Algorithm Analysis

Data Structure

Fall 2022

Comparison of Algorithms

- How do we compare the efficiency of different algorithms?
- Comparing execution time: Too many assumptions, varies greatly between different computers
- Compare number of instructions: Varies greatly due to different languages, compilers, programming styles...

Analysis of Algorithm

The theoretical study of design and analysis of computer algorithms

- Not about programming
- Design: design correct algorithms which minimize cost
  - Efficiency is the design criterion
- Analysis: predict the cost of an algorithm in terms of resource and performance

Basic Goals of Designing Algorithms

Basic goals for an algorithm

- always correct
- always terminates

More, we also care about *performance*

- Tradeoffs between what is possible and what is impossible
- We usually have a deadline
  - E.g., Computing 24-hour weather forecast within 20 hours

Why is algorithm analysis useful?

Computers are always limited in the computational ability and memory

- Resources are always limited
- Efficiency is the center of algorithms

So, we need to learn how to solve a problem in an efficient way

Example 1

Determine whether x is one of A[1], A[2], . . . , A[n] (and retrieve other information about x).

– Algorithm: go through each number in order and compare it to x.

i = 1;

while(i <= n) and (A[i] $\neq x$) do

i = i +1;

if (i > n) then i = 0;

Example 1 (cont.)

- Number of element comparisons?
- Worst case?
- Best case?

```
i = 1;
while(i <= n) and (A[i] ≠ x) do
i = i +1;
if (i > n) then i = 0;
```

Example 2

Square Matrix Multiplication.

$$\begin{bmatrix} c_{11} & \cdots & c_{1n} \\ \vdots & \ddots & \vdots \\ c_{n1} & \cdots & c_{nn} \end{bmatrix} = \begin{bmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{n1} & \cdots & a_{nn} \end{bmatrix} \times \begin{bmatrix} b_{11} & \cdots & b_{1n} \\ \vdots & \ddots & \vdots \\ b_{n1} & \cdots & b_{nn} \end{bmatrix}$$

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$$

Example 2 (cont.)

```
for i = 1 to n do
        for j = 1 to n do
        {
                c[i,j] = 0
                for k = 1 to n do
                        c[i,j] = c[i,j] + a[i,k]*b[k,j];
        }
```

- What is the number of multiplications?
- What is the number of additions?

Growth rate analysis

A further abstraction that we use in algorithm analysis is to characterize in terms of growth classes.

- Matrix multiplication time grows as n^3
- Linear search time grows as n

Why is growth rate important?

Actual execution time assuming 1,000,000 basic operations per second.

Input size	n	$n\lg n$	n^2	n^3	2^n
10	0.00001 sec	3.62e-5 sec	0.0001 sec	0.001 sec	<0.01 sec
100	0.0001 sec	6.52e-4 sec	0.01 sec	1 min	~∞ centuries
1000	0.001 sec	0.00978 sec	1 sec	17.64 min	~∞ centuries
10^4	0.01 sec	0.132 sec	1.692 min	11.76 days	~∞ centuries

Growth classes of functions

- O(g(n)) big oh: upper bound on the growth rate of a function;
  - That is, a function belongs to class O(g(n)) if g(n) is an upper bound on its growth rate
- Ω(g(n)) big omega: lower bound on the growth rate of a function
- Θ (g(n)) big theta: exact bound on the growth rate of a function

Big oh and big omega

- f(n) ∈ O(g(n)) iff there exist c > 0 and n_0> 0 such that f(n) $\leq$ cg(n) for all n $\geq n_0$
- f(n) ∈ Ω(g(n)) iff there exist c > 0 and n_0 > 0 such that f(n) $\geq$ cg(n) for all n $\geq n_0$
- Θ (g(n) ∈ O(g(n)) ∩ Ω(g(n))

Big-O Notation

- The best way is to compare algorithms by the amount of work done in a critical loop, as a function of the number of input elements (*N*)
- **Big-O:** A notation expressing execution time (complexity) as the term in a function that increases most rapidly relative to *N*
- Consider the *order of magnitude* of the algorithm

Common Orders of Magnitude

- O(1): Constant or *bounded* time; not affected by *N* at all
- $O(\log_2 N)$: Logarithmic time; each step of the algorithm cuts the amount of work left in half
- $O(N)$: Linear time; each element of the input is processed
- $O(N \log_2 N)$: $N \log_2 N$ time; apply a logarithmic algorithm N times or vice versa

Common Orders of Magnitude (cont.)

- $O(N^2)$: Quadratic time; typically apply a linear algorithm N times, or process every element with every other element
- $O(N^3)$: Cubic time; naive multiplication of two NxN matrices, or process every element in a three-dimensional matrix
- $O(2^N)$: Exponential time; computation increases dramatically with input size

Big-O Comparison of List Operations

OPERATION	UnsortedList	SortedList
GetItem	O(N)	O(N) linear search $O(\log_2 N)$ binary search
PutItem		
Find	O(1)	O(N) search
Put	O(1)	O(N) moving down
Combined	O(1)	O(N)
DeleteItem		
Find	O(N)	O(N) search
Put	O(1) swap	O(N) moving up
Combined	O(N)	O(N)

What About Other Factors?

- Consider $f(N) = 2N^4 + 100N^2 + 10N + 50$
- We can ignore $100N^2 + 10N + 50$ because $2N^4$ grows so quickly
- Similarly, the 2 in $2N^4$ does not greatly influence the growth
- The final order of magnitude is $O(N^4)$
- The other factors may be useful when comparing two very similar algorithms

Example: Phone Book Search

- Goal: Given a name, find the matching phone number in the phone book
- Algorithm 1: Linear search through the phone book until the name is found
- Best case: O(1) (it’s the first name in the book)
- Worst case: O(N) (it’s the final name)
- Average case: The name is near the middle, requiring N/2 steps, which is O(N)

Example: Phone Book Search (cont.)

