University of Kelaniya Sri Lanka

COSC 21063

Analysis of Algorithms

Lecturer : Ms.D.M.L.Maheshika Dissanayake

Analysis of Algorithms

- There are a number of ways we can look at an algorithm's performance, but usually the aspect of most interest is how fast the algorithm will run.
- In some cases, if an algorithm uses significant storage, we may be interested in its space requirement as well.

Analysis of Algorithms

Worst-case analysis

- The metric by which most algorithms are compared. Other cases we might consider are the average case and best case. However, worst-case analysis usually offers several advantages.

Analysis of Algorithms

Comparing Algorithms

(1) Compare execution times?

Not good: times are specific to a particular computer.

(2) Count the number of statements executed?

Not good: number of statements vary with the programming language as well as the style of the individual programmer.

Analysis of Algorithms

Ideal Solution

Uses a high-level description of the algorithm instead of an implementation

Express running time as a function of the input size n

(i.e., f(n)).

Compare different functions corresponding to running times.

Such an analysis is independent of hardware, software environment.

Analysis of Algorithms

- Associate a "cost" with each statement.
- Find the "total cost“ by finding the total number of times each statement is executed.

Algorithm 1	**Cost**
for(i=0; i<n; i++)	c_2
arr[i] = 0;	c_1
... …… ...	

$$(n+1) \times c_2 + n \times c_1 = (c_2 + c_1) \times n + c_2$$

Analysis of Algorithms

- ***Algorithm 2*** *Cost*

```
sum = 0;                          c1
for(i=0; i<n; i++)                c2
   for(j=0; j<n; j++)             c2
   sum += arr[i][j];              c3
                                  ----------
```

$$c_1 + c_2 \; x \; (n+1) + c_2 \; x \; n \; x \; (n+1) + c_3 \; x \; n^2$$

Analysis of Algorithms

Computational complexity

- The growth rate of the resources an algorithm requires with respect to the size of the data it processes. O - notation is a formal expression of an algorithm's complexity.

Analysis of Algorithms

O-notation:

- The most common notation used to formally express an algorithm's performance.
- O -notation is used to express the upper bound of a function within a constant factor.

Simple Rules for O-Notation

- Constant terms are expressed as O (1). When analysing the running time of an algorithm, apply this rule when you have a task that you know will execute in a certain amount of time regardless of the size of the data it processes. Formally stated, for some constant c:

  $O(c) = O(1)$

- Multiplicative constants are omitted. When analyzing the running time of an algorithm, apply this rule when you have a number of tasks that all execute in the same amount of time. For example, if three tasks each run in time T (n) = n, the result is O (3n), which simplifies to O (n). Formally stated, for some constant c:

  $O(cT) = cO(T) = O(T)$

Simple Rules for O-Notation

- Addition is performed by taking the maximum. When analyzing the running time of an algorithm, apply this rule when one task is executed after another. For example, if T1(n) = n and T2(n) = n2 describe two tasks executed sequentially, the result is O (n) + O (n2), which simplifies to O (n2).

  Formally stated: O(T1)+O(T1+T2) = max (O(T1), O(T2))

- Multiplication is not changed but often is rewritten more compactly. When analyzing the running time of an algorithm, apply this rule when one task causes another to be executed some number of times for each iteration of itself. For example, in a nested loop whose outer iterations are described by T1 and whose inner iterations by T2, if T1(n) = n and T2(n) = n, the result is O (n)O (n), or O (n2). Formally stated: O(T1)O(T2) = O(T1 T2)

- **O(1)** - O(1) describes an algorithm that will always execute in the same time (or space) regardless of the size of the input data set.

```
bool IsFirstElementNull(IList<string> elements)
{
   return elements[0] == null;
}
```

- **O(n)** - O(n) describes an algorithm whose performance will grow linearly and in direct proportion to the size of the input data set.

```
bool ContainsValue(IList<string> elements, string value)
{
   foreach (var element in elements)
   {
      if (element == value) return true;
   }
   return false;
}
```

- **O(n^2)** - O(n^2) represents an algorithm whose performance is directly proportional to the square of the size of the input data set. (Nested Iterations)

```
bool ContainsDuplicates(IList<string> elements)
{
   for (var outer = 0; outer < elements.Count; outer++)
   {
      for (var inner = 0; inner < elements.Count; inner++)
      {
         // Don't compare with self
         if (outer == inner) continue;

         if (elements[outer] == elements[inner]) return true;
      }
   }
   return false;
}
```

- **O(2^n)** - O(2^n) denotes an algorithm whose growth doubles with each additon to the input data set.

```
int Fibonacci(int number)
{
   if (number <= 1) return number;

   return Fibonacci(number - 2) + Fibonacci(number - 1);
}
```

- **O(log n)** - The iterative halving of data sets.

  Ex: Binary Search

- **(n+1)/2**

- Without the dominant term – O(n)
- With the dominant term – n/2 + O(1)

- **nlg(n) – n + 1**

- Without the dominant term – O(nlgn)
- With the dominant term – nlgn + O(n)

Analysis of Algorithms

Examples:

1. 1000n^2 + 50n = ?
2. 100n + 5 = ?
3. 5n^3 + 10n = ?