Database Management Systems

Module 36: Algorithms and Data Structures/1: Algorithms and Complexity Analysis

Partha Pratim Das

Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

ppd@cse.iitkgp.ac.in

Week Recap

- Had a glimpse of Application Programs across various sectors
- Understood the architectures for an application and their classification and evolution
- Glimpsed at architecture for a few sample applications
- Familiarized with the Fundamentals notions and technologies of Web
- Learnt about Scripting and the notions of Servlets
- Learnt to use SQL from a programming language
- Learnt to build Python Web Applications with PostgreSQL using psycopg2 and Flask
- Understood the steps in the Rapid Application Development Process
- Exposed to the issues in Application Performance and Security
- Learnt the distinctive features of Mobile Apps

Module Objectives

- Define Algorithms and its difference with Programs
- Analyze algorithms for performance of time, space, power, etc.
- Introduce Asymptotic notation for representation of complexity
- Consider complexity of common algorithms

Module Outline

- Algorithms and Programs
- Analysis of Algorithms
- Complexity Chart

Algorithms and Programs

- **Algorithm**
  - An algorithm is a *finite sequence* of *well-defined*, computer-implementable (optional) instructions, typically to solve a class of specific problems or to perform a computation.
  - Algorithms are always *unambiguous* and are used as specifications for performing calculations, data processing, automated reasoning, and other tasks.
  - An algorithm must *terminate*
- **Program**
  - A computer program is a collection of instructions that can be executed by a computer to perform a specific task
  - A computer program is usually written by a computer programmer in a programming language.
  - A programs *implements* an algorithm
  - A program *may* or *may not* terminate. For example, an OS

Analysis of Algorithms

Analysis of Algorithms

- **Why?**
  - Set the motivation for algorithm analysis:
  - *Why analyze?*
- **What?**
  - Identify what all need to be analyzed:
  - *What to analyze?*
- **How?**
  - Learn the techniques for analysis:
  - *How to analyze?*
- **Where?**
  - Understand the scenarios for application:
  - *Where to analyze?*
- When?
  - Realize your position for seeking the analysis:
  - *When to analyze?*

Why analyze?

Practical reasons:

- Resources are scarce
- Greed to do more with less
- Avoid performance bugs

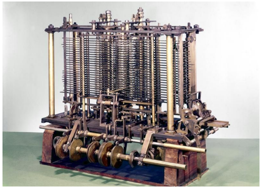
Core Issues:

- **Predict performance**
  - *How much time does binary search take?*
- **Compare algorithms**
  - *How quick is Quicksort?*
- **Provide guarantees**
  - *Size notwithstanding, Red-Black tree inserts in* $O(\log n)$
- **Understand theoretical basis**
  - *Sorting by comparison cannot do better than* $\Omega(n \log n)$

What to analyze?

Core Issue: Cannot control what we cannot measure

- **Time**
  - Story starts here with *Analytical Engine*



  - Most common analysis factor
  - Representative of various related analysis factors like Power, Bandwidth, Processors
  - Supported by Complexity Classes
- **Space**
  - Widely explored
  - Important for hand-held devices
  - Supported by Complexity Classes

What to analyze?

- Sum of Natural Numbers

```
int sum(int n) {
    int s = 0;
    for(; n > 0; --n)
        s = s + n;
    return s;
}
```

- Time $T(n) = n$ (additions)
- Space $S(n) = 2$ (n, s)

What to analyze?

- Find a character in a string

```
int find(char *str, char c) {
    for(int i = 0; i < strlen(str); ++i)
        if (str[i] == c)
            return i;
    return 0;
}
```

$n = \texttt{strlen(str)}$

- Time $T(n) = n$ (compare) $+ n * T(\texttt{strlen(str)}) \approx n + n^2 \approx n^2$
- Space $S(n) = 3$ (str, c, i)

What to analyze?

- Minimum of a Sequence of Numbers

```
int min(int a[], int n) {
    for(int i = 0; i < n; ++i)
        cin >> a[i];

    int t = a[--n];
    for(; n > 0; --n)
        if (t < a[--n])
            t = a[n];
    return t;
}
```

- Time $T(n) = n - 1$ (comparison of value)
- Space $S(n) = n + 3$ (`a[]`'s, `n`, `i`, `t`)

What to analyze?

