Analysis of Algorithms: Growth of functions and Time Complexity

Maunendra Sankar Desarkar

IIT Hyderabad

CS1353: Introduction to Data Structures

Acknowledgements

- Slides adapted from lectures on same topic by
  - Dr. Zahoor Jan
  - Dr. Jeyakesavan Veerasamy
  - Jeremy Johnson

Analysis of Algorithms: Why?

- Algorithms need to be correct (We are talking about deterministic algorithms here).
- Time taken to complete the program should not be unreasonably high.
- … and, algorithms should be scalable.

- Example scenario when the running time (waiting time for user) noticeable / important?
  - Web search
  - Database search
  - Real-time systems with time constraints
    - Withdrawing money from ATM
    - Decision taken by an Autonomous Vehicle

Analysis of Algorithms: How?

- Run the program
- Measure the time

- Is it the right thing to do?
- Same algorithm, same input
  - Different running times are possible.
  - Why?

Factors that determine running time of a program

Factors that determine running time of a program

- Problem size: n
- Basic algorithm / actual processing
- Memory access speed
- CPU/processor speed
- # of processors?
- Compiler/linker optimization?

Running time of a program or transaction processing time

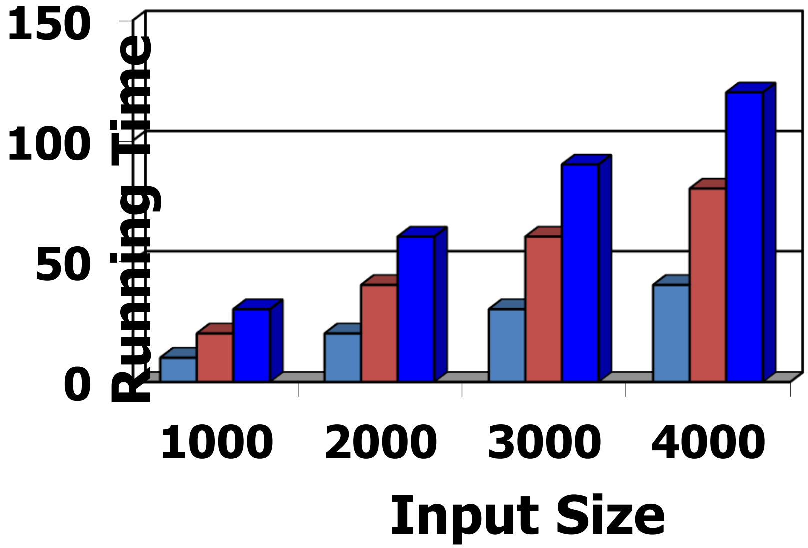
- Amount of input: n → min. linear increase*
- Basic algorithm / actual processing → depends on algorithm!
- Memory access speed → by a factor
- CPU/processor speed → by a factor
- # of processors? → yes, if multi-threading or multiple processes are used.
- Compiler/linker optimization? → ~20%

Time Complexity

- Measure of algorithm efficiency
- Ignore hardware and environment
  - E.g. processor details, threading etc.
- Focus on input size
- Has a big impact on running time.

