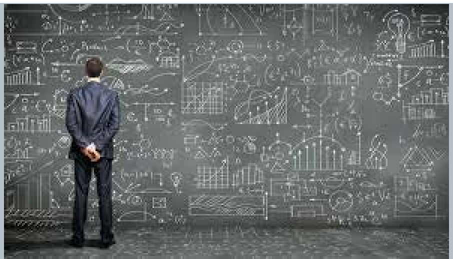
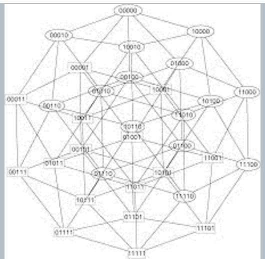
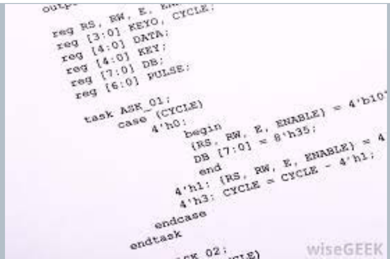
Introduction to Computer Science

William Hsu
Advanced Computation Laboratory
Department of Computer Science and Engineering
Department of Environmental Biology and Fisheries Science
National Taiwan Ocean University

Chapter 5: Algorithms

An algorithm is any set of detailed instructions which results in a predictable end-state from a known beginning. Algorithms are only as good as the instructions given, however, and the result will be incorrect if the algorithm is not properly defined.

- 5.1 The Concept of an Algorithm
- 5.2 Algorithm Representation
- 5.3 Algorithm Discovery
- 5.4 Iterative Structures
- 5.5 Recursive Structures
- 5.6 Efficiency and Correctness

The concept of an algorithm

- Algorithms from previous chapters
  - Converting from one base to another
  - Correcting errors in data
  - Compression
- Many researchers believe that every activity of the human mind is the result of an algorithm

Definition of Algorithm

- An algorithm is an **ordered** set of **unambiguous**, **executable** steps that defines a **terminating** process.
  - Parallel algorithms. (Not step by step)
  - Finite.
  - Solvable vs unsolvable.
  - Effective vs noneffective.
- A Terminating Process
  - Culminates with a result
  - Can include systems that run continuously
  - Hospital systems
  - Long Division Algorithm
- A Non-terminating Process
  - Does not produce an answer
  - Nondeterministic algorithms.

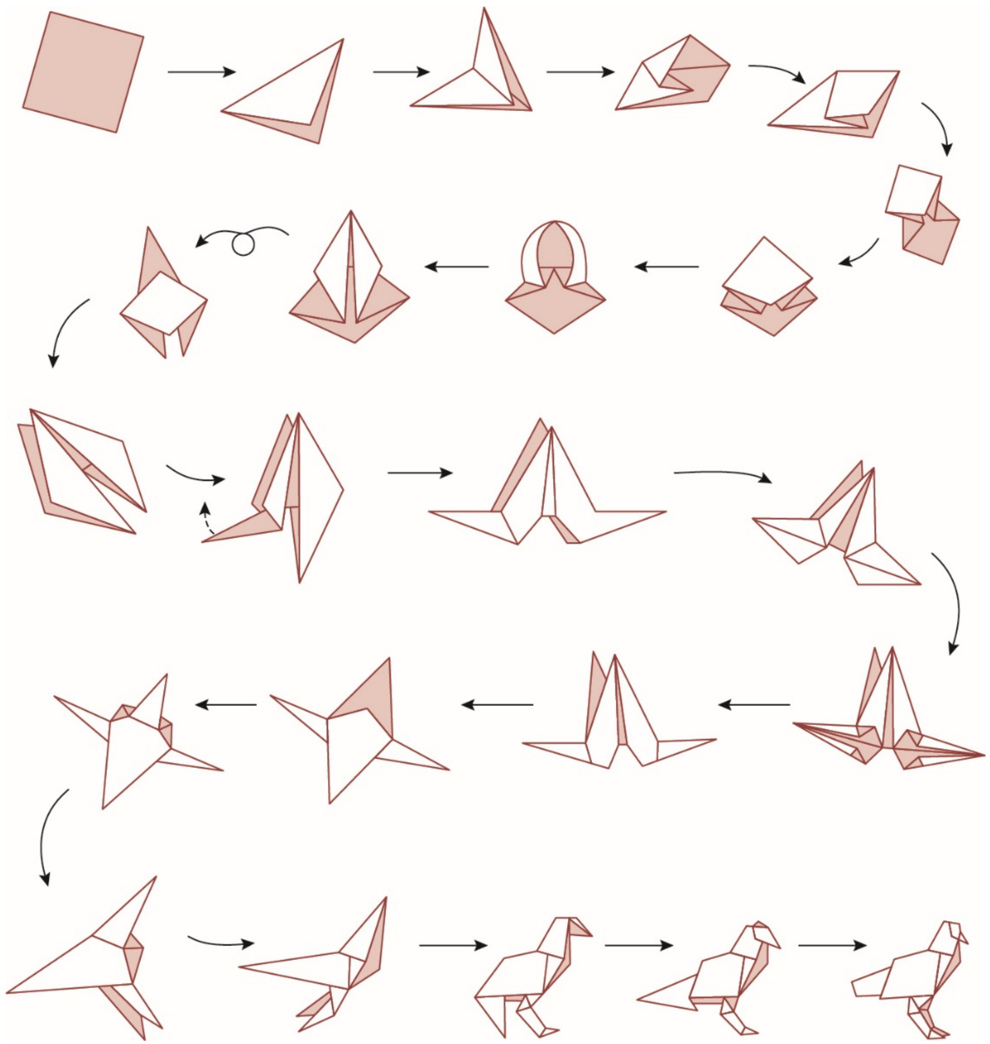
The abstract nature of algorithms

- There is a difference between an algorithm and its representation.
  - Analogy: difference between a story and a book
- A Program is a representation of an algorithm.
- A Process is the activity of executing an algorithm.

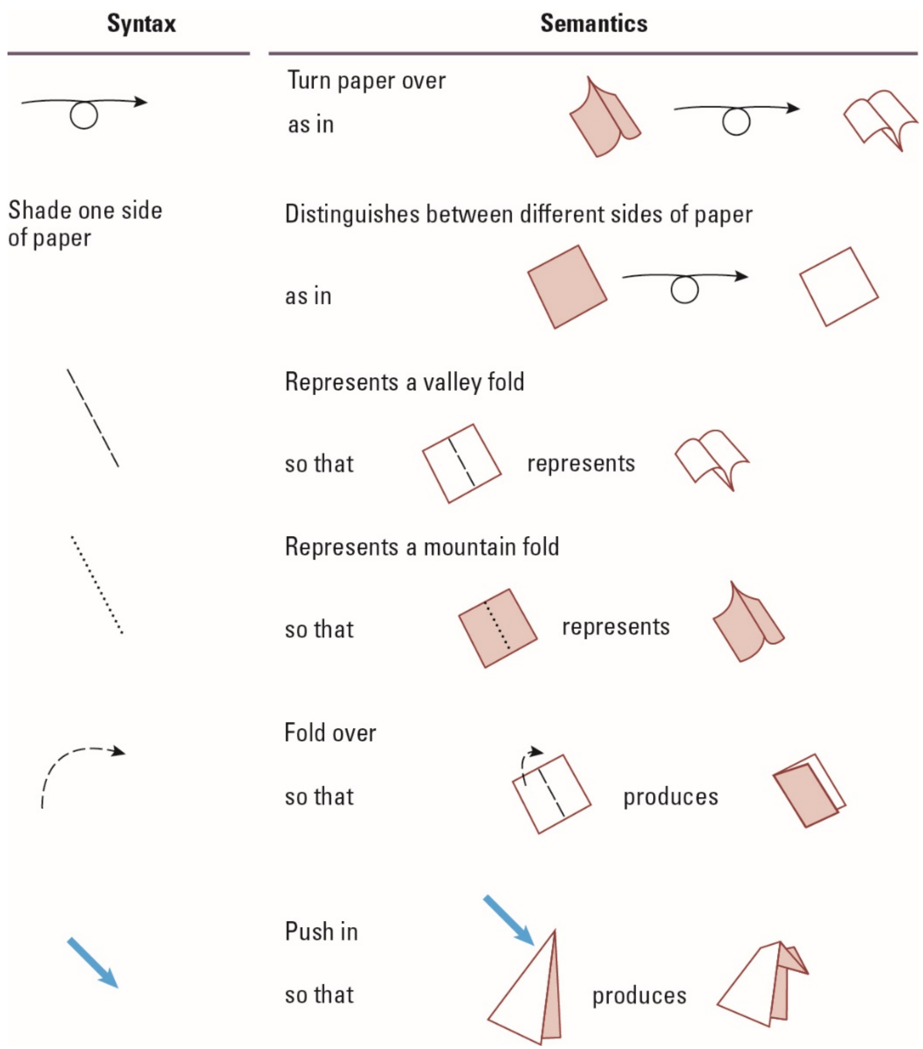
Algorithm Representation

- Requires well-defined primitives.
  - Some form of language. (Natural)
  - A collection of primitives constitutes a programming language.
- Is done informally with Pseudocode
  - Pseudocode is between natural language and a programming language.

Folding a Bird From a Square Piece of Paper



Origami Primitives



Data Type

- Data Type: A data type is a collection of objects and a set of operations that act on those objects.
- Abstract Data Type: An abstract data type(ADT) is a data type that is organized in such a way that the specification of the objects and the operations on the objects is separated from the representation of the objects and the implementation of the operations.

***Structure:**Abstract data type *Natural_Number*

structure Natural_Number is
  objects: an ordered subrange of the integers starting at zero and ending at the maximum integer (*INT_MAX*) on the computer
  functions:
    for all x, y ∈ *Nat_Number*; *TRUE, FALSE* ∈ *Boolean*
    and where +, -, <, and == are the usual integer operations.
    Nat_No Zero ( ) ::= 0
    Boolean Is_Zero(x) ::= **if** (x) **return** *FALSE*
        else return *TRUE*
    Nat_No Add(x, y) ::= **if** ((x+y) <= *INT_MAX*) **return** x+y
        else return *INT_MAX*
    Boolean Equal(x,y) ::= **if** (x== y) **return** *TRUE*
        else return *FALSE*
    Nat_No Successor(x) ::= **if** (x == *INT_MAX*) **return** x
        else return x+1
    Nat_No Subtract(x,y) ::= **if** (x<y) **return** 0
        else return x-y
  end *Natural_Number*

::= is defined as

Designing a pseudocode language

- Choose a common programming language
- Loosen some of the syntax rules
- Allow for some natural language
- Use consistent, concise notation
- We will use a Python-like Pseudocode

Pseudocode Primitives

- Assignment

```
name = expression
```

- Example

```
RemainingFunds = CheckingBalance +
      SavingsBalance
```

› Conditional selection

```
if (condition):
    activity
```

› Example

```
if (sales have decreased):
    lower the price by 5%
```

Pseudocode Primitives (continued)

- Repeated execution

```
while (condition):
    body
```

- Example

```
while (tickets remain to be sold):
    sell a ticket
```

- Indentation shows **nested** conditions

```
if (not raining):
    if (temperature == hot):
        go swimming
    else:
        play golf
else:
    watch television
```

Pseudocode Primitives (continued)

- Define a function

```
def name():
```

- Example

```
def ProcessLoan():
```

- Executing a function

```
if (. . .):
    ProcessLoan()
else:
    RejectApplication()
```

The Procedure *Greetings* in Pseudocode

```
def Greetings():
    Count = 3
    while (Count > 0):
        print('Hello')
        Count = Count - 1
```

Pseudocode Primitives (continued)

- Using parameters

```
def Sort(List):
      .
      .
```

- Executing Sort on different lists

```
Sort(the membership list)
Sort(the wedding guest list)
```

Algorithm discovery

- The first step in developing a program
- More of an art than a skill
- A challenging task

Polya's Problem Solving Steps

1. Understand the problem.
2. Devise a plan for solving the problem.
3. Carry out the plan.
4. Evaluate the solution for accuracy and its potential as a tool for solving other problems.

