2. Write an algorithm to find the largest item in an unsorted singly linked

list with cells containing integers.  
  
Cell: FindLargestCell(Cell: top)

// If the list is empty, return null.

If (top.Next == null) Return null

// Move to the first cell that holds data.

top = top.Next

// Save this cell and its value.

Cell: best\_cell = top

Integer: best\_value = best\_cell.Value  
// Move to the next cell.

top = top.Next

// Check the other cells.

While (top != null)

// See if this cell's value is bigger.

If (top.Value > best\_value) Then

best\_cell = top

best\_value = top.Value

End If

// Move to the next cell.

top = top.Next

End While

Return best\_cell

End FindLargestCell

3. Write an algorithm to add an item at the top of a doubly linked list.  
  
AddAtBeginning(Cell: top, Cell: new\_cell)

// Update the Next links.

new\_cell.Next = top.Next

top.Next = new\_cell

// Update the Prev links.

new\_cell.Next.Prev = new\_cell

new\_cell.Prev = top

End AddAtBeginning

4. Write an algorithm to add an item at the bottom of a doubly linked list.  
  
AddAtEnd(Cell: bottom, Cell: new\_cell)

// Update the Prev links.

new\_cell.Prev = bottom.Prev

bottom.Prev = new\_cell

// Update the Next links.

new\_cell.Prev.Next = new\_cell

new\_cell.Next = bottom

End AddAtEnd

5. If you compare the algorithms you wrote for Exercises 3 and 4 to the

Insert Cell algorithm shown in the section “Doubly Linked Lists,” you

should notice that they look very similar. Rewrite the algorithms you wrote

for Exercises 3 and 4 so that they call the Insert Cell algorithm instead of

updating the list’s links directly.  
  
AddAtBeginning(Cell: top, Cell: new\_cell)

// Insert after the top sentinel.  
InsertCell(top, new\_cell)

End AddAtBeginning

AddAtEnd(Cell: bottom, Cell: new\_cell)

// Insert after the cell before the bottom sentinel.

InsertCell(

6. Write an algorithm that deletes a specified cell from a doubly linked list.

Draw a picture that shows the process graphically.  
  
DeleteCell(Cell: target\_cell)

// Update the next cell's Prev link.

target\_cell.Next.Prev = target\_cell.Prev

// Update the previous cell's Next link.

target\_cell.Prev.Next = target\_cell.Next

End DeleteCell  
  
  
  
  
7. Suppose you have a sorted doubly linked list holding names. Can you

think of a way to improve search performance by starting the search from

the bottom sentinel instead of the top sentinel? Does that change the Big

O run time?  
  
If the name you’re looking for comes nearer the end of the alphabet than the

beginning, such as a name that starts with N or later, you could search the list

backwards, starting at the bottom sentinel. This would not change the O(N) run

time, but it might cut the search time roughly in half in practice if the names are

reasonably evenly distributed.  
  
  
  
8.Write an algorithm for inserting an item in a sorted doubly linked list

where the top and bottom sentinels hold the minimum and maximum

possible values.  
  
  
// Insert a cell in a sorted doubly linked list.

InsertCell(Cell: top, Cell: new\_cell)

// Find the cell before where the new cell belongs.

While (top.Next.Value < new\_cell.Value)  
top = top.Next

End While

// Update Next links.

new\_cell.Next = top.Next

top.Next = new\_cell

// Update Prev links.

new\_cell.Next.Prev = new\_cell

new\_cell.Prev = top

End InsertCell  
  
  
9. Write an algorithm that determines whether a linked list is sorted.  
  
Boolean: IsSorted(Cell: sentinel)

// If the list has 0 or 1 items, it's sorted.

If (sentinel.Next == null) Then Return true

If (sentinel.Next.Next == null) Then Return true

// Compare the other items.

sentinel = sentinel.Next;

While (sentinel.Next != null)

// Compare this item with the next one.

If (sentinel.Value > sentinel.Next.Value) Then Return false

// Move to the next item.

sentinel = sentinel.Next

End While

// If we get here, the list is sorted.

