Advanced Analysis of Algorithms

Fall 2021

Shujaat Hussain
shujaat.Hussain@nu.edu.pk

National University of Computer and Emerging Sciences, Islamabad

About me

- Ph.D (Data Science, Health Informatics) – Kyung Hee University, Korea
- Head, Knowledge Discovery and Data Mining (KDD) Lab
- Current Project from NCAI: ”Re-Designing E-recruitment using AI for Temporal Analysis”

Classroom

- Online course content & coordination
  - Google Classroom

evb7xpa

Text book and reference material

- Introduction to Algorithms (Text Book)

  Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein

  Third Edition, MIT Press
- Any web material (consult authentic material e.g. on some university’s website)

Grading Criteria (Marks Distribution)

- Tentative grading criteria is as follows:
  - Assignments (10%)
  - Quizzes (10%)
  - Mid Term Exam (25%)
  - Project/Paper (15%)
  - Final Exam (40%)

Algorithm

- An algorithm is a well-defined and effective sequence of computation steps that takes some value, or set of values, as input and produces some value, or set of values, as output.

Questions?

- What are algorithms?
- Why is the study of algorithms worthwhile?
- What is the role of algorithms relative to other technologies used in computers?

Correctness of an algorithm

- An algorithm is said to be correct if, for every **input instance**, it halts with the correct output.
- An incorrect algorithm
  - might not halt at all on some input instances, or
  - It might halt with an answer other than desired one.

Problems solved by algorithms

- Sorting/searching are by no mean the only computational problem for which algorithms have been developed.
- Otherwise, we wouldn't have the whole course on this topic
- Practical application of algorithms are <u>ubiquitous</u> and include the following examples

Practical applications

- Internet world
- Electronic commerce
- Manufacturing and other commercial settings
- Shortest path
- Matrices multiplication order
- DNA sequence matching

Common about algorithms

- There are many candidate solutions, most of which are not what we want, finding one that we do want can present quite a challenge.
- There are practical applications (its not just mathematical exercises to develop algorithms.)

Why Study Algorithms and Performance of Algorithms

- Algorithms help us to understand scalability.
- Performance often draws the line between what is feasible and what is impossible.
- Algorithmic mathematics provides a language for talking about program behavior.
- Performance is the currency of computing.
- The lessons of program performance generalize to other computing resources.
- Speed is fun!

Why study algorithms ?

- If you are given two brand new algorithms from two different companies to perform sorting. Which one you would go for ? Lets assume companies are not willing to install the software at your end for testing but are willing to share their pseudo-code with you.
  - You need an objective analysis of both the algorithms before you can choose one. Like:-
    - Scalability of algorithms
    - Real life constraints like time and storage
    - Behavior of the algorithms
    - Quickness (speed is fun)

Example: sorting

- Input: A sequence of n numbers <a1,a2,a3…an>
- Output: A permutation (re-ordering) <b1,b2,b3…bn> of the input sequence such that b1<b2<b3…<bn

Example Sorting (Insertion Sort)

Insertion-Sort(A,n) → A[1 … n]

for j ← 2 to n

    do key ← A[j]

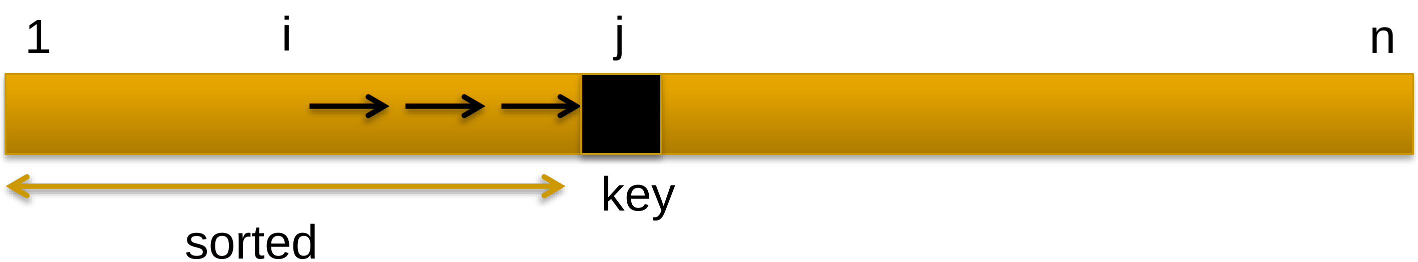
        i ← j-1

        While i > 0 and A[i] > key

            do A[i+1] ← A[i]

                i ← i -1

        A[i+1] = key



Running time of Insertion Sort

- The running time depends on the input: an already sorted sequence is easier to sort.
- Parameterize the running time by the size of the input,
  - short sequences are easier to sort than long ones.
- Generally, we seek upper bounds on the running time, because everybody likes a guarantee.