Polya's Steps in the Context of Program Development

1. Understand the problem.
2. Get an idea of how an algorithmic function might solve the problem.
3. Formulate the algorithm and represent it as a program.
4. Evaluate the solution for accuracy and its potential as a tool for solving other problems.

Getting a Foot in the Door

- Try working the problem backwards.
- Solve an easier related problem.
  - Relax some of the problem constraints.
  - Solve pieces of the problem first (bottom up methodology).
- Stepwise refinement: Divide the problem into smaller problems (top-down methodology).

Ages of Children Problem

- Person A is charged with the task of determining the ages of B’s three children.
  - B tells A that the product of the children’s ages is 36.
  - A replies that another clue is required.
  - B tells A the sum of the children’s ages.
  - A replies that another clue is needed.
  - B tells A that the oldest child plays the piano.
  - A tells B the ages of the three children.
- How old are the three children?

Ages of Children Problem

a. Triples whose product is 36

(1,1,36)	(1,6,6)
(1,2,18)	(2,2,9)
(1,3,12)	(2,3,6)
(1,4,9)	(3,3,4)

b. Sums of triples from part (a)

1 + 1 + 36 = 38	1 + 6 + 6 = 13
1 + 2 + 18 = 21	2 + 2 + 9 = 13
1 + 3 + 12 = 16	2 + 3 + 6 = 11
1 + 4 + 9 = 14	3 + 3 + 4 = 10

Iterative structures

- A collection of instructions repeated in a looping manner
- Examples include:
  - Sequential search algorithm
  - Insertion sort algorithm

Measurements

- Criteria
  - Is it correct?
  - Is it readable?
  - …
- Performance Analysis (machine independent)
  - space complexity: storage requirement
  - time complexity: computing time
- Performance Measurement (machine dependent)

The Sequential Search Algorithm in Pseudocode

```
def Search (List, TargetValue):
    if (List is empty):
        Declare search a failure
    else:
        Select the first entry in List to be TestEntry
        while (TargetValue > TestEntry and entries remain):
            Select the next entry in List as TestEntry
        if (TargetValue == TestEntry):
            Declare search a success
        else:
            Declare search a failure
```

Components of Repetitive Control

Initialize: Establish an initial state that will be modified toward the termination condition

Test: Compare the current state to the termination condition and terminate the repetition if equal

Modify: Change the state in such a way that it moves toward the termination condition

Iterative Structures

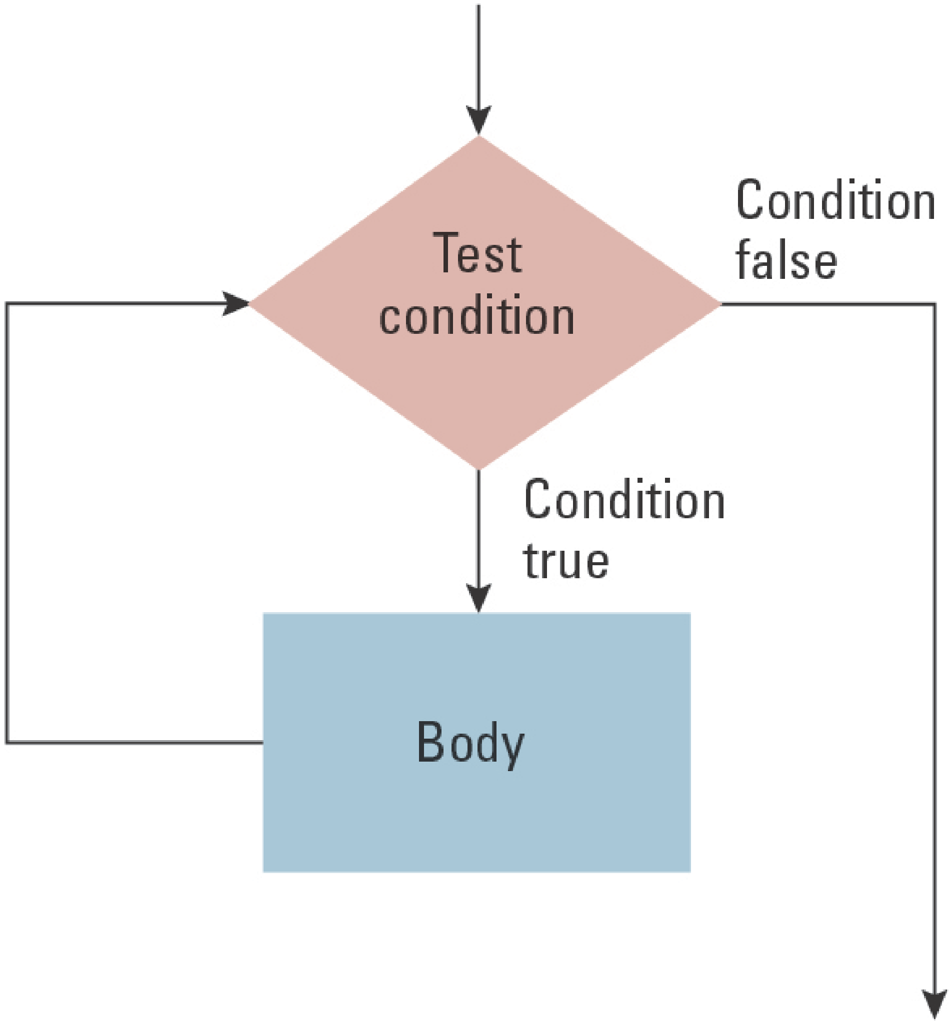
- Pretest loop:

```
while (condition):
    body
```

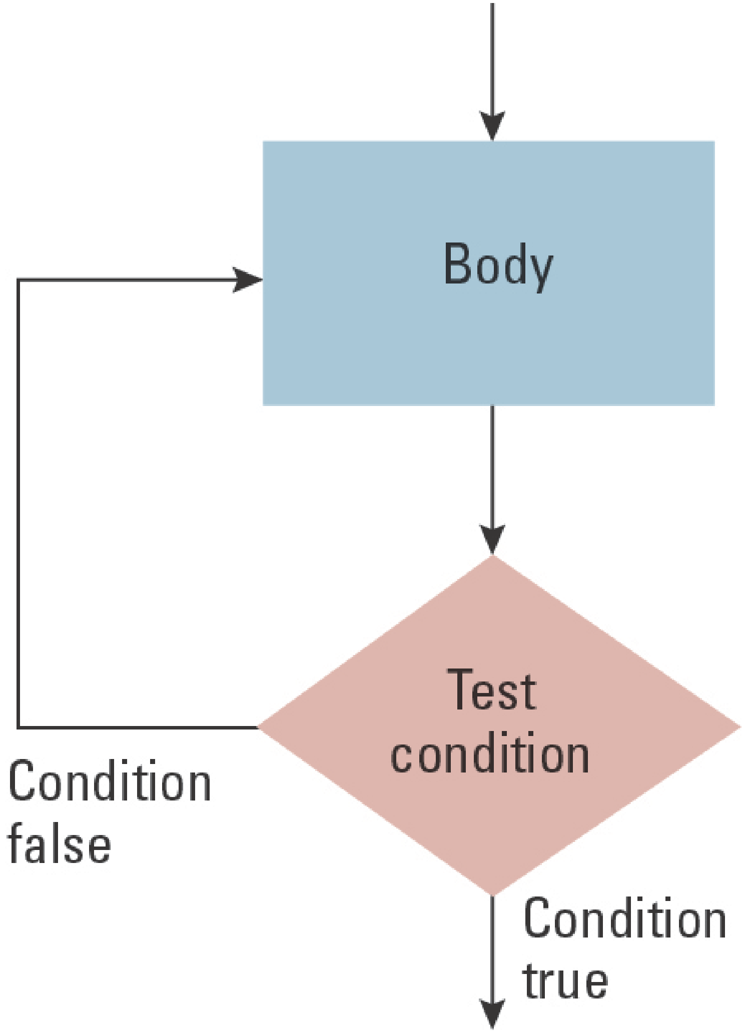
- Posttest loop:

```
repeat:
    body
until(condition)
```

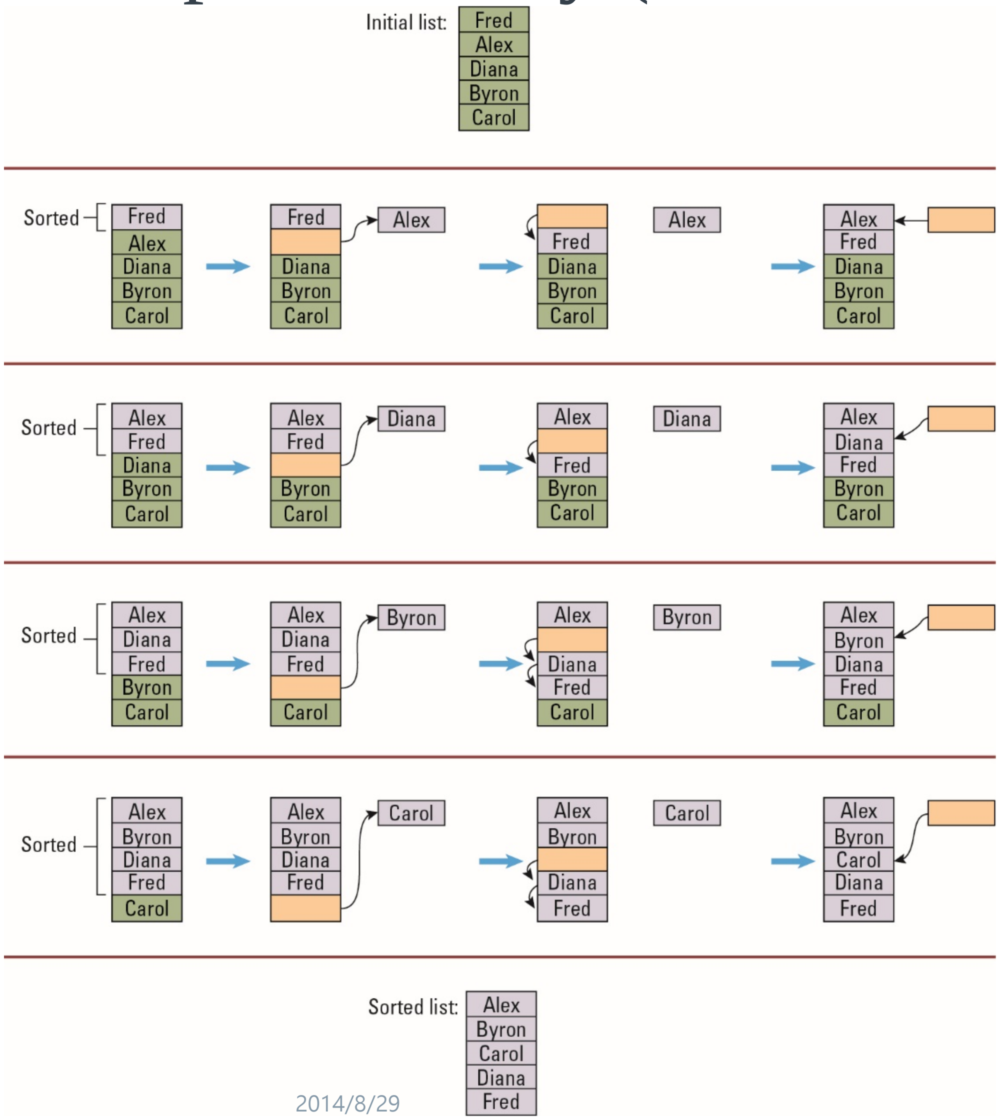
The `while` Loop Structure



The `repeat` Loop Structure



Sorting the List Fred, Alex, Diana, Byron, and Carol Alphabetically (Insertion Sort)



