Algorithms and Running Time

Algorithm Definition

A finite set of statements that guarantees an optimal solution in finite interval of time

Algorithm

- Finite sequence of instructions.
- Each instruction having a clear meaning.

- Each instruction requiring finite amount of effort.

- Each instruction requiring finite time to complete.

Algorithm

- Finite sequence of instructions.

An input should not take the program in an infinite loop

- Each instruction having a clear meaning. Very subjective. What is clear to me, may not be clear to you.

- Each instruction requiring finite amount of effort. Very subjective. Finite on a super computer or a P4?

- Each instruction requiring finite time to complete. Very subjective. 1 min, 1 hr, 1 year or a lifetime?

Good Algorithms?

- Run in less time
- Consume less memory

But computational resources (time complexity) is usually more important

Measuring Efficiency

- The efficiency of an algorithm is a measure of the amount of resources consumed in solving a problem of size n.

- The resource we are most interested in is time
- We can use the same techniques to analyze the consumption of other resources, such as memory space.

- It would seem that the most obvious way to measure the efficiency of an algorithm is to run it and measure how much processor time is needed
- Is it correct

Factors

- Hardware
- Operating System
- Compiler
- Size of input

- Nature of Input
- Algorithm

Which should be improved?

Running Time of an Algorithm

- Depends upon
    - Input Size
    - Nature of Input
- Generally time grows with size of input, so running time of an algorithm is usually measured as function of input size.
- Running time is measured in terms of number of steps/primitive

operations performed

- Independent from machine, OS

Finding running time of an Algorithm / Analyzing an Algorithm

- Running time is measured by number of steps/primitive operations performed
- Steps means elementary operation like
- ,+, *,<, =, A[i] etc

- We will measure number of steps taken in term of size of input

Simple Example (1)

```
// Input: int A[N], array of N integers
// Output: Sum of all numbers in array A

int Sum(int A[], int N)
{
1. int s=0;
```

2. for (int i=0; i< N; i++)

3. s = s + A[i];

4. return s;

}

How should we analyse this?

Simple Example (2)

```
// Input: int A[N], array of N integers
// Output: Sum of all numbers in array
A

int Sum(int A[], int N){
 int [redacted];
1
                                        3 4  s = s + [redacted];
 for (int [redacted]; [redacted]; [redacted] 2
```

5 6 7

[redacted]

8

of for loop, N iteration

Total: 5N + 3

The *complexity function* of the algorithm is : $f(N) = 5N + 3$

1,2,8: Once

3,4,5,6,7: Once per each iteration

}

Simple Example (3)
Growth of 5n+3

Estimated running time for different values of N:

N = 10 => 53 steps

N = 100 => 503 steps

N = 1,000 => 5003 steps

N = 1,000,000 => 5,000,003 steps

As N grows, the number of steps grow in *linear* proportion to N for this function *“Sum”*

What Dominates in Previous Example?

- What about the +3 and 5 in 5N+3?
  - – As N gets large, the +3 becomes insignificant
  - – 5 is inaccurate, as different operations require varying amounts of time and also does not have any significant importance
- Asymptotic Complexity: As N gets large, concentrate on the highest order term:
  - Drop lower order terms such as +3

- Drop the constant coefficient of the highest order term i.e. N
- The 5N+3 time bound is said to "grow asymptotically" like N

Running Time Calculations

- Simple for loop

```
int Sum (int N) {
/* 1 */ int sum = 0;
/* 2 */ for (int i =1; i <= N; i++)
/* 3 */ sum = sum + i * i * i;
/* 4 */ return sum ; }
```

Q: What is the running time?

Line 1 & 4 → 2 units of time → 2 Line 2 → 1 unit (initialize) + $(N+1)$ tests + N Increments → $2N + 2$ Line 3 → 4 units (1 add, 2 muls., 1 assign) * N executions → $4N$ **Total** → $6N + 4$

A: O(N)

Asymptotic Complexity

Three types of notations are used to asymptotically bound and algorithm

- Big-O, Omega, and Theta are formal notational methods for stating the growth of resource needs (efficiency and storage) of an algorithm. • In simple words it describe how much resources(CPU cycles) are needed to execute said algorithm.

