CSE221

Performance analysis

Fall 2021

Young-ri Choi

Acknowledgment: The content of this file is based on the slides of the textbook as well as the slides provided in former lectures at UNIST.

ULSAN NATIONAL INSTITUTE OF SCIENCE AND TECHNOLOGY

Outline

- Performance analysis
- Performance measurement

Performance Evaluation

- Judging programs
- How?
  - Prior estimate
    - Performance analysis
  - Posteriori testing
    - Performance measurement

Performance Analysis

- Theoretical analysis
- Criteria: Complexity
  – Space complexity
  – Time complexity
- Complexity is affected by instance characteristics (size of inputs and outputs)
  – E.g., function f(n) where n is input size

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

Space Complexity

- $S(P) = c + S_p$(instance characteristics)
- Fixed part : c
  - Independent of input/output characteristics
  - Space for instruction, constants, etc.
- Variable part : S_p(instance characteristics)
  - Space for variables whose size is dependent of input/output, recursion stack space, etc.

Since c is constant, we only focus on S_p for space complexity!

Example

```
float Abc(float a, float b, float c)
{
  return a+b+b*c+(a+b-c)/(a+b)+4.0;
}
```

1. What are the instance characteristics?
2. S_p?

Example

```
float Abc(float a, float b, float c)
{
  return a+b+b*c+(a+b-c)/(a+b)+4.0;
}
```

1. What are the instance characteristics?
   : a, b, c
2. $S_p = 0$

Example

```
float Sum(float *a, const int n)
{
   float s = 0;
   for(int i=0; i<n; i++)
     s += a[i];
   return s;
}
```

1. What are the instance characteristics?
2. S_p?

ULSAN NATIONAL INSTITUTE OF SCIENCE AND TECHNOLOGY

Example

```
float Sum(float *a, const int n)
{
  float s = 0;
  for(int i=0; i<n; i++)
    s += a[i];
  return s;
}
```

1. What are the instance characteristics?
   : a, n
2. $S_p = 0$

Example

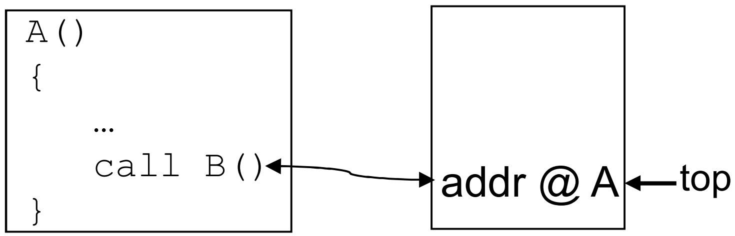
```
float RSum(float *a, const int n)
{
  if(n<=0) return 0;
  else return (Rsum(a, n-1)+a[n-1]);
}
```

1. What are the instance characteristics?
2. S_p ?

Example

```
float RSum(float *a, const int n)
{
  if(n<=0) return 0;
  else return (Rsum(a, n-1)+a[n-1]);
}
```

1. What are the instance characteristics?
   : a, n
2. S_p = A(n+1)
   : A = stack frame size



Time Complexity

- T(P) = compile time + run time
- Compile time: one-time cost (ignore)
- Actual run time: can only be measured
  - Compiler / machine / runtime dependent
  - Difficult to derive exact time for operators
- Use a <u>program step</u> instead

Program Steps (Primitive Operations)

- A segment of program with constant execution time (independent of instance characteristics)

  `e.g., a+b+b*c+(a+b-c)/(a+b)+4.0`

- Set of primitive operations
  - Assigning a value to a variable
  - Calling a function
  - Arithmetic operations (+,-,*,/, ...)
  - Comparing two numbers
  - Indexing into an array
  - Following an object reference
  - Returning from a function...

```
float Sum(float *a, const int n)
{
   float s = 0;
   for (int i = 0; i < n; i++)
   {
       s += a[i];
   }
   return s;
}
```

```
          float Sum(float *a, const int n)
          {
1           float s = 0;
1 * (n + 1) for (int i = 0; i < n; i++)
            {
1 * n           s += a[i];
            }
1           return s;
          }
```

