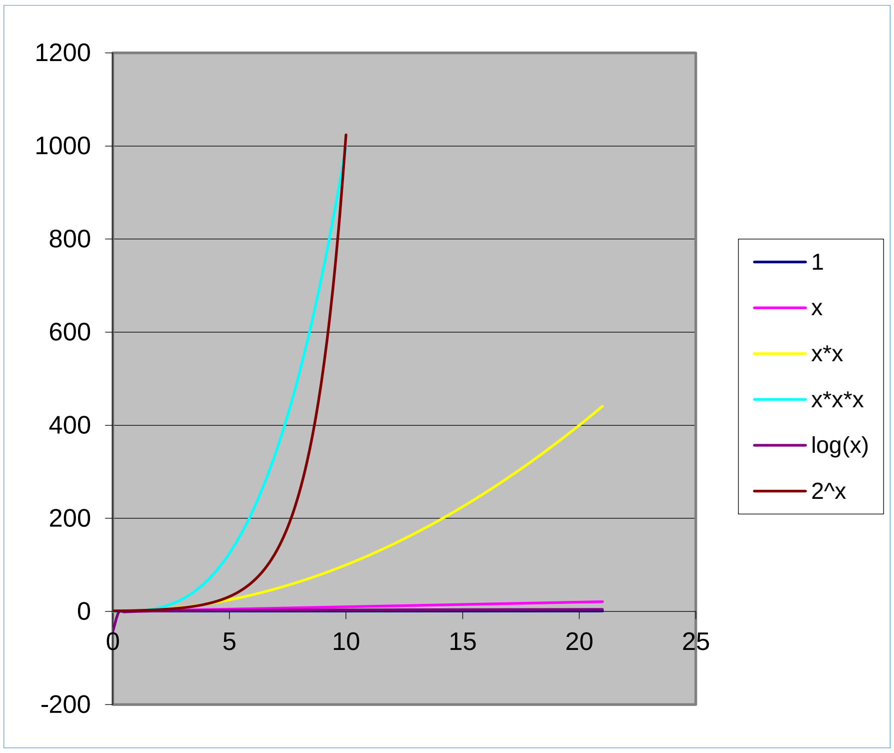
CO3401 Advanced Software Engineering Techniques

Tutorial Exercise Book 2: Algorithms and Efficiency ***with answers***

Efficiency

1. On the same axes sketch graphs of y=1, y=x*x, y=2^x, y=$\log_2$(x), y=x, y = x^3



a) List the functions order by ascending size for large values of x.

*1, log2(x), x, x*x, x3, 2^x*

b) What would be the effect on the ordering of
   1. multiplying each by a different constant
   2. adding to each one or more of the functions which precede it in the original list

There is no effect in either case, as long as you go to large enough x

2. A linear algorithm (T = C*N) to solve a class of problems has been implemented in Visual Basic on a PDA and takes 200 milliseconds to solve a problem of size 10.