Asymptotic Notation

- **Θ, *O*, Ω, *o*, ω**
- Defined for functions over the natural numbers.
  - **Ex:** $f(n) = \Theta(n^2)$.
  - Describes how $f(n)$ grows in comparison to n^2.

- Define a ***set*** of functions; in practice used to compare two function sizes.
- The notations describe different rate-of-growth relations between the defining function and the defined set of functions.

Comparing Functions: Asymptotic Notation

- Big Oh Notation:
  - Upper bound
- Omega Notation:

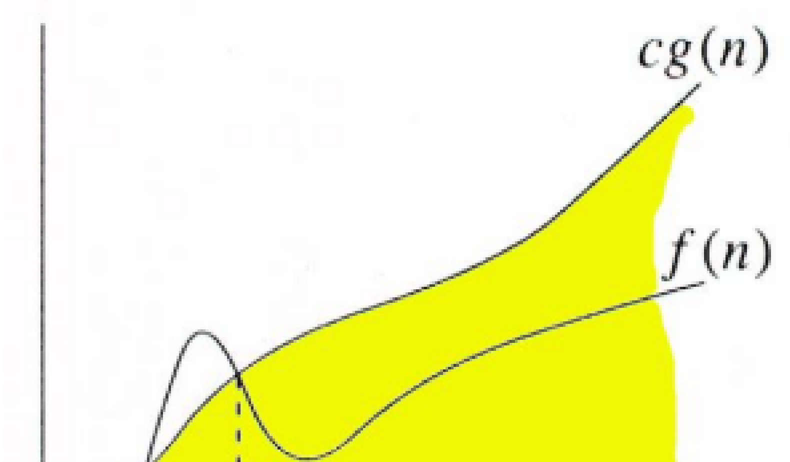
- Lower bound
- Theta Notation:
  - Tighter bound

O-notation

For function $g(n)$, we define $O(g(n))$, big-O of n, as the set:

$O(g(n)) = \{f(n)$:

$\exists$ positive constants c and n_0, such that $\forall n \geq n_0$,



we have $0 \le f(n) \le cg(n)$ }

Intuitively: Set of all functions whose *rate of growth* is the same as or lower than that of $g(n)$.

$g(n)$ is an *asymptotic upper bound* for $f(n)$.

$f(n) = \Theta(g(n)) \Rightarrow f(n) = O(g(n))$.

$\Theta(g(n)) \subset O(g(n))$.

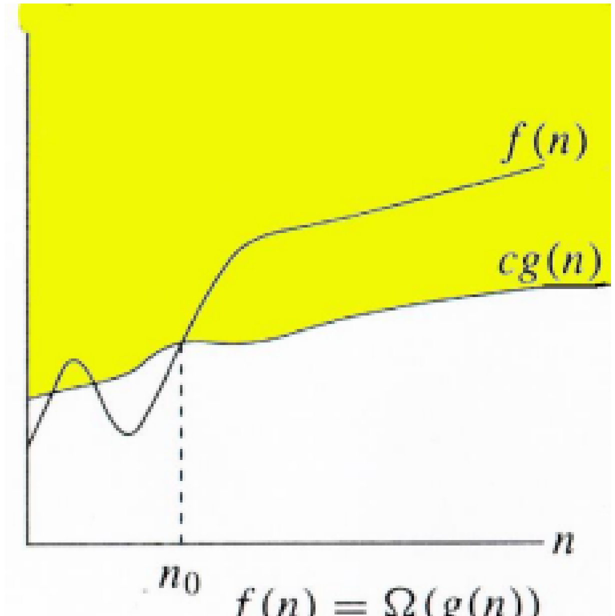
Ω -notation

For function $g(n)$, we define $\Omega(g(n))$, big-Omega of n, as the set:

$\Omega(g(n)) = \{f(n)$:

$\exists$ positive constants c and n_0,

such that $\forall n \ge n_0$,



we have $0 \le cg(n) \le f(n)\}$

Intuitively: Set of all functions whose *rate of growth* is the same as or higher than that of $g(n)$.

$g(n)$ is an *asymptotic lower bound* for $f(n)$.

$f(n) = \Theta(g(n)) \Rightarrow f(n) = \Omega(g(n))$.

$\Theta(g(n)) \subset \Omega(g(n))$.