Step count: 2n + 3

ULSAN NATIONAL INSTITUTE OF SCIENCE AND TECHNOLOGY

```
float Rsum(float *a, const int n)
{
  if(n <= 0)
  {
      return 0;
  }
  else
  {
      return (Rsum(a, n-1)+a[n-1]);
  }
}
```

```
                 float Rsum(float *a, const int n)
                 {
1                    if(n <= 0)
                     {
1                        return 0;
                     }
                     else
                     {
Rsum + 1                 return (Rsum(a, n-1)+a[n-1]);
                     }
                 }
```

Step count: 2n + 2

T(P) = 2 + Rsum(n-1) = 2 + 2 + Rsum(n-2) = ... = 2n + Rsum(0) = 2n + 2

```
void Add(int **a,int **b,int n,int m)
{
  for(int i=0; i<m; i++)
  {
    for(int j=0; j<n; j++)
    {
      c[i][j] = a[i][j]+b[i][j];
    }
  }
}
```

```
          void Add(int **a,int **b,int n,int m)
          {
1 *(m + 1)  for(int i=0; i<m; i++)
            {
1 *(n + 1) *m for(int j=0; j<n; j++)
              {
1 *n *m         c[i][j] = a[i][j]+b[i][j];
              }
            }
          }
```

Step count: $2nm + 2m + 1$

Best, Worst, Average Step Count

- What if expressions depend on complex characteristics?
- Ex) linear search

```
        int search(float *a, float k, int n)
        {
1*?       for(int i=0; i<n; i++)
          {
1*?           if(a[i] == k)
              {
1                 return i;
              }
          }
        }
```

? = n(worst case), 1(best case), n/2(average)

average = ∑(all cases) / # cases = (1+2+3+4+…+n) / n

Comparing Step Count

- What step count means?
  - How the run time changes when the instance characteristics changes
  - e.g., how much the program becomes slower if the input size grows

- P_A : $2n + 1$
- P_B : $3n^2$
  - When n is doubled, P_A becomes 2x slower but P_B becomes 4x slower

Comparing Step Count

- P_A : $n^2 + 200$
- P_B : $n^2 + 100$
  - When n = 1, running time of P_A and P_B become 201 and 101, respectively (1.99x difference)
  - When n = 100, running time of P_A and P_B become 10200 and 10100, respectively (1.009x difference)
  - If n becomes very large, P_A and P_B converges (n^2 term dominates)
- P_A runs same as P_B *asymptotically!*

Asymptotic Analysis

- Asymptotic behaviour
  - behaviour as input n approaches infinity
    - input: instance characteristics
  - dominant factor prevails
    - constants become "insignificant"

Asymptotic Analysis

- We need relatively easy ways to compare functions as problem size n vary:
  - A is 'at most' as fast/big as B
  - A is 'at least' as fast/big as B
  - A is 'equal' in performance/size to B
- Asymptotic behaviour
  - The behaviour as some key parameter n increases towards infinity

Big-O

- Formally, given functions *f(x), g(x),*

$$f(x) = O(g(x))$$

  if there exist positive constants c and x_0 such that $f(x) \leq cg(x)$ for all $x \geq x_0$

- Meaning of Big-O
  - For sufficiently large x, *f* is bounded by *g* with a scalar factor
  - *f* is "at most" *g* beyond some value x_0
    - *g* is an upper bound of *f*

Examples

- $3n+2 = O(n)$?
  - *Yes.* $3n + 2 \le 4n$ for all $n \ge 2$
- $1000n^2+100n-6 = O(n^2)$?
  - *Yes.* $1000n^2 + 100n - 6 \le 1001n^2$ for all $n \ge 100$
- $6*2^n+n^2 = O(2^n)$?
  - *Yes.* $6*2^n + n^2 \le 7*2^n$ for all $n \ge 4$