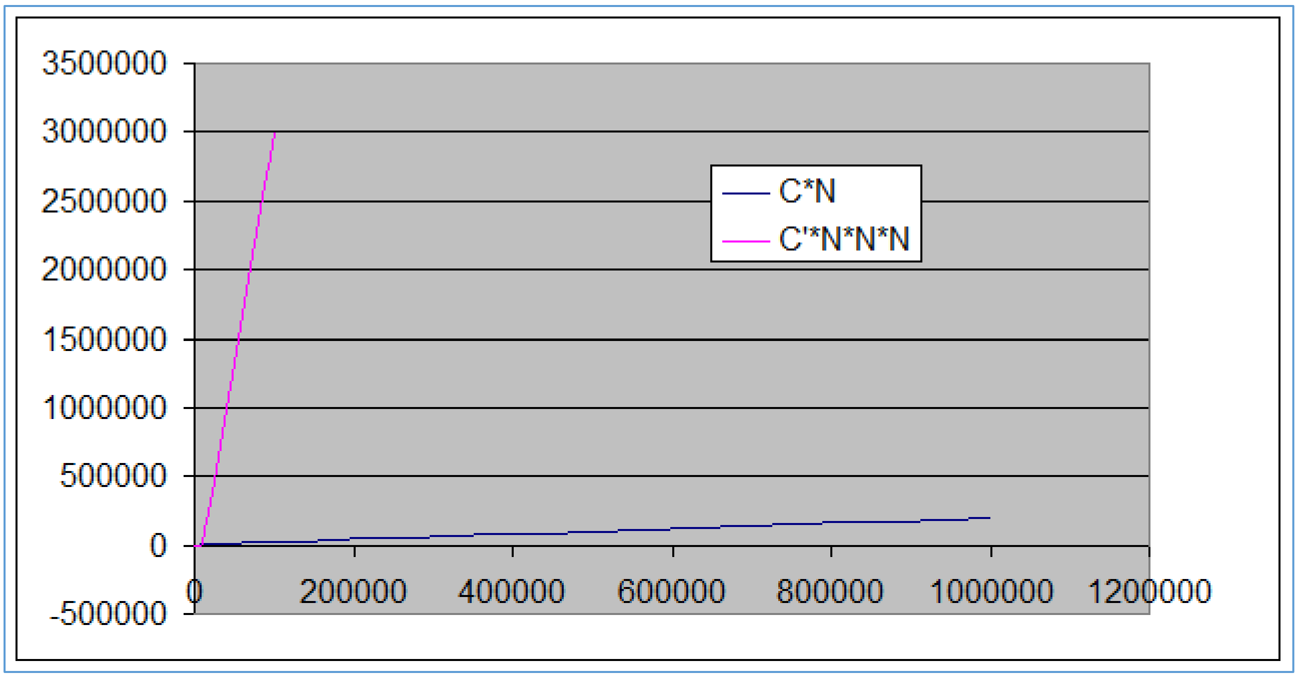
A cubic algorithm (T = C'*N^3) to solve the same class of problem has been implemented in Optimised Fortran on a super-computer. It takes 3 microseconds to solve a problem of size 10.

a) Calculate values of C and C'

*T = C * N*
*0.2 = C * 10*
C = 0.02

*T = C'*N3*
*0.000003 = C'*103*
*C' = 3*10-9*

b) Use Excel to sketch a graph of the time taken for both programs for N = powers of 10 from 1 to 6 (i.e. N=10,100, ...)



c) If the cost of supercomputer time is £2000 per hour and the cost of waiting for a result is £4000 per hour, what values of N should be solved on a PDA?

*Cost of PDA is 4000*60*60*0.02*N*
*Cost of super computer is (6000+2000)*60*60*0.000000003*N*N*N*
*Costs are equal if 4000*60*60*0.02*N = (4000+2000)*60*60*0.000000003*N*N*N*
*4*0.02 = 6 * 3*10-9 * N * N*
*N*N = (4/9)*107 = (2/3)*1000*10 –0.5*
approximately 2000

3.

a) At the Summer Ball, Hercule Parrot has learnt that one of the 128 guests has been given a slow-acting poison via the champagne used for the Toast. The treatment is too expensive and painful to be given to all the potential victims, so it is necessary to discover by analysis which glass contained the poison.

An expensive chemical test can detect the poison in champagne. Devise an efficient algorithm (plan of testing) to locate the poisoned glass at minimum cost. Assume that the test is very sensitive and there is plenty of champagne left in each glass. How many tests are needed?

Mix a drop from any 64 glasses – do the test, if positive the poisoned glass is in this set, if not it's in the other.
Split the positive group in half, mix a drop from each, test etc.
128 -> 64 -> 32 -> 16 -> 8 -> 4 -> 2 -> 1   7 tests.
The duration of this process will be 7 test times.

There is a cunning way of doing it faster, but it still requires 7 tests.
Label each glass with a binary code representing a number from 0 – 127
For test 1, mix a drop from each glass with a 1 in the first position of the binary number
For test 2, mix a drop from each glass with a 1 in the second position of the binary number
For test 7, mix a drop from each glass with a 1 in the seventh (least significant) position of the binary number

Treat the results of the test as 1 if true, 0 if false and write the results of the test in the corresponding position – that will tell you the number of the poisoned glass.

Since the tests can be done in parallel, the time is just 1 test time.

b) Following a murder in the arts department, Hercule believes that the murder weapon has been hidden in one of 27 identical sculptures. This will make the sculpture slightly heavier than the others. A balance scale is available which can take up to 10 sculptures in each pan.

Devise a plan that requires a minimum number of weighings to discover the murder weapon. Clearly you do not want to break the sculptures of innocent people. How many weighings are necessary?