Problems solved by algorithms

- Sorting/searching are by no mean the only computational problem for which algorithms have been developed.
- Otherwise, we wouldn’t have the whole course on this topic
- Practical application of algorithms are <u>ubiquitous</u> and include the following examples

Practical applications

- Internet world
- Electronic commerce
- Manufacturing and other commercial settings
- Shortest path
- Matrices multiplication order
- DNA sequence matching

Algorithms and other advanced technologies

- Hardware with high clock rates, pipelining and superscalar architecture.
- Easy to use graphical user interface (GUI's)
- Object oriented systems.
- Local-area and wide-area networking.

- Are algorithms as important as above technologies?

Complexity Analysis

Want to achieve platform-independence

- Use an abstract machine that uses *steps* of time and *units* of memory, instead of seconds or bytes
  - each elementary operation takes 1 step
  - each elementary instance occupies 1 unit of memory

Analysing an Algorithm

- Simple statement sequence

  $s_1;\ s_2;\ \ldots.\ ;\ s_k$

  - $O(1)$ as long as k is constant
- Simple loops

  ```
  for(i=0; i<n; i++) { s; }
  ```

  where `s` is $O(1)$

  - Time complexity is $O(n)$
- Nested loops

  ```
  for(i=0; i<n; i++)
      for(j=0; j<n; j++) { s; }
  ```

  - Complexity is $O(n^2)$

This part is *O(n)*

Analysing an Algorithm

- Loop index doesn't vary linearly

```
h = 1;
while ( h <= n ) {
        s;
        h = 2 * h;
        }
```

  - `h` takes values 1, 2, 4, … until it exceeds n
  - There are $1 + \log_2 n$ iterations
  - Complexity $O(\log n)$

Analysing an Algorithm

- Loop index depends on outer loop index

```
for(j=0;j<=n;j++)
    for(k=0;k<j;k++){
        s;
        }
```

- Inner loop executed
  - 1, 2, 3, …., n times

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

∴ Complexity $O(n^2)$

Distinguish this case - where the iteration count increases (decreases) by a factor ç $O(n^k)$ from the previous one - where it changes by a factor ç $O(\log n)$

Analysing an Algorithm

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

int Sum(int A[], int N)
{
   int s=0;
   for (int i=0; i< N; i++)
      s = s + A[i];
   return s;
}
```

How should we analyse this?

Analysing an Algorithm

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

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

    for (int i=0; i< N; i++)
  ②                ③       ④
       s = s + A[i];
  ⑤          ⑥     ⑦
    return s;
}             ⑧
```

1,2,8: Once

3,4,5,6,7: Once per each iteration of for loop, N iteration

Total: 5N + 3

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

Analysing an Algorithm

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

What is fundamental is that the time is *linear* in N.
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

Asymptotic Complexity

- The 5N+3 time bound is said to "grow asymptotically" like N
- This gives us an approximation of the complexity of the algorithm
- Ignores lots of (machine dependent) details, concentrate on the bigger picture

Comparing Functions: Asymptotic Notation

- Big Oh Notation: Upper bound
- Omega Notation: Lower bound
- Theta Notation: Tighter bound