Algorithm 2: Since the phone book is sorted, we can use a more efficient search

1) Check the name in the middle of the book
2) If the target name is less than the middle name, search the first half of the book
3) If the target name is greater, search the last half
4) Continue until the name is found

Example: One Problem & Two Algorithms

- ***Problem: Calculate the sum of the integers from 1 to N, i.e., 1+2+3+…+(N-1) + N***
- ***Alg #1***

```
sum = 0;
for (count=1; count<=N; count++)
    sum = sum + count;
```

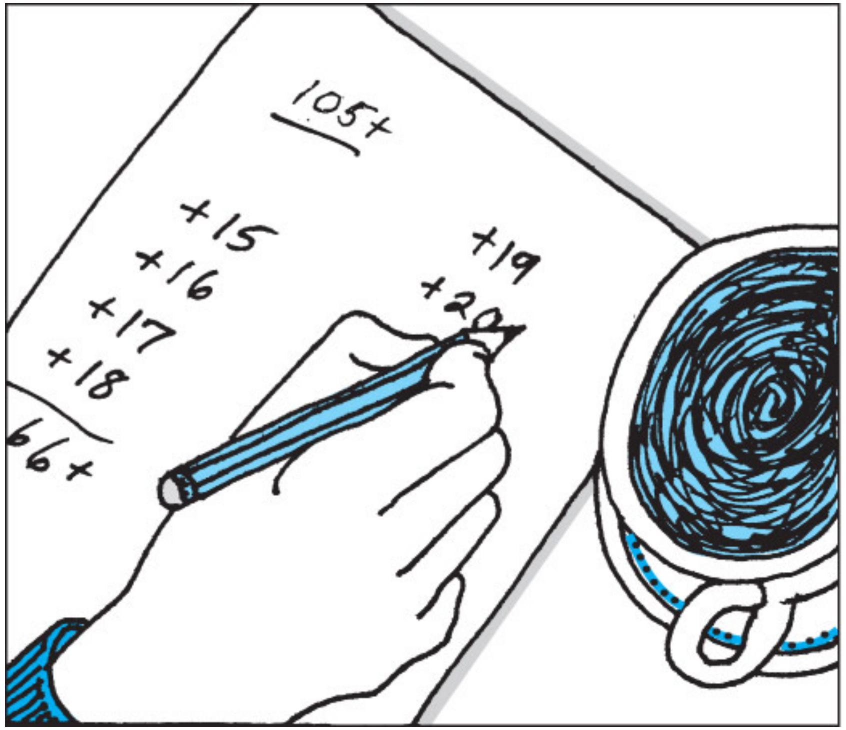
- ***Alg #2***

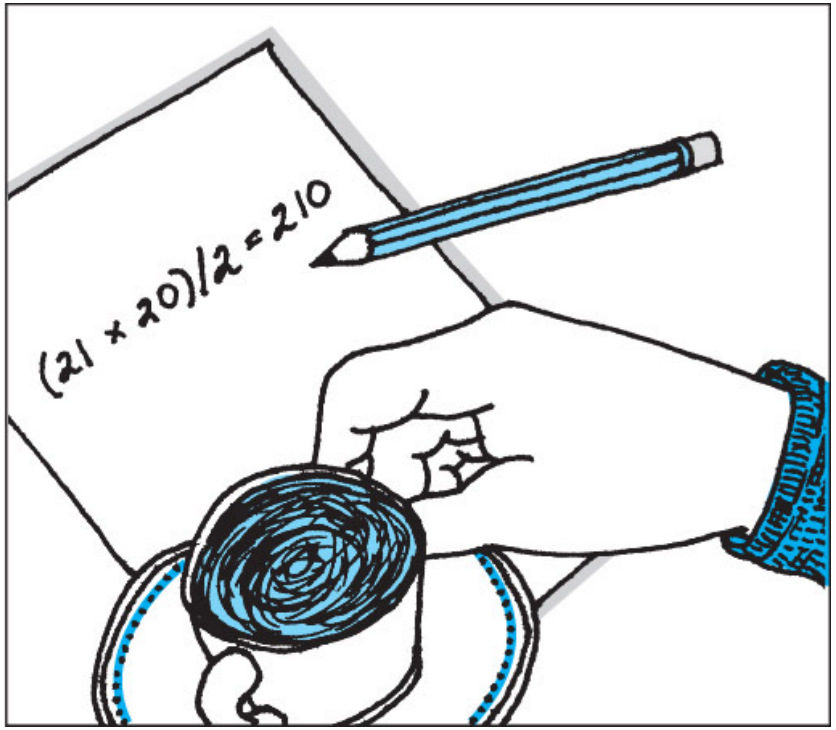
```
sum = (1+N)*N/2;
```

Can you see that both are correct?

Comparison of Two Algorithms

Which is faster?





How to compare Algorithms

1. Compare the actual running time on a computer
2. Compare the number of instructions/statements executed
    - varies with languages used and programmer’s style
    - Count the number of passes through a critical loop in algorithm
3. Count representative operations performed in the algorithm

One Problem & Two Algorithms

- ***Problem: Calculate the sum of the integers from 1 to N, i.e., 1+2+3+…+(N-1) + N***
- ***Alg #1***

```
sum = 0;
for (count=1; count<=N; count++)
    sum = sum + count;
```

performs N addition/assignment operations

- ***Alg #2***

```
sum = (1+N)*N/2;
```

Performs one addition, one multiplication and one division