Divide the sculptures into 3 groups. Weigh any two of the groups. The heavier group contains the weapon. If they balance, the unweighed group contains the weapon. Repeat until just one sculpture is left.
27 -> 9 -> 3 -> 1 This requires 3 tests

4. A free-text system contains 0.1 gigabytes of information organised in 2 kilobyte disc blocks. The system allows a user to enter one or more keywords and then searches for any blocks which contain all the specified keywords.

Having done problem 3a, a programmer has suggested that each 2k block should have a 128-bit tag associated with it. The value of this tag would be calculated by XORing together the ASCII values of each character in potential keywords to give a 7-bit value and setting the bit in the tag whose position is given by this value.
Discuss whether this will lead to an improvement in performance and whether the tag should be stored along with each block or whether the tag should be in a separate file (and if so how).

If the bit position is calculated for a keyword, it is not present in the block if that bit position is not set. If it is set, it may be present (but it could be another word which sets the bit). This will save time if the number of collisions is not high, and the number of searches is large – to spread the cost of the initial calculations. The tag should be in a separate file – can test lots of blocks without having to fetch the block data. Tags can be stored sequentially in the file since they will be accessed in the same order as the blocks. Alternatively, how about having 128 files which just store the numbers of blocks which sets that bit?

5. The following is a simple version of bubblesort. It can be improved by terminating as soon as the array is sorted and by reducing the size of the part of the array that is processed each time around the loop.

```
for (int i = 0; i < size; i++)
   for j := 0; j < size-1; j++)
          if (a[j] > a[j+1])
                 swap(a[j],a[j+1])
```

a) What is the algorithmic complexity of the above code (assume that comparison of array elements is the expensive operation)?

*Roughly size*size*
There will be size(size-1) comparisons*

b) Modify the above code to incorporate the changes outlined.

```
Boolean swaps = true;
for (int i = 0; i < size && swaps; i++)
{
   swaps = false;
   for j := 0; j < size-1-i; j++)
   {
          if (a[j] > a[j+1])
          {
                 swap(a[j],a[j+1]);
                 swaps = true;
          }
   }
}
```

c) Estimate the algorithmic complexity of the modified code.

*Still roughly size*size on average – worst case is the same as (a). However, the best case is size*

6. In a (simplified) mergesort, a file of n items is split into n separate files, these are then merged in pairs to produce n/2 files each containing 2 items in order. The pairwise merging process (called a pass) continues until a single sorted file is produced.

a) Estimate the average number of comparisons required per pass

First time requires n/2 comparisons – 1 comparison to create each file of size 2
*Second time requires 2 to 3 comparisons to create n/4 files of size 4 – n/2 to 3*n/4*
*Third time requires 4-7 comparisons to create n/8 files of size 8 – n/2 to 7*n/8*
each pass requires something between n/2 and n comparisons

b) How many passes are needed for a file of size n (assume n is a power of 2)

log2 n

c) What is the algorithmic complexity of the merge algorithm? (Assuming that a comparison is the most important operation).

n log2 n*

d) Discuss how realistic that assumption is.

Depends on how big the items are, however moving information from disk to computer is likely to be expensive – better to load a memory full of items and sort and output them to create large ordered files as a starting point.

7. A thousand readings are to be taken from a laboratory instrument in approximately one minute. Each reading will be in the range 0-999999. It is necessary to keep a count of the number of occurrences of each reading. Evaluate the following three methods:

i. Keep a linked list of each reading that has occurred along with its count.

*In worst case, each reading requires searching entirely through the existing list for an item which isn't there, giving something like 500*1000 comparisons. Requires memory for 1000 values (data and pointer)*

ii. Keep an array of the counts indexed by the reading. Hence if reading 333 occurs, increment element 333.

Requires 1 million locations
Each look up is in constant time

iii. Keep an array of about 6 thousand elements, each holding a reading that has occurred and its count. To add a reading to the array, find the remainder on dividing the reading by the size of the array and start searching the array from that position until either an empty element is found (then, insert the reading with a count of 1), or the reading is found (then, increment the count).