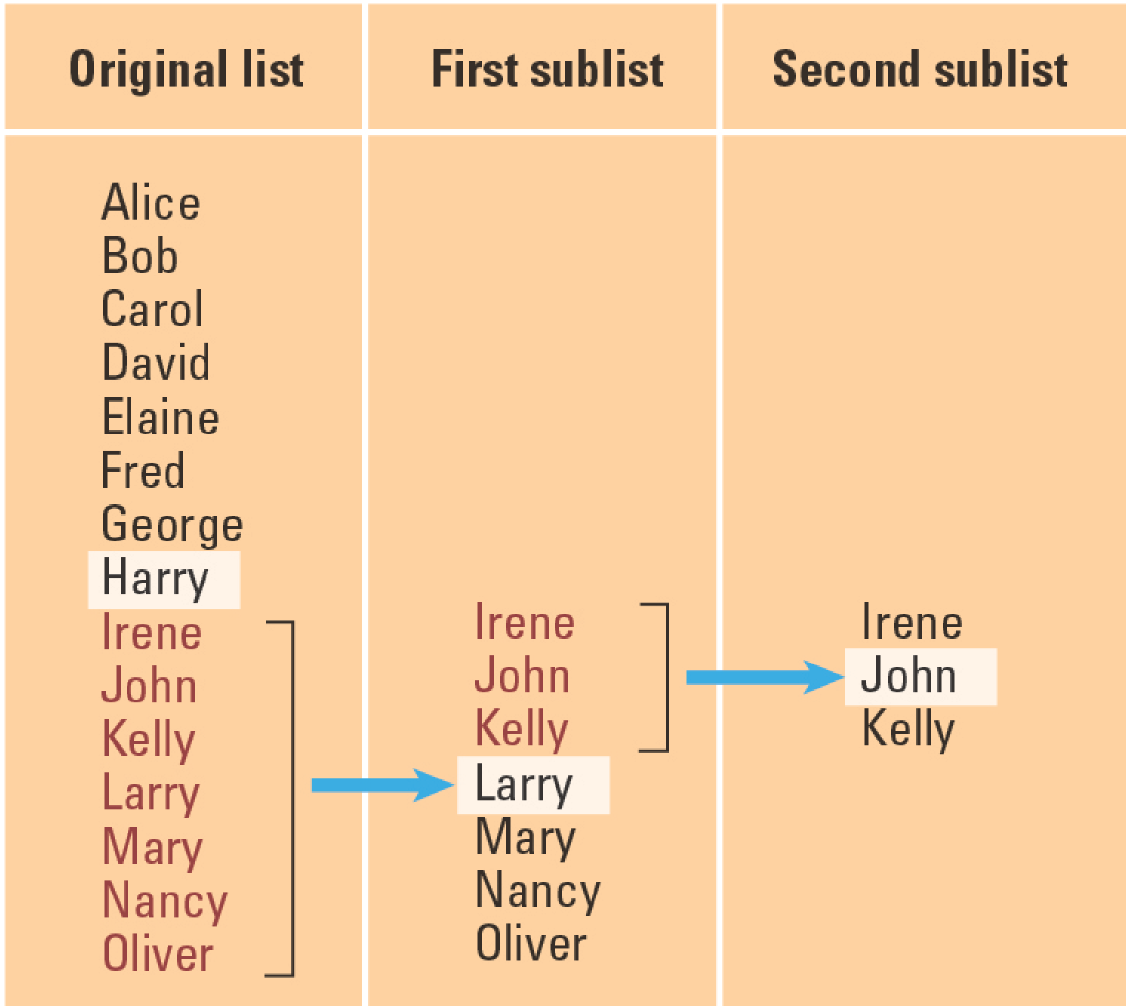
The Insertion Sort Algorithm Expressed in Pseudocode

```
def Sort(List):
    N = 2
    while (N <= length of List):
        Pivot = Nth entry in List
        Remove Nth entry leaving a hole in List
        while (there is an Entry above the
                  hole and Entry > Pivot):
            Move Entry down into the hole leaving
            a hole in the list above the Entry
        Move Pivot into the hole
        N = N + 1
```

Recursion

- Repeating the set of instructions as a subtask of itself.
- Multiple activations of the procedure are formed, all but one of which are waiting for other activations to complete.
- Example: The Binary Search Algorithm

Applying Our Strategy to Search a List for the Entry John



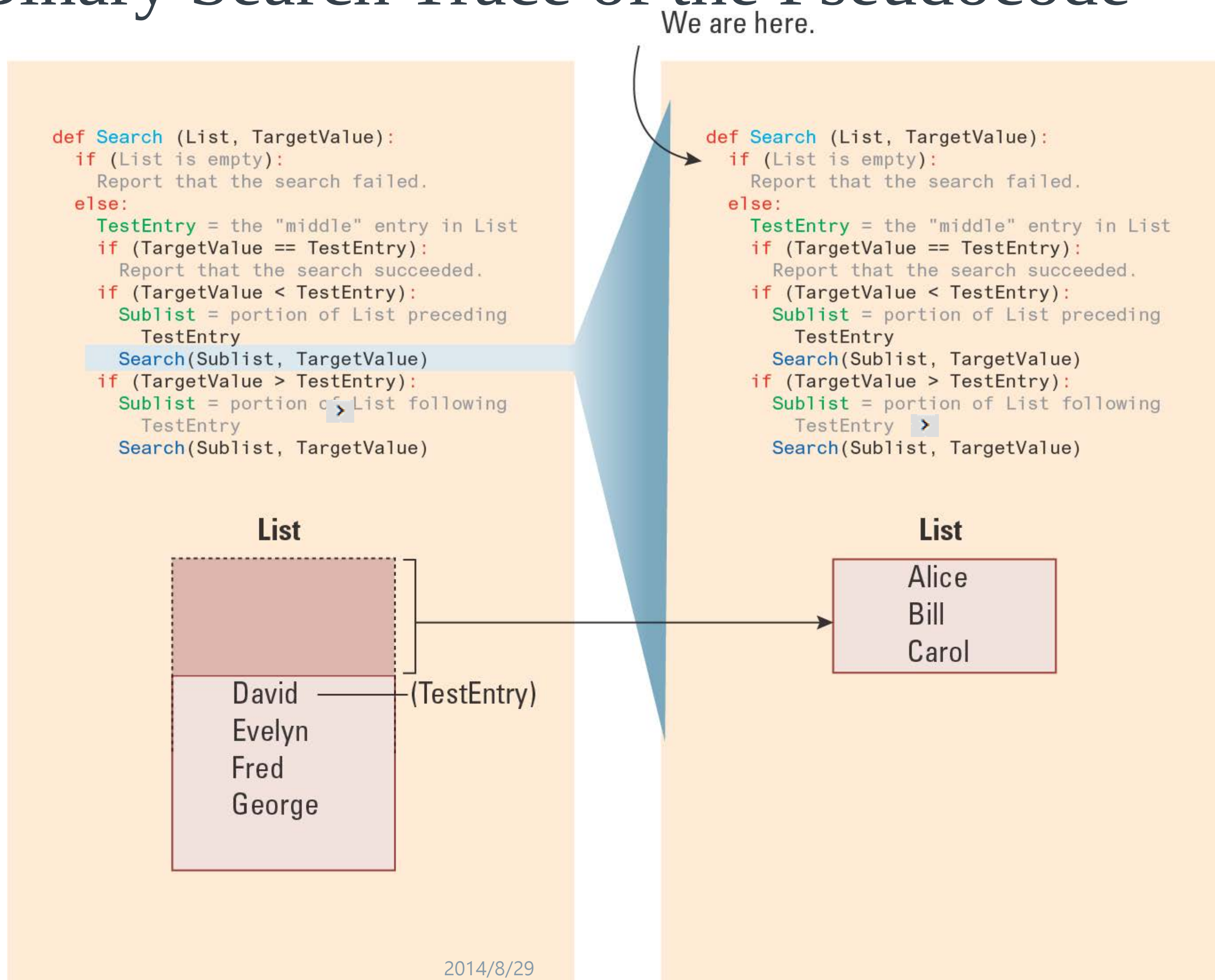
A First Draft of the Binary Search Technique

```
if (List is empty):
    Report that the search failed
else:
    TestEntry = middle entry in the List
    if (TargetValue == TestEntry):
        Report that the search succeeded
    if (TargetValue < TestEntry):
        Search the portion of List preceding TestEntry for
        TargetValue, and report the result of that search
    if (TargetValue > TestEntry):
        Search the portion of List following TestEntry for
        TargetValue, and report the result of that search
```

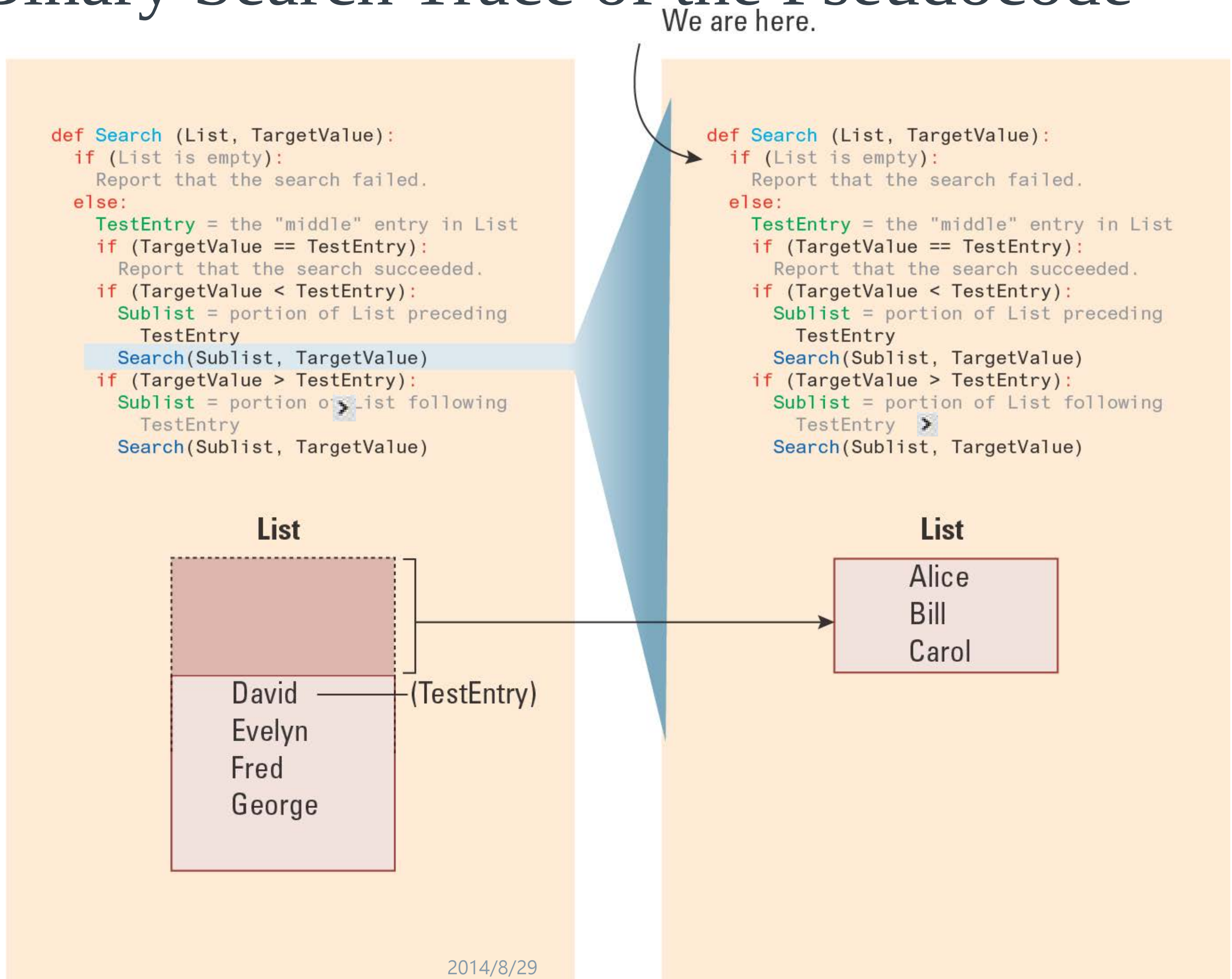
The Binary Search Algorithm in Pseudocode

```
def Search(List, TargetValue):
    if (List is empty):
        Report that the search failed
    else:
        TestEntry = middle entry in the List
        if (TargetValue == TestEntry):
            Report that the search succeeded
        if (TargetValue < TestEntry):
            Sublist = portion of List preceding TestEntry
            Search(Sublist, TargetValue)
        if (TargetValue > TestEntry):
            Sublist = portion of List following TestEntry
            Search(Sublist, TargetValue)
```

