Algorithm Analysis

We begin with informal definitions of some important notions that are frequently used in this lecture notes.

- Algorithm. A finite sequence of instructions, each of which has a clear meaning and can be executed with a finite amount of effort in finite time.

- Data Type. Data type of a variable (also called an object) is the set of values that the object may assume.

- Abstract Data Type (ADT): An ADT is a set of elements with a collection of well-defined operations.

    - The operations can take as operands not only instances of the ADT but other types of operands or instances of other ADTs.
    - Similarly results need not be instances of the ADT
    - At least one operand or the result is of the ADT type in question.

Object-oriented languages such as C++ and Java provide explicit support for expressing ADTs by means of classes.

Examples of ADTs include vector, list, stack, queue, set, tree, graph, etc.

- Data Structures: An implementation of an ADT is a translation into statements of a programming language:

    - the declarations that define a variable to be of that ADT type
    - the operations defined on the ADT (using procedures of the programming language)

An ADT implementation chooses a data structure to represent the ADT. Each data structure is built up from the basic data types of the underlying programming language using the available data structuring facilities, such as arrays, structs, pointers, files, sets, etc.

Example: A Queue is an ADT which can be defined as a sequence of elements with operations such as Queue(Q), IsEmpty(Q), EnQueue(x, Q), and DeQueue(Q). This can be implemented using data structures such as

- array
- singly linked list
- doubly linked list
- circular array

Complexity of Algorithms

It is very convenient to classify algorithms based on the relative amount of time or relative amount of space they require and specify the growth of time or space

requirements as a function of the input size. Thus, we have the notions of:

- Time Complexity: Running time of the program as a function of the size of input
- Space Complexity: Amount of computer memory required during the program execution, as a function of the input size

Big Oh Notation

- A convenient way of describing the growth rate of a function and hence the time complexity of an algorithm.

Let n be the size of the input and f (n), g(n) be positive functions of n.

Definition: Big Oh. f (n) is O(g(n)) if and only if there exists a real, positive constant C and a positive integer n_0 such that $f(n) \leq C\, g(n)\ \forall\ n \geq n_0$

- Note that O(g(n)) is a class of functions.
- The "Oh" notation specifies asymptotic upper bounds
- O(1) refers to constant time. O(n) indicates linear time; O(nk) (k fixed) refers to polynomial time; O(log n) is called logarithmic time; $O(n^2)$ refers to exponential time, etc.

Examples

- Let $f(n) = n^2 + n + 5$. Then:
  - f(n) is $O(n^2)$
  - f(n) is $O(n^3)$
  - f(n) is not O(n)
- Let $f(n) = 3^n$
  - f(n) is $O(4^n)$
  - f(n) is not $O(2^n)$
- If $f_1(n)$ is $O(g_1(n))$ and $f_2(n)$ is $O(g_2(n))$, then
  - $f_1(n) + f_2(n)$ is $O(\max(g_1(n), g_2(n)))$

An Example: Complexity of Mergesort: Mergesort is a divide and conquer algorithm, as outlined below. Note that the function mergesort calls itself recursively. Let us try to determine the time complexity of this algorithm.

```
Procedure mergesort {
Input: list L, integer n.
Output: Sorted list

     if (n == 1)
          return (L);
     else {
          Split L into two halves L1 and L2;
          return (merge(mergesort (L1, n/2), (mergesort (L2,
      n/2));
     }
```

```
} // end Procedure mergesort
```

Let T(n) be the running time of *mergesort* on an input list of size n. Then,
$T(n) \leq C_1$ (if $n = 1$) (C_1 is a constant)

$$\leq \underbrace{2\,T\left(\frac{n}{2}\right)}_{\text{two recursive calls}} + \underbrace{C_2 n}_{\text{cost of merging}} \quad (\text{if } n > 1)$$

If $n = 2^k$ for some k, it can be shown that $T(n) \leq 2^k T(1) + C_2 k 2^k$

That is, T(n) is O(n log n).