Requires 6000 elements
Each look up is close to constant time, but in worst case might have to do 1000 comparisons. This is extremely unlikely

Reasoning

8. Partitioning an Array

A partition of an array about a partition value reorganises its contents into two sections. The elements in the first section of the array are less than or equal to the partition value: the elements in the second section are greater than the partition value.

i. Design an algorithm to partition an array about the last element in the array. e.g

```
Input:  2 1 1 3 3 2 4 4 1 1
Output: 1 1 1 1 3 2 4 4 2 3

Input:  0 9 9 0
Output: 0 0 9 9

Input:  0 9 9 7
Output: 0 7 9 9
```

The following is pretty simple:

```
static partition(int [] theArray){
    int boundary = 0;
    int limit = theArray.length-1;
    for (int pos = 0; pos < limit; pos++){
        if (theArray[pos] <= theArray[limit]){
            swap(theArray, boundary, pos);
            boundary++;
        }
    }
    swap(theArray,boundary,limit);
}
```

ii. Are you sure it is correct? How do you know?

Testing is one possibility for increasing my confidence in it, but I can't test every possible input. In fact, it doesn't work for an empty array, since it returns the same result as for an array of size 1. On the other hand, is it fair to expect the routine to handle an array of size zero since the specification is to partition about the last element, and there isn't a last element in this case.

We could consider the concept of "pre-conditions" here – things that the specification allows the routine to assume are true (e.g. there is at least one element). Alternatively, we need to modify the specification to define what the right answer is for an empty array.

Reasoning is another possibility:
Initially, anything that is to the left of boundary is <= the partition element (because there isn't anything to the left of boundary).
Each time round the loop, we move pos closer to the last element in the array, stopping when all previous elements have been examined. We also ensure that if an element in the current position is less than or equal to the partition element, we move it to the current boundary position and increment the boundary – so everything to the left of boundary is <= the partition element.
Finally, we swap the item in the last position with the item in the boundary position, this ensures that everything upto and including the boundary position is now <= the partition element.

iii. Use a single extra swap to modify the algorithm to partition about the first element in the array.

Swap the first and the last items.
Of course, it would probably be better to re-write the algorithm to sweep from right to left.

iv. Modify the algorithm to return the position marking the boundary between the partitions.

9. Cumulative Frequency

Given an array of counts, each position (p) in the cumulative frequency array would contain the sum of values from the first array from the first element to the element corresponding to p.

Mathematically

input d – an integer array of size n

output cf – an integer array of size n, such that for $0 \leq i < n$,

$$cf[i] = \sum_{p=0}^{p=i} d[p]$$

If we have an array representing the number of people coming to a shop on each day of the week, the first element cumulative frequency array would contain the number who came on Monday, the second the total on Monday and Tuesday, the third the total up to and including Wednesday, …

e.g.

Frequencies:	21	3	5	8	9	21	1
Cumulative frequencies:	21	24	29	37	46	67	68

The following code accepts the frequencies in d and puts the cumulative frequency array into cf;

```
int cf [] = new int[d.length];  // create the output array

for (int pos = 0; pos < d.length; pos++){
  int total = 0;

     // add up the current and the preceding values
  for (int i = 0; i <= pos; i++)
     total = total + d[i];

     // and store it in the output
  cf[pos] = total;
}
```

i. Does it work?
The inner loop calculates the cumulative frequency for the current position.
The outer loop goes through each position, calculating its cf and inserting it.

ii. Estimate its complexity
The basic operation: addition of elements in the array (Could discuss if this is reasonable – any other addition e.g. the incrementing of the for loop variable is just as “complex” and time consuming – however, the number of these are likely to be proportional to the number of additions involving array elements, so our assumption is ok)

Simple analysis: nested loops -> O(n2)

More careful analysis the Number of additions in each iteration of the outer loop is:
1, 2, 3, …n
Σn for n from 1 to n is n(n+1)/2
Again O(n2)

iii. Derive a more efficient algorithm

```
int cf [] = new int[d.length];  // create the output array
cf[0]= d[0];
for (int pos = 1; pos < d.length; pos++)
{
  cf[pos] = d[pos] + cf[pos-1];
}
```

iv. Demonstrate that it works.

At the start of the loop cf[0] is correct
After each iteration, cf[0] .. cf[pos] are correct
After the last iteration cf[0] .. cf[n] are correct

v. Estimate its complexity. Is the difference significant?

