ONLINE LEARNING

innovative • entrepreneurial • global

Algorithm Efficiency Analysis

SCSJ2013 Data Structures & Algorithms

Nor Bahiah Hj Ahmad & Dayang Norhayati A. Jawawi

Faculty of Computing

Objectives

Students are expected to be able to do the following:

- Understand algorithm efficiency analysis.
- Able to measure algorithm efficiency using big O notation.

What is algorithm analysis?

Study the efficiency of algorithms when the input size grow, based on the number of steps, the amount of computer time and the space usage.

Analysis of algorithms

- Algorithm analysis concern with the size and growth of data run on a particular algorithm.
- However, algorithm analysis should be independent of :
  - Specific implementations (programming language such as C, C++ or Java)
  - Specific Computer hardware (computer chips, OS, or speed)
  - Particular set of data (string, int, float)

Analysis of algorithms

Three possible states in algorithm analysis:

Worst case

- Longest running time for *any* input of size *n*
- A determination of the maximum amount of time that an algorithm requires to solve problems of size *n*

Best Case

- Shortest running time for *any* input of size *n*
- A determination of the minimum amount of time that an algorithm requires to solve problems of size *n*

Average Case

- Average running time for *all* inputs of size *n*
- A determination of the average amount of time that an algorithm requires to solve problems of size *n*

Big O Notation

- Complexity time can be represented by

  Big 'O' notation.
- Notation that used to show the complexity time of algorithms.
- Big 'O' notation is denoted as *O(f(n)).*

  whereby

  O – "the order of"

  f(n)- algorithm's growth-rate function

Example, O(1), O($log_x n$), O(n), (On $log_x n$), O(n^2)

Big O Notation Example

Notation	Execution time
$O(1)$	Constant function, independent of input size, n Example: Finding the first element of a list.
$O(\log_x n)$	Problem complexity increases slowly as the problem size increases. Example: Solve a problem by splitting into constant fractions of the problem (e.g., throw away ½ at each step)
$O(n)$	Problem complexity increases linearly with the size of the input, n Example: counting the elements in a list.

Big O Notation Example cont..

$O(n \log_x n)$	Log-linear increase - Problem complexity increases a little faster than n Characteristic: Divide problem into subproblems that are solved the same way Example: mergesort
$O(n^2)$	Quadratic increase. Problem complexity increases fairly fast, but still manageable Characteristic: Two nested loops of size n
$O(n^3)$	Cubic increase. Practical for small input size, n.
$O(2^n)$	Exponential increase - Increase too rapidly to be practical Problem complexity increases very fast Generally unmanageable for any meaningful n Example: Find all subsets of a set of n elements

Big O Notation Algorithm

Notation	Codes
O(1) *Constant*	`int counter = 1;` `cout << "cout  execution times" << counter;`

$O(log_x n)$
Logarithmic
	`int counter = 1; int i = 0;`
	`for (i = x; i <= n; i = i * x)`
	`{ // x must be > than 1`
	`   cout << "cout  execution times" << counter ;`
	`   counter++;`
	`}`
	`// Ex: if x = 2 and n = 16`
	`// i = 2, 4, 8, 16`

Big O Notation

<table>
<tr><td>O(n)
Linear</td><td>

```
int counter = 1; int i = 0;
for (i = 1; i <= n; i++) {
        cout << "cout  execution times" << counter ;
        counter++;
}
```

</td></tr>
<tr><td>O(n log_xn)
Linear Logarithmic</td><td>

```
int counter = 1; int i = 0; int j = 1;

for (i = x; i <= n; i = i * x)
{ // x must be > than 1
   while (j <= n) {
       cout << "cout  execution times" << counter;
       counter++; j++;
   }
}
```

</td></tr>
</table>

Big O Notation

- **Example of algorithm for common function:**

O(*n*^{2})
Quadratic
	`int counter = 1;`
	`int i = 0;`
	`int j = 0;`
	`for (i = 1; i <= n; i++) {`
	`    for (j = 1; j <= n; j++) {`
	`    cout << "cout  execution times" << counter;`
	`    counter++;`
	`    }`
	`}`

Big O Notation Algorithm

$O(n^3)$
Cubic
	`int counter = 1;`
	`int i = 0;`
	`int j = 0;`
	`int k = 0;`
	`for (i = 1; i <= n; i++) {`
	`  for (j = 1; j <= n; j++) {`
	`    for (j = 1; j <= n; j++) {`
	`      cout << "cout  execution times " << counter;`
	`      counter++;`
	`     }`
	`   }`
	`}`

Big O Notation Algorithm

O(2^n) Exponential	`int counter = 1;`
	`int i = 1;`
	`int j = 1;`
	`while (i <= n) {`
	`        j = j * 2;`
	`        i++;`
	`}`
	`for (i = 1; i <= j; i++) {`
	`    cout << "cout  execution times" << counter;`
	`    counter++;`
	`}`

Order of increasing complexity

$O(1) < O(\log_x n) < O(n) < O(n \log_2 n) < O(n^2) < O(n^3) < O(2^n)$

Notasi	n = 8	n = 16	n = 32
$O(1)$	1	1	1
$O(\log_2 n)$	3	4	5
$O(n)$	8	16	32
$O(n \log_2 n)$	24	64	160
$O(n^2)$	64	256	1024
$O(n^3)$	512	4096	32768
$O(2^n)$	256	65536	4294967296

Determine the number of steps

Algorithm	No. of Steps
void sample8 ( )	0
{	0
int n, x, i=1;	1
while (i<=n)	n
{	0
x++;	n.1 = n
i++;	n.1 = n
}	0
}	0
Number of Steps	**1 + 3n**

Consider the largest factor : 3n

and remove the coefficient : O(n)

Conclusion and Summary

- Algorithm analysis to study the efficiency of algorithms when the input size grow, based on the number of steps, the amount of computer time and space
- Can be done using Big O notation by using growth of function.
- Order of growth for some common function:
  - $O(1) < O(\log_x n) < O(n) < O(n \log_2 n) < O(n^2) < O(n^3) < O(2^n)$
- Three possible states in algorithm analysis best case, average case and worst case.

http://comp.utm.my/