Return true

End IsSorted

10. Insertion sort and selection sort both have a run time of O(N2). Explain

why selection sort takes longer in practice.  
Insertionsort takes the fi rst item from the input list and then fi nds the place in the

growing sorted list where that item belongs. Depending on its value, sometimes

the item will belong near the beginning of the list, and sometimes it will belong

near the end. The algorithm won’t always need to search the whole list, unless

the new item is larger than all the items already on the sorted list.

In contrast, when selectionsort searches the unsorted input list to fi nd the largest

item, it must search the whole list. Unlike insertionsort, it can never stop the

search early.

11. Write a program that builds a multithreaded linked list of the planets, as

described in the section “Multithreaded Linked Lists.” Let the user click

a radio button or select from a combo box to display the planets ordered

by the different threads. (Hints: Make a Planet class with fields Name,

Distance To Sun, Mass, Diameter, Next Distance, Next Mass, and Next Diameter.

Then make an Add Planet To List method that adds a planet to the threads

in sorted order.)

12. Write a program that implements the tortoise-and-hare algorithm.

Algorithm A, the original:

tortoise = head

rabbit = head

length = 0

if rabbit == null: return length

rabbit, length = step(rabbit), length+1

if rabbit == null: return length

while rabbit != tortoise:

rabbit, length = step(rabbit), length+1

if rabbit==null: return length

rabbit, length = step(rabbit), length+1

if rabbit==null: return length

tortoise = step(tortoise)

return infinity

-----------------------------------------------------------------------------------------

Algorithm B, the leaping version:

tortoise = head

rabbit = head

length,gap = 0, 0

if rabbit == null: return "Finite list of length "+string(length)

rabbit, length, gap = step(rabbit), length+1, gap+1

if rabbit == null: return "Finite list of length "+string(length)

limit = 1

while rabbit != tortoise:

if gap==limit: tortoise, gap, limit = rabbit, 0, limit\*2

rabbit, length, gap = step(rabbit), length+1, gap+1

if rabbit==null: return "Finite list of length "+string(length)

return "List with loop of size "+string(gap)

---------------------------------------------------------------------------------------

def brent(f, x0):

# main phase: search successive powers of two

power = lam = 1

tortoise = x0

hare = f(x0) # f(x0) is the element/node next to x0.

while tortoise != hare:

if power == lam: # time to start a new power of two?

tortoise = hare

power \*= 2

lam = 0

hare = f(hare)

lam += 1

# Find the position of the first repetition of length λ

mu = 0

tortoise = hare = x0

for i in range(lam):

# range(lam) produces a list with the values 0, 1, ... , lam-1

hare = f(hare)

# The distance between the hare and tortoise is now λ.

# Next, the hare and tortoise move at same speed until they agree

while tortoise != hare:

tortoise = f(tortoise)

hare = f(hare)

mu += 1

return lam, mu

--------------------------------------------------------------------------------------------

def floyd(f, x0):

# Main phase of algorithm: finding a repetition x\_i = x\_2i.

# The hare moves twice as quickly as the tortoise and

# the distance between them increases by 1 at each step.

# Eventually they will both be inside the cycle and then,

# at some point, the distance between them will be

# divisible by the period λ.

tortoise = f(x0) # f(x0) is the element/node next to x0.

hare = f(f(x0))

while tortoise != hare:

tortoise = f(tortoise)

hare = f(f(hare))

# At this point the tortoise position, ν, which is also equal

# to the distance between hare and tortoise, is divisible by

# the period λ. So hare moving in circle one step at a time,

# and tortoise (reset to x0) moving towards the circle, will

# intersect at the beginning of the circle. Because the

# distance between them is constant at 2ν, a multiple of λ,

# they will agree as soon as the tortoise reaches index μ.

# Find the position μ of first repetition.

mu = 0

tortoise = x0

while tortoise != hare:

tortoise = f(tortoise)

hare = f(hare) # Hare and tortoise move at same speed

mu += 1

# Find the length of the shortest cycle starting from x\_μ

# The hare moves one step at a time while tortoise is still.

# lam is incremented until λ is found.

lam = 1

hare = f(tortoise)

while tortoise != hare:

hare = f(hare)

lam += 1

return lam, mu