WIA / WIB 1002
Data Structures

The Efficiency of Algorithms

Motivation

- Efficiency of computers remains an issue
- consider 3 algorithms that finding sum of 1+2+...+n:

<table>
<tr><th>Algorithm A</th><th>Algorithm B</th><th>Algorithm C</th></tr>
<tr>
<td><pre>
sum = 0
for i = 1 to n
    sum = sum + i
</pre></td>
<td><pre>
sum = 0
for i = 1 to n
{
    for j = 1 to i
        sum = sum + 1
}
</pre></td>
<td><pre>
sum = n * (n + 1) / 2
</pre></td>
</tr>
</table>

Which one is faster if n is big?

```java
// Computing the sum of the consecutive integers from 1 to n:
long n = 10000; // ten thousand

// Algorithm A
long sum = 0;
for (long i = 1; i <= n; i++)
   sum = sum + i;
System.out.println(sum);

// Algorithm B
sum = 0;
for (long i = 1; i <= n; i++)
{
   for (long j = 1; j <= i; j++)
      sum = sum + 1;
} // end for
System.out.println(sum);

// Algorithm C
sum = n * (n + 1) / 2;
System.out.println(sum);
```

Measuring efficiency

- An algorithm has both time and space requirements, called its complexity
- When we assess an algorithm’s complexity, we are not measuring how involved or difficult it is.
- we measure an algorithm’s :
  - time complexity—the time it takes to execute
  - space complexity—the memory it needs to execute
- But we will pay more attention to space complexity, since it is usually more important.

Problem size

- Problem size - the number of items that an algorithm processes.
- For example, if you are searching a collection of data, the problem size is the number of items in the collection
- if problem size is small - efficiency of algorithm is not important.
- it will be different if problem size is large

Growth rate function

- Actual time for an algorithm to solve a problem varies - depends on hardware, implementation of algorithm, problem size and etc.
- Growth rate function - measures how an algorithm's time requirement grows as the problem size grows.
- By comparing the growth-rate functions of two algorithms, you can see whether one algorithm is faster than the other for large-size problems

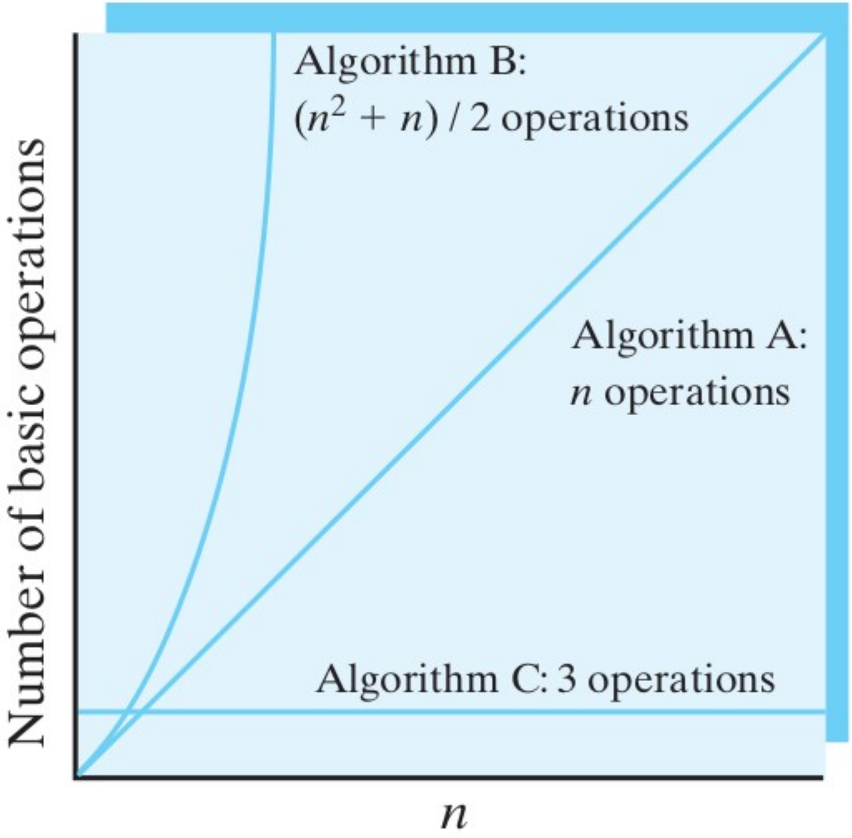
Counting Basic Operations

- An algorithm's basic operation is the most significant contributor to its total time requirement.

	Algorithm A	Algorithm B	Algorithm C
Additions	n	$n(n+1)/2$	1
Multiplications			1
Divisions			1
Total basic operations	$\mathbf{n}$	$\mathbf{(n^2+n)/2}$	**3**

Algorithm B requires time directly proportional to $(n^2 + n)/2$, and Algorithm C requires time that is constant and independent of the value of n.

The number of basic operations required by the algorithms



- We only interested in the case where problem size, n is large.
- So, when we compare algorithms, we consider only the dominant term in each growth-rate function.
  - $(n^2 + n)/2$ behaves like n^2 when n is large
  - So instead of using $(n^2 + n)/2$ as Algorithm B’s growth-rate function, we can use n^2
  - Algorithm A requires time proportional to n
  - Algorithm C is independent from n.

The relative magnitudes of common growth-rate functions:

$$1 < \log(\log n) < \log n < \log^2 n < n < n \log n < n^2 < n^3 < 2^n < n!$$

n	$\log(\log n)$	$\log n$	$\log^2 n$	n	$n \log n$	n^2	n^3	2^n	$n!$
10	2	3	11	10	33	10^2	10^3	10^3	10^5
10^2	3	7	44	100	664	10^4	10^6	10^{30}	10^{94}
10^3	3	10	99	1000	9966	10^6	10^9	10^{301}	10^{1435}
10^4	4	13	177	10,000	132,877	10^8	10^{12}	10^{3010}	$10^{19,335}$
10^5	4	17	276	100,000	1,660,964	10^{10}	10^{15}	$10^{30,103}$	$10^{243,338}$
10^6	4	20	397	1,000,000	19,931,569	10^{12}	10^{18}	$10^{301,030}$	$10^{2,933,369}$

* binary log

Best, Worst, and Average Cases

- time need for some algorithms not only depends on problem size, but also the data set.
- Example:
  - linear serach can be very fast if the key is at the first position of the list - best case
  - but can be one of the slowest if at the last position of the list - worst case
  - average case - average time for all data
- best and worst cases seldom occur

Big O notation (Big Oh)

- Computer scientists use Big O notation to represent an algorithm’s complexity
- Instead of saying that Algorithm A has a time requirement proportional to n, we say that A is O(n).
- Read as “Big O of n” or “order of at most n.”
- Algorithm B has a time requirement proportional to n^2 , we say that B is $O(n^2)$
- Algorithm C is O(1)

Time Complexity for Search algorithms

- Linear search:
  - Best : O(1)
  - Average: O(n)
  - Worst : O(n)
- Binary search:
  - Best : O(1)
  - Average: O(log n)
  - Worst : O(log n)

Data Structure Operations

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	O(1)	O(n)	O(n)	O(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Stack	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Singly-Linked List	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Binary Search Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)

Array Sorting Algorithms

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Selection Sort	O(n^2)	O(n^2)	O(n^2)	O(1)
Mergesort	O(n log(n))	O(n log(n))	O(n log(n))	O(n)
Bubble Sort	O(n)	O(n^2)	O(n^2)	O(1)
Insertion Sort	O(n)	O(n^2)	O(n^2)	O(1)