Algorithm Analysis

Fall 2020

School of Software Engineering

South China University of Technology

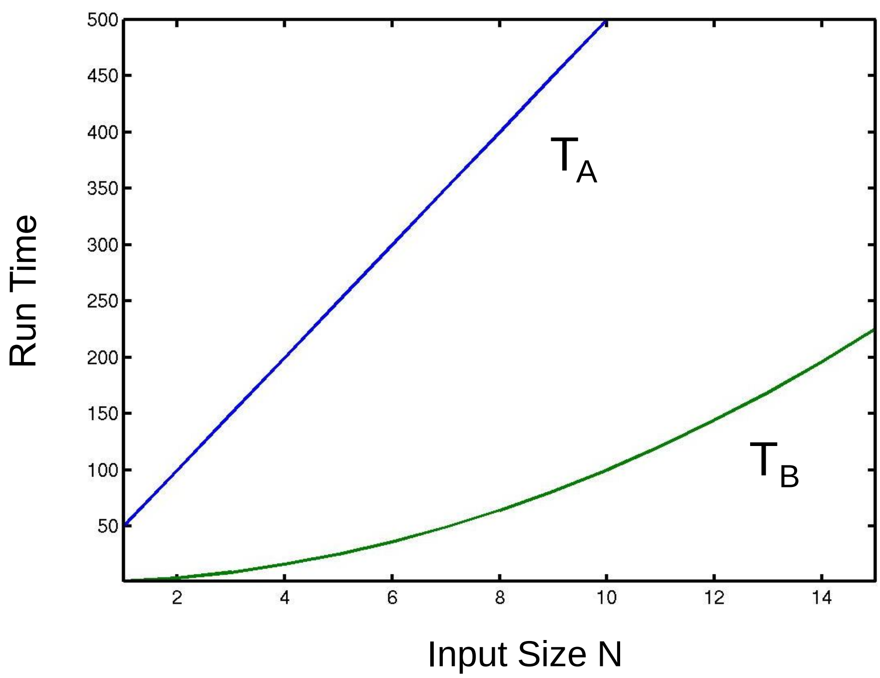
Content

- Simple Model of Computation
- Definitions of Big-Oh and Other Notations
- Common Functions and Growth Rates
- Worst Case vs. Average Case Analysis
- How to Perform Analyses
- Comparative Examples

Algorithm Analysis

- 1. Why do we analyze algorithms?

  - Suppose you are given two algorithms A and B for solving a problem.
  - The running times $T_A(N)$ and $T_B(N)$ of A and B as a function of input size N are given

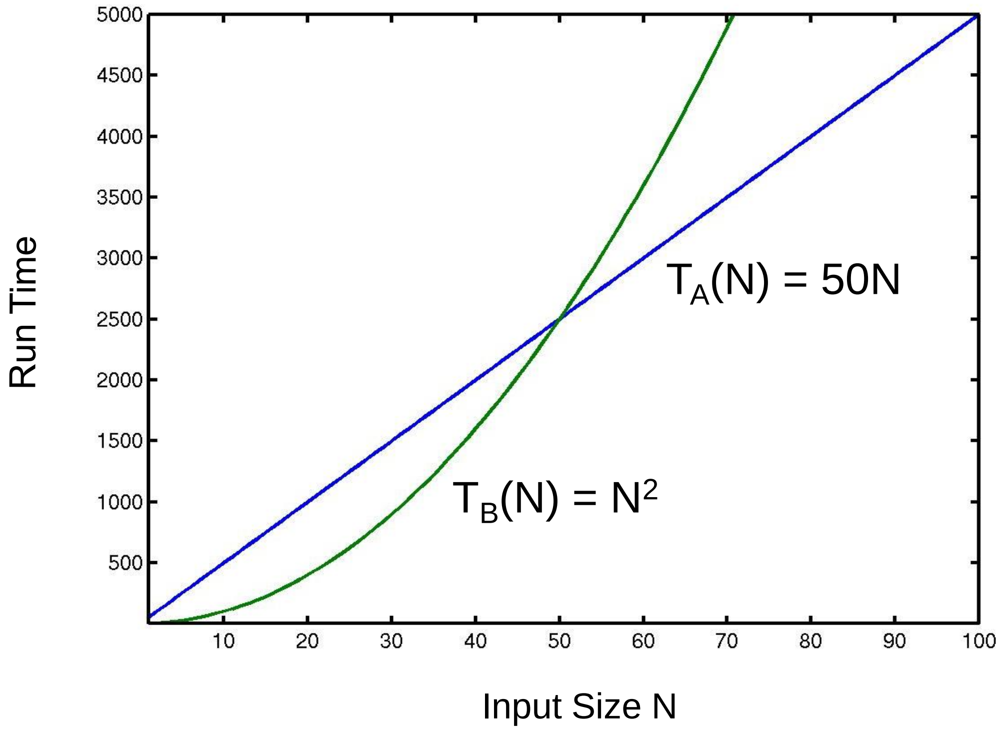


Which is better?

