CMPT 307 (Fall 2019)

1 Analysis of Algorithms (You should also read Chapter 2 of CLRS).

This topic is concerned primarily with

- memory requirement, called space complexity, analysis of algorithms
- time requirement, called time complexity, analysis of algorithms

We will focus on time complexity only since the techniques used to measure this complexity subsume the techniques to determine the space complexity.

In order to measure the time complexity, we will not use the execution time of the program as the measure of work done by the algorithm, since the execution time

1. varies with the computer
2. is dependent on the programming language
3. is affected by biased implementation
4. is a time consuming process

The objective is to find a measure of work that tells us things about the efficiency of the algorithm independent of computer, programming language, the programmer etc.

1.1 Our computer (model of computation)

Our computer has

- unlimited random access memory
- no Input/Output mechanism. The input data reside in the memory at the beginning of the computation, and the output data are left in the memory at the end.

- no underflow/overflow. This means that the word size is arbitrarily large.
- the same execution time for an operation, whether it is a comparison, addition, or multiplication etc.

The time complexity of an algorithm is measured as a function of the problem size. The problem sizes of some examples are:

Searching for an element in a list of n elements	n
Sorting a list of n elements	n
Multiplying two matrices of sizes $m \times n$ and $n \times p$	m, n, p
Determining whether an element x is a prime number	number of bits to represent x

1.2 Counting

There are two acceptable approaches in the counting process.

Worst case: We obtain the bound on the largest possible counts on input of a given size. It generally captures efficiency in practice. We will be mostly concerned with this bound.

Average Case We obtain the bound on the average possible counts on random input of a given size. It is hard (sometimes impossible) to accurately model real instances by random distributions. Algorithm tuned for a certain distribution may perform poorly on other inputs.

Let D_n be the set of all inputs of size n. Let I denote one particular input instance. Let $t(I)$ denote the time the algorithm takes to solve problem P with input I. Let $W(n)$ and $A(n)$ denote the worst case and average complexity of the algorithm to solve P respectively. Then $W(n)$ and $A(n)$ can be written as follows.

$$W(n) = max_{I \in D_n} t(I) \text{ and } A(n) = \frac{\sum_{\forall I \in D_n} t(I)}{|D_n|}$$

We now discuss the following sections in our quest for counting in order to analyze algorithms.

- Operation counts
- Step counts
- Counting Cache misses (this will be discussed later)

1.2.1 Operation Counts

One way to estimate the time complexity of a method is to select one or more operations, such as add, multiply, and compare and determine how many of each is done. For this to work we need to determine the dominant operations and then just count them.

Let us consider some examples. For each of the problems, we identify the worst case situation when the algorithm is made to work the most. We then perform the operation count for that given scenario.

A: Largest Element:

The function Max finds the maximum element of a set of n elements. There are n elements stored in an array. The array is assumed to be in the memory. The output is the index of the maximum element, and is left in the memory.

```
Function Max(T[1..n])
{
        if (n < 1) return -1;
        index = 1;
        for i = 2 to n
        {
                if (T[index] < T[i]) then index = i;
                i++;
        }
        return index;
}
```

Function Max works the most when the elements are listed in the array in increasing order. In this situation, the instruction *index* = *i* will be executed $n-1$ times. If the focus is on the comparison operation, then for an array of size n, in the worst case, function Max will perform $n-1$ comparisons. We also notice that i++ will execute in total $n-1$ additions as well. Thus we can conclude that the worst case operation count is proportional to n, the size of the input. Therefore, function Max has linear complexity in the worst case situation. **This implies that if the input size is doubled, the execution time of function Max is also doubled.**

In the average case we notice that the instruction *index*=*i* will not be executed $n-1$ times. May be on an average, about half of the times it will be executed. Still the statement i++ will be executed $n-1$ times. Therefore, the average case

complexity of function Max is still linear.

B: Sequential Search

The following function searches an array of n elements for a key. Again the array is assumed to be in the memory. The output is the index of the array containing the key.

```
Function Search(T[1..n], key)
{
        i = 1;
        while (i <= n & T[i] != key)
        {
                i++;
        }
        if (i = n + 1) then return -1
                else return i;
}
```

The worst case scenario is when the key is not in the array. In this case i++ will be executed n times. The dominant operation is the comparison operation. Therefore, the time complexity of the algorithm is proportional to the number of the comparison operation count. This implies that the worst case complexity of Search is linear.