Big Oh Notation

If f(N) and g(N) are two complexity functions, we say

$$f(N) = O(g(N))$$

(read "f(N) as order g(N)", or "f(N) is big-O of g(N)")

if there are constants c and N_0 such that for $N > N_0$,

$$f(N) \leq c * g(N)$$

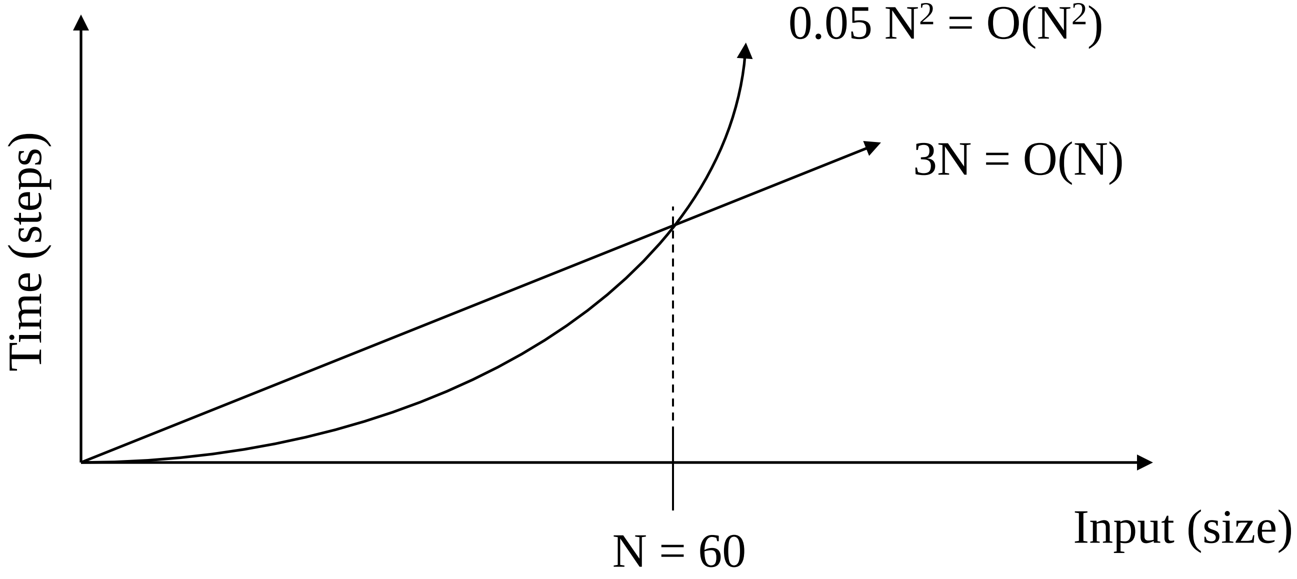
for all sufficiently large N.

Polynomial and Intractable Algorithms

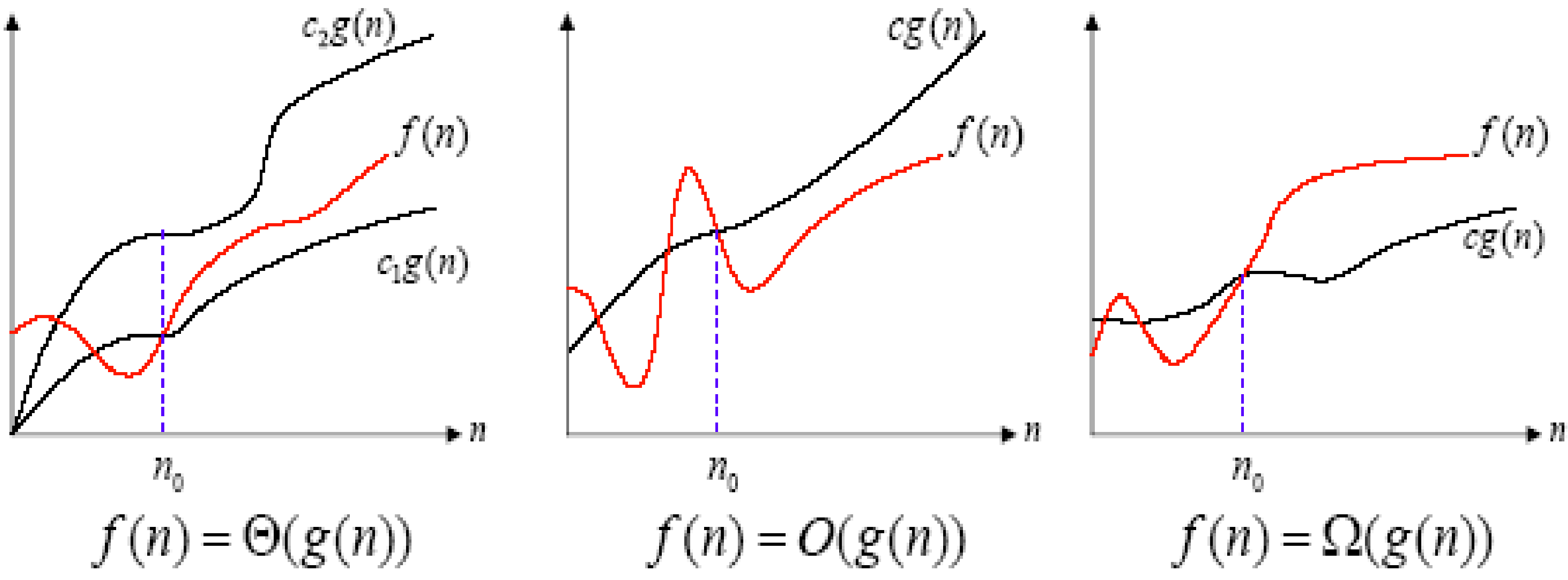
- Polynomial Time complexity
  - An algorithm is said to be polynomial if it is $O(n^d)$ for some integer d
  - Polynomial algorithms are said to be efficient
    - They solve problems in reasonable times!
- Intractable algorithms
  - Algorithms for which there is no *known* polynomial time algorithm
  - *We will come back to this important class later*

