CSE221

Lecture 5: Performance analysis

Hyungon Moon

Slide credits: The textbook authors, Won-ki Jeong, Tsz-Chiu Au, and Myeongjae Jeon

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

- Paper and pencil – theoretical analysis
- Do not need computer or program
- Criteria: Complexity
  - Space complexity
  - Time complexity
- Instance characteristics (size of inputs and outputs) affect complexity
  - Complexity is a function of characteristics, e.g., (f(n)) where n is input size

Space Complexity

- S(P) = c + Sp(inst. char.)
- Fixed part : c
  - Independent of input/output characteristics
  - Space for instruction, simple variables, constants, etc
- Variable part : Sp(inst. char.)
  - Space for variables whose size is dependent on the input/output (e.g. list length).

Since c is constant, we only focus on Sp(inst. char.) for space complexity!

Example

```
void 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. Sp?

Example

```
void 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. Sp = 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. Sp?

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. Sp = 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. Sp ?

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. Sp = A(n+1)
   : A = stack frame size

Time Complexity

- T(P) = compile time + run time
- Compile time is independent of instance characteristics - ignore
- Actual runtime can only be measured with executions
  - Compiler / machine / runtime dependent
  - Difficult to derive exact time for operators
- Use a program step instead

Primitive Operations (Program Steps)

- A segment of program independent of instance characteristics
  - Corresponds to low-level instruction with constant exec. time
- 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;
     }
```

Total count: 2n + 3

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

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

Total count: 2n + 2

T(P) = Rsum(n-1) + 2 = Rsum(n-2) + 2 + 2 = ... = 2*n + Rsum(0) = 2*n+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)
{
  for(int i=0; i<m; i++)
  {
    for(int j=0; j<n; j++)
    {
      c[i][j] = a[i][j]+b[i][j];
    }
  }
}
```

1 *(m + 1)

1 *(n + 1) *m

1 *n *m

Total count: 2nm + 2m + 1

Best, Worst, Average Step Count

- What if expressions depends 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)

Comparing Step Count

- P_A : 2n + 3,
- P_B : 2n + 2

- $P_A > P_B$

- Is P_B faster than P_A?
  - Yes and No, because the running time of each step might be different in P_A and P_B

Comparing Step Count

- Meaning of step count
  - 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

- We need an approximate yet intuitive means to compare models (functions) as problem size N vary:
  - Model A is 'at most' as fast or as big as model B
  - Model A is 'at least' as fast or as big as model B
  - Model A is 'equal' in performance/size to model 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
    - *g* is an upper bound of *f*

Big-O

- $A(t) = O(t)$ ("at most" or "of the order t"),
- $B(t) = O(t^2)$ ("at most" or "of the order t^2"),

- Informally, big-Oh can be used to identify the simplest function that bounds (above) a more complex function, as the parameter gets (asymptotically) bigger

Examples

- $3n+2 = O(n)$?

Examples

- $1000n^2+100n-6 = O(n^2)$?

Examples

- $6*2^n+n^2 = O(2^n)$?

Examples

- How good the bound is?
  - $3n+2 = O(n) = O(n^2)$?