The average case complexity can be estimated as follows. We count the number of comparisons. Clearly, it is the average of the cases: when the index of the search key is at location 1, or location 2, or location 3, so on. The total comparison cost is $1+2+3+....+n$ which is $\frac{n(n+1)}{2}$. Therefore, the average complexity of Search is $\frac{n(n+1)/2}{n} = \frac{n+1}{2}$. Note that we have assumed that all the elements are distinct and each is searched for with equal frequency.

C: Insertion Sort

Insertion sort is performed by inserting one element at a time to a partially sorted list. An array of n elements is given. The algorithm sorts the elements in increasing order. The algorithm is an in-place algorithm. A sorting algorithm is said to be in-place if it requires very little additional space besides the initial array holding the elements that are to be sorted.

```
Function InsertSort(T[1..n])
{
        for i = 2 to n
        {
                x = T[i];
                j = i - 1;
                while (j > 0 & x < T[j])
                {
                        T[j+1] = T[j];
                        j--;
                }
                T[j+1] = x;
        }
}
```

The worst case scenario is when the while loop is executed i times during the i^{th} for loop. This happens when the input list has elements in decreasing order (try with a simple example). In the worst case, the while loop (comparison operation) will be executed i times when $i = 2, 3, ..., n$. Therefore, the total comparison cost is $\sum_{i=2}^{i=n} i$ which is $\frac{n(n+1)}{2} - 1$. This says that as the input size is doubled, the comparison cost is quadrupled. Thus the running time of InsertionSort is quadratic.

1.3 Step Counts

The operation-count method of estimating time complexity counts only the dominant operations. In the step-count method, we count for the time spent in all parts of the algorithm. The step count is a function of the problem size.

It is assumed that each step takes one unit time to execute. This time is independent of the problem size. Thus a step like *return* $a + b + b * c + (a + b -$

$c)/(a+b)+4$; also takes one unit of time. We also count a statement such as $x = y$ as a single step.

The tables below give the step counts of the three algorithms mentioned above.

A: Largest Element:

```
=============================================================================
  Statement                                    |   Frequency (worst case)
=============================================================================
Function Max(T[1..n])                          |           0
{
   if (n < 1) return -1;                       |           1
   index = 1;                                  |           1
   for i = 2 to n                              |           n-1
   {
      if (T[index] < T[i]) then index = i;     |           n-1
      i++;                                     |           n-1
   }
   return index;                               |           1
}
=============================================================================
  Total step count                             |           3n
=============================================================================
```

B: Sequential Search

```
=========================================================================
   Statement                                 |   Frequency (worst case)
=========================================================================
Function Search(T[1..n], key)                |           0
{
   i = 1;                                    |           1
   while (i <= n & T[i] != key)              |           n
   {
      i++;                                   |           n
   }
   if (i = n + 1) then return -1             |           1
      else return i;
}
=========================================================================
   Total step count                          |           2n + 2
=========================================================================
```

C: Insertion Sort

```
=========================================================================
   Statement                                 |   Frequency (worst case)
=========================================================================
Function InsertSort(T[1..n])                 |           0
{
   for i = 2 to n                            |           n-1
   {
      x = T[i];                              |           n-1
      j = i - 1;                             |           n-1
      while (j > 0 & x < T[j])               |           1+2+3+ ... + n-1
      {
         T[j+1] = T[j];                      |           1+2+3+ ... + n-1
         j--;                                |           1+2+3+ ... + n-1
      }
      T[j+1] = x;                            |           n-1
```

```
   }
}
=============================================================================
   Total step count                          |          4(n-1) + 3n(n-1)/2
=============================================================================
```

The frequency of an executed statement is called the **order of magnitude of the statement**. The **order of magnitude of an algorithm** is the sum of the order of magnitude of all of its statements.

