

- More on algorithms
 - Iterative structures
 - Recursive structures
 - Efficiency
 - Correctness
 - Verification of software

These slides were based on those of C Johnson, E McCreath and W Liang.

Iterative Structures

- Structures in which a collection of instructions is repeated in a looping manner
- Consider searching through a list of items, that are in sorted order, for a particular item
- Develop an algorithm that determines whether or not that element is in the list
 - In the list = success
 - Not in the list = failure

Iterative Structures

- Continue to search through the list as long as
 - There are more items to investigate
 - The target item is “greater” than that currently being examined
- Algorithm is called *sequential search*
- Is this algorithm “efficient” for really long lists ?

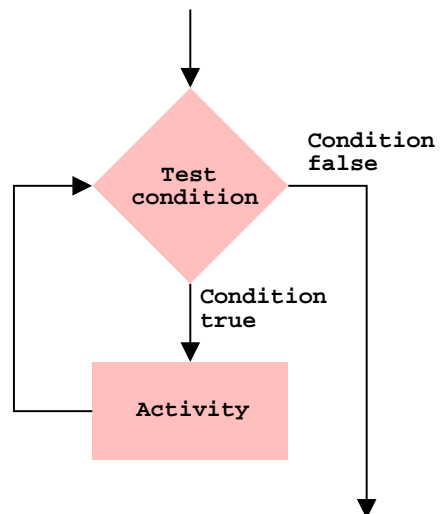
Sequential Search

```
procedure SeqSearch(List, TargetValue)
if (List empty)
then
  (Declare search a failure)
else
  (Select first entry in the list to be TestEntry;
  while (TargetValue > TestEntry and
  there are more entries to be considered)
  do (Select the next entry in the List as TestEntry);
  if (TargetValue = TestEntry)
  then (Declare search a success)
  else (Declare search a failure)
)
```

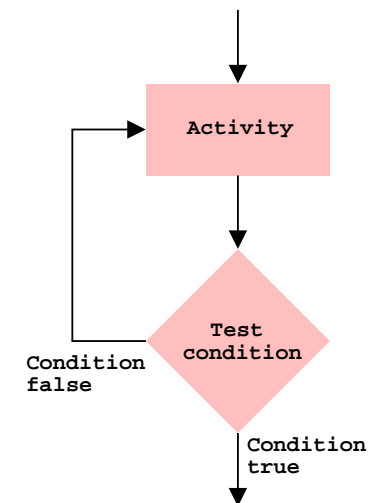
Loop Control

- **while** (*condition*) **do** (*body*)
 - check the *condition*
 - execute the *body*
 - check the *condition*
 - execute the *body* . . .
- Control of a loop consists of
 - Establishing an initial state
 - Compare current state to the termination condition
 - change the state

while (...) do (...)



repeat (...) until (...)



Insertion sort

- Sorts a list of elements into some order
- Sorts the list “within itself”
- e.g. sort 6 1 4 2 3 into increasing order
 - the first element (6) by itself is in order
 - the first two elements (6 1) are not
 - swap the first two elements to give 1 6 4 2 3
 - now the first two elements (1 6) by themselves are in order
 - the first three elements (1 6 4) are not
 - swap the 2nd and 3rd elements to give 1 4 6 2 3

Insertion sort

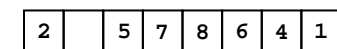
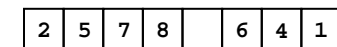
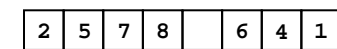
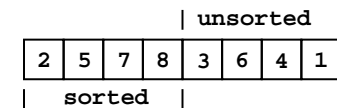
- now the first three elements (1 4 6) are in order
- the first four elements (1 4 6 2) are not
 - swap the 3rd and 4th elements and then the 2nd and 3rd elements to give 1 2 4 6 3
- now the first four elements (1 2 4 6) are in order
- the first five elements (1 2 4 6 3) are not
 - swap the 4th and 5th elements and then the 3rd and 4th elements to give 1 2 3 4 6

Insertion sort

- At any point the list has a sorted section and an unsorted section



- select the first element in the unsorted part
- move elements from the sorted part that are greater than this element, to the right
- store the element in the 'space' that is created



Insertion Sort

```

procedure Sort(List)
  N ← 2
  while ( the value of N does not exceed the length of List)
  do
    (Select the Nth element in List as Pivot;
     Move Pivot to temporary location leaving a gap;
     while (there is an element to the left of the gap
            that is greater than Pivot)
     do
       (move the element that is left of the gap, to the right)
     Move Pivot into the gap;
     N ← N + 1
  )