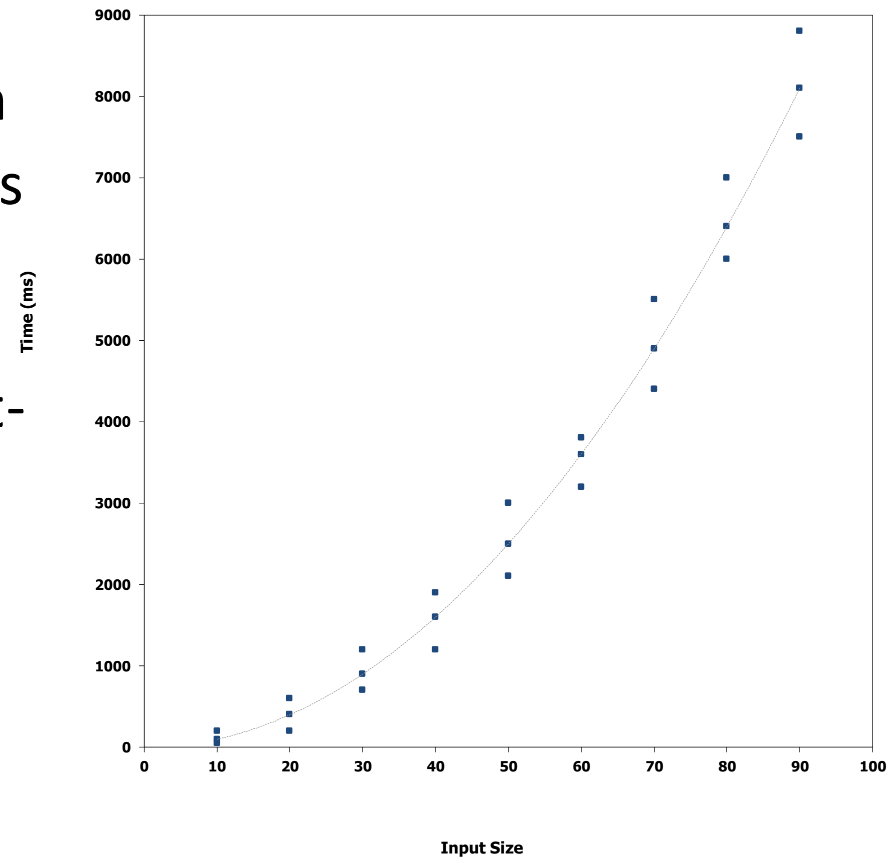
Running Time

- The running time of an algorithm typically grows with the input size.
- We (generally) focus on the worst case running time.
  - Easier to analyze
  - Crucial to applications such as games, finance, robotics, …



Experimental Studies

- Write a program implementing the algorithm
- Run the program with inputs of varying size and composition
- Use a function, like the built-in clock() function, to get an accurate measure of the actual running time
- Plot the results



Limitations of Experiments

- It is necessary to implement the algorithm, which may be difficult
- Results may not be indicative of the running time on other inputs not included in the experiment.
- In order to compare two algorithms, the same hardware and software environments must be used

Theoretical Analysis

- Uses a high-level description of the algorithm instead of an implementation
- Characterizes running time as a function of the input size, *n*.
- Takes into account all possible inputs
- Allows us to evaluate the speed of an algorithm independent of the hardware/software environment

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){
    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.
N iterations

Total: 5N + 3

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

Simple Example: 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

- 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

Coding example #1

```
for ( i=0 ; i<n ; i++ )
    m += i;
```

Coding example #2

```
for ( i=0 ; i<n ; i++ )
        for( j=0 ; j<n ; j++ )
                sum[i] += entry[i][j];
```

Coding example #3

```
for ( i=0 ; i<n ; i++ )
        for( j=0 ; j<=i ; j++ )
            m += j;
```

Coding example #4

```
i = 1; tot = 0;
while (i < n) {
     tot += i;
     i = i * 2;
}
```

Example #4: equivalent # of steps?

```
i = n;
while (i > 0) {
    tot += i;
    i = i / 2;
}
```

Coding example #5

```
for ( i=0 ; i<n ; i++ )
    for( j=0 ; j<n ; j++ )
        for( k=0 ; k<n ; k++ )
            sum[i][j] += entry[i][j][k];
```

Coding example #6

```
for ( i=0 ; i<n ; i++ )
        for( j=0 ; j<n ; j++ )
             sum[i] += entry[i][j][0];

for ( i=0 ; i<n ; i++ )
        for( k=0 ; k<n ; k++ )
             sum[i] += entry[i][0][k];
```

Coding example #7

```
for ( i=0 ; i<n ; i++ )
        for( j=0 ; j< sqrt(n) ; j++ )
            m += j;
```

Coding example #8

```
for ( i=0 ; i<n ; i++ )
        for( j=0 ; j< sqrt(995) ; j++ )
            m += j;
```

Coding example #8 : Equivalent code

```
for ( i=0 ; i<n ; i++ )
{
        m += j;
        m += j;
        m += j;
        …
        m += j;    // 31 times
}
```

COMPARING FUNCTIONS: ASYMPTOTIC NOTATION

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

Big Oh Notation [1]

