CPT108 Data Structures and Algorithms

Lecture 4

Analysis of Algorithms: Asymptotic Notations

Selection of Algorithms (last lecture)

Choice of Algorithms

- Criteria:
    - Time efficiency – how fast?
    - Space efficiency – memory requirement?
    - Development cost – reuse existing / proven ones?
- May also include:
    - Communication methods and bandwidth
        - synchronous or asynchronous communications?
        - any issues due to communication overhead?

Algorithm Analysis

What is *Algorithm*?

- A finite set of rigorous instructions used to solve a specific problem, or to achieve a goal
  - takes a set of values as input, and
  - produce a value, or set of values, as output
- When executing an instruction on computer
  - *Time* is needed!
    - ⇒ How to measure the performance of an algorithm?
      ... especially when the size of data is *large*!
    - ⇒ How to compare different algorithms?

➡ *Algorithm Analysis*

Algorithm Analysis (cont.)

Analyse algorithms can help us:

- To understand the *performance* of an algorithm
- To understand the *differences* between different algorithms
  - To eliminate bad algorithms earlier
  - To pinpoints the bottlenecks, which are worth coding carefully
- To *predict* how much resources that the algorithm is needed, e.g.:
  - Computational time
  - Memory
  - Development cost
  - Communication bandwidth, if required
  - etc.

Algorithm Analysis (cont.)

Different approaches

- Empirical
  - Run an implemented systemed on *real-world* data. Notion of benchmarks.
- Simulational
  - Run an implemented systemed on *simulated* data.
- Analytical
  - Use theoretic-model data with a theoretical model system. This is what we do in CPT108!

Outline

Performance analysis: Running time (cont.)

How the algorithm performs when the problem size increases?

E.g., for a problem size n,

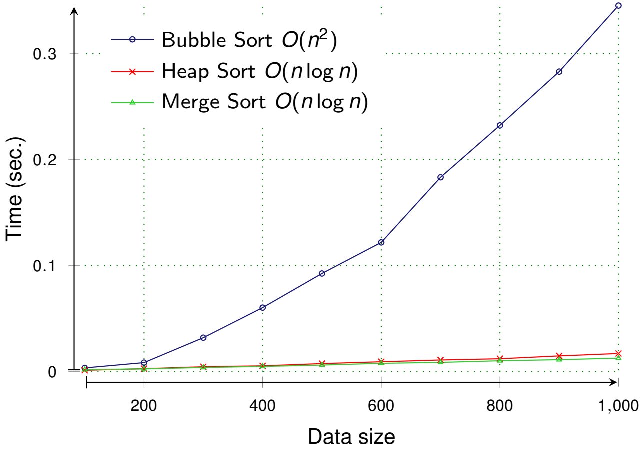
Execution time $T = 20n^2 + \boxed{100n + 360}$

The importance of the lower order part decrease as $n \uparrow$

$$\xrightarrow{n \to \infty} 20n^2 \in O(n^2)$$

size n	Execution time T	$20 * n^2$	
2	640	80	(13%)
4	1080	320	(30%)
8	2440	1280	(52%)
16	7080	5120	(72%)
32	24040	20480	(85%)
64	88680	81920	(92%)
128	340840	327680	(96%)

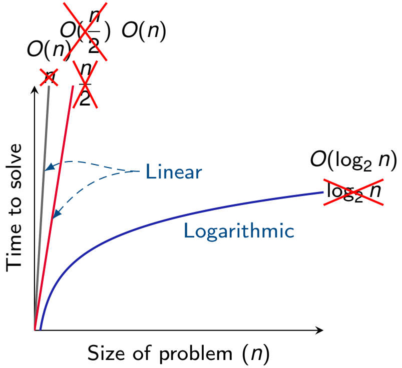
Performance of Different Sorting Algorithms



Performance analysis: Running time

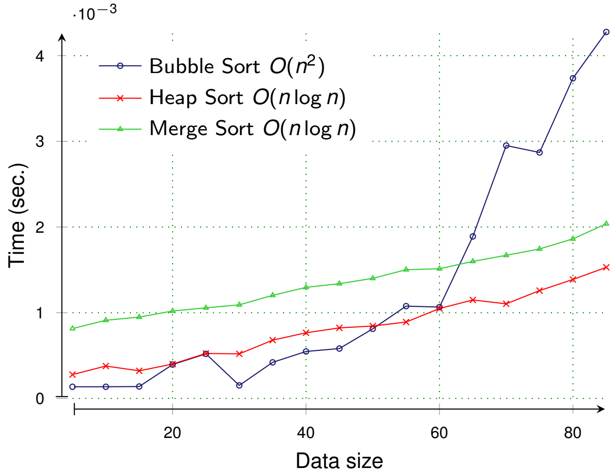
Running time

- What is the efficiency of these algorithms?
- How do we measure them?



In addition, computer scientist also throw away the coefficient of the higher order term since the effect of the coefficient is also not that significant as $n\uparrow$.