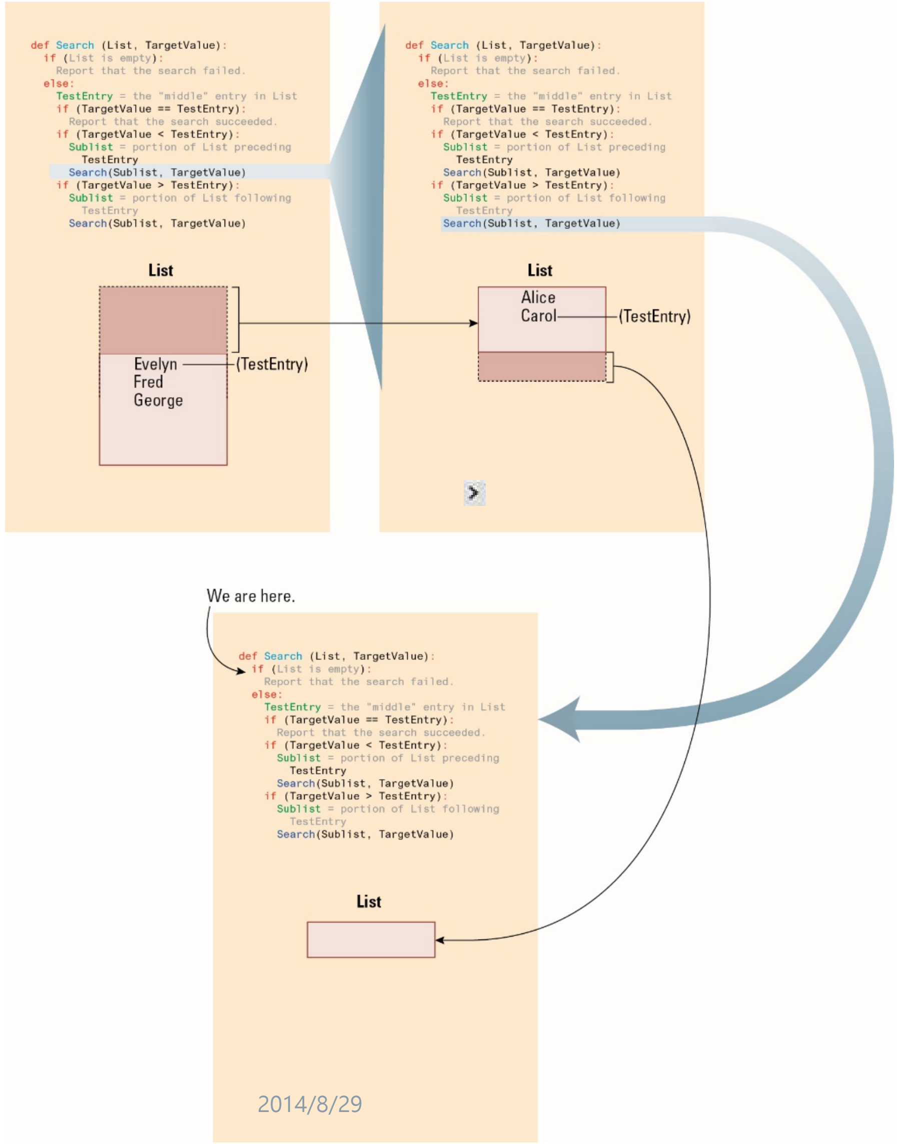
Binary Search Trace of the Pseudocode



Binary Search Trace of the Pseudocode



Binary Search Trace of the Pseudocode



Recursive control

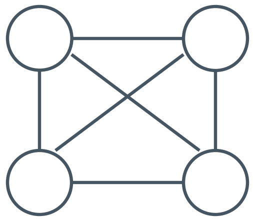
› Requires initialization, modification, and a test for termination (base case)

› Provides the illusion of multiple copies of the function, created dynamically in a telescoping manner

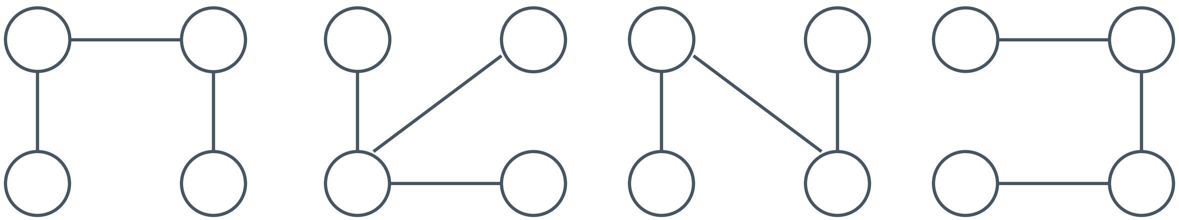
› Only one copy is actually running at a given time, the others are waiting

Spanning trees

- Suppose you have a connected undirected graph.
  - Connected: every node is reachable from every other node.
  - Undirected: edges do not have an associated direction.
- A **spanning tree** of the graph is a connected subgraph in which there are no cycles.



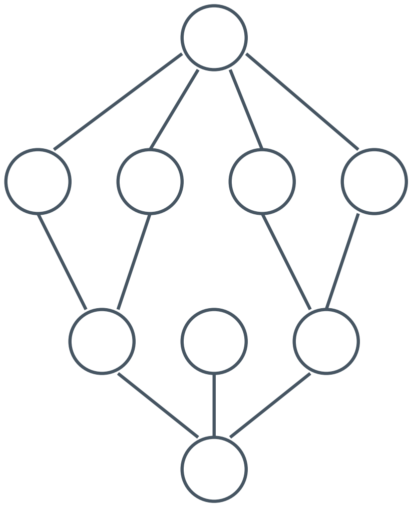
A connected, undirected graph



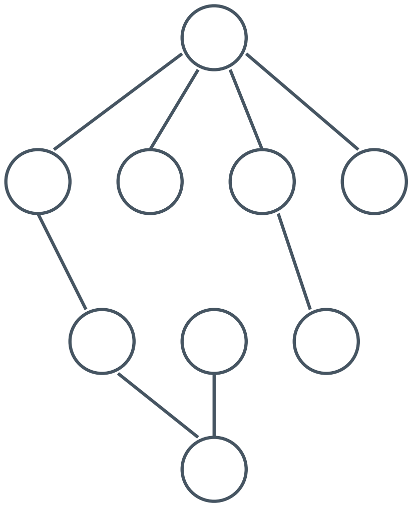
Four of the spanning trees of the graph

Constructing a spanning tree

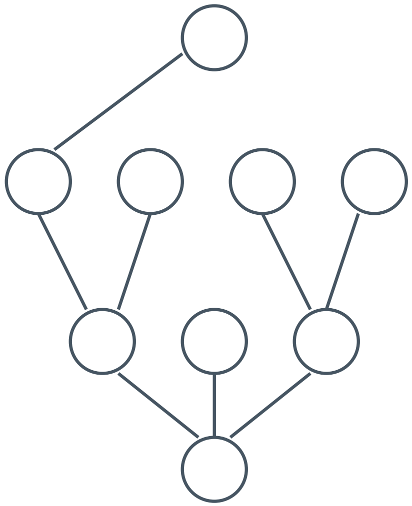
```
pick an initial node and call it part of the
spanning tree
do a search from the initial node:
    each time you find a node that is not in the
    spanning tree, add to the spanning tree both the
    new node and the edge you followed to get to it
```



An undirected graph



One possible result of a BFS starting from top



One possible result of a DFS starting from top

Minimum cost spanning trees

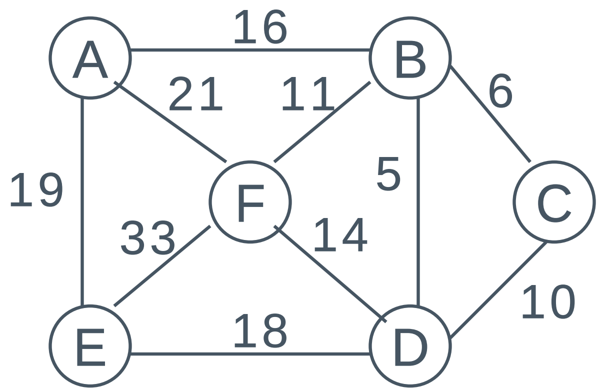
- Suppose you want to supply a set of houses (say, in a new subdivision) with:
  - electric power
  - water
  - sewage lines
  - telephone lines
- To keep costs down, you could connect these houses with a spanning tree (of, for example, power lines).
  - The houses are not all equal distances apart.
- To reduce costs even further, you could connect the. houses with a minimum-cost spanning tree.

