Faster sorting may be possible if we can constrain the nature of the input.
Example 1. Counting Sort

- **Counting Sort** applies when the elements to be sorted come from a finite (and preferably small) set.
- For example, the elements to be sorted are integers in the range \([0…k-1]\), for some fixed integer \(k\).
- We can then create an array \(V[0…k-1]\) and use it to count the number of elements with each value \([0…k-1]\).
- Then each input element can be placed in exactly the right place in the output array in constant time.
## Counting Sort

<table>
<thead>
<tr>
<th>Input:</th>
<th>1 0 0 1 3 1 1 3 1 0 2 1 0 1 1 1 2 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output:</td>
<td>0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 2 2 3 3 3</td>
</tr>
</tbody>
</table>

- Input: N records with integer keys between [0…3].
- Output: **Stable** sorted keys.
- Algorithm:
  - Count frequency of each key value to determine transition locations
  - Go through the records in order putting them where they go.
CountingSort

Input: 1 0 0 1 3 1 1 3 1 0 2 1 0 1 1 2 2 1 0
Output: 0 0 0 0 0 1 1 1 1 1 1 1 2 2 2 3 3
Index: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18

Stable sort: If two keys are the same, their order does not change.

Thus the 4th record in input with digit 1 must be
the 4th record in output with digit 1.

It belongs at output index 8, because 8 records go before it
ie, 5 records with a smaller digit & 3 records with the same digit

Count These!
CountingSort

<table>
<thead>
<tr>
<th>Input:</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 0 1 3 1 1 3 1 0 2 1 0 1 1 2 2 1 0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Output:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Index:</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18</td>
</tr>
</tbody>
</table>

Value v:

<table>
<thead>
<tr>
<th># of records with digit v:</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3</td>
</tr>
<tr>
<td>5 9 3 2</td>
</tr>
</tbody>
</table>

N records. Time to count? $\Theta(N)$
## CountingSort

<table>
<thead>
<tr>
<th>Input:</th>
<th>1 0 0 1 3 1 1 3 1 0 2 1 0 1 1 2 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output:</td>
<td></td>
</tr>
<tr>
<td>Index:</td>
<td>0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18</td>
</tr>
</tbody>
</table>

Value v: | 0 | 1 | 2 | 3 |
---------|---|---|---|---|
# of records with digit v: | 5 | 9 | 3 | 3 |
# of records with digit < v: | 0 | 5 | 14 | 17 |

N records, k different values. Time to count? $\Theta(k)$
## CountingSort

<table>
<thead>
<tr>
<th>Input:</th>
<th>1 0 0 1 3 1 1 3 1 0 2 1 0 1 1 2 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output:</td>
<td>0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 2 2 2 3 3</td>
</tr>
<tr>
<td>Index:</td>
<td>0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Value v:</th>
<th>0 1 2 3</th>
</tr>
</thead>
<tbody>
<tr>
<td># of records with digit &lt; v:</td>
<td>0 5 14 17</td>
</tr>
</tbody>
</table>

= location of first record with digit v.
CountingSort

<table>
<thead>
<tr>
<th>Input:</th>
<th>1 0 0 1 3 1 1 3 1 0 2 1 0 1 1 2 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output:</td>
<td>0 ? 1</td>
</tr>
<tr>
<td>Index:</td>
<td>0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18</td>
</tr>
</tbody>
</table>

Value v: | 0 | 1 | 2 | 3 |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Location of first record with digit v.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Algorithm: Go through the records in order putting them where they go.
Loop Invariant

- The first \( i-1 \) keys have been placed in the correct locations in the output array.
- The auxiliary data structure \( v \) indicates the location at which to place the \( i^{th} \) key for each possible key value from \([1..k-1]\).
CountingSort

Algorithm: Go through the records in order putting them where they go.
CountingSort

Input:

Output:

Index:

Value v:

Location of next record with digit v.

Algorithm: Go through the records in order putting them where they go.
CountingSort

Input:

| 0 | 0 | 1 | 3 | 1 | 1 | 3 | 1 | 0 | 2 | 1 | 0 | 1 | 1 | 2 | 2 | 1 | 0 |

Output:

| 0 | 0 | 1 |

Index:

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 |

Value v:

| 0 | 1 | 2 | 3 |

| 1 | 6 | 14 | 17 |

Location of next record with digit v.