Asymptotic Complexity

- Order of magnitude – higher one dominates
  - O(1) : constant
  - O(log n) : logarithmic
  - O(n) : linear
  - O(nlog n) : log linear
  - O(n^2) : quadratic
  - O(n^3) : cubic
  - O(2^n) : exponential
  - O(n!) : factorial

Higher

Informative Big-O

- How good the bound is?
  - $3n+2 = O(n) = O(n^2)$?
  - Both are correct statement
  - $O(n)$ is tighter bound and more informative
  - $O(n^2)$ is less informative and not used

Big-Oh Rules

- If is $f(n)$ a polynomial of degree d, then $f(n)$ is $O(n^d)$, i.e.,
  1. Drop lower-order terms
  2. Drop constant factors
- Use the smallest possible class of functions
  - Say "$2n$ is $O(n)$" instead of "$2n$ is $O(n^2)$"
- Use the simplest expression of the class
  - Say "$3n + 5$ is $O(n)$" instead of "$3n + 5$ is $O(3n)$"

Theta and Omega

- Omega (Ω) meaning "at least" (lower bound):

$$f(x) = \Omega(g(x))$$

  - If there exist positive constants c and x_0 such that $cg(x) \leq f(x)$ for all $x \geq x_0$

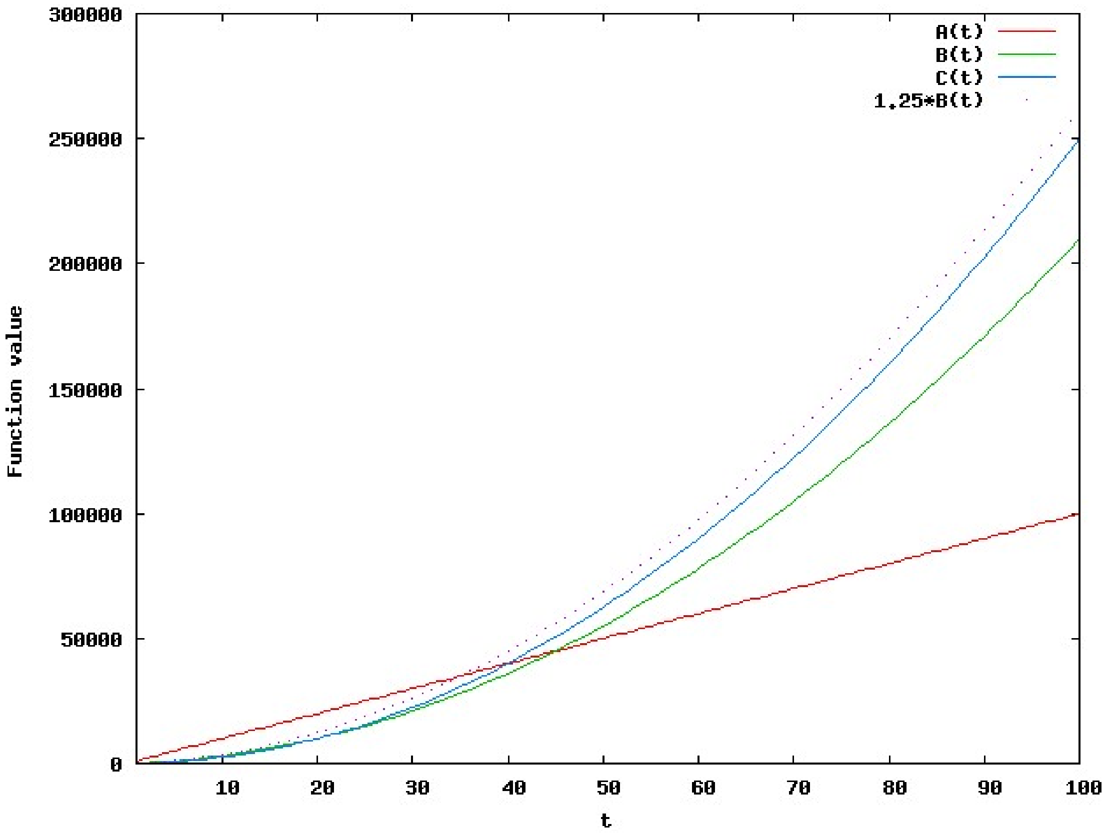
- Theta (Θ) "equals" or "goes as":