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

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

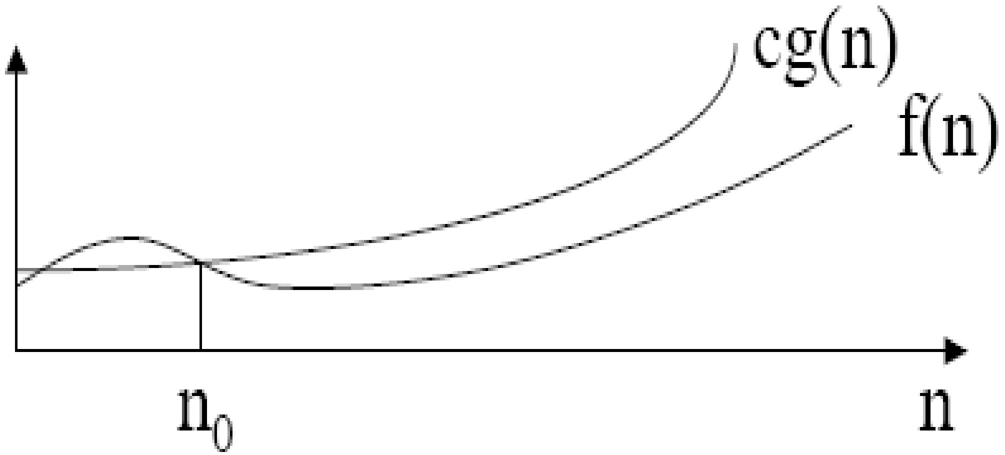
(read "f(N) is order g(N)", or "f(N) is big-O of g(N)")

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

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

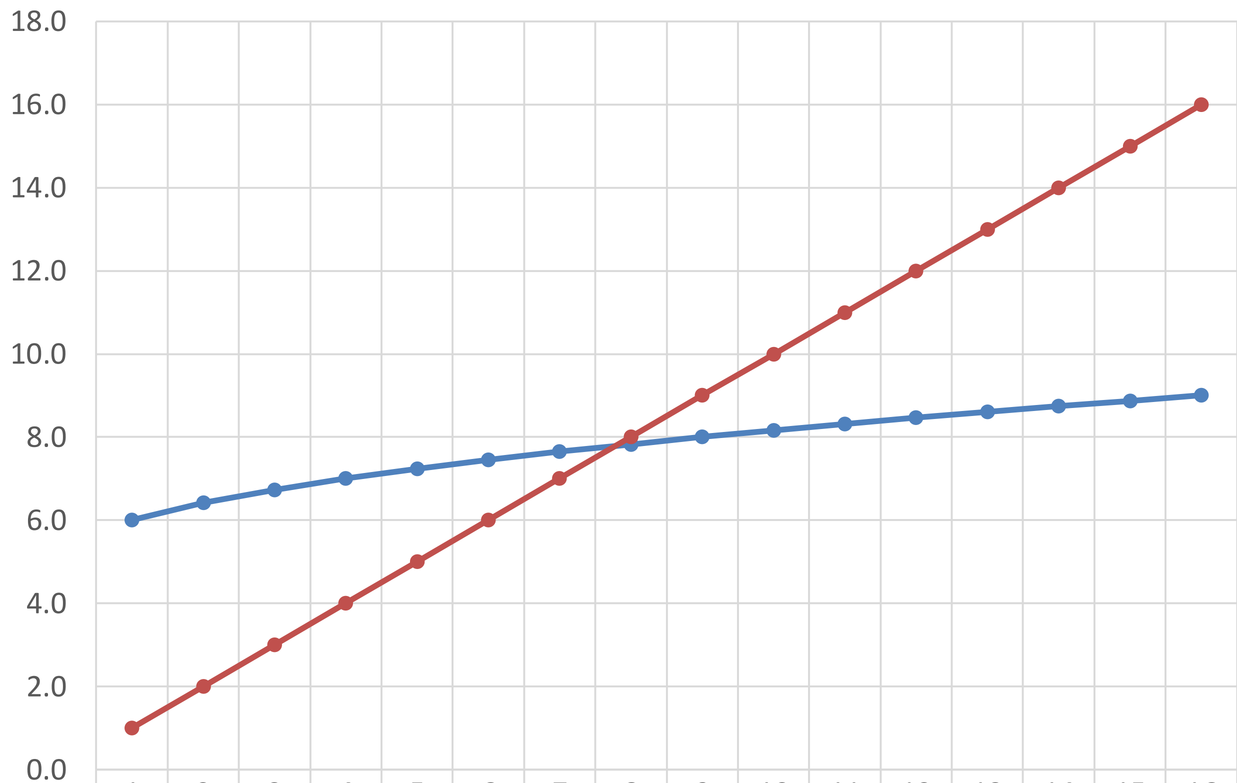
for all sufficiently large N.