Performance of Different Sorting Algorithms



Asymptotic notations

- To establish a relative order among functions for *large* data size, n
- Commonly used notations in measuring computational time complexities include:
  - Upper bound $O(g(n))$
  - Lower bound $\Omega(g(n))$
  - Tight bound $\Theta(g(n))$

O (“Big-Oh”) Notation

- A asymptotic notation used to measure the time taken to run an algorithm as the size of the problem increase
  - i.e., it calculates the *worse-case* running time (upper bound) of an algorithm

O Notation (cont.)

Major categories of algorithms:

- $O(1)$ – Constant of steps or time
- $O(\log_2 n)$ (or $O(\lg n)$, $O(\log n)$) – Logarithmic
- $O(n)$ – Linear
- $O(n^k)$ – Polynomial
- $O(2^n)$ – Exponential – *intractable*

Ω *“Big-Omega” Notation*

- A asymptotic notation used to measure the time complexity of an algorithm in its best-case

Similarly, we have the following major categories:

- $\Omega(1)$
- $\Omega(\log_2 n)$ (or $\Omega(\lg n)$, $\Omega(\log n)$)
- $\Omega(n)$
- $\Omega(n^k)$
- $\Omega(2^n)$

Θ *"Big-Theta" Notation*

- Used to specify the asymptotic bounds (both upper and lower bounds) of an algorithm
  - The bound is the tightest possible
  - i.e., the case when the upper bound (O) and the lower bound (Ω) are equals

and again, we have the following major categories:

- $\Theta(1)$
- $\Theta(\log_2 n)$ (or $\Theta(\lg n)$, $\Theta(\log n)$)
- $\Theta(n)$
- $\Theta(n^k)$
- $\Theta(2^n)$

Asymptotic analysis: General Rules

- When $T(n)$ is a polynomial of degree k,
  i.e., $T(n) = c_k n^k + c_{k-1} n^{k-1} + \cdots + c_0$,
  therefore $T(k) = O(n^k)$
- If $T(n)$ is a logarithmic function,
  i.e., $T(n) = \log n$,
  therefore $T(n) = O(\log n)$

Asymptotic analysis: General Rules (cont.)

- Loops (`for` loop, `while` loops)
  - At most the running time of the statements inside the loop $\times$ the number of iterations
  - $O(n)$

```
for (i=0; i<n; i++) {
  // do something with O(1)
}
```

- Nested loops
  - Running time of the statement equals to the product fo the sizes of all loops
  - $O(mn)$, or $O(n^2)$ if $m = n$

```
for (i=0; i<n; i++) {
  for (j=0; j<m; j++) {
    // do something with O(1)
  }
}
```

Asymptotic analysis: General Rules (cont.)

- Consecutive/ sequential executions
  - When two separate code are executed one after the other, just add them together
  - $O(n + m)$

```
for (i=0; i<n; i++) {
  // do something with O(1)
}
for (j=0; j<m; j++) {
  // do something else with O(1)
}
```

- Conditional: `if S1 else S2`
  - never more than the running time of the *test* plus the *larger* running time of `S1` and `S2`
  - $O(1) + O($`S1`$)$ or $O(1) + O($`S2`$)$

Factors affecting the running time

- Factors affecting the running time
  - Computer
    - Speed of the processor, application currently running on the machine, (available) memory size (a.k.a. free memory), etc.
  - Compiler
  - Algorithm used
  - Implementation, i.e., programming skills
  - Input to the algorithm, such as the content and the data size,
    - E.g., sorting and matrices multiplication
- Machine model assumed
  - Instructions are executed one after another, with *no* concurrent operations
    - i.e., *NO* parallel processing!

Some Notes

Bounds are for algorithms, rather than problems

- A problem can be solved with several algorithms, some are more efficient than others

Bounds are for the algorithms, rather than programs

- Programs are just implementations of an algorithm, and almost always the details of the program do not affect the bounds

An Example

```
int sum(int n) {
  int sum;
  sum = 0;
  for (int i = 0; i <= n; i++)
    sum += i*i*i;
  return sum;
}
```

1 initiation

$n+1$ test

n increment

```
sum = sum + i * i * i;
```

	$T(n)$	
1	0	Variable declaration
2	1	Variable assignment
3	$2n+2$	$(1)+(n+1)+(n)$ operations
4	$4n$	4 operations executed n times
5	1	Result return

$$\therefore \text{ Total execution time} = (1)+(2n+2)+(4n)+(1)$$
$$= 6n+4 \quad \Rightarrow O(N)$$

Selection Problem: An Exercise

Given a set of N numbers, determine the k^{th} largest value, where $k \leq N$.

Algorithm 1

- Read N numbers into an array
- Sort the array in decreasing order by some algorithm
  (you can assume it as $O(n \log n)$)
- Return the element in position k

Selection Problem: An Exercise (cont.)

Algorithm 2

- Read the k elements in the array and sort them in decreasing order
- Each remaining element is read one-by-one
  - If smaller than the k^{th} element, then it is ignored
  - Otherwise, it is placed in its correct sport in the array, bumping one element out of the array
- The element in the k^{th} position is returned as the answer

Selection Problem: An Exercise (cont.)

- Which algorithm is better when
  - $n = 100$ and $k = 100$?
  - $n = 100$ and $k = 1$?
- What happens when
  - $n = 1{,}000{,}000$ and $k = 500{,}000$?

Reading

- Chapter 3, Cormen (2022)