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

  - If there exist positive constants c_1, c_2, and x_0 such th at $c_1 g(x) \leq f(x) \leq c_2 g(x)$ for all $x \geq x_0$

Graph

- For *t > 45*, *B(t)* is always greater than *A(t) : A=O(B)*



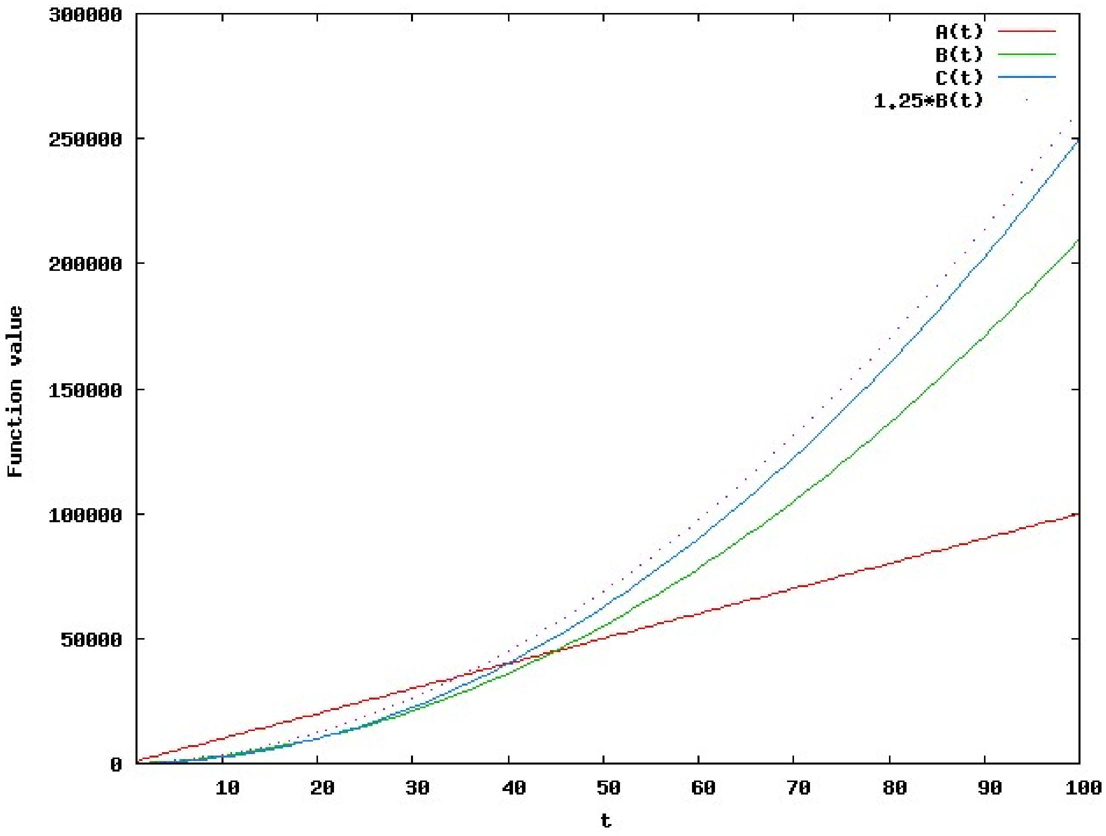
$$A(t) = 1000t$$

$$B(t) = 100t + 20t^2$$

$$C(t) = 25t^2$$

Graph

- Can we say B=*O(A)*?