- Minimum of a Sequence of Numbers

```
int min(int n) {
    int x;
    cin >> x;
    int t = x;
    for(; n > 1; --n) {
        cin >> x;
        if (t < x)
            t = x;
    }
    return t;
}
```

- Time $T(n) = n - 1$ (comparison of value)
- Space $S(n) = 3$ (n, x, t)

How to analyze?

- **Counting Models**
- **Asymptotic Analysis**
- Generating Functions
- Master Theorem

How to analyze?: Counting Models

Counting Models

- Core Idea: Total running time = Sum of cost $\times$ frequency for all operations
  - Need to analyze program to determine set of operations
  - **Cost** depends on machine, compiler
  - **Frequency** depends on algorithm, input data
- Machine Model: Random Access Machine (RAM) Computing Model
  - Input data & size
  - Operations
  - Intermediate Stages
  - Output data & size

How to analyze?: Counting Models

- Factorial (Recursive)

```
int fact(int n) {
    if (0 != n) return n*fact(n-1);
    return 1;
}
```

  - Time $T(n) = n - 1$ (multiplication)
  - Space $S(n) = n + 1$ (n's in recursive calls)

- Factorial (Iterative)

```
int fact(int n) {
    int t = 1;
    for(; n > 0; --n)
        t = t * n;
    return t;
}
```

  - Time $T(n) = n$ (multiplication)
  - Space $S(n) = 2$ (n, t)

How to analyze?: Asymptotic Analysis

Asymptotic Analysis

- Core Idea: Cannot compare actual times; hence compare *Growth* or how time increases with input size
  - Function Approximation (tilde (˜) notation)
  - Common Growth Functions
  - Big-Oh ($O(.)$), Big-Omega ($\Omega(.)$), and Big-Theta ($\Theta(.)$) Notations
  - Solve recurrence with Growth Functions

How to analyze?: Asymptotic Analysis


```
int count = 0;
for (int i = 0; i < N; i++)
    for (int j = i+1; j < N; j++)
        if (a[i] + a[j] == 0)
            count++;
```

Function Approximation (tilde (˜) notation)

Operation	Frequency	Approximation
variable declaration	$N+2$	$\sim N$
assignment statement	$N+2$	$\sim N$
less than compare	$\frac{1}{2}(N+1)(N+2)$	$\sim \frac{1}{2}N^2$
equal to compare	$\frac{1}{2}N(N-1)$	$\sim \frac{1}{2}N^2$
array access	$N(N-1)$	$\sim N^2$
increment	$\frac{1}{2}N(N-1)$ to $N(N-1)$	$\sim \frac{1}{2}N^2$ to $\sim N^2$

- Estimate running time (or memory) as a function of input size N. Ignore lower order terms
  - when N is large, terms are negligible
  - when N is small, we don't care

$f(n) \sim g(n)$ **means**

$$\lim_{N\to\infty} \frac{f(n)}{g(n)} = 1$$

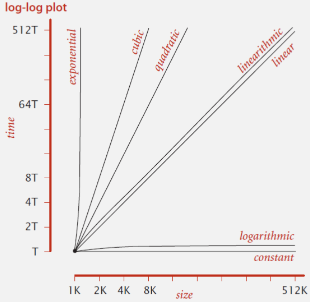
How to analyze?: Asymptotic Analysis

Common order-of-growth classifications

Good news. The set of functions

$1,\ \log N,\ N,\ N\log N,\ N^2,\ N^3$, and 2^N

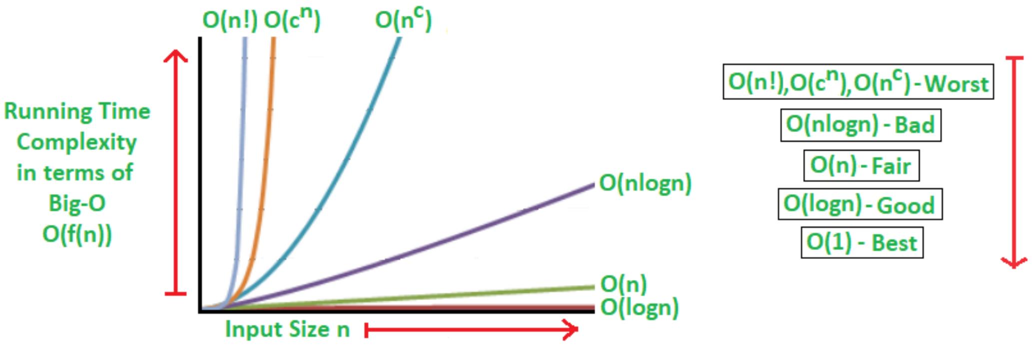
suffices to describe the order of growth of most common algorithms.