Minimum-cost spanning trees

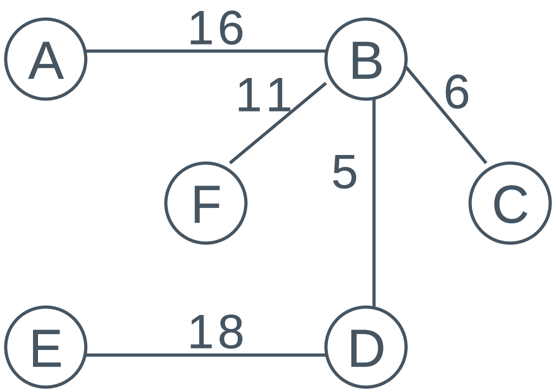
› Suppose you have a connected undirected graph with a weight (or cost) associated with each edge

› The cost of a spanning tree would be the sum of the costs of its edges

› A minimum-cost spanning tree is a spanning tree that has the lowest cost



A connected, undirected graph



A minimum-cost spanning tree

Finding spanning trees

- There are two basic algorithms for finding minimum-cost spanning trees, and both are greedy algorithms

- **Kruskal's algorithm**: Start with no nodes or edges in the spanning tree, and repeatedly add the cheapest edge that does not create a cycle
  - Here, we consider the spanning tree to consist of edges only

- **Prim's algorithm**: Start with any one node in the spanning tree, and repeatedly add the cheapest edge, and the node it leads to, for which the node is not already in the spanning tree.
  - Here, we consider the spanning tree to consist of both nodes and edges

Kruskal’s algorithm

```
T = empty spanning tree;
E = set of edges;
N = number of nodes in graph;

while T has fewer than N - 1 edges {
   remove an edge (v, w) of lowest cost from E
   if adding (v, w) to T would create a cycle
      then discard (v, w)
      else add (v, w) to T
}
```

- Finding an edge of lowest cost can be done just by sorting the edges
- Efficient testing for a cycle requires a fairly complex algorithm (UNION-FIND) which we don’t cover in this course

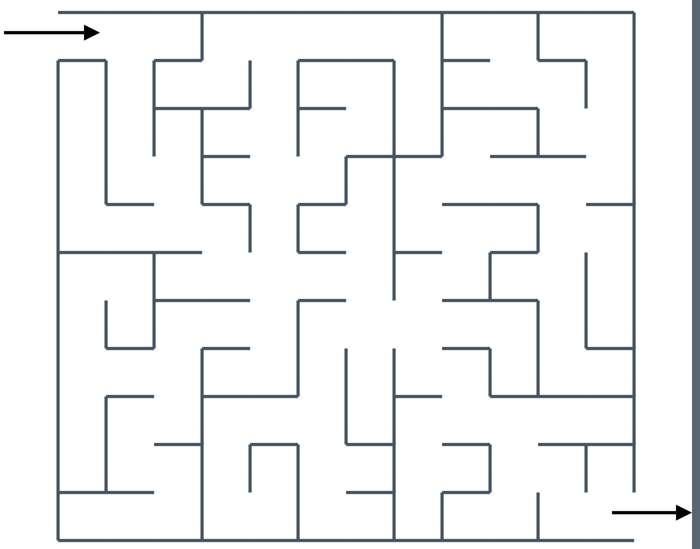
Prim's algorithm

```
T = a spanning tree containing a single
node s;
E = set of edges adjacent to s;
while T does not contain all the nodes {
   remove an edge (v, w) of lowest cost from E
   if w is already in T then discard edge (v, w)
   else {
      add edge (v, w) and node w to T
      add to E the edges adjacent to w
   }
}
```

- An edge of lowest cost can be found with a priority queue
- Testing for a cycle is automatic
  - Hence, Prim's algorithm is far simpler to implement than Kruskal's algorithm

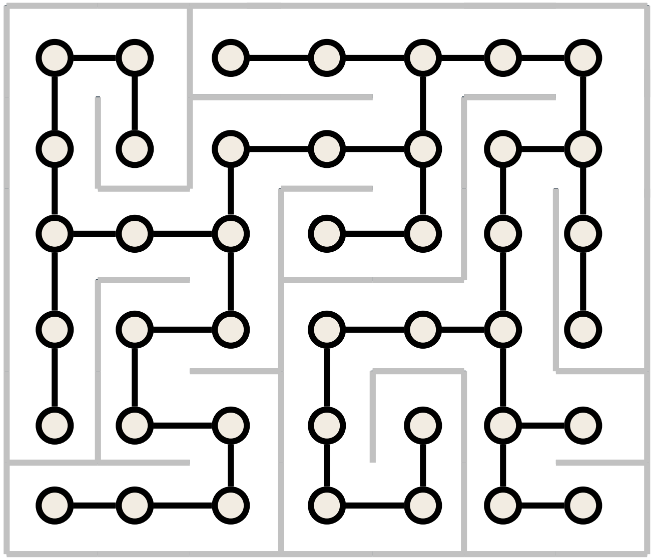
Mazes

- Typically,
  - Every location in a maze is reachable from the starting location
  - There is only one path from start to finish
- If the cells are “vertices” and the open doors between cells are “edges,” this describes a spanning tree
- Since there is exactly one path between any pair of cells, any cells can be used as “start” and “finish”
- This describes a spanning tree



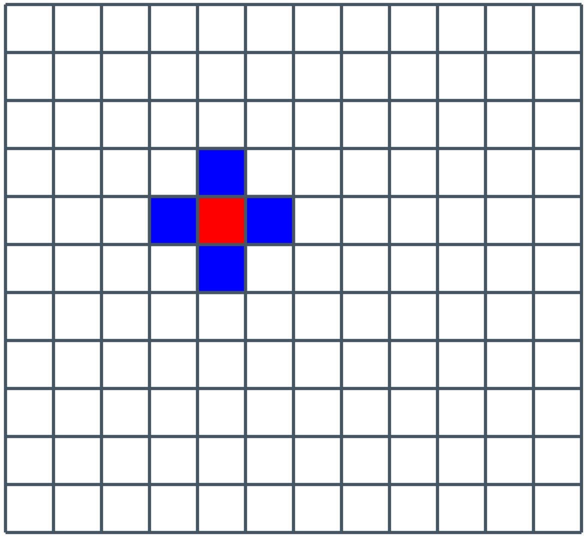
Mazes as spanning trees

- While not every maze is a spanning tree, most can be represented as such
- The nodes are “places” within the maze
- There is exactly one cycle-free path from any node to any other node



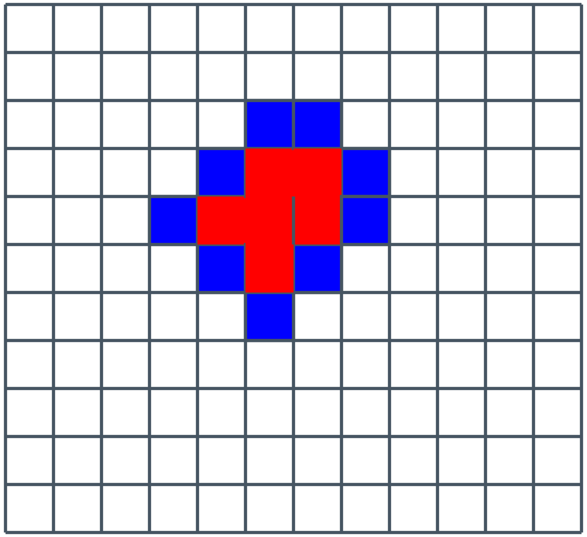
Building a maze I

- This algorithm requires two sets of cells
  - the set of cells already in the spanning tree, IN
  - the set of cells adjacent to the cells in the spanning tree (but not in it themselves), called the FRONTIER
- Start with all walls present
- Pick any cell and put it into IN (red)
- Put all adjacent cells, that aren't in IN, into FRONTIER (blue)



Building a maze II

- Repeatedly do the following:
  - Remove any one cell C from FRONTIER and put it in IN
  - Erase the wall between C and some one adjacent cell in IN
  - Add to FRONTIER all the cells adjacent to C that aren't in IN (or in FRONTIER already)
  - Continue until there are no more cells in FRONTIER
  - When the maze is complete (or at any time), choose the start and finish cells



Efficiency and correctness

- The choice between efficient and inefficient algorithms can make the difference between a practical solution and an impractical one
- The correctness of an algorithm is determined by reasoning formally about the algorithm, not by testing its implementation

Efficiency

- Measured as number of instructions executed
- Uses big theta notation:
- Example: Insertion sort is in $\Theta(n^2)$
- Incorporates best, worst, and average case analysis

Algorithm Analysis

- Space complexity
  - How much space is required
- Time complexity
  - How much time does it take to run the algorithm
- Often, we deal with estimates!

Space Complexity

- Space complexity = The amount of memory required by an algorithm to run to completion
  - [**Core dumps** = the most often encountered cause is “dangling pointers”]
- Some algorithms may be more efficient if data completely loaded into memory
  - Need to look also at system limitations
  - E.g. Classify 2GB of text in various categories [politics, tourism, sport, natural disasters, etc.] – can I afford to load the entire collection?

Space Complexity

1. Fixed part: The size required to store certain data/variables, that is independent of the size of the problem:

   - e.g. name of the data collection

   - same size for classifying 2GB or 1MB of texts

2. Variable part: Space needed by variables, whose size is dependent on the size of the problem:

   - e.g. actual text

   - load 2GB of text VS. load 1MB of text

Space Complexity

- S(P) = c + S(instance characteristics)
  - c = constant
- Example:

```
float summation(const float (&a)[10], int n )
{
    float s = 0;
    int i;
    for(i = 0; i<n; i++) {
        s+= a[i];
    }
    return s;
}
```

- Space requirement: one for `n`, one for a [passed by reference!], one for `i` → constant space!

Time Complexity

- Often more important than space complexity
  - space available (for computer programs!) tends to be larger and larger
  - time is still a problem for all of us

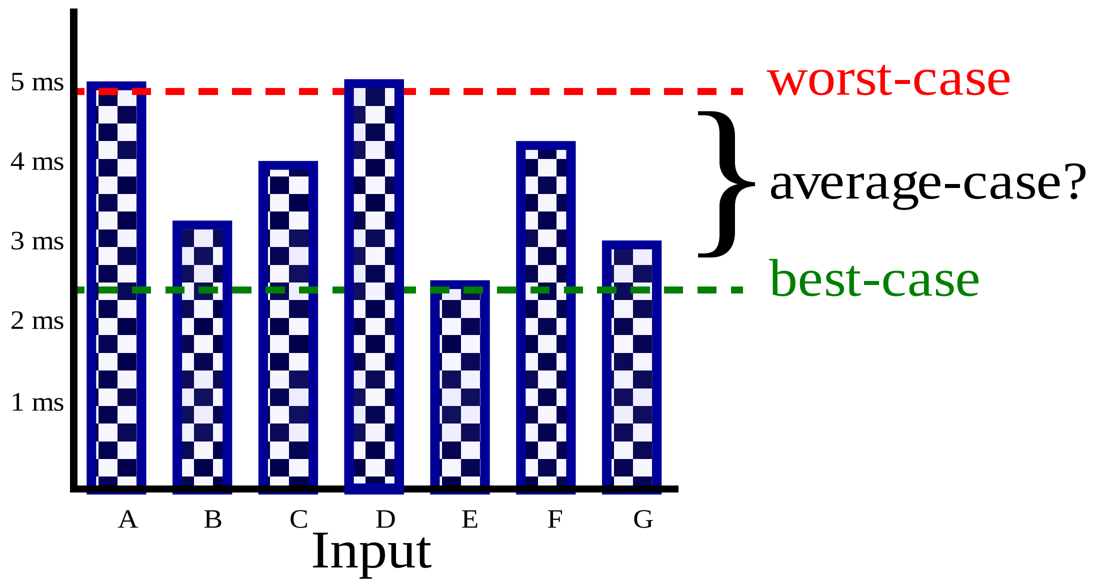
- 3-4GHz processors on the market
  - still …
  - researchers estimate that the computation of various transformations for 1 single DNA chain for one single protein on 1 TerraHZ computer would take about 1 year to run to completion