$$A(t) = 1000t$$

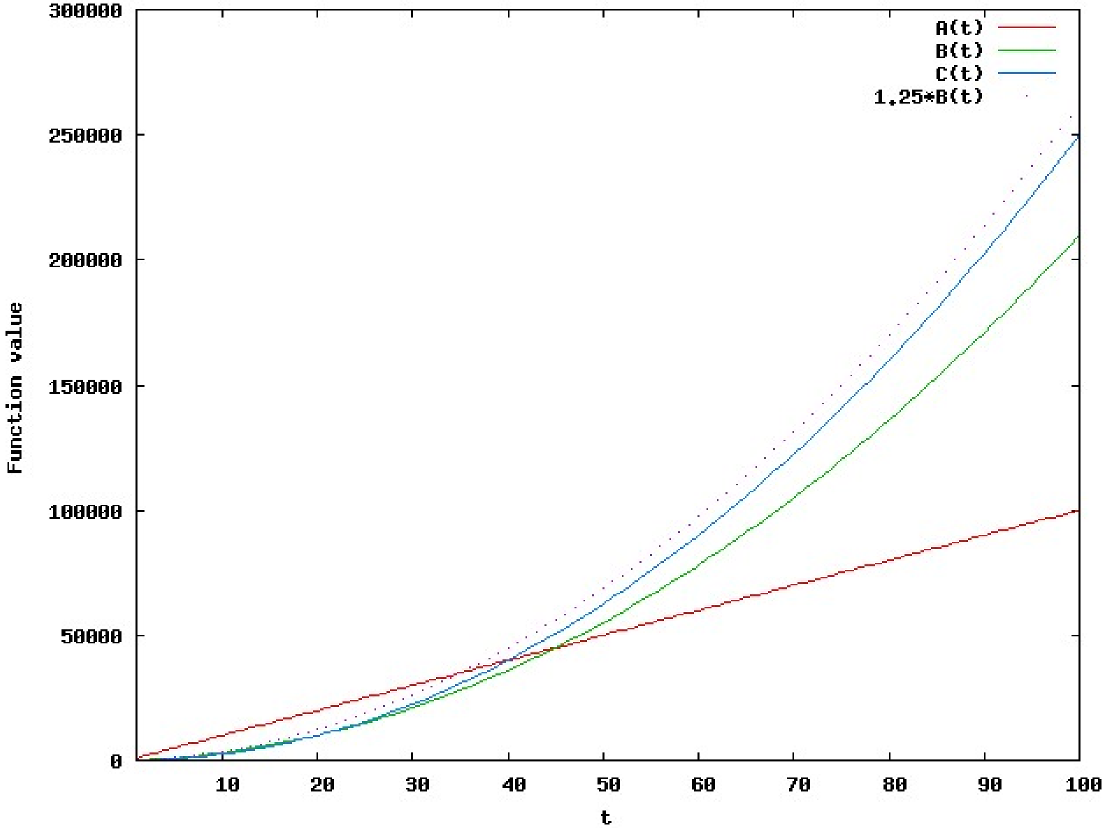
$$B(t) = 100t + 20t^2$$

$$C(t) = 25t^2$$

ULSAN NATIONAL INSTITUTE OF SCIENCE AND TECHNOLOGY

Graph

- For *t > 20, B(t)* is always less than C*(t) : ?*
- For *t > 0, 1.25*B(t)* is always greater than *C(t) : ?*