Typical orders of growth

Courtesy: Algorithms by Robert Sedgewick & Kevin Wayne

How to analyze?: Asymptotic Analysis



How to analyze?: Asymptotic Analysis

Common order-of-growth classifications

order of growth	name	typical code framework	description	example	$T(2N) / T(N)$
1	**constant**	`a = b + c;`	statement	add two numbers	1
$\log N$	**logarithmic**	`while (N > 1) { N = N / 2; ... }`	divide in half	binary search	~ 1
N	**linear**	`for (int i = 0; i < N; i++) { ... }`	loop	find the maximum	2
$N \log N$	**linearithmic**	[see mergesort lecture]	divide and conquer	mergesort	~ 2
N^2	**quadratic**	`for (int i = 0; i < N; i++) for (int j = 0; j < N; j++) { ... }`	double loop	check all pairs	4
N^3	**cubic**	`for (int i = 0; i < N; i++) for (int j = 0; j < N; j++) for (int k = 0; k < N; k++) { ... }`	triple loop	check all triples	8
2^N	**exponential**	[see combinatorial search lecture]	exhaustive search	check all subsets	$T(N)$

Courtesy: Algorithms by Robert Sedgewick & Kevin Wayne

Asymptotic notation

For a given function $g(n)$, we denote by $O(g(n))$ the set of functions:

$$O(g(n)) = \{f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that}$$
$$0 \leq f(n) \leq cg(n), \text{for all } n > n_0\}$$

- We use O-notation to give an upper bound on a function, to within a constant factor.
- When we say that the running time of A is $O(n^2)$, we mean that there is a function $f(n)$ that is $O(n^2)$ such that for any value of n, no matter what particular input of size n is chosen, the running time on that input is bounded from above by the value $f(n)$.
- Equivalently, we mean that the worst-case running time is $O(n^2)$.

Where to analyze?

Algorithmic Situation

- Core Idea: Identify data configurations or scenarios for analysis
  - Best Case
    - Minimum running time on an input
  - **Worst Case**
    - Running time guarantee for any input of size n
  - **Average Case**
    - Expected running time for a random input of size n
  - Probabilistic Case
    - Expected running time of a randomized algorithm
  - Amortized Case
    - Worst case running time for any sequence of n operations

Complexity Chart

Big-O Algorithm Complexity Cheat Sheet

Common Data Structure Operations

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	Θ(1)	Θ(n)	Θ(n)	Θ(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Stack	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Queue	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Singly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Skip List	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
Hash Table	N/A	Θ(1)	Θ(1)	Θ(1)	N/A	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
Cartesian Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(n)	O(n)	O(n)	O(n)
B-Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Red-Black Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Splay Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(log(n))	O(log(n))	O(log(n))	O(n)
AVL Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
KD Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)

Source: Know Thy Complexities! (06-Apr-2021)

Big-O Algorithm Complexity Cheat Sheet

Array Sorting Algorithms

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(log(n))
Mergesort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(n)
Timsort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)
Heapsort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(1)
Bubble Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Insertion Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Selection Sort	Ω(n^2)	Θ(n^2)	O(n^2)	O(1)
Tree Sort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(n)
Shell Sort	Ω(n log(n))	Θ(n(log(n))^2)	O(n(log(n))^2)	O(1)
Bucket Sort	Ω(n+k)	Θ(n+k)	O(n^2)	O(n)
Radix Sort	Ω(nk)	Θ(nk)	O(nk)	O(n+k)
Counting Sort	Ω(n+k)	Θ(n+k)	O(n+k)	O(k)
Cubesort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)

Source: Know Thy Complexities! (06-Apr-2021)

Module Summary

- Need for analyzing the running-time and space requirements of a program
- Asymptotic growth rate or order of the complexity of different algorithms
- Worst-case, average-case and best-case analysis

Slides used in this presentation are borrowed from http://db-book.com/ with kind permission of the authors.
Edited and new slides are marked with "PPD".