Algorithm Analysis

- 1. Why do we analyze algorithms?

  - For large N, the running time of A and B is:



Now which algorithm would you choose?

Algorithm Analysis

- 1. Why do we analyze algorithms?

- 2. How do we measure the efficiency of an algorithm?
  - A. Time it on my computer.
  - B. Compare its time to that of another algorithm that has already been analyzed.
  - C. Count how many instructions it will execute for an arbitrary input data set.

Algorithm Analysis

- Suppose there are n inputs.
- We'd like to find a time function **T(n)** that shows how the execution time depends on **n**.

$\boxed{T(n) = 3n + 4}$ $\boxed{T(n) = e^n}$ $\boxed{T(n) = 2}$

Model of Computation

- Simple Model of Computation
  - instructions are executed sequentially
  - has the standard repertoire of simple instructions
    - it takes exactly one time unit to do addition, multiplication, comparison, and assignment
  - assume that the model has fixed-size integers(ex. 32-bits) and no fancy operations
  - assume infinite memory

What to Analyze

- What to Analyze
  - Running **time** required
  - Memory or disk **space** required to run the program and store the data structure
- Main factors
  - the algorithm used
  - the **input** to the algorithm
  - Not include the programming language, compiler,…

Analyze the algorithms rather than the programs

Mathematical Definitions

- **Big-Oh**

$T(N) = O(f(N))$ if there are positive constants c and n_0 such that $T(N) \leq cf(N)$ when $N \geq n_0$.

- **Big-Omega**

$T(N) = \Omega(g(N))$ if there are positive constants c and n0 such that $T(N) \geq cg(N)$ when $N \geq n0$.

- **Big-Theta**

$T(N) = \Theta(h(N))$ if and only if $T(N) = O(h(N))$ and $T(N) = \Omega(h(N))$.

- **Little-oh**

$T(N) = o(p(N))$ if, for all positive constants c, there exists an n_0 such that $T(N) < cp(N)$ when $N > n_0$.

“Big-Oh”

$T(N) = O(f(N))$ if there are positive constants c and n_0 such that $T(N) \le cf(N)$ when $N \ge n_0$.

We say “T(N) has order f(N).”

“f(N) is an **upper bound** on T(N)”

We try to simplify T(N) into one or more **common functions**.

Ex. 1 $T(N) = 3N + 4$

T(N) is linear. Intuitively, f(N) should be N.

More formally,

$T(N) = 3N + 4 \le 3N + 4N, \; N \ge 1$

$T(N) \le 7N, \; N \ge 1$

So T(N) is of order N.

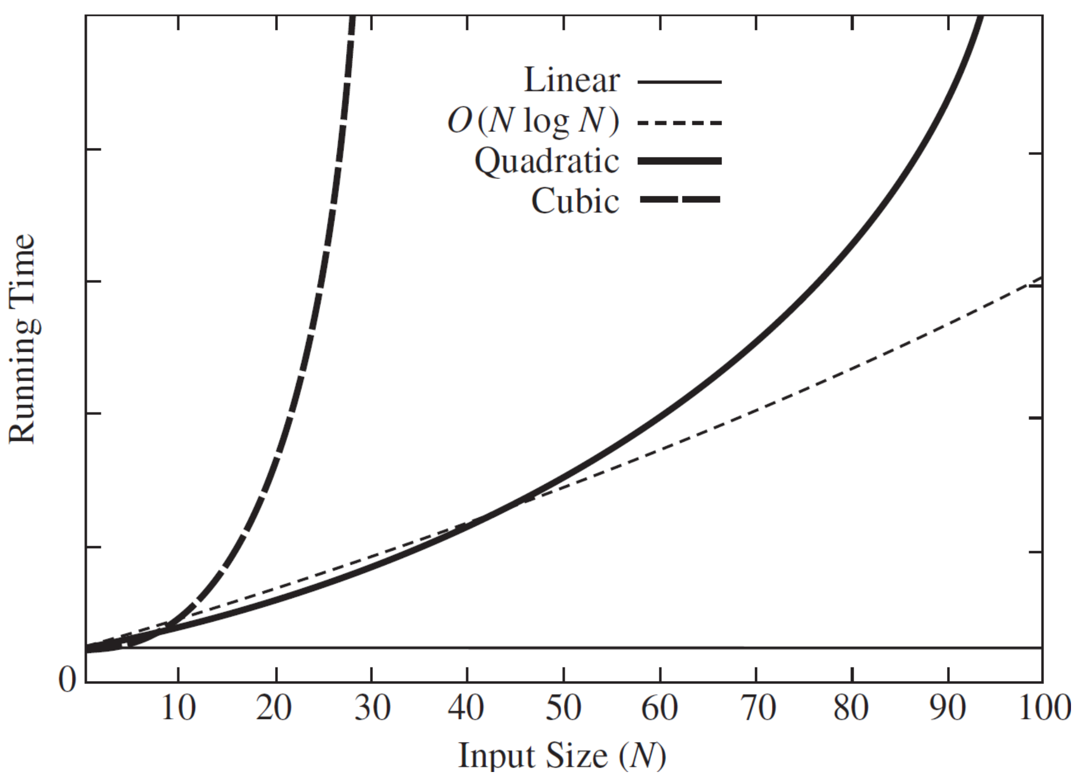
"Common Functions to Use"

