UNIVERSITY OF WATERLOO

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

ECE 250 – ALGORITHMS AND DATA STRUCTURES

Midterm Examination (5 pages) Instructor: R.E.Seviora 1 hour, Oct 25, 2002

SOLUTIONS		**V1.0**	
1.	2.	3.	Total:

Do all problems. The number in brackets denotes the weight of the problem (out of 100). If information appears to be missing from a problem, make a reasonable assumption, state it and proceed. If the space to answer a question is insufficient, use the last (overflow) page. Closed book. Numerical calculators only.

PROBLEM 1 [35]

A. Simplified Model Analysis

1 Determine the running time predicted by the simplified computer model for the method sumSquares:

```
1 public class Example {
2    public static int sumSquares (int n) {
3      int sum = 0;
4      for (int i=0; i<=n; i++)
5        sum += i*i;
6      return sum;
7    }
8    //…
9 }
```

Statement	*Running Time (Simplified Model)*
3	*2*
4a	*2*
4b	*3(n+2)*
4c	*4(n+1)*
5	*6(n+1)*
6	*2*
TOTAL	***13n+22***

B. Estimates of Running Time

A simplified model analysis of a program resulted in the following expression for its running time:

$$T(n) = 5n^2 + 20n + 20$$

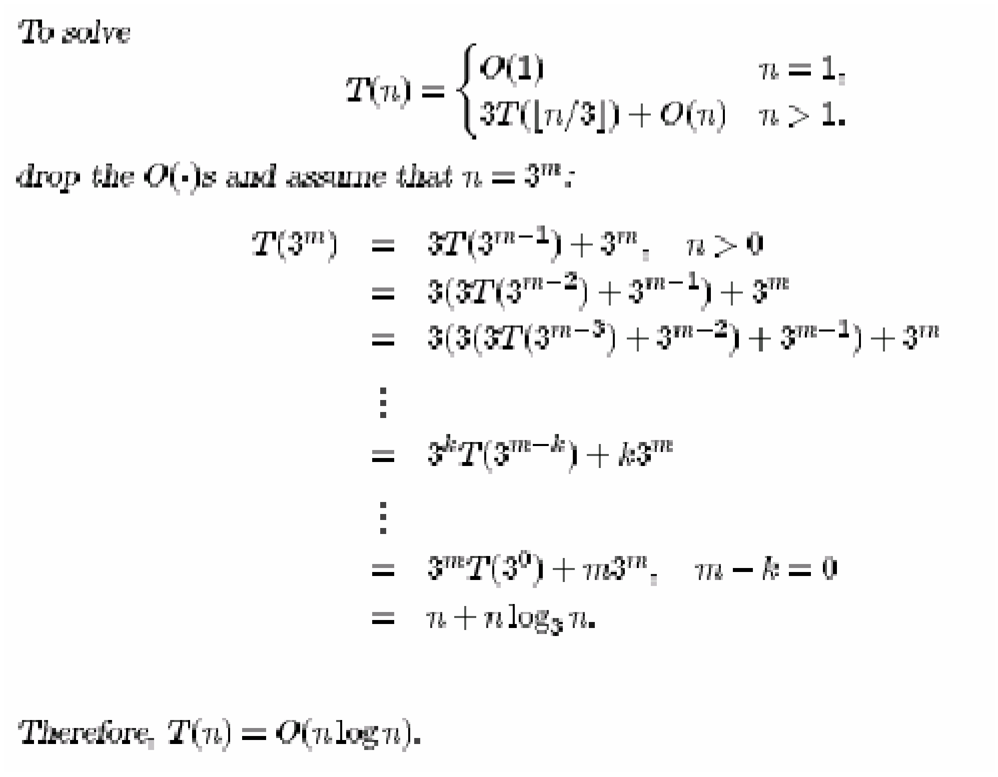
You were asked to run the program on a fairly large problem (n=100,000). The fastest computer you can use has a modern processor running with 1GHz clock. Obtain a *rough* estimate of the time needed to execute the program on this computer.

$$T(n) = [5\times(10^5)^2 + 20\times10^5 + 20]/10^9 \cong 5\times10^1\,\text{sec} = 50\,\text{sec}$$

C. Recurrences

Solve the following recurrence and state the resulting tight big Oh bound for T(n). You may assume that n is a power of three.

$$T(n) = \begin{cases} O(1) & n = 1 \\ 3T(\lfloor n/3 \rfloor) + O(n) & n > 1 \end{cases}$$



