Introduction to Analysis of Algorithms

Analysis of Algorithms

- To determine how efficient an algorithm is we compute the amount of time that the algorithm needs to solve a problem.
- Given two algorithms for the same problem, the preferred one is the faster.

Time Complexity

- The *time complexity* of an algorithm is a function that gives the amount of time that the algorithm takes to complete.
- The time complexity depends on the size of the input, or the number of data items in the input.

Time Complexity

- The time complexity of an algorithm is denoted as *t(n) or f(n)*, where n is the size of the input.
- The time complexity is non-decreasing in the size of the input, i.e. the amount of time needed by an algorithm cannot decrease as the size of the input increases. (For example, copying a large array cannot take less time than making a copy of a smaller array.)

Asymptotic Growth

- Let

$$t(n) = 15n^2 + 45\ n$$

be the time complexity of an algorithm. For each value of n the function indicates the amount of time required by the algorithm.

So t(1) is the running time of the algorithm when the input has 1 data item, t(100) is the running time of the algorithm when the input has 100 data items, and so on.

The following table shows the value of the above function t(n) for various values of n. Assume that the times are given in microseconds (1 microsecond = 0.000001 second)

No. of items **n**	**$15n^2$**	**45n**	**$15n^2 + 45n$**
1	15	45	60
2	60	90	150
5	375	225	600
10	1,500	450	1,950
100	150,000	4,500	154,500
1,000	15,000,000	45,000	15,045,000
10,000	1,500,000,000	450,000	1,500,450,000
100,000	150,000,000,000	4,500,000	150,004,500,000
1,000,000	15,000,000,000,000	45,000,000	15,000,045,000,000

Asymptotic Growth

When trying to decide whether an algorithm is efficient we are only interested in the value of its time complexity for large values of n, because for small values of n the running time of an algorithm is very small. (For example, in the previous table, for values of n smaller than 100 the running times are much smaller than 1 second. However, for n = 1 million, the running time of the algorithm is 173 days.)

Asymptotic Growth

For large values of n, the value of the time complexity function is mainly determined by the largest term in the function. For example, for the above time complexity $t(n) = 15n^2 + 45n$ the largest term is $15n^2$. Notice that for large values of n, the value of $15n^2$ is much bigger than the value of 45n.

We say that $15n^2$ asymptotically dominates 45n and so that t(n) has the same asymptotic behaviour as $15n^2$.

Big-Oh Notation

There is a mathematical notation called the order or big-Oh notation for expressing the asymptotic growth of a time complexity function.

A formal definition for the big-Oh notation will be given in the second year course on data structures an algorithms. The big-Oh notation captures the running time of an algorithm independently of how it is implemented and executed, i.e., independently of the programming

Big-OhNotation

language in which it is implemented, of the computer in which it is run and of the operating system used by such a computer.

Roughly speaking, since different computers differ in speed by a constant factor (a factor that does not depend on the algorithm being executed and the size of the input on which the algorithm operates), when expressing the asymptotic growth of a time complexity

Big-Oh Notation

function using the big-Oh notation, any constant factors in the function are ignored.

So, for example the time complexity function $t(n) = 15n^2 + 45n$, grows asymptotically as fast as $15n^2$, which in big-Oh notation is denoted as $O(n^2)$ since, as explained above, the constant factor 15 is ignored. A factor is constant if its value does not depend on the size of the input. (Note that 15 does not depend on n.)

Big-Oh Notation

- The asymptotic growth of the time complexity function of an algorithm is referred to as the ***order of*** the time complexity of the algorithm.
  - ***Example***: ***O(n^2)*** means that the time taken by the algorithm grows quadratically as n increases
  - ***O(1)*** means constant time, independent of the size of the input

Big-Oh Notation

Given a time complexity function of the form

$$t(n) = t_1(n) + t_2(n) + \ldots + t_k(n)$$

Where the number k of terms is constant (i.e. independent of n) the *order* of t(n) is determined by the largest term $t_i(n)$.

Some examples of computing the *order* of several functions are given in the next page.

Determining Time Complexity

When computing the time complexity of an algorithm, we focus our attention on the most expensive parts of the algorithm, namely the loops and the recursive calls. We will look at recursive calls later, for now we will look at how to compute the time complexity of loops.

Determining Time Complexity

The time complexity function of an algorithm gives the running time of the algorithm. From just the description of an algorithm we cannot determine the amount of time, say in seconds, that it would need to execute. So, how can we determine the time complexity of an algorithm like this one?

```
x = 0;
for (int i = 0; i < n; i++)
        x = x + 1;
```

Determining Time Complexity

When analyzing an algorithm we cannot determine its actual running time, but we can estimate it if we count the number of basic or primitive operations that the algorithm performs.

A basic or primitive operation is an operation that takes constant time (i.e. independent of the size of the input). Some examples of primitive operations are: assigning a value to a variable, comparing the values of two variables and adding two values.