$O(1)$	constant
$O(\log n)$	log base 2
$O(n)$	linear
$O(n \log n)$	
$O(n^2)$	quadratic
$O(n^3)$	cubic
$O(2^n)$ or $O(e^n)$	exponential
$O(n+m)$	
$O(nm)$	
$O(n^m)$	

Growth Rates

The idea of the definitions is to establish a relative order among functions.

The **growth rate** is the rate at which the cost of the algorithm grows as the input size grows



“simplifying rules”

Rule 1: If $T1(N) = O(f(N))$ and $T2(N) = O(g(N))$, then
(a) $T1(N) + T2(N) = O(f(N) + g(N))$
(b) $T1(N) * T2(N) = O(f(N) * g(N))$.

Suppose we get $T_1(N) = O(f(N))$ and $T_2(N)=O(g(N))$,

$f(N) = 4N^2 + 6$, $g(N) = 3N$, $T(N) = T_1(N)+T_2(N) =?$

$$
\begin{aligned}
T(N) = T_1(N)+T_2(N) &= O(f(N) + g(N)) \\
&= 4N^2 + 3N + 6 \\
&= O(N^2)
\end{aligned}
$$

intuitively $O(\max(f(N), g(N)))$

“simplifying rules”

Rule 2: If $T(N)$ is a polynomial of degree k, then
$T(N) = \Theta(N^k)$.

Rule 3: $\log^k N = O(N)$ for any constant k.

Rule 4: If $T(n) = O(g(n))$ and $g(n) = O(h(n))$, then
$T(n) = O(h(n))$. – transitive*

Rule 5: If $T(n) = O(kg(n))$ for any constant $k > 0$,
then $T(n) = O(g(n))$. – ignore the constant*.

“Common Functions to Use”

Suppose we get $T(N) = 4N^2 + 3N + 6$.

Is $T(N) = O(N^2)$?

Is $T(N) = O(N^3)$?

Generally, we look for the smallest f(N) that bounds T(N).

We want a common function that is a least upper bound.

If $T(N) = c_k N^k + c_{k-1} N^{k-1} + \dots + c_0$.

$T(N) = O(N^k)$.

N^k is the dominant term.

How to Analyze

- How to analyze
  - Asymptotic algorithm analysis
    - Measures the efficiency of an algorithm, as the input size becomes large – growth rate.
    - best-case, often of little interest
    - average-case, often reflects typical behavior
    - worst-case, represents a guarantee for performance on any possible input.

Need to know enough about the input data distribution to do average-case analysis

How to Analyze

Given an array containing **n** integers, suppose the sequential search algorithm is adopted

Q1: The cost of finding the largest value

Always *c**n

Q2: The cost for finding a particular value K

May be different for different inputs

Best case:

if the first integer is K – examine 1 value

Worst case:

if only the last integer is K – examine n values

Average case:

If the sequential search is performed on different inputs for many times – examine n/2 values on average

How to Analyze

Step 1. Counting	T(N)
Step 2. Simplifying	O(f(N))

```
//Loop
int sumit( int v[ ],  int num)  {
  sum = 0;                          c1
  for (i = 0;  i  <  num;  i++)     c2*num
    sum = sum + v[i];               c3*num
  return sum                        c4
}
```

T(num) = (c2 + c3)* num + (c1 + c4)

= k1 * num + k2

= O(num) throw away leading constants

We say T(n) of the algorithm is in O(n)

How to Analyze

```
// Nested Loops

int sum(int n) {
  int sum = 0;                c1
  for (i=1; i<=n; i++)        c2*n
    for (j=1; j<=n; j++)      c3*n
      sum++;                  c4*n^2
}
```

$$T(n) = c1 + (c2 + c3)*n + c4*n^2$$
$$= O(n^2)$$ throw away low order terms

We say T(n) of the algorithm is in $O(n^2)$

How to Analyze

```
// compare the cost of the two loop codes
sum = 0;
for(k=1; k<=n; k*=2)  //do log(n) times
  for(j=1; j<=n; j++)   //do n times
    sum++;
```