Comparing Functions

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



Asymptotic notation



- Example:
  - $f(n) = 3n^5+n^4 = \Theta(n^5)$

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. When *n* doubles, `log` *n* increases by a constant, but does not double until *n* increases to n^2. 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.

Size does matter

What happens if we double the input size N?

N	$\log_2 N$	N	$N \log_2 N$	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	$\sim 10^{76}$

Review of Three Common Sets

$g(n) = O(f(n))$ means $c \times f(n)$ is an *Upper Bound* on $g(n)$

$g(n) = \Omega(f(n))$ means $c \times f(n)$ is a *Lower Bound* on $g(n)$

$g(n) = \Theta(f(n))$ means $c_1 \times f(n)$ is an *Upper Bound* on $g(n)$
and $c_2 \times f(n)$ is a *Lower Bound* on $g(n)$

These bounds hold for all inputs beyond some threshold n_0.

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;
```
- 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)
  - 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
- 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:
- $0 + 1 + 2 + \ldots + (N-1) = O(N^2)$

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="<<sum;
```

$O(N^2)$

$O(N)$

$O(1)$

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 $O(N^2)$

Properties of the O notation

- Constant factors may be ignored
  - $\forall\ k > 0,\ kf$ is $O(f)$
- Higher powers grow faster
  - n^r is $O(n^s)$ if $0 \leq r \leq s$
- Fastest growing term dominates a sum
  - If f is $O(g)$, then $f + g$ is $O(g)$

    eg $an^4 + bn^3$ is $O(n^4)$
- Polynomial's growth rate is determined by leading term
  - If f is a polynomial of degree d, then f is $O(n^d)$

Properties of the O notation

- f is $O(g)$ is transitive
  - If f is $O(g)$ and g is $O(h)$ then f is $O(h)$
- Product of upper bounds is upper bound for the product
  - If f is $O(g)$ and h is $O(r)$ then fh is $O(gr)$
- Exponential functions grow faster than powers
  - n^k is $O(b^n)$ $\forall$ $b > 1$ and $k \geq 0$
    $e.g.\ n^{20}$ is $O(1.05^n)$
- Logarithms grow more slowly than powers
  - $\log_b n$ is $O(n^k)$ $\forall$ $b > 1$ and $k > 0$
    $e.g.\ \log_2 n$ is $O(n^{0.5})$

Important!

Properties of the O notation

- All logarithms grow at the same rate
  - $\log_b n$ is $O(\log_d n)\ \forall\ \ b, d > 1$

- Sum of first n r^{th} powers grows as the $(r+1)^{th}$ power
  - $\sum_{k=1}^{n} k^r$ is $\Theta(\ n^{r+1}\ )$

  $e.g.\quad \sum_{k=1}^{n} k = \frac{n(n+1)}{2}$ is $\Theta(\ n^2\ )$