Examples

- Highest order term dominates

$$f(n) = a_m n^m + a_{m-1} n^{m-1} + \ldots + a_1 n + a_0 = O(n^m)$$

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 $x \geq x_0$

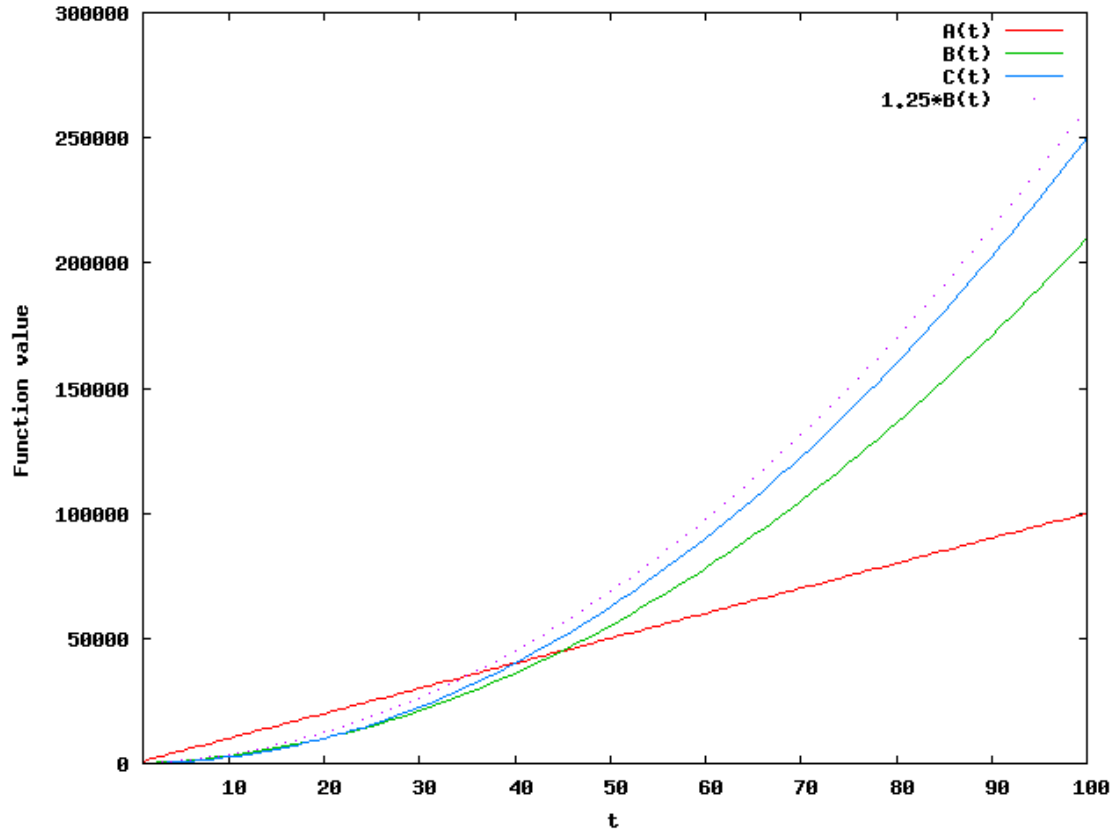
- Theta (Θ) "equals" or "goes as":

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

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

Graph

- For $t > 45$, $B(t)$ is always greater than $A(t)$: ?



