Computer Science COMP-250B

Midterm, Feb 23, 2016, 13:05-14:25.

O P E N • B O O K S •/• O P E N • N O T E S

1) Consider the following recursive function definition:

$$f(n)=\begin{cases} n^2 & \text{if } n<4 \\ f(n-1)+2f(n-2)+4f(n-3) & \text{if } n\geq 4 \end{cases}$$

a) Write an iterative algorithm to compute this function for any n ≥1.

```
f1=1; f2=4; f3=9;
FOR i = 1 to n-1
  tmp = f3+2*f2+4*f1;
  f1 = f2;
  f2 = f3;
  f3 = tmp;
return f1.
```

b) Why not simply compute this function recursively if it was defined recursively?

If computed recursively this function will repeatedly compute the same values on and on, whereas the iterative version computes each f(i) exactly once and no more.

2)

Write *any* algorithm that runs in time $\Theta(n^2 \log^2 n)$ in worse case. **Explain** why this is its running time. I don't care what it does. I only care about its running time…

```
WhatEver(array int A)

n = A.length;
FOR i=1 TO n
  FOR j=1 TO n
     x=n; WHILE x>1 DO { x=x/2; y=n;
                         WHILE y>1 DO y=y/2 }
```

3) Prove by mathematical induction that for n ≥ 1,

$$\sum_{i=1}^{n} i^2 = n(n+1)(2n+1)/6.$$

Base case: n=1, $\sum_{i=1}^{1} i^2 = 1^2 = 1 = 1*2*3/6 = 1(1+1)(2*1+1)/6.$

Induction step: Assume for induction hypothesis that for n≥1,

we already have $\sum_{i=1}^{n} i^2 = n(n+1)(2n+1)/6$. Let's prove for n+1.

$$\begin{aligned}\sum_{i=1}^{n+1} i^2 = (n+1)^2 + \sum_{i=1}^{n} i^2 &= (n+1)^2 + n(n+1)(2n+1)/6 \\ &= (n+1)(\ n+1+n(2n+1)/6\ ) \\ &= (n+1)(\ 2n^2+7n+6\ )/6 \\ &= (n+1)((n+1)+1)(2(n+1)+1)/6.\end{aligned}$$

4)
Consider the following arithmetic expression

```
5 + 6 * 4 + (6 + 5 * (2 + 1)).
```

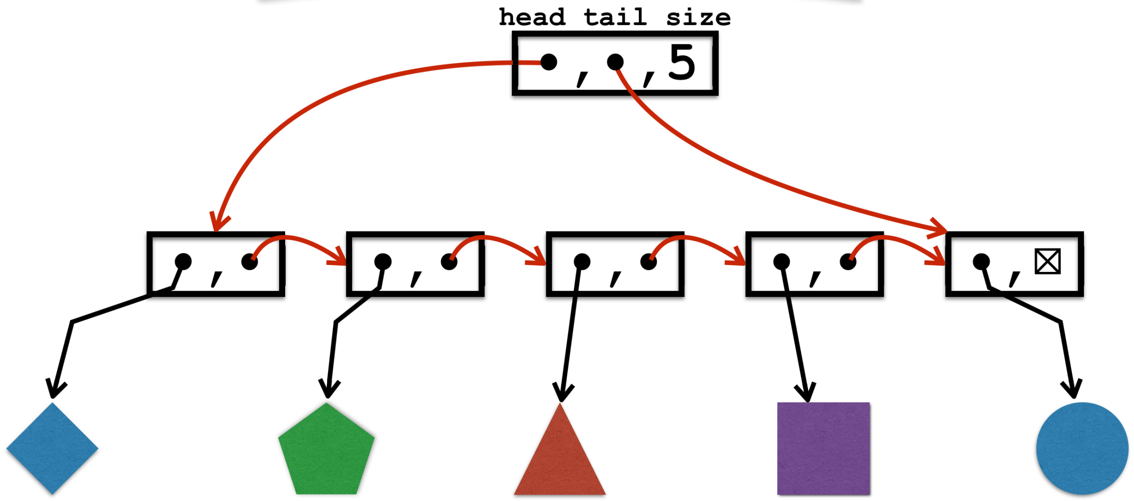
Give the successive states of the *argument* and *operator* stacks that will result of computing (from left to right) the value corresponding to this expression.

```
                                                          1
                                                      2   2   3   3
                  4                           5   5   5   5   5   5   5  15
          6   6   6  24           6   6   6   6   6   6   6   6   6   6  21  21
  5   5   5   5   5   5  29  29  29  29  29  29  29  29  29  29  29  29  29  29  50

                                                      +   +
                                                  (   (   (   (
                                              *   *   *   *   *   *
                                      +   +   +   +   +   +   +   +   +
              *   *           (   (   (   (   (   (   (   (   (   (   (   (
  _   +   +   +   +   +   +   +   +   +   +   +   +   +   +   +   +   +   +   +   _
```

5) Consider the situation where you are given a Singly Linked List of objects as in *Lecture 5*. An example is given below :

```
class  SLinkedList{
   SNode          head;
   SNode          tail;
   integer        size;
}
```



A) Write (Java-like) pseudo-code to break this list (call it MyList) into two sub-lists (call them MyLeftList and MyRightList) of nearly equal sizes (difference of sizes is ≤ 1). In the example above your algorithm would produce two sub-lists of sizes 3 and 2.

```
BREAK(MyList)
Middle = MyList.head;
FOR i = 1 to MyList.size/2
  Middle = Middle.next;

MyLeftList.head = MyList.head; MyLeftList.tail = Middle;
MyRightList.head = Middle.next; MyRightList.tail = MyList.tail;
Middle.next = null;
MyLeftList.size = MyList.size/2;
MyRightList.size = MyList.size - MyLeftList.size;
RETURN [MyLeftList,MyRightList]
```

B) Write (Java-like) pseudo-code to merge two such sub-lists once they each have been sorted[*].

```
MERGE(MyLeftList,MyRightList)
NEW MyList; NEW MyList.Tail; MyList.head = MyList.tail;
WHILE MyLeftList.head≠null AND MyRightList.head≠null DO
|   IF MyLeftList.head.element.Bigger(MyRightList.head.element)
|   THEN MyList.tail.next = MyRightList.head;
|            MyList.tail = MyList.tail.next;
|            MyRightList.head = MyRightList.head.next;
|   ELSE MyList.tail.next = MyLeftList.head;
|            MyList.tail = MyList.tail.next;
|            MyLeftList.head = MyLeftList.head.next;
WHILE MyLeftList.head≠null DO
|   MyList.tail.next = MyLeftList.head;
|   MyList.tail = MyList.tail.next;
|   MyLeftList.head = MyLeftList.head.next;
WHILE MyRightList.head≠null DO
|   MyList.tail.next = MyRightList.head;
|   MyList.tail = MyList.tail.next;
|   MyRightList.head = MyRightList.head.next;
MyList.head = MyList.head.next;
MyList.size = MyLeftList.size + MyRightList.size;
RETURN MyList;
```

C) Write (Java-like) pseudo-code combining A) and B) to perform merge sort of the elements of such a list.
**** Make sure the asymptotic running time remains O(n log n). ****

```
MERGESORT(MyList)
IF MyList.size ≤ 1 THEN RETURN MyList;
[L,R] = BREAK(MyList);
RETURN MERGE(MERGESORT(L),MERGESORT(R));
```

[*] Assume any object contains a (boolean valued) method named `Bigger` to compare it with another object of the same type. For instance, in the above example I can compare the blue-triangle with the green-pentagon using

MyList.head.element.Bigger(MyList.head.next.element).