To solve

$$T(n) = \begin{cases} O(1) & n = 1, \\ 3T(\lfloor n/3 \rfloor) + O(n) & n > 1. \end{cases}$$

drop the $O(\cdot)$s and assume that $n = 3^m$:

$$\begin{aligned} T(3^m) &= 3T(3^{m-1}) + 3^m, \quad n > 0 \\ &= 3(3T(3^{m-2}) + 3^{m-1}) + 3^m \\ &= 3(3(3T(3^{m-3}) + 3^{m-2}) + 3^{m-1}) + 3^m \\ &\vdots \\ &= 3^k T(3^{m-k}) + k3^m \\ &\vdots \\ &= 3^m T(3^0) + m3^m, \quad m - k = 0 \\ &= n + n \log_3 n. \end{aligned}$$

Therefore, $T(n) = O(n \log n)$.

PROBLEM 2 [40]

A. Asymptotic Analysis

The implementation of the class `StackAsArray` shown below removes the constraint of fixed stack size by expanding the internal array whenever `push` would result in array overflow. (Note: the method `System.arraycopy` copies `count` items from `array` to `tmp`.)

Compute the *average* time for the `push` operation. To develop your answer, you may use the table shown below the program.

```
public class StackAsArray extends AbstractContainer implements Stack {
  protected Object[] array;
  protected int initSize;
  public StackAsArray(int size) {
    array = new Object[size];
    initSize = size;
  }
  public void push(Object obj) {
    if (count==array.length) {
      Object[] tmp = new Object [array.length + initSize];
      System.arraycopy (array, 0, tmp, 0, count);
      array = tmp;
    }
    array [count++] = obj;
  }
  //
}
```

Let p=push time; i=init stack size; Nb=time to create an array of length N;
Nc=time to copy N items from old to new array (by System.arraycopy)

Example table, initSize=i=3

push number	0	1	2	3	4	5	6	7	8	9	10
push time		*p*	*p*	*p*	*p*	*p*	*p*	*p*	*p*	*p*	*p*
resize time					*6b+3c*			*9b+6c*			*12b+9c*

Let n=mi. Consider the total time needed for n=mi pushes:

$$T(mi) = mip + \sum_{k=1}^{m-1}[(k+1)ib + kic] = mip + \sum_{i=1}^{m-1}[kd + e] = mip + d\frac{(m-1)m}{2} + (m-1)e$$

$$T_{avg}(push) = \frac{1}{mi}(above) = p + \frac{d(m-1)}{2i} + \frac{(m-1)}{mi}e = O(m) + O(1) = O(n)$$

Here, d=ib+ic and e=ib.

B. Asymptotic Bounds

(i) The limit rules are helpful in determining the asymptotic relationships between functions. In the table below, state whether the relationship in the column header always holds (Y) or not (N), for the limit given stated in row header. f(n) and g(n) are nonnegative functions defined on n ≥ 0.

$\lim_{n\to\infty} f(n)/g(n)$	f(n)= O(g(n))	g(n)= O(f(n))	f(n)= Ω(g(n))	g(n)= Ω(f(n))	f(n)= o(g(n))	g(n)= o(f(n))	*f(n)= Θ(g(n))*
0	*Y*	*N*	*N*	*Y*	*Y* †	*N*	*N*
a > 0	*Y*	*Y*	*Y*	*Y*	*N*	*N*	*Y*
∞	*N*	*Y*	*Y*	*N*	*N*	*Y*	*N*

(ii) Justify your answer for the table entry labeled with † (i.e. f(n)=o(g(n)), lim f/g = 0).

Definition of small oh: f(n)=o(n) iff f(n)=O(g(n)) but f(n)≠Θ(g(n)).

(To answer question (i), it is useful to add one column to the table. The column shows whether f and g are in the Θ relationship.)

C. Memory Requirements (Project 2)

A major objective of Project 2 was to represent a polynomial in x

$$a_n x^n + a_{n-1} x^{n-1} + .. + a_1 x + a_0$$

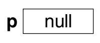
where n ≥ 0 is the degree of the polynomial, using Java arrays. The implementation was required to be reasonably lean in terms of computing time, and minimal: the array length at all times was to be *n+1*.