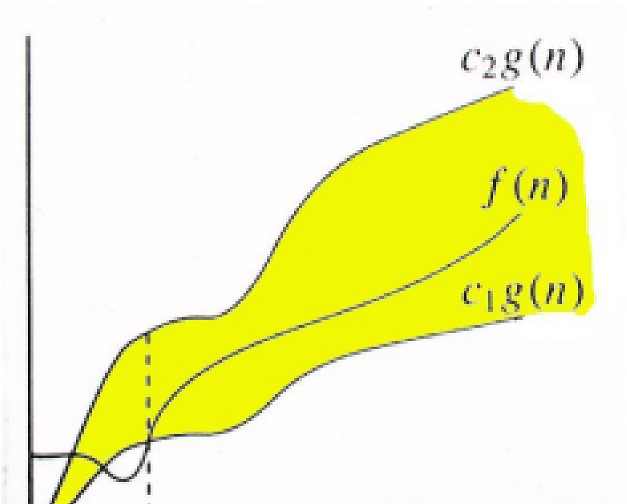
Θ-notation

For function $g(n)$, we define $\Theta(g(n))$, big-Theta of n, as the set:

$\Theta(g(n)) = \{f(n)$:

$\exists$ positive constants c_1, c_2, and n_0,

such that $\forall n \ge n_0$,



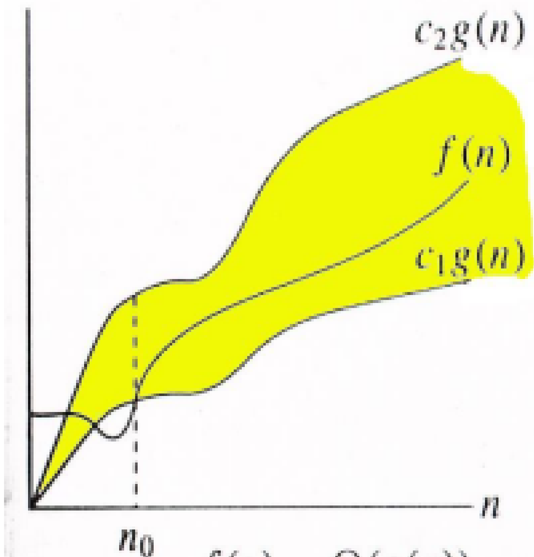
we have $\mathbf{0 \le c_1 g(n) \le f(n) \le c_2 g(n)}$

}

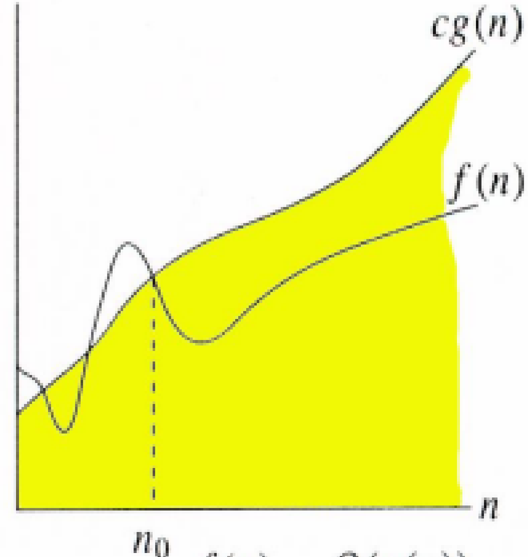
Intuitively: Set of all functions that have the same *rate of growth* as $g(n)$.

$\boldsymbol{g(n)}$ **is an *asymptotically tight bound* for** $\boldsymbol{f(n)}$**.**