```

Recursive Structures

- Recursion is an alternative to iteration
- Iteration (using loops) involves repeating a set of instructions in a way such that the set is completed and repeated
- Recursion involves repeating the set of instructions as a set of itself e.g.
 - Factorial
 - Fibonacci numbers

Binary Search Algorithm

- Again, we need to see if a particular item is in a list
- Consider finding a name in a phone book
 - Check 'middle' entry (first entry in 2nd half)
 - If that's what you're looking for, you're done!
 - If the 'middle' entry is more than what you want, look in the first half
 - If the 'middle' entry is less than what you want, look in the second half
 - If item is not in List, algorithm will proceed to subdivide List until empty

Binary Search Algorithm

Original list	Original list	Original list
Alice		
Bob		
Carol		
David		
Elaine		
Fred		
George		
→ Harry		
Irene	Irene	Irene
John	John →	John
Kelly	Kelly	Kelly
Larry →	Larry	
Mary	Mary	
Nancy	Nancy	
Oliver	Oliver	

Binary Search

```
procedure Search(List, Target)
if (List empty)
then (report failure)
else [
    select 'middle' entry to be Test;
    if (Target = Test)
    then (report success);
    else (
        if (Target < Test)
        then (apply procedure Search to portion
            of list preceding Test)
        else (apply procedure Search to portion
            of list following Test)
    )
]
```

Recursion

- Each stage of the repetition is a subtask of the previous stage
- The illusion created by a recursive procedure is the existence of multiple copies of the procedure
- Recursive systems are dependent on proper control (as are loops)
- Need to test for termination condition (often called the base case)

Algorithm Efficiency

- Although today's machine execute millions of instructions per second, efficiency is still a major concern
- Suppose we have a 'big' list, say 16,384 elements
- Which algorithm would you use to search for an item
- Sequential search or Binary search ?
- Why?

Algorithm Efficiency

- How many comparisons does sequential search make on average
- About half the length of the list; some searches take more; some take less
- For a 16,384 element list, sequential search will make around 8,192 comparisons per search, on average
- If each comparison takes 10^{-3} seconds, such a search would average out to be 8.192 seconds (not very quick for not such a big list!)

Algorithm Efficiency

- Binary search proceeds by targeting the 'middle' entry
- If this is not the target then the search space is cut in half
- Starting with 16,384 elements, we continue with 8,192
- Then with 4,096, 2,048, 1,024, 512, ...
- Continuing like this, the target will be found (or algorithm will fail) after at most 16 comparisons
- If each comparison takes 10^{-3} seconds, such a search would average 0.016 seconds (quite quick!)

Best-case, Worst-case, Average-case

- Sequential search makes around $n/2$ comparisons on average for a list of length n
- Binary search makes around $\lg n (= \log_2 n)$ comparisons in its worst-case scenario for a list of length n

The performance of Insertion sort

- The best-case for insertion sort is a list of elements that are already sorted
- So it requires around $n - 1$ comparisons to conclude the list is already sorted
- The worst-case for insertion sort is a list of elements that are sorted in reverse order
- How many comparisons are needed to sort the list?

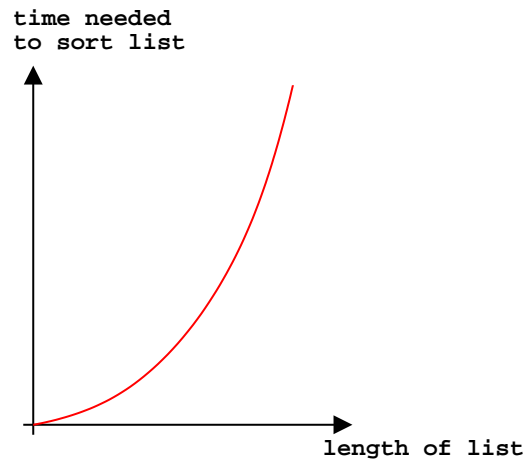
The performance of Insertion sort

- The first pivot (second item) is compared to one other entry
- The second pivot (third item) is compared to two other entries
- The final pivot is compared to $n - 1$ other entries
- Total number of comparisons is