$$A(t) = 1000t$$

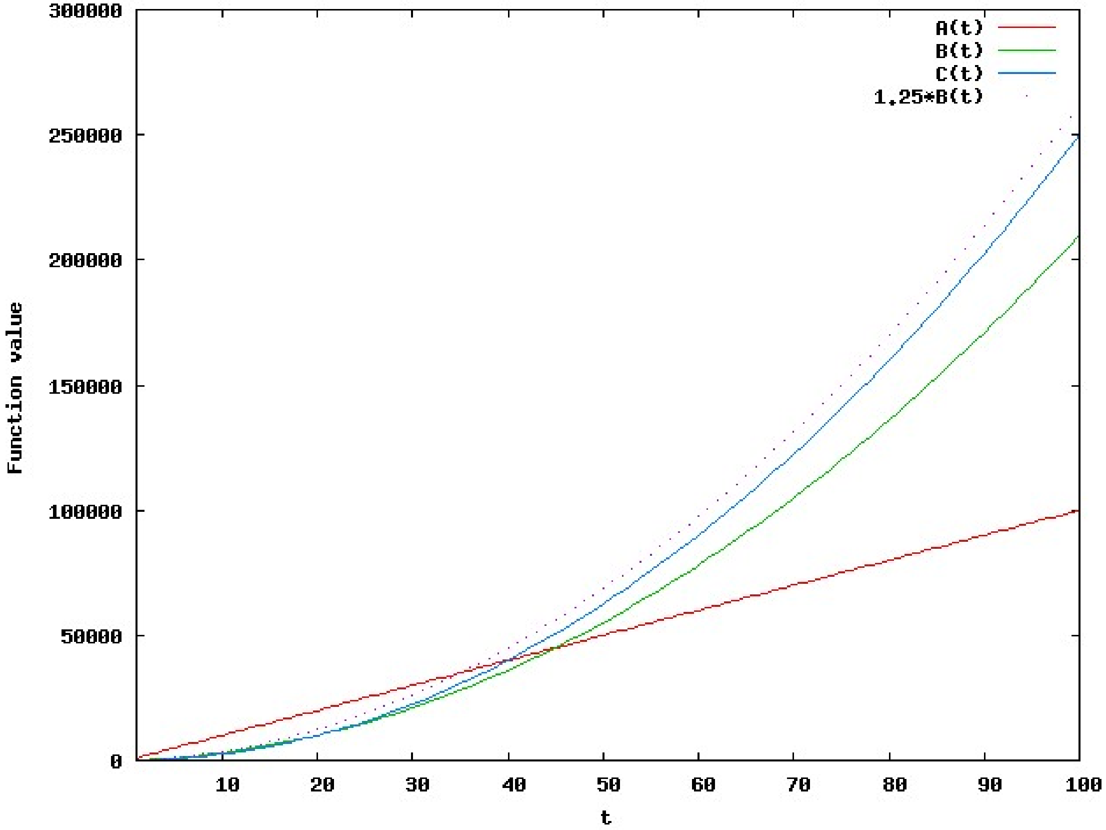
$$B(t) = 100t + 20t^2$$

$$C(t) = 25t^2$$

Graph

C=Θ(B)!

- For $t > 20$, $B(t)$ is always less than $C(t)$: $C=\Omega(B)$
- For $t > 0$, $1.25*B(t)$ is always greater than $C(t)$: $C=O(B)$