Algorithm: Go through the records in order putting them where they go.
CountingSort

Input: 1 0 0 1 3 1 1 3 1 0 2 1 0 1 1 2 2 1 0
Output: 0 0 1 1
Index: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18

Value v: 0 1 2 3
Location of next record with digit v:

Algorithm: Go through the records in order putting them where they go.
**CountingSort**

| Input: | 1 0 0 1 3 1 1 3 1 0 2 1 0 1 1 2 2 1 0 |
| Output: | 0 0 1 1 |
| Index: | 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 |

Location of next record with digit \( v \).

**Value \( v \):**

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>7</td>
<td>14</td>
<td>17</td>
</tr>
</tbody>
</table>

**Algorithm:** Go through the records in order putting them where they go.
CountingSort

Algorithm: Go through the records in order putting them where they go.

Input: 1 0 0 1 3 1 1 3 1 0 2 1 0 1 1 2 2 1 0

Output: 0 0 1 1 1

Index: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18

Location of next record with digit v.

Value v: 0 1 2 3

2 7 14 18
### CountingSort

<table>
<thead>
<tr>
<th>Input:</th>
<th>1 0 0 1 3 1 1 3 1 0 2 1 0 1 1 2 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output:</td>
<td>0 0 1 1 1 1 1 3</td>
</tr>
<tr>
<td>Index:</td>
<td>0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18</td>
</tr>
<tr>
<td>Value v:</td>
<td>0 1 2 3</td>
</tr>
<tr>
<td>Location of next record with digit v:</td>
<td>2 8 14 18</td>
</tr>
</tbody>
</table>

Algorithm: Go through the records in order putting them where they go.
CountingSort

<table>
<thead>
<tr>
<th>Input:</th>
<th>1 0 0 1 3 1 1 3 1 0 2 1 0 1 1 2 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output:</td>
<td>0 0 1 1 1 1</td>
</tr>
<tr>
<td>Index:</td>
<td>0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18</td>
</tr>
</tbody>
</table>

Value v: 0 1 2 3 2 9 14 18

Location of next record with digit v.

Algorithm: Go through the records in order putting them where they go.
CountingSort

<table>
<thead>
<tr>
<th>Input:</th>
<th>100131 1310210112210</th>
<th>Output:</th>
<th>0011111133</th>
<th>Index:</th>
<th>0123456789101112131415161718</th>
</tr>
</thead>
</table>

Value v:  

| 0123 | 291419 | Location of next record with digit v. |

Algorithm: Go through the records in order putting them where they go.
CountingSort

<table>
<thead>
<tr>
<th>Input:</th>
<th>1 0 0 1 3 1 1 3 1 0 2 1 0 1 1 2 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output:</td>
<td>0 0 0 1 1 1 1 1 3 3</td>
</tr>
<tr>
<td>Index:</td>
<td>0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18</td>
</tr>
</tbody>
</table>

Value v: 0 1 2 3 2 10 14 19

Location of next record with digit v.

Algorithm: Go through the records in order putting them where they go.
## CountingSort

**Input:**

|   | 1 | 0 | 0 | 1 | 3 | 1 | 1 | 3 | 1 | 0 | 2 | 1 | 0 | 1 | 1 | 2 | 2 | 1 | 0 |
**Output:**

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>3</th>
</tr>
</thead>
</table>
**Index:**

<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>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
<th>17</th>
<th>18</th>
</tr>
</thead>
</table>

**Value v:**

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>3</td>
<td>10</td>
<td>14</td>
<td>19</td>
</tr>
</tbody>
</table>

**Location of next record with digit v.**

**Algorithm:** Go through the records in order putting them where they go.
### CountingSort

**Input:**

| 1000131113102101112210 |

**Output:**

| 0001111111233 |

**Index:**

| 0123456789101112131415161718 |

#### Value v:

| 0123 |

#### Location of next record with digit v.

Algorithm: Go through the records in order putting them where they go.
**CountingSort**

| Input: | 0 | 0 | 0 | 1 | 3 | 1 | 1 | 3 | 1 | 0 | 2 | 1 | 0 | 1 | 1 | 2 | 2 | 1 | 0 |
| Output:| 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 2 | 3 | 3 |
| Index: | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 |

**Value v:**

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>10</td>
<td>15</td>
<td>19</td>
</tr>
</tbody>
</table>

**Algorithm:** Go through the records in order putting them where they go.

Location of next record with digit v.
CountingSort

Input: 1 0 0 1 3 1 1 3 1 0 2 1 0 1 1 2 2 1 0
Output: 0 0 0 0 0 1 1 1 1 1 1 1 1 1 2 2 2 3 3
Index: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18

Location of next record with digit v.