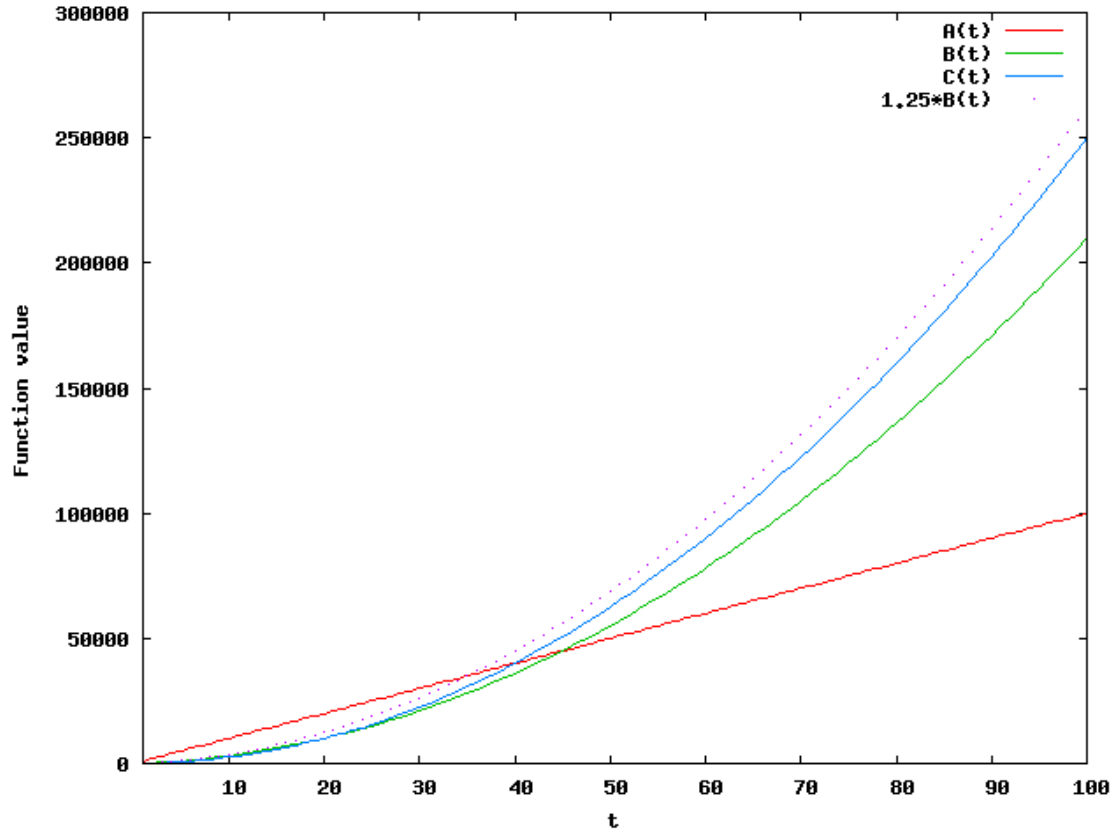
$$A(t) = 1000t$$

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

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

Graph

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



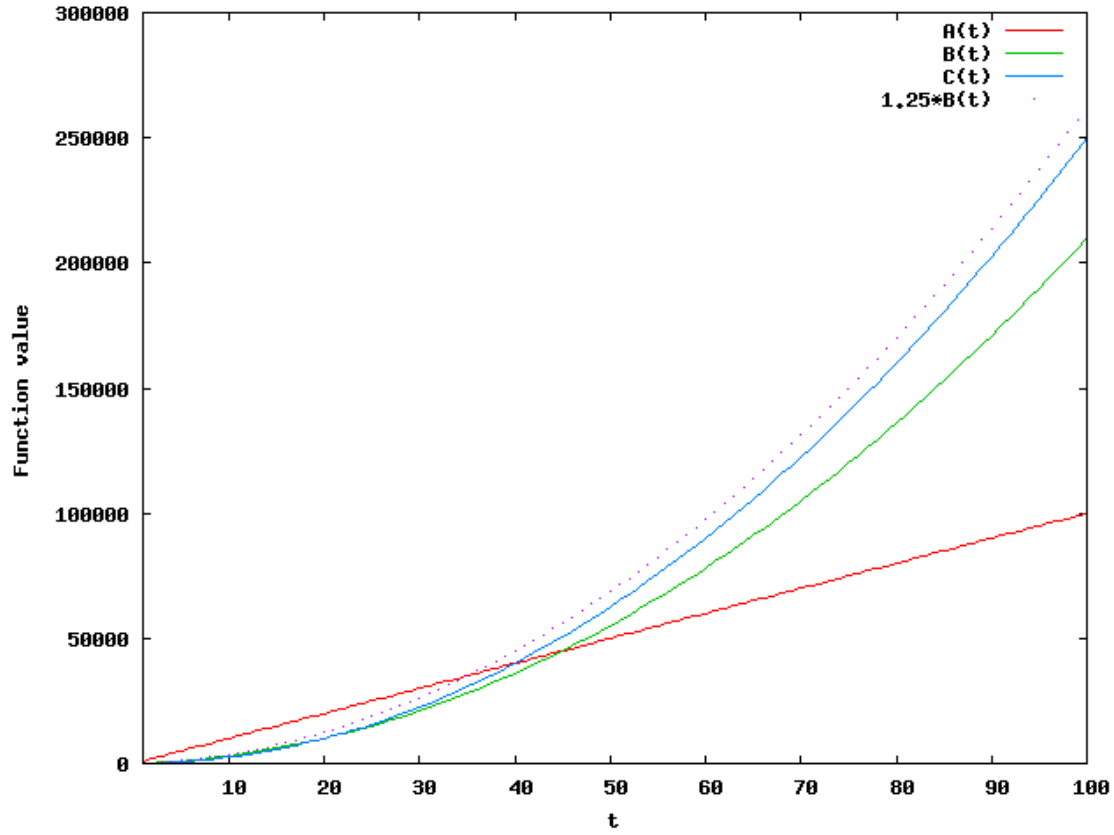
$$A(t) = 1000t$$

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