- Algorithms running time is an important issue

Running Time

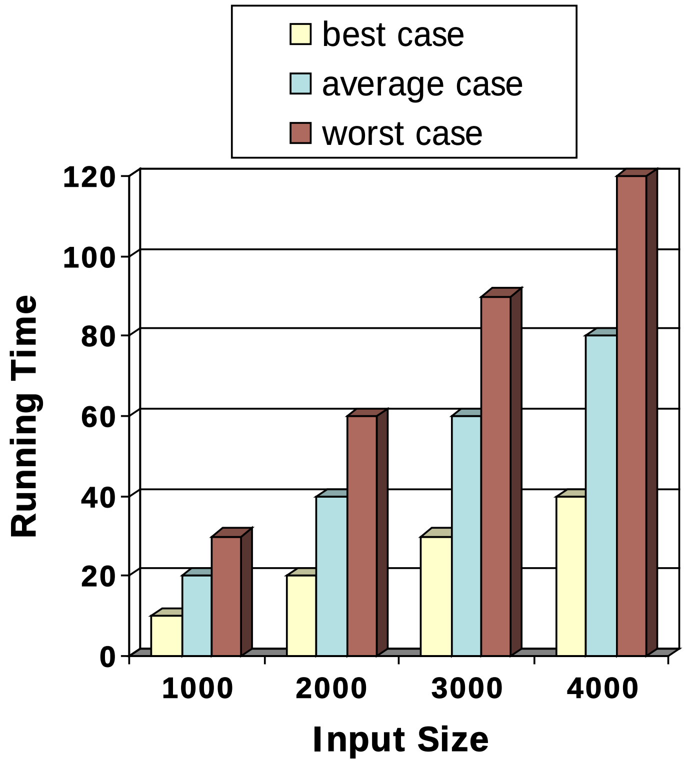
› Suppose the program includes an *if-then* statement that may execute or not: → variable running time

› Typically algorithms are measured by their ***worst case***



Running Time

- The running time of an algorithm varies with the inputs, and typically grows with the size of the inputs.
- To evaluate an algorithm or to compare two algorithms, we focus on their relative rates of growth wrt the increase of the input size.
- The average running time is difficult to determine.
- We focus on the worst case running time
  - Easier to analyze
  - Crucial to applications such as finance, robotics, and games



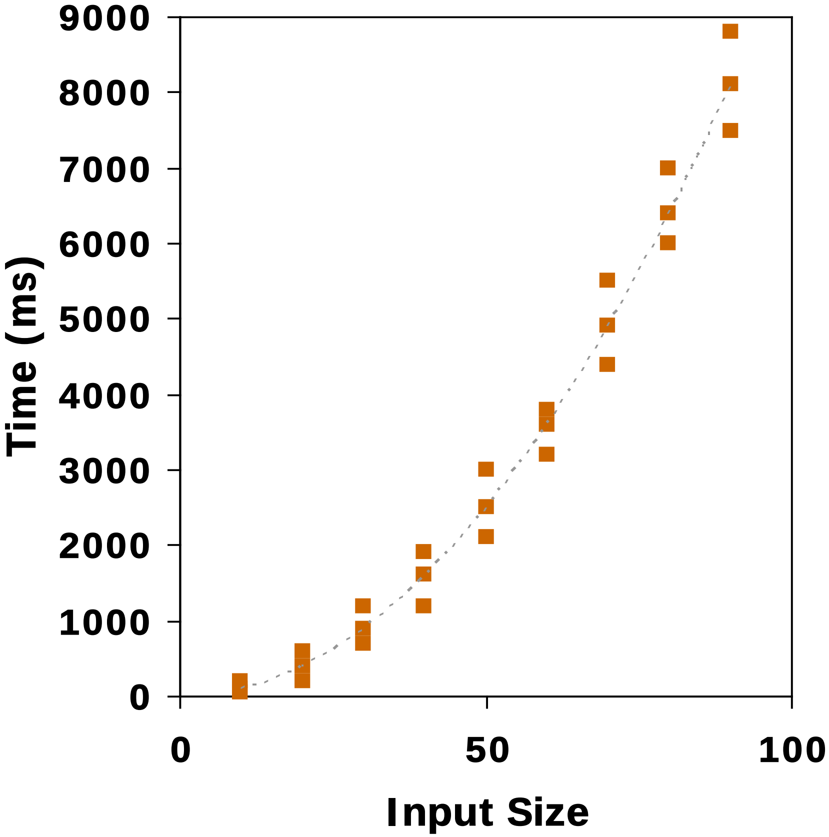
Running Time

- Problem: prefix averages
  - Given an array X
  - Compute the array A such that A[i] is the average of elements X[0] … X[i], for i=0..n-1
- Sol 1
  - At each step i, compute the element X[i] by traversing the array A and determining the *sum* of its elements, respectively the average
- Sol 2
  - At each step i update a *sum* of the elements in the array A
  - Compute the element X[i] as *sum/I*

Big question: Which solution to choose?

Experimental Approach

- Write a program to implement the algorithm.
- Run this program with inputs of varying size and composition.
- Get an accurate measure of the actual running time (e.g. system call date).
- Plot the results.
- Problems?



Limitations of Experimental Studies

- The algorithm has to be implemented, which may take a long time and could be very difficult.
- Results may not be indicative for the running time on other inputs that are not included in the experiments.
- In order to compare two algorithms, the same hardware and software must be used.

Use a Theoretical Approach

- Based on high-level description of the algorithms, rather than language dependent implementations
- Makes possible an evaluation of the algorithms that is independent of the hardware and software environments

→ **Generality**

Pseudocode

- High-level description of an algorithm.
- More structured than plain English.
- Less detailed than a program.
- Preferred notation for describing algorithms.
- Hides program design issues.

Example: find the max element of an array

Algorithm ***arrayMax***(***A***, ***n***)
  Input array ***A*** of ***n*** integers
  Output maximum element of ***A***

  currentMax $\leftarrow$ ***A***[0]
  for ***i*** $\leftarrow$ 1 **to** ***n*** – 1 **do**
    if *A*[*i*] > ***currentMax*** **then**
      currentMax $\leftarrow$ ***A***[***i***]
  return ***currentMax***

Primitive Operations

- The basic computations performed by an algorithm
- Identifiable in pseudocode
- Largely independent from the programming language
- Exact definition not important
- Use comments
- Instructions have to be basic enough and feasible!

- Examples:
  - Evaluating an expression
  - Assigning a value to a variable
  - Calling a method
  - Returning from a method

Low Level Algorithm Analysis

- Based on primitive operations (low-level computations independent from the programming language)
- E.g.:
  - Make an addition = 1 operation
  - Calling a method or returning from a method = 1 operation
  - Index in an array = 1 operation
  - Comparison = 1 operation etc.
- Method: Inspect the pseudo-code and count the number of primitive operations executed by the algorithm

Counting Primitive Operations

- By inspecting the code, we can determine the number of primitive operations executed by an algorithm, as a function of the input size.

Algorithm *arrayMax*(A, n)	# operations
currentMax $\leftarrow A[0]$	2
for $i \leftarrow 1$ **to** $n - 1$ **do**	$2 + n$
if $A[i] >$ *currentMax* **then**	$2(n - 1)$
currentMax $\leftarrow A[i]$	$2(n - 1)$
{ increment counter i }	$2(n - 1)$
return *currentMax*	1
Total	$7n - 1$

Estimating Running Time

- Algorithm $arrayMax$ executes $7n$ – 1 primitive operations.
- Let's define
  - a:= Time taken by the fastest primitive operation
  - b:= Time taken by the slowest primitive operation
- Let $T(n)$ be the actual running time of arrayMax. We have
$$a\ (7n - 1) \leq T(n) \leq b(7n - 1)$$
- Therefore, the running time $T(n)$ is bounded by two linear functions.

Growth Rate of Running Time

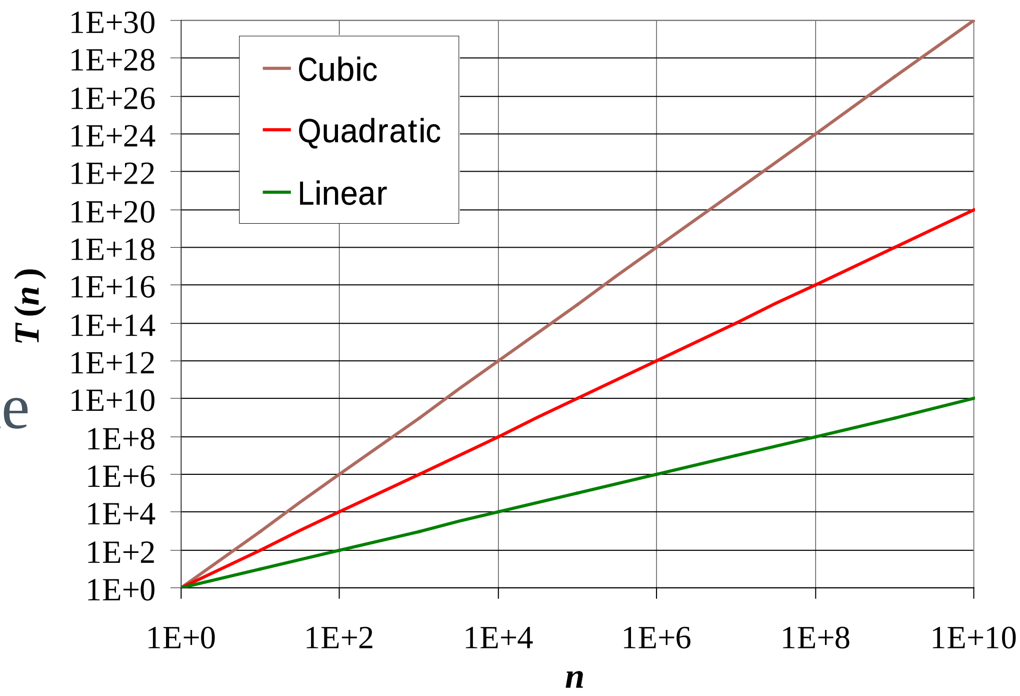
- Changing computer hardware / software
  - Affects $T(n)$ by a constant factor
  - Does not alter the growth rate of $T(n)$

- The linear growth rate of the running time $T(n)$ is an intrinsic property of algorithm $arrayMax$

Growth Rates

- Growth rates of functions:
  - Linear $\approx n$
  - Quadratic $\approx n^2$
  - Cubic $\approx n^3$

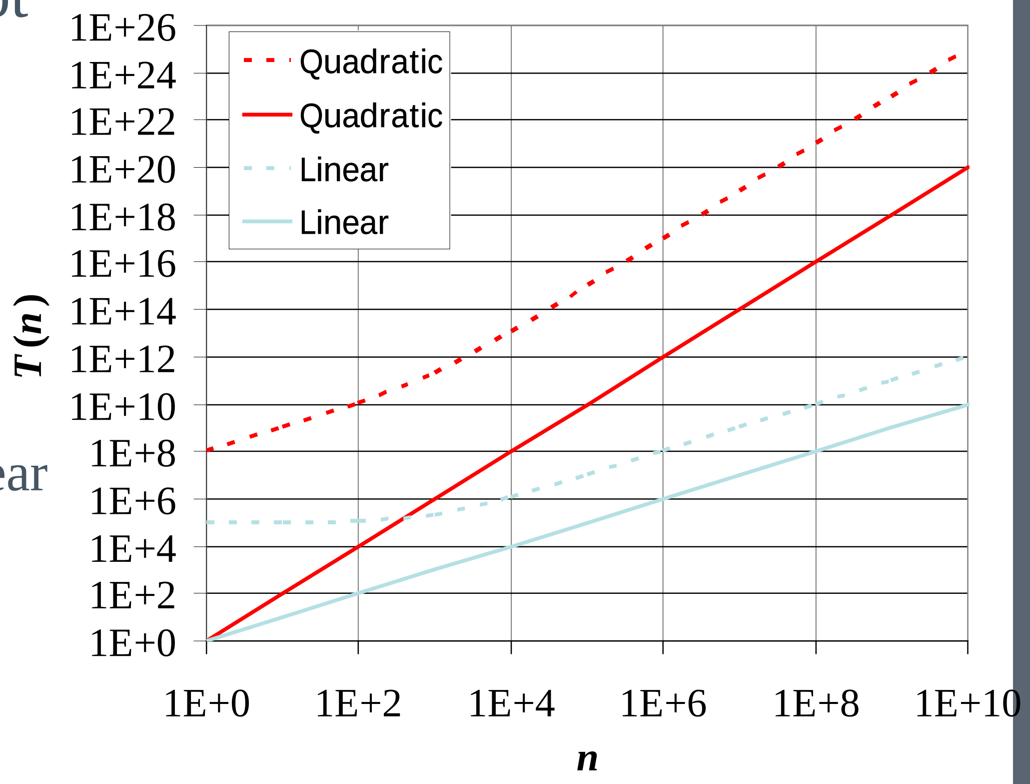
- In a **log-log** chart, the slope of the line corresponds to the growth rate of the function