The typical PolynomialAsArray class implementation looked as follows:

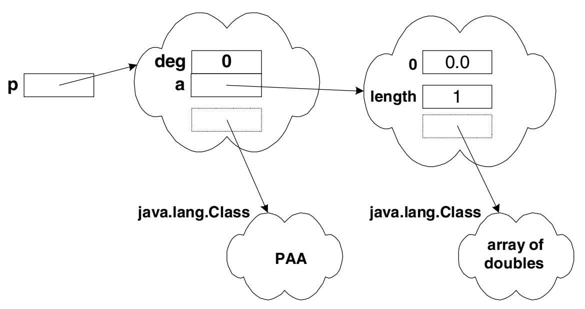
```
class PolynomialAsArray {
   int deg;
   double[] a;                //array of coefficients
   PolynomialAsArray () {     //constructs the polynomial 0x0
     a = new double[1];
     a[0]=0.0;
     deg = 0;  }
   //etc
}
```

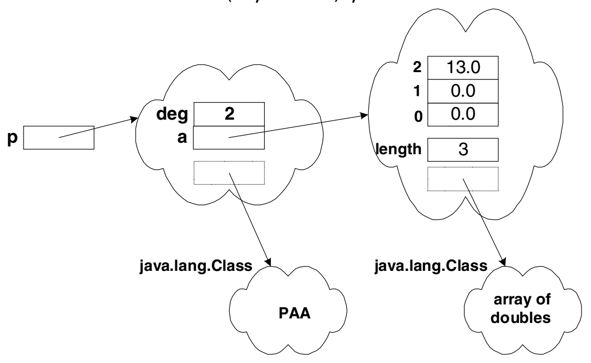
(i) For the above `PolynomialAsArray` class, draw a diagram that shows the objects stored in the memory of the computer after each of the following statements.

```
PolynomialAsArray p;
```



```
p = new PolynomialAsArray();          p.setCoefficient(2,13.0);
```





(ii) Derive an expression for the amount of memory required to represent a `PolynomialAsArray` of degree N. Use `S(int), S(double) and S(ref)` to denote the memory required to store an `int`, a `double`, and an object reference, respectively.

$$sizeof(PAA\_object) + sizeof(array\_object) =$$
$$= [S(\text{int}) + S(ref)] + [S(\text{int}) + S(double) \times (N+1)] =$$
$$= 2S(\text{int}) + S(ref) + (N+1) \times S(double)$$

Note: the above answer did not include the memory needed to store reference to the class object (dashed outline in 2(i)) This adds 2S(ref) to the sum.

PROBLEM 3 [25]

A. Visitors

Consider a container class whose instances will contain Java `Integer` objects. Design an `AveragingVisitor` class which implements the interface `Visitor` and also includes the method `computeAverage`. This method returns the average of the values of `Integer` objects presented to the `visit` method of the visitor since its creation. Recall that the method `intValue()` returns the value of the integer wrapped in an instance of `Integer`.

For illustration, assume that, after creating an instance `myAV` of the `AveragingVisitor`, the method `myAv.visit` is invoked *four* times, with the actual parameter referring to Integers containing 2, 1, 4, 3, respectively. At this point, the invocation of `computeAverage()` would return 2.5.

For simplicity, you are not required to consider the case when `computeAverage` is invoked prematurely (i.e. before the first invocation of `visit`).

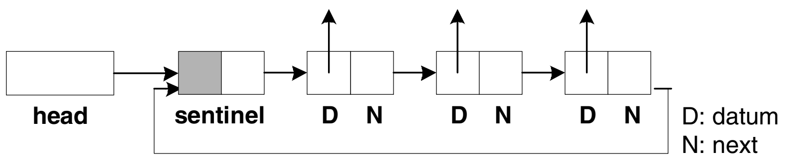
```
public class AveragingVisitor extends AbstractVisitor {

  //fields
  int sum = 0;     //stores the accumulated sum
  int count = 0;   //stores the count of objects presented to the visit method

  //methods
  public void visit (Object o) {
    sum += ((Integer) o).intValue();
    count++;
  }
  public boolean isDone()  {
    return false;
  }
  public float computeAverage() {
    return ((float) sum)/((float) count);
  }
}
```

B. Linked Lists

Implement the `prepend(Object o)` method for the singly-linked circular list with sentinel. (Observation: the implementation is very short, perhaps1-2 lines. Its brevity illustrates one advantage of sentinel-based lists.)



```
public class LinkedListWithSentinel  {
  protected Element head;
  public final class Element {
    Object datum;
    Element next;
    //etc.
  }
  public void prepend (Object o) {

    head.next=new Element(o,head.next);

  }
}
```

OVERFLOW SHEET [Please identify the question(s) being answered.]