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

Graph

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



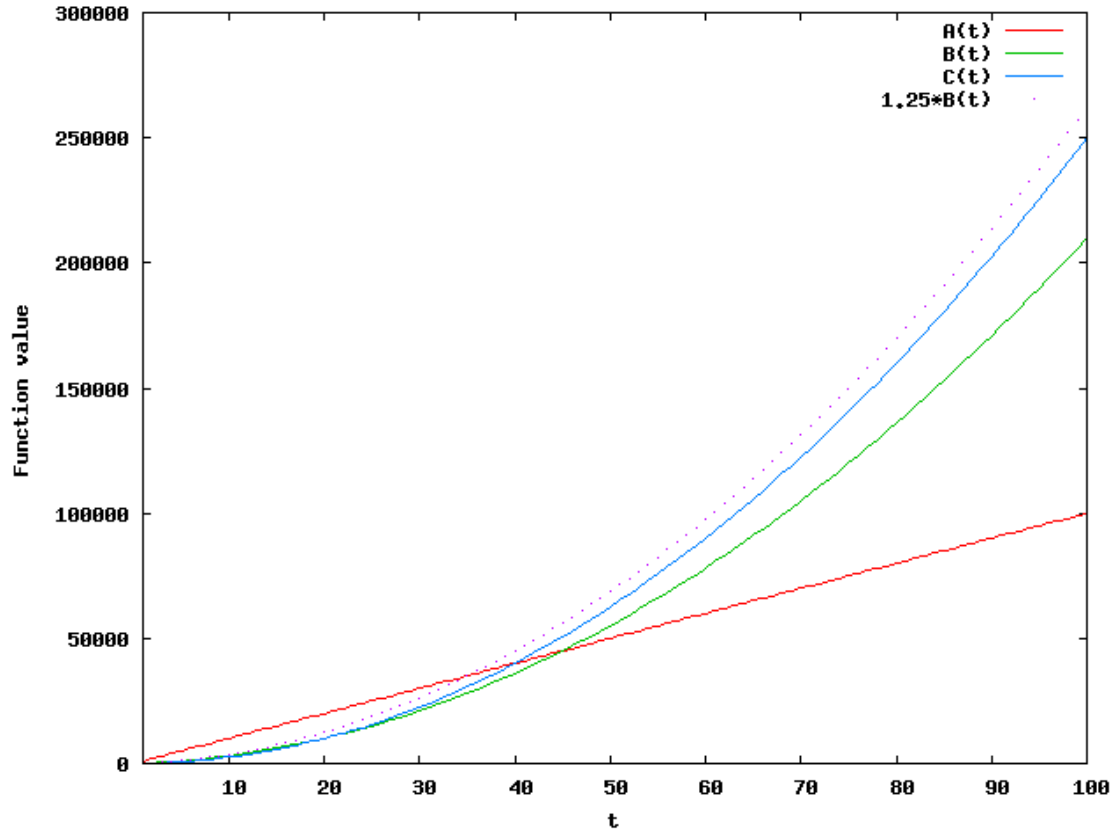
$$A(t) = 1000t$$

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

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

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) : ?*



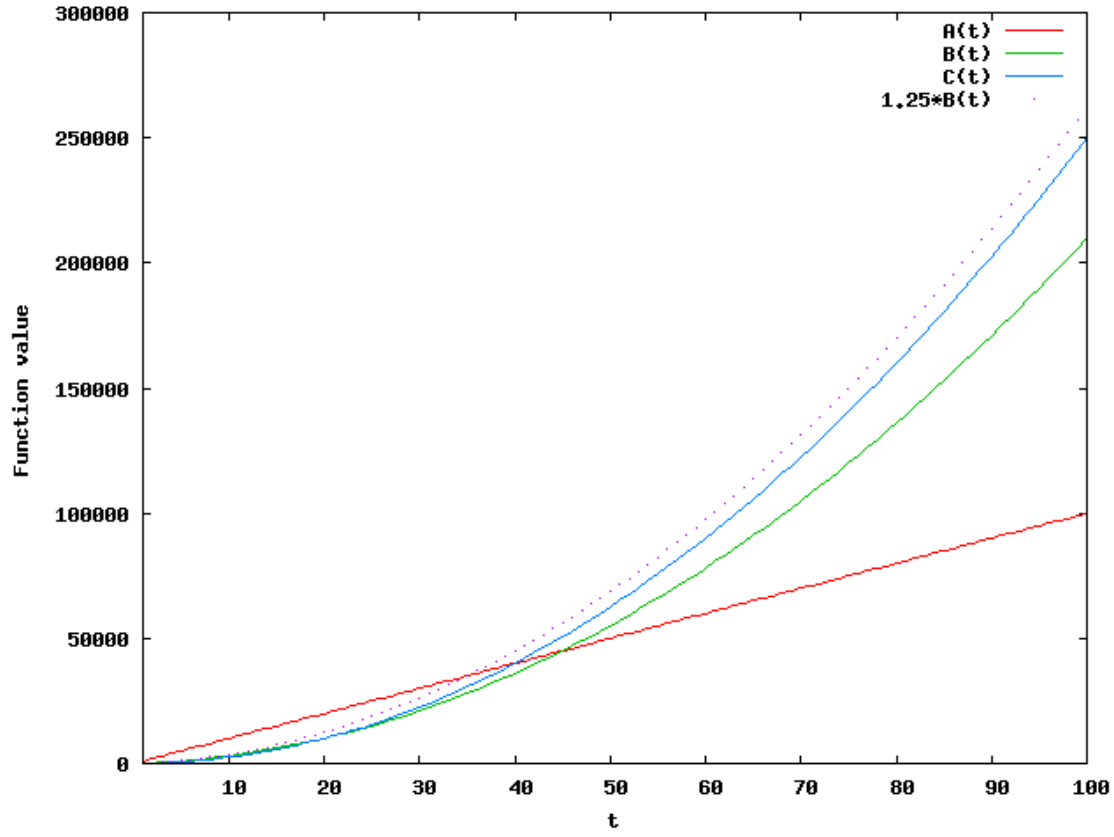