In this double loop, the cost is

$$T(n) = \Theta\left(\sum_{i=1}^{\log n} n\right) = \Theta(n \log n)$$

How to Analyze

```
// compare the cost of the two loop codes
sum = 0;
for(k=1; k<=n; k*=2)    //do log(n) times
  for(j=1; j<=k; j++)     //do k times
    sum++;
```

In this double loop, the cost is

$$\Theta\left(\sum_{i=1}^{\log n} 2^i\right) = \Theta(2^{\log n}) = \Theta(n)$$

$$\sum_{0}^{\log n} 2^i = 2^{\log n+1} - 1 = 2n - 1$$

How to Analyze

```
// assume array A contains n values,
//  random takes constant time c1 and
// sort takes cnlogn steps
for (i=0; i<n; i++) {
  for (j=0; j<n; j++)
    A[j] = random(n);
  sort(A, n);
}
```

Determine Θ in average case

$$\Theta(n(c_1n+cnlogn))= \Theta(n^2logn)$$

How to Analyze

```
// assume A[] contains a random permutation
// of the values from 0 to n-1
sum=0;
for(i=0; i<n; i++)
  for(j=0; A[j] !=i; j++)
    sum++;
```

7	**5**	**2**	**1**	**0**	**4**	**6**	**3**

j=0 j=1 j=4

Determine Θ in average case

$$\Theta(n*n/2)=\Theta(n^2)$$

How to Analyze

```
// A simple assignment to
// an integer variable
a = b;
```

$T(N) = \Theta(1)$

How to Analyze

- While loop – similar to for loop
- If-then-else statement – the greater of the costs for the then and else clauses.
- Switch statement – the most expensive branch
- Subroutine call – add the cost of subroutine
- Recursive subroutine – express the cost by a recurrence relation and then find the closed-form solution.

How to Analyze

```
// Here || is the string concatenation operator
string t (int n){
  if (n == 1) return '(1) ';
  else return '(' || n || t(n - 1) || ') '
}
```

$T(n) = T(n-1) + c$, for $n>1$, with $T(1) = c_1$

That is ,
$T(n) = c*(n-1) + c_1 = \Theta(n)$

How to Analyze

// Multiple Parameters
// A picture with P pixels; each pixel take one of C color
// values; Sort the colors w.r.t. the number of pixels with
// the color (Θ(C log C))

```
for (i=0; i<C; i++)  // Initialize count
  count[i] = 0;
for (i=0; i<P; i++)  // Look at all pixels
  count[value(i)]++; // Increment count
sort(count);         // Sort pixel counts
```

The cost is $\Theta(C) + \Theta(P) + \Theta(C \log C) = \Theta(P + C \log C)$

Can we drop $\Theta(P)$ or $\Theta(C \log C)$?

Notes

Best, Worst, Average-Cases
vs.
Upper, Lower Bounds

- U/L bounds refer to the algorithm's growth rate.
- B/W/A cases refer to a certain type of inputs which cause the shortest/average/longest running time among all the inputs in study.
- U/L bounds can be used to describe the running time of an algorithm in its [best, worst, average] case

Notes

- For an algorithm with $T(n) = c_1n^2+c_2n$ in the average case (c_1, $c_2>0$) ,
  - $c_1n^2+c_2n <= (c_1+c_2)n^2$ for all $n>1$, $T(n) <= cn^2$ for $c=c_1+c_2$ and $n_0=1$.
  - $c_1n^2+c_2n >= c_1n^2$ for all $n>1$, $T(n) >= c_1n^2$ for $n_0=1$.
  - $T(n)$ is in $\Theta(n^2)$ in the average case
- Reading the value from the first position in an array takes constant time regardless of the size of the array.
  - $T(n) = c$ for the (best, worst, and average) cases.
  - Traditionally, we say that the algorithm is in $O(1)$.

Notes

Analyze an algorithm

vs.

Analyze a problem

- The upper bound for a problem cannot be worse than the upper bound for the best known algorithm

- If a problem is in $\Omega(f(n))$, every algorithm that solves the problem is in $\Omega(f(n))$, even algorithms that we have not thought of.

Notes

- Other kinds of analysis
  - Amortized –worst case averaged over a sequential operations
  - Textbook Chapter 11

  - Common case – 80%-20%

Homework

- Read textbook Ch2. Please pay more attention on section 2.4.3 “the Maximum Subsequence Sum Problem”
- Exercise 2.7 3) and 4)
- Exercise 2.20
- Deadline: to be confirmed.