Constant Factors

- The growth rate is not affected by
  - constant factors or
  - lower-order terms

- Examples
  - $10^2n + 10^5$ is a linear function
  - $10^5n^2 + 10^8n$ is a quadratic function



Asymptotic Notation

- Need to abstract further
- Give an “idea” of how the algorithm performs
- n steps vs. $n + 5$ steps
- n steps vs. n^2 steps

Example

- Complexity of $c_1 n^2 + c_2 n$ and $c_3 n$
  - for sufficiently large of value, $c_3 n$ is faster than $c_1 n^2 + c_2 n$
  - for small values of n, either could be faster
    - c_1=1, c_2=2, c_3=100 --> $c_1 n^2 + c_2 n \leq c_3 n$ for $n \geq 98$
    - c_1=1, c_2=2, c_3=1000 --> $c_1 n^2 + c_2 n \leq c_3 n$ for $n \geq 998$
  - break even point
    - no matter what the values of c_1, c_2, and c_3, the n beyond which $c_3 n$ is always faster than $c_1 n^2 + c_2 n$

Problem

- Fibonacci numbers
  - $F[0] = 0$
  - $F[1] = 1$
  - $F[i] = F[i-1] + F[i-2]$ for $i \geq 2$
- Pseudo-code
- Number of operations

Iterative summing of a list of numbers

```
float sum(float list[ ], int n){
    float tempsum = 0; count++; /* for assignment */
    int i;
    for (i = 0; i < n; i++) {
          count++;                /*for the for loop */
          /* for assignment */
          tempsum += list[i]; count++;
    }
    count++;          /* last execution of for */
    return tempsum;
    count++;          /* for return */
 }
```

2n + 3 steps

Asymptotic Notation Big-O

- Definition
  $f(n) = O(g(n))$ iff there exist positive constants c and n_0 such that $f(n) \leq cg(n)$ for all n, $n \geq n_0$.
- Examples
  - $3n + 2 = O(n)$ /* $3n + 2 \leq 4n$ for $n \geq 2$ */
  - $3n + 3 = O(n)$ /* $3n + 3 \leq 4n$ for $n \geq 3$ */
  - $100n + 6 = O(n)$ /* $100n + 6 \leq 101n$ for $n \geq 10$ */
  - $10n^2 + 4n + 2 = O(n^2)$ /* $10n2 + 4n + 2 \leq 11n^2$ for $n \geq 5$ */
  - $6 * 2^{\mathrm{n}} + n^2 = O(2^n)$ /* $6 * 2^2 + n^2 \leq 7 * 2^{\mathrm{n}}$ for $n \geq 4$ */

Common Big-Os

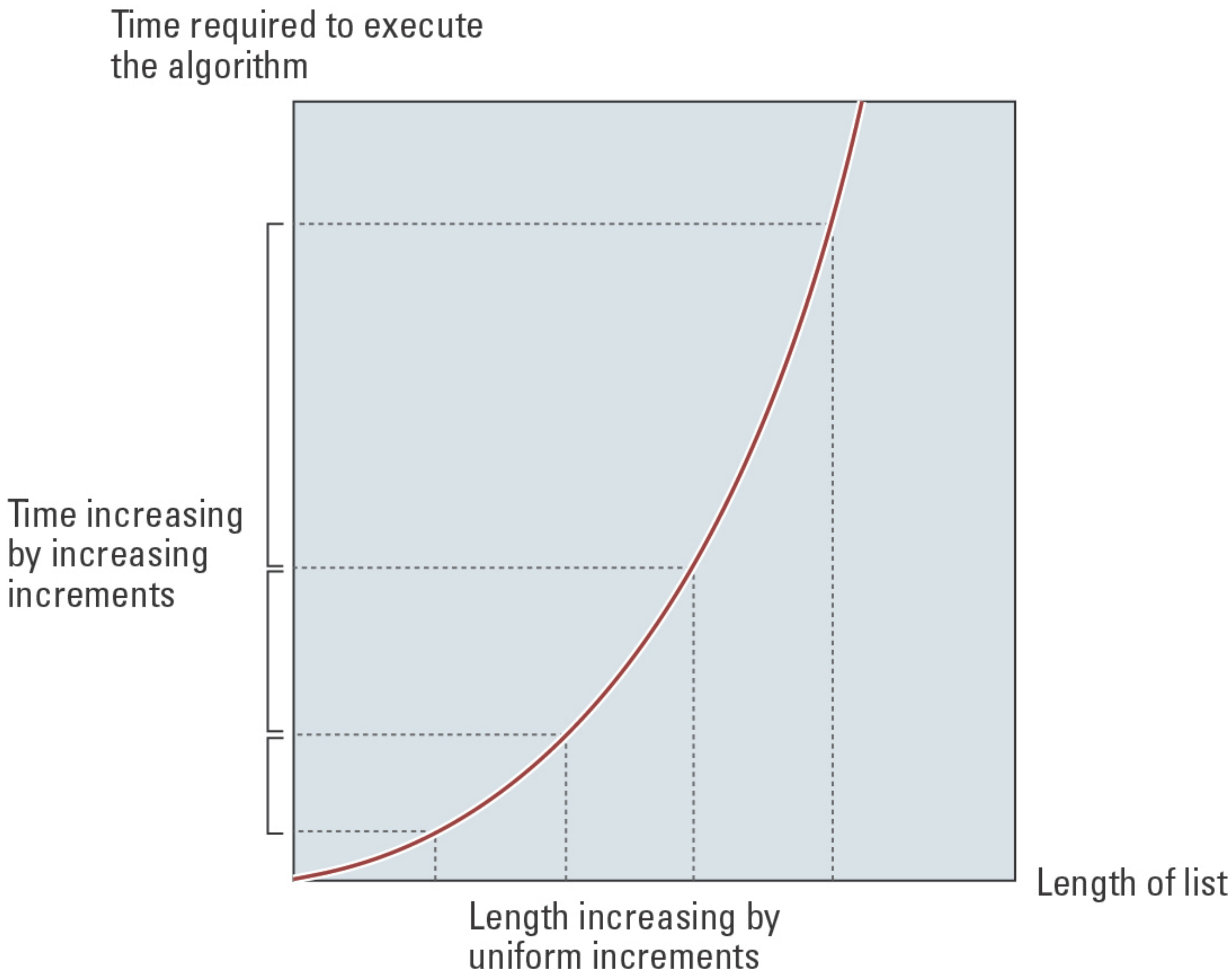
- $O(1)$: constant
- $O(\log n)$
- $O(n)$: linear
- $O(n \log n)$
- $O(n^2)$: quadratic
- $O(n^3)$: cubic
- $O(2^n)$: exponential
- $O(n^n)$: super exponential

Applying the Insertion Sort in a Worst-case Situation

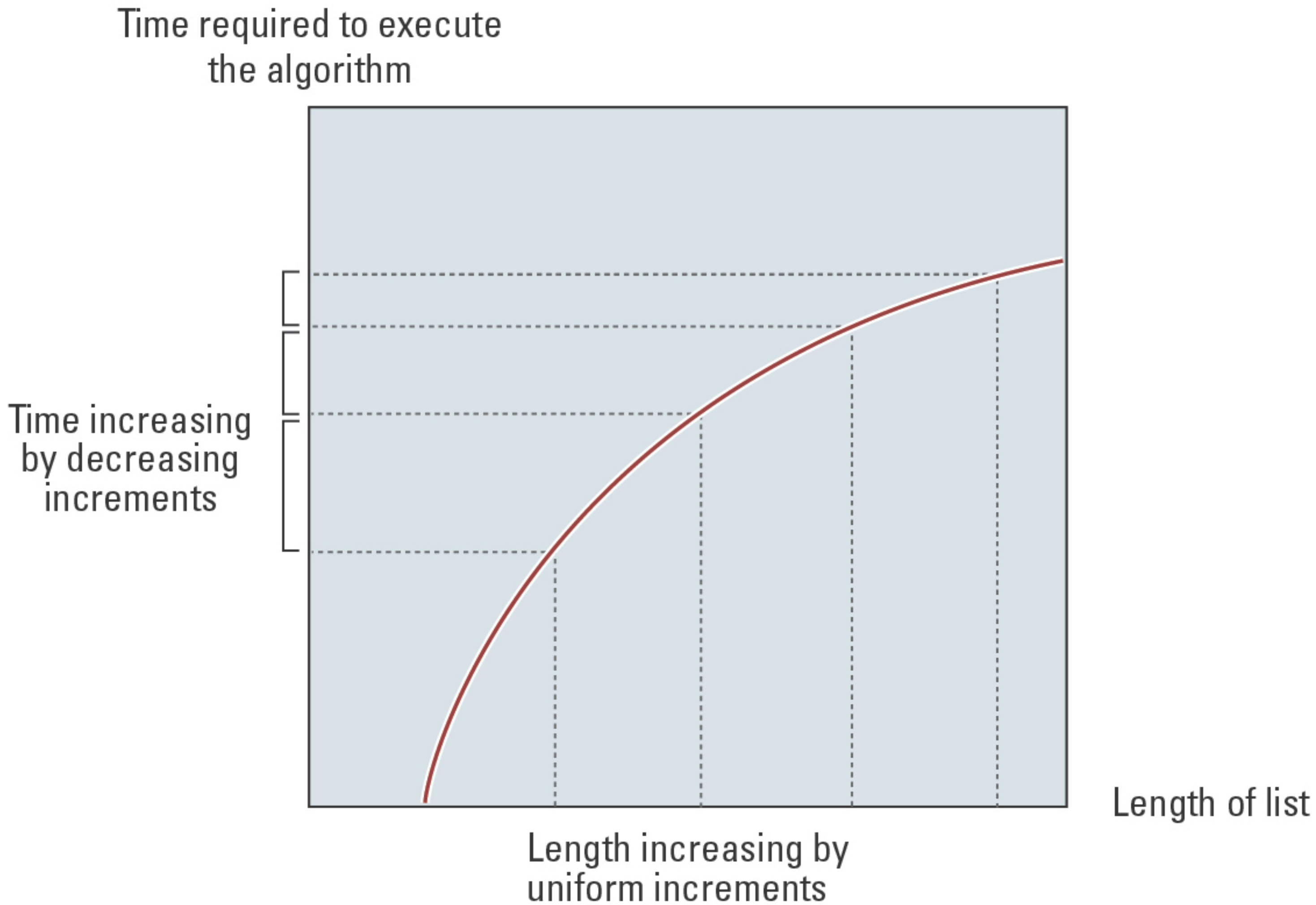
Comparisons made for each pivot

Initial list	1st pivot	2nd pivot	3rd pivot	4th pivot	Sorted list
Elaine	1 Elaine	3 David	6 Carol	10 Barbara	Alfred
David	David	2 Elaine	5 David	9 Carol	Barbara
Carol	Carol	Carol	4 Elaine	8 David	Carol
Barbara	Barbara	Barbara	Barbara	7 Elaine	David
Alfred	Alfred	Alfred	Alfred	Alfred	Elaine