Big Oh Notation [2]



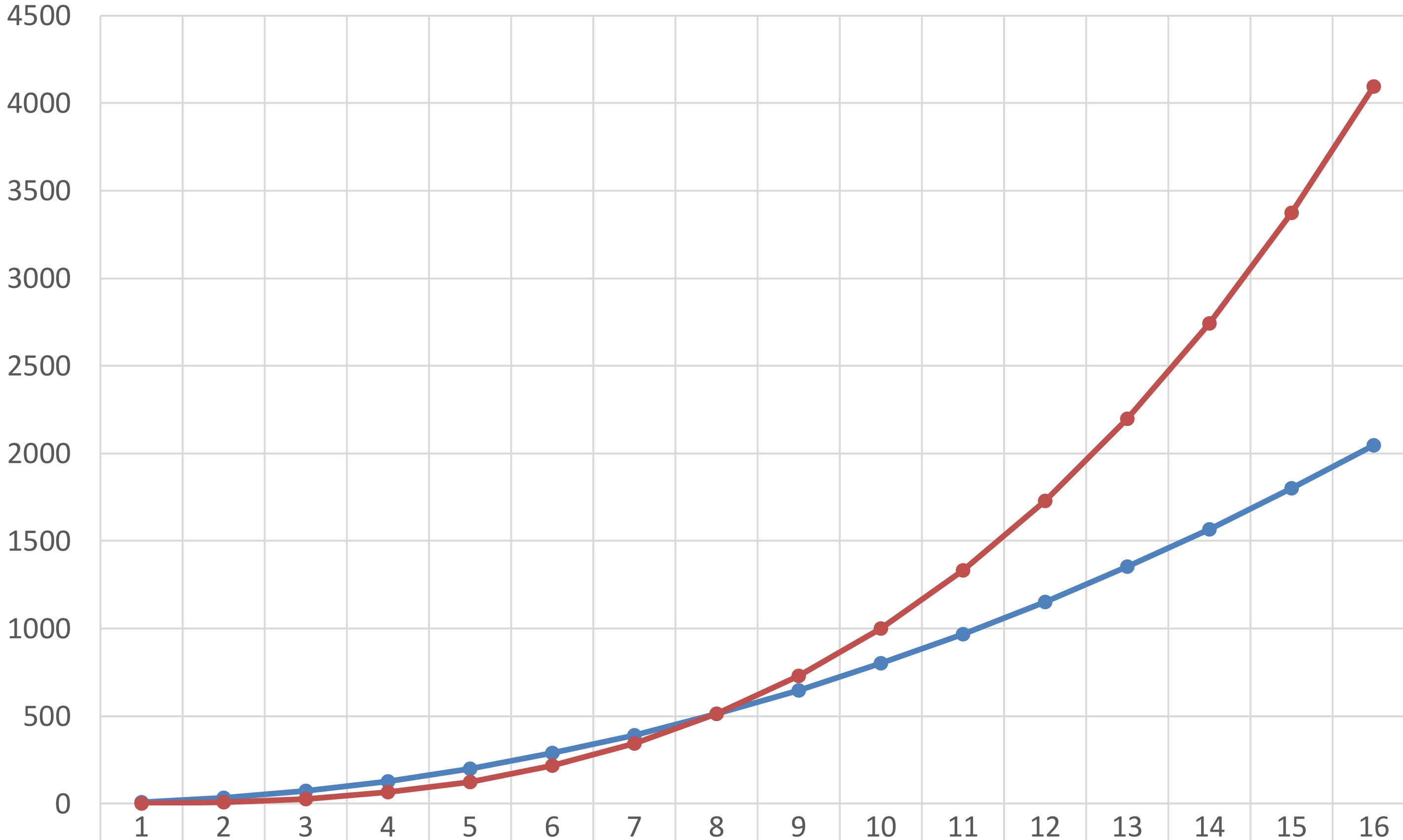
- Function cg(n) always dominates f(n) to the right of n_0

Functions and upper bounds



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
f(n)=sqrt(n)+5	6.0	6.4	6.7	7.0	7.2	7.4	7.6	7.8	8.0	8.2	8.3	8.5	8.6	8.7	8.9	9.0
g(n)=n	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Functions and upper bounds



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
f(n)=8n^2	8	32	72	128	200	288	392	512	648	800	968	1152	1352	1568	1800	2048
g(n)=n^3	1	8	27	64	125	216	343	512	729	1000	1331	1728	2197	2744	3375	4096

f(n)=8n^2 g(n)=n^3

Exercise: Time complexity relations

- Using the definition of Big-O notation, show that:
  - $7n + 8 = O(n)$
  - $2n^2 + 3n + 8 = O(n^2)$
  - $n^2 - n = O(n^2)$
  - $\log(n!) = O(n \log n)$
  - $\log \log n = O(\log n)$
  - $3 \log_4 n + \log_2 \log_2 n = O(\log_2 n)$
  - $\sum_{k=0}^{n} 3^k = O(3^n)$