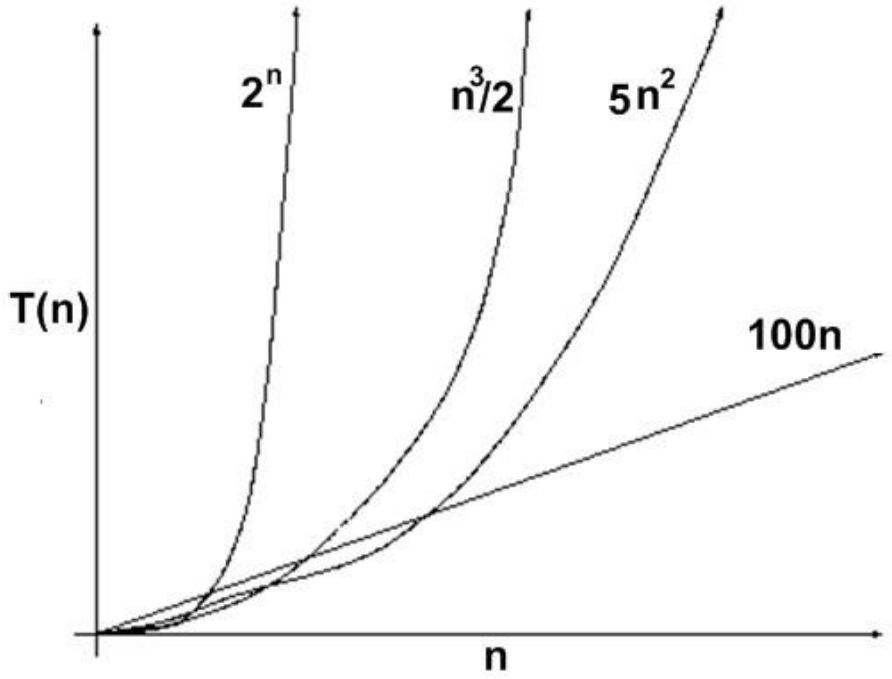

Figure 1: Growth rates of some functions

Role of the Constant: The constant C that appears in the definition of the asymptotic upper bounds is very important. It depends on the algorithm, machine, compiler, etc. It is to be noted that the big ”Oh” notation gives only asymptotic complexity. As such, a polynomial time algorithm with a large value of the constant may turn out to be much less efficient than an exponential time algorithm (with a small constant) for the range of interest of the input values. See Figure ?? and also Table 1.1.

Worst Case, Average Case, and Amortized Complexity

- Worst case Running Time: The behavior of the algorithm with respect to the worst possible case of the input instance. The worst-case running time of an algorithm is an upper bound on the running time for any input. Knowing it gives

us a guarantee that the algorithm will never take any longer. There is no need to make an educated guess about the running time.

T(n)	Maximum Problem Size that can be solved in		
	100 Time Units	1000 Time Units	10000 Time Units
100 n	1	10	100
$5\ n^2$	5	14	45
$n^3/2$	7	12	27
2^n	8	10	13

Table 1: Growth rate of functions

- Average case Running Time: The expected behavior when the input is randomly drawn from a given distribution. The average-case running time of an algorithm is an estimate of the running time for an ”average” input. Computation of average-case running time entails knowing all possible input sequences, the probability distribution of occurrence of these sequences, and the running times for the individual sequences. Often it is assumed that all inputs of a given size are equally likely.
- Amortized Running Time: Here the time required to perform a sequence of (related) operations is averaged over all the operations performed. Amortized analysis can be used to show that the average cost of an operation is small, if one averages over a sequence of operations, even though a simple operation might be expensive. Amortized analysis guarantees the average performance of each operation in the worst case.

1. For example, consider the problem of finding the minimum element in a list of elements.
   Worst case = O(n)
   Average case = O(n)
2. Quick sort
   Worst case = $O(n^2)$
   Average case = O(n log n)
3. Merge Sort, Heap Sort
   Worst case = O(n log n)
   Average case = O(n log n)
4. Bubble sort
   Worst case = $O(n^2)$
   Average case = $O(n^2)$
5. Binary Search Tree: Search for an element

Worst case = O(n)
Average case = O(log n)

Big Omega and Big Theta Notations

The Ω notation specifies asymptotic lower bounds.

Definition: Big Omega. f(n) is said to be Ω(g(n)) if ∃ a positive real constant C and a positive integer n0 such that $f(n) \geq Cg(n)$ ∀ $n \geq n_0$
An Alternative Definition : f(n) is said to be Ω(g(n)) iff ∃ a positive real constant C such that f (n) ≥ Cg(n) for infinitely many values of n.
The Θ notation describes asymptotic tight bounds.

Definition: Big Theta. f (n) is Θ(g(n)) iff ∃ positive real constants C_1 and C_2 and a positive integer n0, such that $C_1 g(n) \leq f(n) \leq C_2 g(n)$ ∀ $n \geq n_0$

An Example:
Let $f(n) = 2n^2 + 4n + 10$. f(n) is $O(n^2)$. For,

$f(n) \leq 3n^2$ ∀ $n \geq 6$
Thus, C = 3 and $n_0 = 6$

Also,

$f(n) \leq 4n2$ ∀ $n \geq 4$
Thus, C = 4 and n0 = 4

f(n) is $O(n^3)$

In fact, if f(n) is $O(n^k)$ for some k, it is $O(n^h)$ for h > k
f(n) is not O(n).

Suppose ∃ a constant C such that $2n^2 + 4n + 10 \leq Cn$ ∀$n \geq n_0$ This can be easily seen to lead to a contradiction. Thus, we have that:
f(n) is $\Omega(n^2)$ and f(n) is $\Theta(n^2)$