1.4 Discussion

An algorithm may contain thousands of statements. It is not practical to find the order of magnitude of all of its statements. Since we are interested in finding the growth of the running time as the input size increases, we identify a subset of statements which dominates the running time of the algorithm. In InsertionSort routine we realize that the most of the work, in relative terms, is done in the *while* loop. Thus just by focussing on that piece, we notice that the running time of the algorithm is quadratic. In some way we are interested in determining the growth rate of the running time ignoring the constants and lower order terms. This complexity is known as the the asymptotic complexity meaning the growth rate of the running time of the algorithm as the problem size gets larger and larger to infinity. Therefore the worst case asymptotic complexities of Max, Search and InsertionSort routines are $O(n), O(n)$ and $O(n^2)$ respectively.

1.5 Examples

1. The following program tests if an array of elements of size n has any value duplicated. There are two nested loops. Therefore, the asymptotic behaviour is quadratic.

```
bool duplicate = false;
for ( int i = 0; i < n; ++i ) {
    for ( int j = 0; j < n; ++j ) {
        if ( i != j && A[ i ] == A[ j ] ) {
            duplicate = true;
            break;
        }
    }
    if ( duplicate ) {
        break;
    }
}
```

1.6 Rule of thumb

1. Simple programs can be analyzed by counting the nested loops. A single loop over n items yields linear complexity. A loop within a loop generally yields quadratic complexity. A loop within a loop within a loop generally yields a cubic complexity.

2. Given a series of *for* loops that are sequential, the slowest of them determines the asymptotic behaviour of the program. Two nested loops followed by a single loop is asymptotically the same as the nested loops alone. Here the nested loops dominate the simple loop.

3. Improving the asymptotic behaviour of the algorithm often increases its performance in a significant way. This is much more than any smaller optimizations such as a faster programming language.

1.7 Effects of Increasing Input Size

The following table is obtained from `https://cs.lmu.edu/~ray/notes/alganalysis/`.

The Effects of Increasing Input Size

Suppressing leading constant factors hides implementation dependent details such as the speed of the computer which runs the algorithm. Still, you can some observations even without the constant factors.

For an algorithm of complexity	If the input size doubles, then the running time
1	stays the same
$\log n$	increases by a constant
n	doubles
n^2	quadruples
n^3	increases eight fold
2^n	is left as an exercise for the reader

1.8 Effects of Faster Computer

Again the following tables are obtained from `https://cs.lmu.edu/~ray/notes/alganalysis/`.

The Effects of a Faster Computer

Getting a faster computer allows to solve larger problem sets in a fixed amount of time, but for exponential time algorithms the improvement is pitifully small.

For an algorithm of complexity	If you can solve a problem of this size on your 100MHz PC	Then on a 500MHz PC you can solve a problem set of this size	And on a supercomputer one thousand times faster than your PC you can solve a problem set of this size
n	100	500	100000
n^2	100	223	3162
n^3	100	170	1000
2^n	100	102	109

More generally,

$T(n)$	On Present Computer	On a computer 100 times faster	On a computer 1000 times faster	On a computer one BILLION times faster
n	N	100N	1000N	1000000000N
n^2	N	10N	31.6N	31623N
n^3	N	4.64N	10N	1000N
n^5	N	2.5N	3.9N	63N
2^n	N	N + 6.64	N + 9.97	N + 30
3^n	N	N + 4.19	N + 6.3	N + 19

1.9 Example discussed in the class

In the following figure, there are four cases whose step counts we need to compute. Looking at the cases, we see that in the worst case the step *sum++* is the dominant step. We have discussed the cases 1,2, and 3 have been discussed in the class. I will analyze case 4 here. In the class I have missed an important observation (thanks to the student for noting this).

```
1.
    sum = 0
    for(i = 0; i < n; i++)
        for( j = 0; j < n * n; j++)
            sum++;

2.
    sum = 0
        for(i = 0; i < n; i++)
            for(j = 0; j < i; j++)
                sum++;

3.
    sum = 0
    for(i = 0; i < n; i++)
        for(j = 0; j < i * i; j++)
            for( k = 0; k < j; k++)
                sum++;

4.
    sum = 0
    for(i = 0; i < n; i++)
        for(j = 1; j < i * i; j++)
            if( j % i == 0)
                for(k = 0; k < j; k++)
                    sum++;
```

Case 4 is very similar to Case 3 except for the line 24. The **if** statement in line 24 will be true when $j = i, 2i, 3i, \ldots, (i-1) * i$. Number of times line 26 will be executed is then

$$\sum_{i=0}^{i=n-1} \{i + 2i + 3i + \ldots + (i-1) * i\}$$