$$A(t) = 1000t$$

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

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

Graph

- 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 $C(t)$: $C=O(B)$



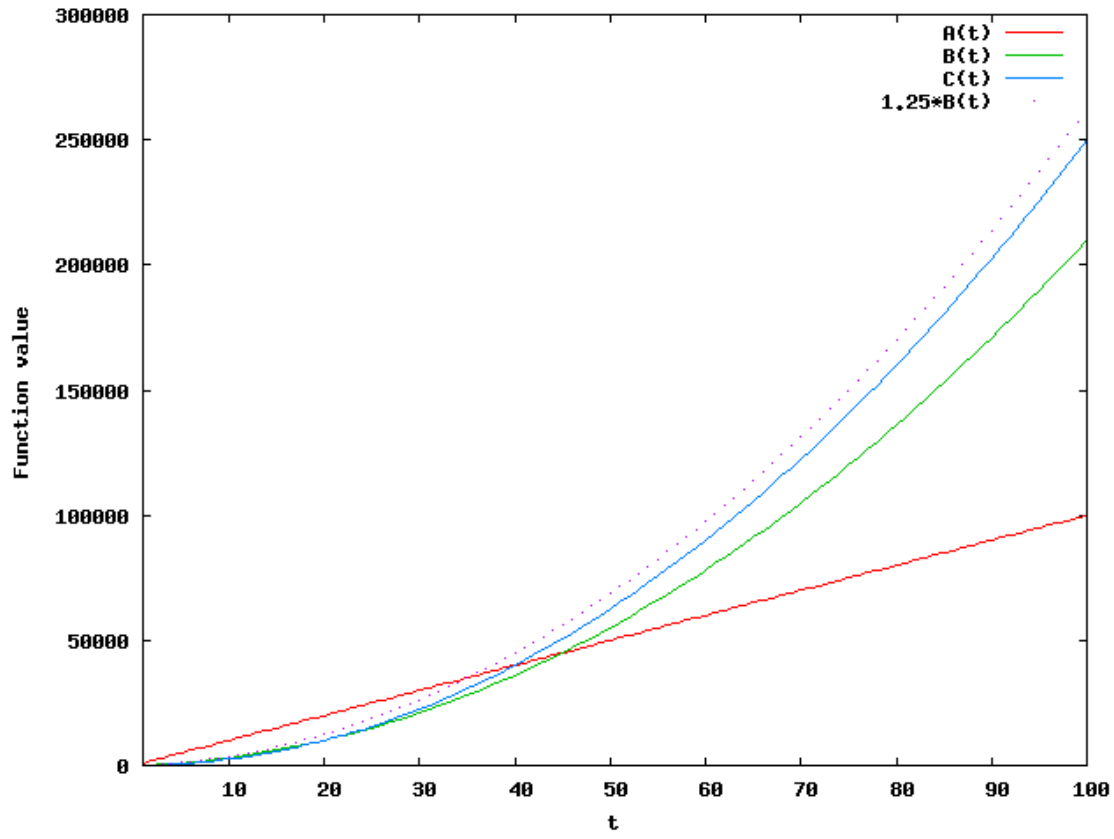
$$A(t) = 1000t$$

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

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