Graph of the Worst-case Analysis of the Insertion Sort Algorithm



Graph of the Worst-case Analysis of the Binary Search Algorithm



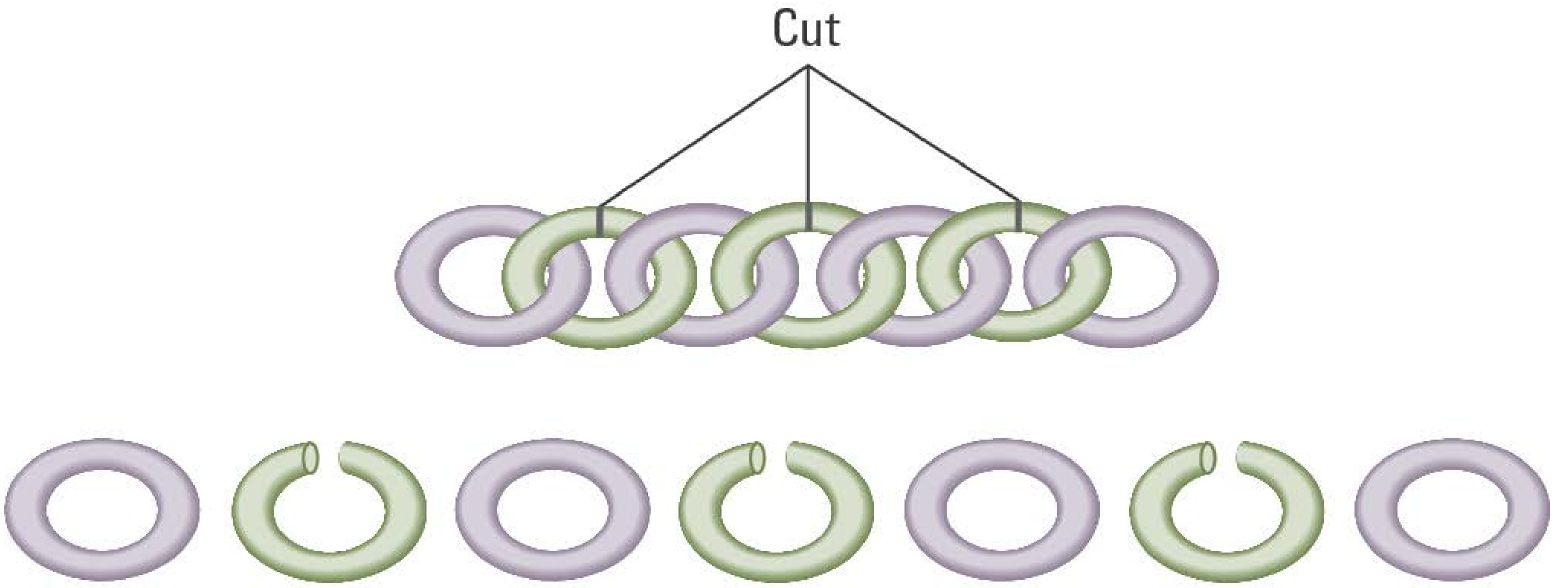
Software Verification

- Proof of correctness (with formal logic)
  - Assertions
    - Preconditions
    - Loop invariants
- Testing is more commonly used to verify software
- Testing only proves that the program is correct for the test cases used

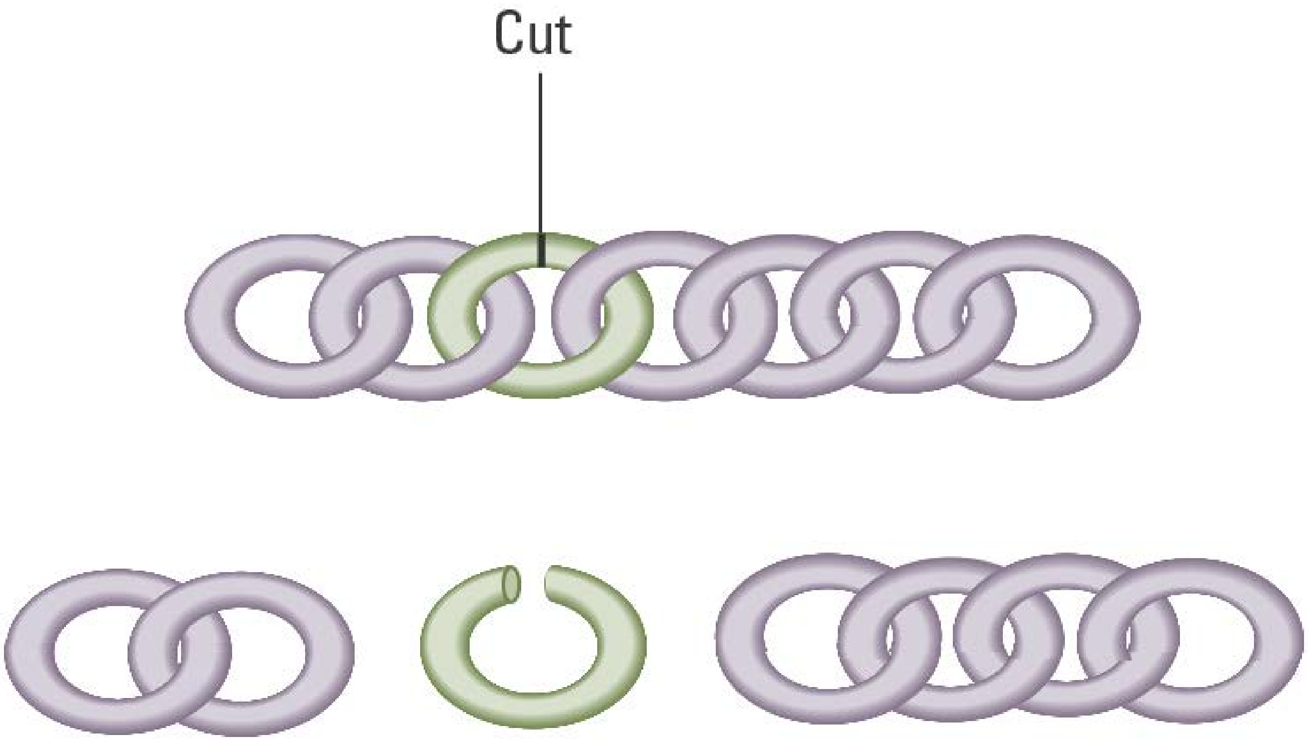
Chain Separating Problem

- A traveler has a gold chain of seven links.
- He must stay at an isolated hotel for seven nights.
- The rent each night consists of one link from the chain.
- What is the fewest number of links that must be cut so that the traveler can pay the hotel one link of the chain each morning without paying for lodging in advance?

Separating the Chain Using Only Three Cuts



Solving the Problem with Only One Cut



A Wolf, a Goat, and a Cabbage

- A man finds himself on a riverbank with a wolf, a goat, and a head of cabbage. He needs to transport all three to the other side of the river in his boat. However, the boat has room for only the man himself and one other item (either the wolf, the goat, or the cabbage). In his absence, the wolf would eat the goat, and the goat would eat the cabbage. Show how the man can get all these "passengers" to the other side.

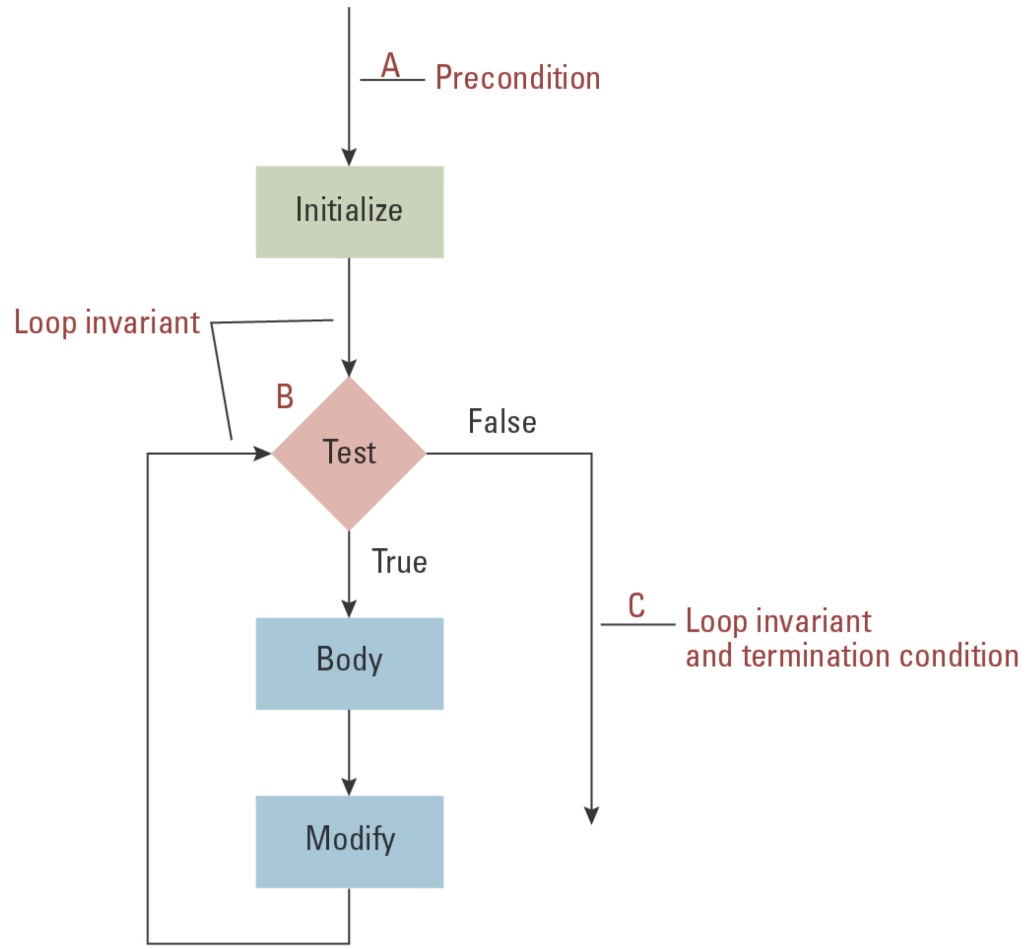
Ferrying Soldiers

- A detachment of 25 soldiers must cross a wide and deep river with no bridge in sight. They notice two 12-year-old boys playing in a rowboat by the shore. The boat is so tiny, however, that it can only hold two boys or one soldier. How can the soldiers get across the river and leave the boys in joint possession of the boat? How many times does the boat pass from shore to shore in your algorithm?

A Fake Among Eight Coins

- There are eight identical-looking coins; one of these coins is counterfeit and is known to be lighter than the genuine coins. What is the minimum number of weightings needed to identify the fake coin with a two-pan balance scale without weights?

The Assertions Associated with a Typical `while` Structure



Loop Invariants

```
int j = 9;
for(int i=0; i<10; i++)
  j--;
```

- In this example it is true (for every iteration):
  - `i + j == 9`.
  - A weaker invariant that is also true is that `i >= 0 && i <= 10`.

Loop Invariants

```
int max(int n,const int a[n]) {
   int m = a[0];
   // m equals the maximum value in a[0...0]
   int i = 1;
   while (i != n) {
      // m equals the maximum value in a[0...i-1]
      if (m < a[i])
         m = a[i];
         // m equals the maximum value in a[0...i]
      ++i;
      // m equals the maximum value in a[0...i-1]
   }
   // m equals the maximum value in a[0...i-1], and i==n
   return m;
}
```