Value v:

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>14</td>
<td>17</td>
<td>19</td>
</tr>
</tbody>
</table>

Total = \( \Theta(N+k) \)
Example 2. RadixSort

Input:
- A stack of $N$ punch cards.
- Each card contains $d$ digits.
- Each digit between $[0...k-1]$

Output:
- Sorted cards.

Digit Sort:
- Select one digit
- Separate cards into $k$ piles based on selected digit (e.g., Counting Sort).

Stable sort: If two cards are the same for that digit, their order does not change.
RadixSort

<table>
<thead>
<tr>
<th>344</th>
<th>125</th>
<th>125</th>
</tr>
</thead>
<tbody>
<tr>
<td>125</td>
<td>333</td>
<td>224</td>
</tr>
<tr>
<td>333</td>
<td>134</td>
<td>134</td>
</tr>
<tr>
<td>134</td>
<td>143</td>
<td>325</td>
</tr>
<tr>
<td>224</td>
<td>225</td>
<td>134</td>
</tr>
<tr>
<td>334</td>
<td>243</td>
<td>333</td>
</tr>
<tr>
<td>143</td>
<td>344</td>
<td>334</td>
</tr>
<tr>
<td>225</td>
<td>333</td>
<td>143</td>
</tr>
<tr>
<td>325</td>
<td>334</td>
<td>243</td>
</tr>
<tr>
<td>243</td>
<td>325</td>
<td>344</td>
</tr>
</tbody>
</table>

Sort wrt which digit first?
The most significant.

All meaning in first sort lost.
RadixSort

344 125 333 134 224 334 143 224 344 134 243 344 134 225 325 225 325 243 325

Sort wrt which digit first?

The least significant.

Sort wrt which digit Second?

The next least significant.
RadixSort

344
125
333
134
224
334
143
225
325
243

Sort wrt which digit first?

333
143
243
344
134
224
334
125
225
325

Sort wrt which digit Second?

The least significant.

2 24
1 25
2 25
3 25
3 33
1 34
3 34
1 43
2 43
3 44

The next least significant.

Is sorted wrt least sig. 2 digits.
RadixSort

Is sorted wrt first i digits.

Sort wrt i+1st digit.

Is sorted wrt first i+1 digits.

These are in the correct order because sorted wrt high order digit.
RadixSort

Is sorted wrt first i digits.

1 24
2 25
3 25
3 33
1 34
3 34
1 43
2 43
3 44

Sort wrt i+1st digit.

1 25
1 34
1 43
2 24
2 25
2 43
3 25
3 33
3 34
3 44

Is sorted wrt first i+1 digits.

These are in the correct order because was sorted & stable sort left sorted.
Loop Invariant

- The keys have been correctly stable-sorted with respect to the $i-1$ least-significant digits.
Running Time

\textbf{Radix-Sort}(A, d)
\begin{algorithmic}
  \STATE \textbf{for} \( i \leftarrow 1 \) \textbf{to} \( d \)
  \STATE \hspace{1em} \textbf{do} use a stable sort to sort array \( A \) on digit \( i \)
\end{algorithmic}

Running time is \( \Theta(d(n + k)) \)

Where
\begin{itemize}
  \item \( d \) = \# of digits in each number
  \item \( n \) = \# of elements to be sorted
  \item \( k \) = \# of possible values for each digit
\end{itemize}
Example 3. Bucket Sort

- Applicable if input is constrained to finite interval, e.g., [0…1).
- If input is random and uniformly distributed, expected run time is $\Theta(n)$. 
Bucket Sort

insert $A[i]$ into list $B[[n \cdot A[i]]]$
Loop Invariants

• Loop 1
  – The first \( i-1 \) keys have been correctly placed into buckets of width \( 1/n \).

• Loop 2
  – The keys within each of the first \( i-1 \) buckets have been correctly stable-sorted.
PseudoCode

**BUCKET-SORT**(A, n)

for $i \leftarrow 1$ to $n$
    do insert $A[i]$ into list $B[[n \cdot A[i]]]$

for $i \leftarrow 0$ to $n - 1$
    do sort list $B[i]$ with insertion sort

concatenate lists $B[0], B[1], \ldots, B[n - 1]$

return the concatenated lists

*Expected Running Time*

$\Theta(1)$

$\Theta(1) \times n$

$\Theta(n)$

$\Theta(n)$
Examples of Iterative Algorithms

• Binary Search
• Partitioning
• Insertion Sort
• Counting Sort
• Radix Sort
• Bucket Sort

• Which can be made stable?
• Which sort in place?
• How about MergeSort?
End of Iterative Algorithms