Each element is examined precisely once: O(n)

Recursion

10. Show how the following data structures or problems could be described in a recursive way – that is showing that the data structure or problem can be broken down in to parts that are similar to the original. For example, a tree data structure consists of a node with left and right sub-trees.

11. Also identify the "base case" – the simplest form of the data structure or problem. In this case, it is the NULL or empty tree.

a. A linked list.

Recursive description: first item followed by a list

Base case: empty list

b. Finding a route from one city to another across a network of roads linking cities

Recursive description: first item followed by a list

Base case: empty list

c. An integer

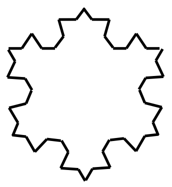
Recursive description: most significant digit + integer

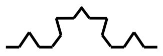
Base case: single digit

d. Find best move at chess / draughts / checkers

Recursive description: make a move, find best move for opponent

Base case: game is over



e. A snowflake - concentrate on the following shape: 

Recursive description:

shape(n) = shape(n-1), turn 60, shape(n-1), turn –120, shape(n-1), turn 60, shape(n-1)

Base case: straight line

f. The windows filing system (folders & files)

Recursive description: A folder contains files and folders

Base cases: an empty folder or a file

g. XML documents: e.g.

```
<module>
   <details>
         <title>co3401</title>
         <department>computing</department>
   </details>
   <students>
         <student> <name>able</name> …. </student>
         <student> <name>baker</name> …. </student>

   </students>
</module>
```

Recursive description: An element contains other elements or text

Base cases: an empty element or text

12. What do the following routines do:

```
Node *routineA(int val, Node * list){
if(list == NULL){
return new Node(val, NULL);
}
else if (list->value >= val){
return new Node(val, list);
}
else {
list->next = routineA(val, list->next);
return list;
}
}
```

Insert a value into a list sorted in ascending order

```
Node *routineB(int val, Node * list){
if(list == NULL){
return NULL;
}
else if (list->value == val){
return list->next;
}
else {
list->next = routineB(val, list->next);
return list;
}
}
```

Delete the first occurrence of an item from a list

```
Node *routineC(int val, Node * list){
if(list == NULL){
return NULL;
}
else if (list->value == val){
return routineC(val, list->next);
}
else {
list->next = routineC(val, list->next);
return list;
}
}
```

Delete all occurrences of an item from a list

```
void printA(Node * list){
if (list != NULL){
printf("%i ", list->value);
printA(list->next);
}
}
```

Print a list in order

```
void printB(Node * list){
if (list != NULL){
printB(list->next);
printf("%i ", list->value);
}
}
```

Print a list in reverse order

13. Convert printA and printB to iterative methods.

point to first item

```
while not pointing to NULL
        Print item
        Move to next item
```

Insert items one at a time into initially empty temporary list:

```
        point to first item
        while not pointing to NULL
                add item to front of temporary list
                Move to next item
        Print the temporary list
```

Alternatively (work down the list reversing the pointers then work back, printing each item and restoring the pointers – roughly same time but no extra space. Following code is untested)

```
if list is not empty
        previous = first item
        current = previous -> next;
        while (current != null){
                next = current -> next
                current ->next = previous
                previous = current
                current = next
        }
print previous->value
current = previous -> next
previous ->next = null
while (previous != head){
        print current->value
        next = current->next
        current->next = previous
        previous = current
        current = next
}
```

14. A tree class is defined as

```
class tree {
    int val;
    tree * left;
    tree * right;
}
```

Write a recursive routine to check it two trees are equal: boolean isEqual(tree * first, tree * second)

```
boolean isEqual(tree * first, tree * second){
        if (first == null)
                return second == null;
        else if (second == null)
                return false;
        else if (first->val == second->val)
                return isEqual(first->left, second->left) && isEqual(first->right, second->right);
        else
                return false;
}
```

Discuss how you could write the routine iteratively. Which would be better?

Difficult.

```
while(true){
    if (first == null && second == null){
        if (stack is not empty){
            pop first
            pop second
        }
        else
            return true
    }
    else if(first != null && second != null){
        if(first->val == second->val){
            push second->left
            push first->left
            first = first->right
            second = second->right
        }
        else
            return false
    }
    else return false
}
```

Probably use similar space and time: recursion is simpler

Discuss whether your conclusions would be the same for a routine to compare two lists?

No iterative routine is simpler