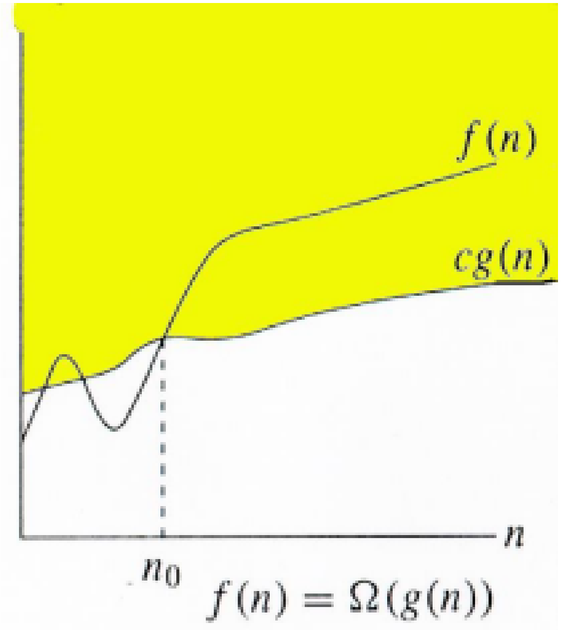
Relations Between Θ, *O*, Ω



$f(n) = \Theta(g(n))$



$f(n) = O(g(n))$



Asymptotic Notation

- O notation: asymptotic “less than”:
  - f(n)=O(g(n)) implies: f(n) “≤” g(n)
- Ω notation: asymptotic “greater than”:

- f(n)= Ω (g(n)) implies: f(n) “≥” g(n)

- Θ notation: asymptotic “equality”:
  - f(n)= Θ (g(n)) implies: f(n) “=” g(n)

Best worst and average time complexity

- Best case: The algorithm take as min time as it can.
  - Searching item in array
  - Found first item as key
- Worst case: The algorithm take max time as it can
  - Searching item in array
  - Found the last item/ did not found item
- Average case: The algorithm takes average time

- Found in middle of array (Just example)

Best worst average vs Notations

- These two can be applied on both the best case and the worst case for linear search:
- best case: first element you look at is the one you are looking for
  - $\Omega(1)$: you need *at least* one lookup
  - $O(1)$: you need *at most* one lookup
- worst case: element is not present

Compare with finding max in array

  - $\Omega(n)$: you need *at least n* steps until you can say that the element you are looking for is not present

- O(n): you need *at most n* steps until you can say that the element you are looking for is not present

- But often we do only want to know the upper bound or tight bound as the lower bound has not much practical information.

Example : Comparing Functions

4000

- Which function is $10\, n^2$ Vs n^3 better?

3500 3000 2500 2000 1500
1000 500
0
1 2 3 4 5 6 7 8 9 10 11 12 13
14 15

10 n^2
n^3

Comparing Functions

- As inputs get larger, any algorithm of a smaller order will be more efficient than an algorithm of a larger order

$0.05\ N^2 = O(N^2)$ $3N =$

$O(N)$

Time (steps)

29

Input (size)

$N = 60$

Big-Oh Notation

- Even though it is correct to say "7n - 3 is $O(n^3)$", a better statement is "7n - 3 is $O(n)$", that is, one should make the approximation as tight as possible
- Simple Rule:

  Drop lower order terms and constant factors

$7n-3$ is $O(n)$

$8n^2 \log n + 5n^2 + n$ is $O(n^2 \log n)$

Performance Classification

f(*n*) Classification

1 ***Constant*: run time is fixed, and does not depend upon n. Most instructions are executed once, or only a few times, regardless of the amount of information being processed**

`log n` ***Logarithmic*: when *n* increases, so does run time, but much slower. Common in programs which solve large problems by transforming them into smaller problems.**

n ***Linear*: run time varies directly with *n*. Typically, a small amount of processing is done on each element.**

n `log` n **When *n* doubles, run time slightly more than doubles. Common in programs which break a problem down into smaller sub-problems, solves them independently, then combines solutions**

n^2 *Quadratic*: when *n* doubles, runtime increases fourfold. Practical only for small problems; typically the program processes all pairs of input (e.g. in a double nested loop).

n^3 *Cubic*: when n doubles, runtime increases eightfold

2^n *Exponential*: when n doubles, run time squares. This is often the result of a natural, "brute force" solution.

Classes of Complexities

- Constant: O(c),
- Logarithmic: $O(\log_c n)$,
- Linear: O(n),
- Quadratic: $O(n^2)$,
- Cubic: $O(n^3)$,
- Polynomial: $O(n^c)$

- Exponential: $O(c^n)$

Size does matter

What happens if we double the input size N?