Analysing Loop Execution

For the algorithm below, the primitive operations are =, <, ++, +. Outside the loop one operation is performed (=). In each iteration of the loop four operations are performed (<, ++, =, +). Since the loop is repeated n times, the total number of operations performed is

4n+1. Roughly speaking, this means that the running time of the algorithm is proportional to 4n+1. The order of the function 4n+1 is O(n) and so is the order of the time complexity of the algorithm.

```
x = 0;
for (int i=0; i<n; i++)
    x = x + 1;
```

Time Complexity

Since the number of primitive operations performed by an algorithm and the time complexity of the algorithm have the same order, we can say that the time complexity of an algorithm is the same as the number of primitive operations that it performs.

So, for the algorithm above we say that its time complexity is $t(n) = 4n+1$. Note that we are not really interested in the exact value of this function, but we only wish to know the *order* of the time complexity. So we say that the above algorithm as time complexity $O(n)$.

Notice that since any constant factors in the time complexity function are ignored when computing the order of the time complexity, we do not need to count exactly the number of primitive operations performed by the algorithm. For example, for the same algorithm above

We can say that each iteration of the for loop performs a constant number k of operations. Since the loop is repeated n times, the total number of operations performed by the for loop is kn. Outside the loop a constant number k’ of additional operations are performed, so the total number of operations performed by the algorithm is kn + k’, which is O(n).

Note that we obtained the same result, namely that the time complexity is O(n) even though we did not count the exact number of operations.

```
x = 0;
   for (int i=0; i<n; i++)
       x = x + 1;
```

However, we need to be very careful when counting the number of iterations of any loop, as a wrong value for the number of operations might lead to the wrong order for the time complexity.

Nested loops

When computing the time complexity of an algorithm with nested loops, like the algorithm below, we usually consider the innermost loop first and work our way outward.

```
for (int i=0; i<n; i++) {
        x = x + 1;
        for (int j=0; j<n; j++)
                y = y – 1;
 }
```

Each iteration of the inner loop performs a constant number k of operations. Every time that the loop is performed, it iterates n times (as j takes values 0, 1, …, n-1), therefore the total number of operations performed by the inner loop is kn.

Outside the inner loop, but inside the outer one, an additional constant number k' of operations are performed (x = x+1), so each iteration of the outside loop performs k' + kn operations.

The outer loop is repeated n times, so the total number of operations performed by this algorithm is $t(n) = n(k' + kn) = k'n + kn^2$. Note that in this function the term kn^2 asymptotically dominates the term $k'n$, so ignoring constant factor we get that $t(n)$ is $O(n^2)$.

More Loop Analysis Examples

```
x = 0;
for (int i=0; i<n; i=i+2) {
    x = x + 1;
}
```

For the above algorithm, each iteration of the loop performs a constant number k of operations. The loop is repeated n/2 times, as in each iteration the value of i increases by 2. Furthermore, outside the loop an additional constant number k' of operations are performed.

Hence, the total number of operations performed by the algorithm is k' + kn/2. The time complexity of this algorithm, then is O(n).

More Loop Analysis Examples

```
x = 0;
for (int i=1; i<=n; i=i*2) {
    x = x + 1;
}
```

For the above algorithm, outside the loop a constant number k’ of operations are performed. Also each iteration of the loop performs a constant number k of operations. Counting the total number of iterations of the loop is more complicated. Let us look at the value of i in each iteration:

Iteration number	**Value of i**
1	$1 = 2^0$
2	$2 = 2^1$
3	$4 = 2^2$
4	$8 = 2^3$
…	…
1 + log n	$n = 2^{\log n}$

Hence, the number of iterations performed by the loop is 1 + log n. The total number of operations performed by the loop is t(n) = k’ + k(1 + log n) = k’ + k + k log n. The dominating term is k log n, so the time complexity is O(log n).

More Loop Analysis Examples

```
x = 0;
for (int i=0; i<n; i++)
   for (int j = i, j < n, j ++) {
               x = x + 1;
}
```

The inner loop performs a constant number k of operations in each iteration. The inner loop repeats once for each value of j between i and n-1, thus the number of iterations is n-i. The number of operations performed by the inner loop is then k(n-i). Note that this expression depends on the value of the variable i.

The outer loop is repeated once for each value of i between 0 and n-1, and as we saw in each iteration the number of operations performed by the inner loop is k(n-i). Hence the total number of operations performed by this loop is
k(n-0) + k(n-1) + k(n-2) + … + k(n-(n-1)) =
kn + k(n-1) + k(n-2) + … + k(1) = k $\sum_{i=1}^{n} i$ = kn(n+1)/2 =
$kn^2/2$ + kn/2.

Outside the outer loop a constant number k’ of additional operations are performed, so the total number of operations performed by the algorithm is k’ + kn/2 + $kn^2/2$. the dominating term is $kn^2/2$, so the time complexity of the algorithm is $O(n^2)$.