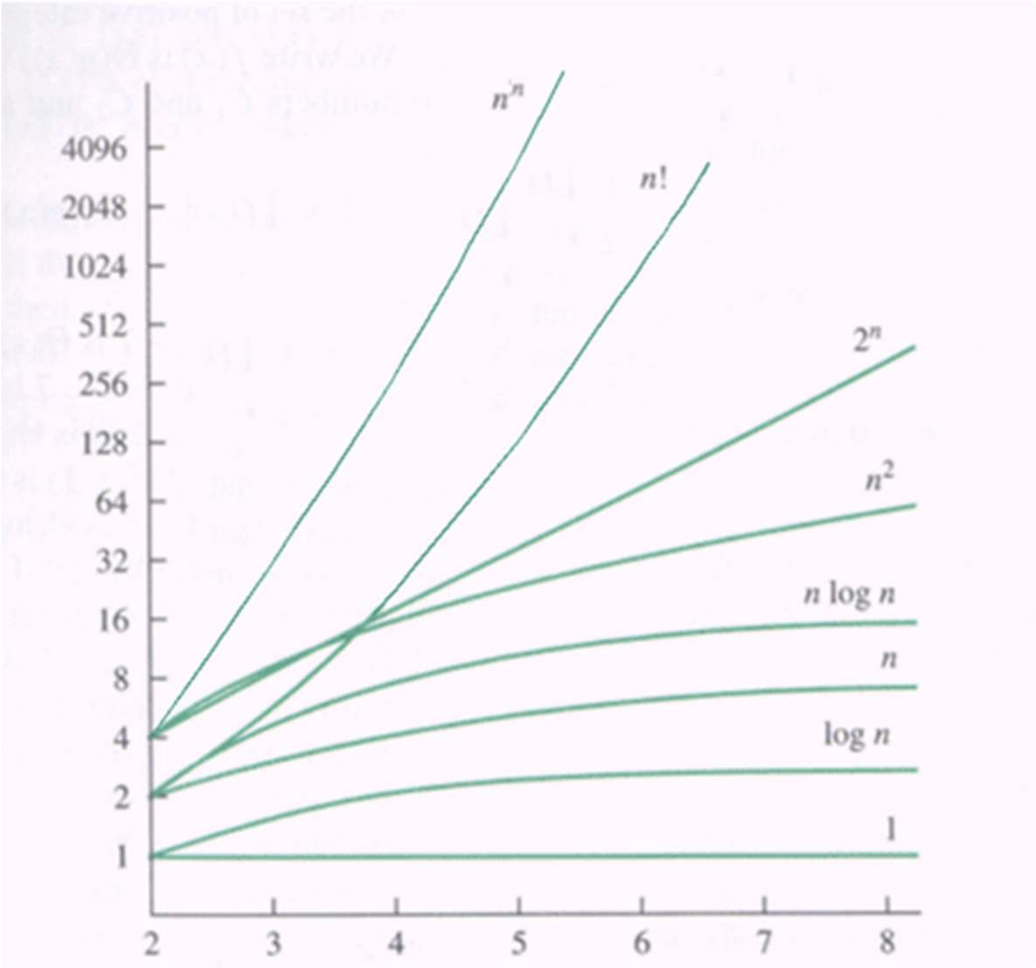


$$A(t) = 1000t$$

$$B(t) = 100t + 20t^2$$

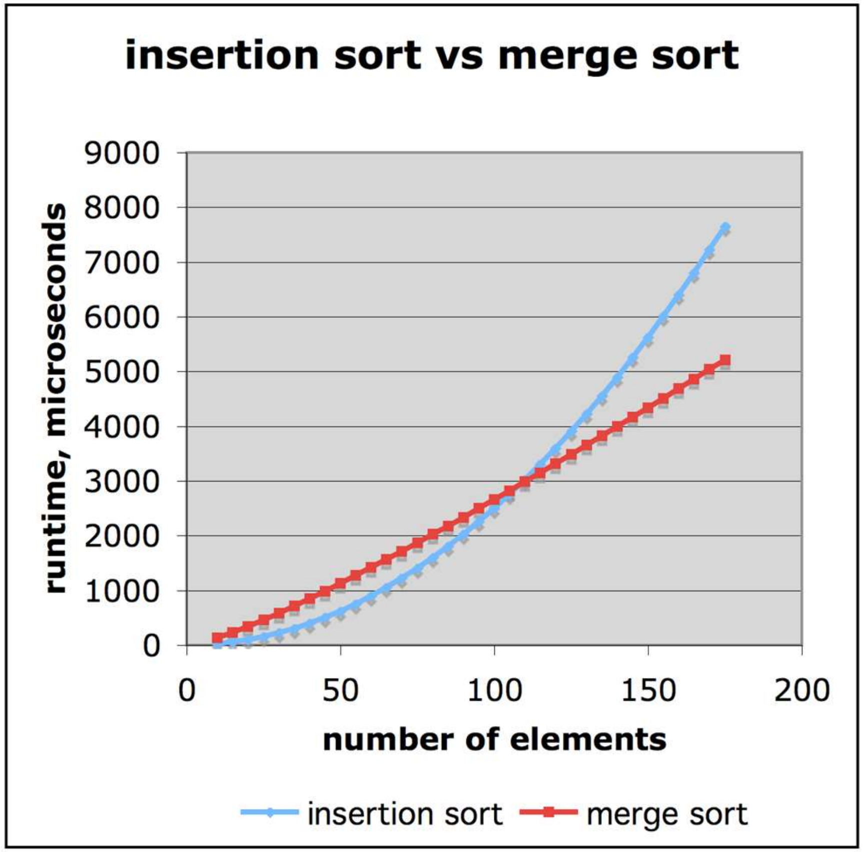
$$C(t) = 25t^2$$

Plot of Functions



Picture credit: https://www.cs.odu.edu/~cs381/cs381content/function/growth.html

Comparison of Two Algorithms



insertion sort is

$n^2 / 4$

merge sort is

$2 \ n \lg n$

sort a million items?

insertion sort takes roughly 70 hours

while

merge sort takes roughly 40 seconds

This is a slow machine, but if 100x as fast then it's 40 minutes versus less than 0.5 seconds

Stallman

Run time for 1billion steps / sec computer

n	n	$n\log_2 n$	n^2	n^3	n^4	n^{10}	2^n
10	.01 μs	.03 μs	.1 μs	1 μs	10 μs	10 s	1 μs
20	.02 μs	.09 μs	.4 μs	8 μs	160 μs	2.84 h	1 ms
30	.03 μ	.15 μ	.9 μ	27 μ	810 μ	6.83 d	1 s
40	.04 μs	.21 μs	1.6 μs	64 μs	2.56 ms	121 d	18 m
50	.05 μs	.28 μs	2.5 μs	125 μs	6.25 ms	3.1 y	13 d
100	.10 μs	.66 μs	10 μs	1 ms	100 ms	3171 y	$4*10^{13}$ y
10^3	1 μs	9.96 μs	1 ms	1 s	16.67 m	$3.17*10^{13}$ y	$32*10^{283}$ y
10^4	10 μs	130 μs	100 ms	16.67 m	115.7 d	$3.17*10^{23}$ y	
10^5	100 μs	1.66 ms	10 s	11.57 d	3171 y	$3.17*10^{33}$ y	
10^6	1 ms	19.92 ms	16.67 m	31.71 y	$3.17*10^7$ y	$3.17*10^{43}$ y	

μs = microsecond = 10^{-6} seconds; ms = milliseconds = 10^{-3} seconds
s = seconds; m = minutes; h = hours; d = days; y = years

ULSAN NATIONAL INSTITUTE OF SCIENCE AND TECHNOLOGY

Performance Measurement

- Use timer function
  - CPU time only for serial program
  - Wall-clock time for parallel program
- total = start – stop
- Timer has accuracy limitation
  - 1/100 sec, etc
- Multiple runs and average gives accurate time

Timing in C

```
clock_t start, stop;

start = clock(); /* set start to current time in
                    hundredths of a second */

/* code to be timed comes here */

stop = clock(); /* set stop to current time */

runTime = (double) (stop – start)/
                        CLOCKS_PER_SEC;
```

Multiple Runs

```
do {
      counter++;
      start = clock();
      doSomething();
      stop = clock();
      elapsedTime += stop – start;
   } while (elapsedTime < 1000)
elapsedTime /= counter;
```

Problem?

ULSAN NATIONAL INSTITUTE OF SCIENCE AND TECHNOLOGY

Questions?