N $\log_2$N 5N N $\log_2$N N^2 2^N ~~8 3 40 24 64 256~~ 16 4 80 64 256 65536 32 5 160 160 1024 $\sim 10^9$ 64 6 320 384 4096 $\sim 10^{19}$ 128 7 640 896 16384 $\sim 10^{38}$ 256 8 1280 2048 65536 $\sim 10^{76}$

Size does matter

- Suppose a program has run time O(n!) and the run time for n = 10 is 1 second

  For n = 12, the run time is 2 minutes
  For n = 14, the run time is 6 hours
  For n = 16, the run time is 2 months
  For n = 18, the run time is 50 years
  For n = 20, the run time is 200 centuries

Standard Analysis Techniques

- Constant time statements
- Analyzing Loops
- Analyzing Nested Loops
- Analyzing Sequence of Statements
- Analyzing Conditional Statements

Constant time statements

- Simplest case: O(1) time statements
- Assignment statements of simple data types
  int x = y;
- Arithmetic operations:
  x = 5 * y + 4 - z;
- Array referencing:
  A[j] = 5;
- Array assignment:
  ∀ j, A[j] = 5;
- Most conditional tests:
  if (x < 12) ...

Analyzing Loops

- Any loop has two parts:
  - How many iterations are performed?
  - How many steps per iteration?

```
int sum = 0,j;
for (j=0; j < N; j++)
sum = sum +j;
```

  - Loop executes N times (0..N-1)
  - 4 = O(1) steps per iteration
- Total time is N * O(1) = O(N*1) = O(N)

Analyzing Loops

- What about this **for** loop?

```
int sum =0, j;
for (j=0; j < 100; j++)
sum = sum +j;
```

- Loop executes 100 times
- 4 = O(1) steps per iteration
- Total time is 100 * O(1) = O(100 * 1) = O(100) = O(1)

Analyzing Nested Loops

- Treat just like a single loop and evaluate each level of nesting as needed:

```
int j,k;
for (j=0; j<N; j++)
for (k=N; k>0; k--)
sum += k+j;
```

- Start with outer loop:
- How many iterations? N
- How much time per iteration? Need to evaluate inner loop
- Inner loop uses O(N) time
- Total time is N * O(N) = O(N*N) = $O(N^2)$

Analyzing Nested Loops

- What if the number of iterations of one loop depends on the counter of the other?

```
int j,k;
for (j=0; j < N; j++)
for (k=0; k < j; k++)
sum += k+j;
```

- Analyze inner and outer loop together:
- Number of iterations of the inner loop is:
- $0 + 1 + 2 + ... + (N-1) = O(N^2)$

Analyzing Sequence of Statements

- For a sequence of statements, compute their complexity functions individually and add them up

```
for (j=0; j < N; j++)
        for (k =0; k < j; k++)
        sum = sum + j*k;
       for (l=0; l < N; l++)
       sum = sum -l;

       cout<<"Sum="<<s
       um;
```

$O(N^2)$

$O(N)$ $O(1)$

Total cost is $O(N^2) + O(N) + O(1) = O(N^2)$

SUM RULE

Analyzing Conditional Statements

What about conditional statements such as

```
if (condition)
statement1;
else
statement2;
```

where statement1 runs in $O(N)$ time and statement2 runs in $O(N^2)$ time?

We use "worst case" complexity: among all inputs of size N, that is the maximum running time?

The analysis for the example above is $O(N^2)$

Time complexity familiar tasks

Growth rate $O(1)$

Task

Getting a specific element from a list (array) Dividing a list in half, dividing one halve in half, etc Binary Search

Scanning (brute force search) a list

Nested **`for`** loops (k levels)

MergeSort

BubbleSort

Generate all subsets of a set of data

Generate all permutations of a set of data

$O(\log_2 N)$

$O(\log_2 N)$

$O(N)$

$O(N^k)$

$O(N \log_2 N)$ $O(N^2)$

$O(2^N)$

$O(N!)$

Data

types

Primitive Data

Types in C/C++

Primitive data types

1. Integer Types

- **int**: A basic integer type**short int** (or short): A short integer type, usually 2 bytes (16 bits).
- **long int** (or long): A long integer type
- **long long int** (or long long): A longer integer type
- **unsigned variants**: unsigned int, unsigned short, unsigned long, unsigned long long — store only non-negative values and extend the positive range.

2. Character Types

- **char**: A character type

3. Floating-Point Types

- **float**: Single-precision floating-point type
- **double**: Double-precision floating-point type
- **long double**: Extended-precision floating-point type

4. Boolean Type

- **bool**: Represents Boolean values (true or false). In C++

Thank you