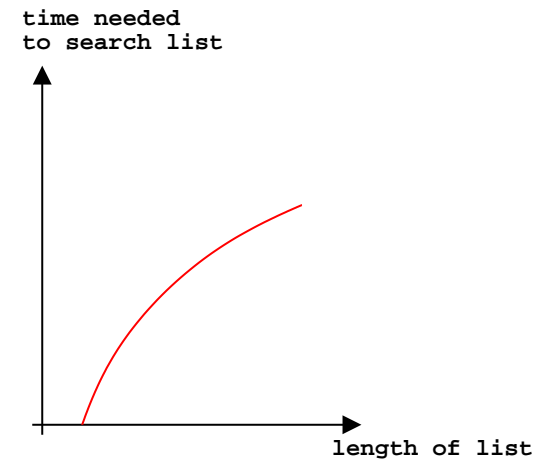
$$1 + 2 + 3 + \dots + n - 2 + n - 1 = \sum_{i=1}^{n-1} i = ??$$

- This is worst-case, what about average-case?

The worst-case performance of Insertion sort

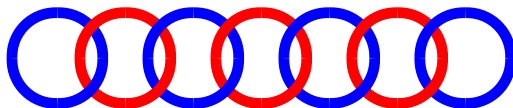


The worst-case performance of binary Search

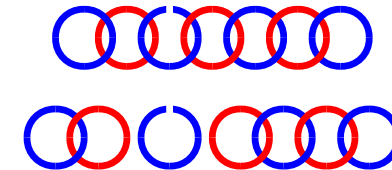


Software verification

- Recall Phase 4 of Polya's analysis of problem solving
- A traveller must stay in an isolated hotel for 7 nights
- He has a gold chain of seven links
- Each night costs one link of the chain
- What is the fewest number of links that must be cut so that he can pay for the room each morning without paying in advance?



Software verification



- How do we know the solution is correct?
- Since a single link must be given to the hotel on the first morning, at least one cut must be made
- Since, in our solution, only one cut is made, the solution must be optimal

Software verification

- A program that *is believed to be* correct and a program that **is** correct are not the same thing
- A proof of correctness assumes that certain conditions (pre-conditions) hold at the start of a program's execution
- The next step is to consider how the consequences of these conditions propagate through the program
- e.g. if a certain statement about Y is known prior to executing $X \leftarrow Y$, then the same statement can be made about X after the instruction has been executed

Software verification

- A proof of correctness proceeds by identifying statements (assertions) at various points in the program
- the result is a collection of assertions that lead to the point in the program where the last assertion was established
- If the assertion at the end of the program corresponds to the desired output, we can assume the program to be correct

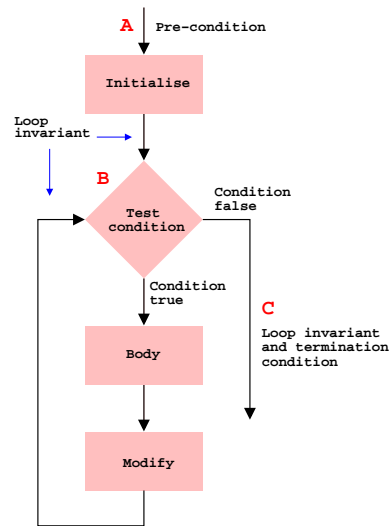
Software verification

- An assertion in a loop that is true each time that point in a loop is reached is called a loop invariant
- Suppose, during the execution of a while loop, we can establish that a particular assertion is true, each time the termination test is performed
- Then, when the loop terminates we can conclude that the loop invariant and the termination both hold

Software verification

- If these combined statements imply the desired output, showing that the initialisation and modification components of the loop lead to the termination condition, completes the proof of correctness

Loop Invariants



Software verification

- Compare this to our example of insertion sort
- The outer loop is based on the loop invariant
 - Each time the termination test is performed, the entries in the list from position 1 to N-1 are sorted
- The termination condition is
 - The value of N is greater than the length of the list
- If the loop terminates we know both conditions must be satisfied → the list is sorted

Insertion Sort

procedure Sort(List)

$N \leftarrow 2$

while (the value of N does not exceed the length of List)

do

(Select the Nth element in List as Pivot;

Move Pivot to temporary location leaving a gap;

while (there is an element to the left of the gap
that is greater than Pivot)

do

(move the element that is left of the gap, to the right)

Move Pivot into the gap;

$N \leftarrow N + 1$

)

Reading and Self Assessment

- Required reading:
 - Chapter 5, sections 3, 4, 5 and 6
- Questions:
 - 5.3 : 2, 3
 - 5.4 : 1, 2, 3, 4, 5
 - 5.5 : 1, 2, 3, 4
 - 5.6 : 1