**Timing: Efficiency of Algorithms**

Four general considerations for comparing algorithms:

1. writing time

2. storage

3. maintenance (how easy to modify and understand)

4. execution time (fastest is best)

Two methods used to compare algorithms:

1. statistics with random inputs (empirical)

2. growth function (running time function) – value of the function is directly proportional to the time requirement. It measures how an algorithm’s time requirement grows as the problem size grows.

Goal: compare orders of magnitude of growth functions

Note: can always time algorithm using a clock

A. Inefficient algorithms are not big problems with small data sets.

B. We compare (assume) the worst case

C. We assume all statements take the SAME time

D. We assume all statements take ONE UNIT of time

E. When n is large only the dominant term is important

O(1) < O(log n) < O(n) < O(n log n) < O(n^2) … O(2^n)

How to know how many times a loop will execute?

upper bound – lower bound + 1

Book Review

pg203

Example 1:

for i = 1 to n

sum = sum + I;

this will execute n times 🡪 O(n)

Example 2:

for i = 1 to n

for j = 1 to i

sum = sum +1

this will execute (n/2) (n^2 +1) 🡪 O(n^2)

Example 3:

sum = n\*(n+1) /2

this will execute 1 time 🡪 O(1)

Pg 204 Question 1:

Pg 212 Question 5:

for i = 1 to n

for j = 1 to 5

sum = sum +1

this will execute 5\*n = 5n 🡪 O(n)

Pg 214 Question 8:

for(i = 0 to n -2)

for (r = i+1 to n-1)

statement

0 (n-1) – (0 + 1) +1 = n-1

1 (n -1) – (1 +1) +1 = n -2

2 -3

3 -4

4 -5

. .

. .

n-2 0

Pg216

Question 9: What is the Big-O of remove method for array / linked list

Best case O(1) / same

Worst case O(n) / same

Question 10: What is Big-O of replace method for array / linked list

Best case O(1) / same

Worst case O(1) / O(n)

Question 11: getEntry for array / linked list

Best case O(1) / same

Worst case O(1) / O(n)

Question 12: contains method for array / linked list

Best Case O(1) / same

Worst Case O(n) / same

Question 13: display method for array / linked list

Worst Case and Best Case O(n) / same

**Recursion**

What are the two ways of repetition?

1. loops

2. recursion

Why use recursion over loops?

-may be more efficient

-may be more clear than some iterative versions with loop

Note: iterative version would have one additional loop than its recursive version

Definition:

A definition is recursive if it is described in terms of itself

What are the properties of recursive methods?

1. method is defined in terms of itself

2. there is a base case (escape clause) that allows for termination

What are the properties of method calls?

1. the recursive call diminishes the problem size

2. the base case is eventually reached in every case

n!

public int f(int i)

{

if(i == 1)

return 1;

else

return f(int i -1) \* i;

}

Pg 231

Question 1: Write a recursive void method that skips n lines of output.

public void skip(int n)

{

if(n >= n)

{

System.out.println();

skip(n-1);

}

}

Question 2: Describe a recursive algorithm that draws a given number of concentric circles.

public void draw(int n)

{

drawCircle(n); draw(n\*(3/4)); }

drawCircle(givenNumber, givenDiameter, givenPoint)

{

if(givenNumber >=1)

Draw a circle whose diameter is givenDiameter and whose center is a givenPoint.

drawCircle(givenNumber -1, givenDiamter\*(4/3), givenPoint)

}

Note: Fibonacci Sequence is O(2^n) if done recursively

Java records the current state of the method’s execution, including the values of its parameters and local variables as well as the location of the current instruction. Each record, called an activation record, provides a snapshot of a method’s state during its execution.

Records are placed into an ADT called a stack. The stack organizes the records chronologically so that the record of the currently executing method is on top.

Recursive method uses more memory than an iterative method because each recursive call generates an activation record.

Too many recursive calls can case the error message “stack overflow”. This means that the stack of activation records has become full. This means that the method has used too much memory.

Pg234

Question 3: Write a recursive void method countUp(n) that counts up from 1 to n

public void countUp(int n)

{

if(n >=1)

{

countUp(n-1);

System.out.println(n)

}

}

Pg235

Question 4: Write a recursive method that computes the product of the integers from 1 to n.

public int product(int n)

{

if(n > 1)

return product(n-1) \* n;

else

return 1;

}

While there are several ways to process arrays with recursive algorithms there is only two ways to process linked lists with recursive algorithms.

**Sorting**

Selection sort – check each number, find the smallest, place in the front

🡪 O(n^2)

Insertion sort – first value is already sorted, start at second and compare to sorted values before it 🡪O(n^2)

Shell sort – uses same logic as bubble sort (according to lecture notes) vs insertion sort (book notes)

-key is to think about Gap = n/2 🡪 until gap = 1 (then do normal insertion sort)

🡪 O(n^2)

Merge sort – merges already sorted sublists

- given 2 arrays that are already sorted we then create a 3rd array.

-according to the book: divides an array into halves, sorts the two halves, and then merges them together

-aka divide and conquer

-Note: natural merge sort is when we split array into sub arrays based on ordered items

🡪 O(nlogn)

Quicksort – idea is to select a pivot and partition the array based on that pivot

1. one element will be the pivot

2. rearrange the array based on the pivot

3. pivot is in the position that it will occupy in the final sorted array

4. elements in positions before the pivot are less than or equal to the pivot

4. elements in positions after the pivot are greater than or equal to the pivot

This arrangement is called a partition of the array – creating a partition divides the array into two pieces

Note: Be sure to do the scan from the RIGHT before the scan from the left

🡪O(n^2)

Radix sort

-aka buckets; initially radix sort distributes the elements into buckets according to the character (digit) at one end of the string; the sort then collects the strings and distributes them again among the buckets according to the character or digit in the next position

🡪 O(n)

joke, book, back, dig, desk, word, fish, ward, dish, wit, deed, fast, dog, bend

1st: dig, dog, wit, word, ward, deed, bend, joke, fish, dish, book, back, desk, fast

2nd: back, deed, dig, dog, joke, bend, book, word, ward, fish, dish, desk, fast, wit

3rd: back, ward, fast, deed, bend, desk, dig, fish, dish, wit, dog, joke, book, word

4th: back, bend, book, deed, desk, dig, dish, dog, fast, fish, joke, ward, wit, word

Default: ascending order (low to high)

**Sorted Lists**

Definition: a list that preserves an order

Instead of calling a sort on unsorted data, we continually preserve the order when adding

Note:

add cannot have a position parameter

remove cannot have a position parameter

getPostion returns the position if there, and returns the “would-be” position (ie. -3”

What about nodes? How does insert work?

public Node nodeBefore(node first, new)

{

if(first == null)

{

first = new;

length++;

return new;

}

if(first.data.compareTo(new.data) <=0)

{

new.next = first;

first = new;

length++;

return new;

}

current = first;

before = first;

while(current != null)

{

if(current.data.compareTo(new.data) <= 0)

{

new.next = before.next;

before.next = new;

length++;

return new;

}

else

{

before = current;

current = current.next;

}

}

current.next = new;

length++;

return new;

}