Graph

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



$$A(t) = 1000t$$

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

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

Asymptotic Complexity

- Order of magnitude
  - 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

Plot

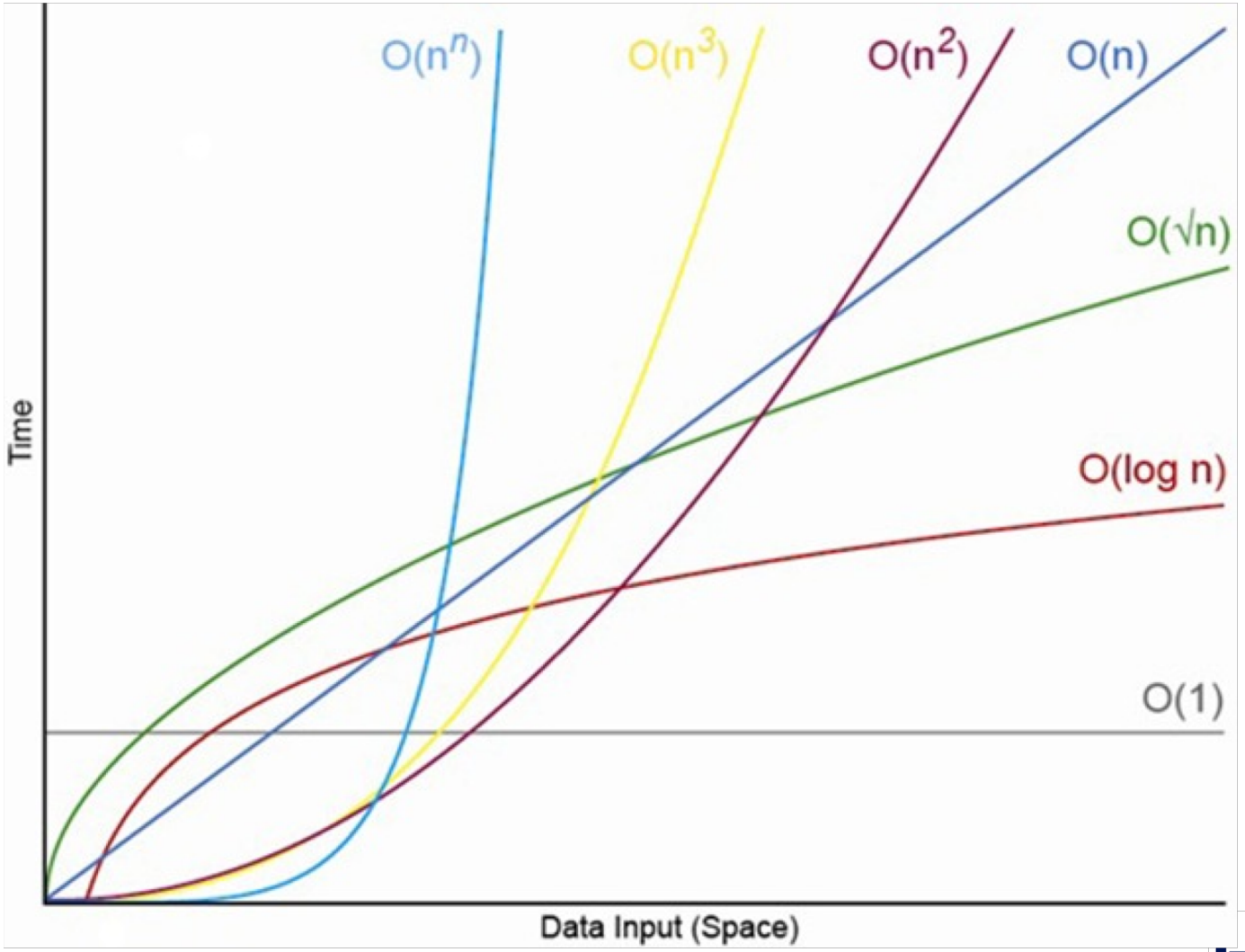
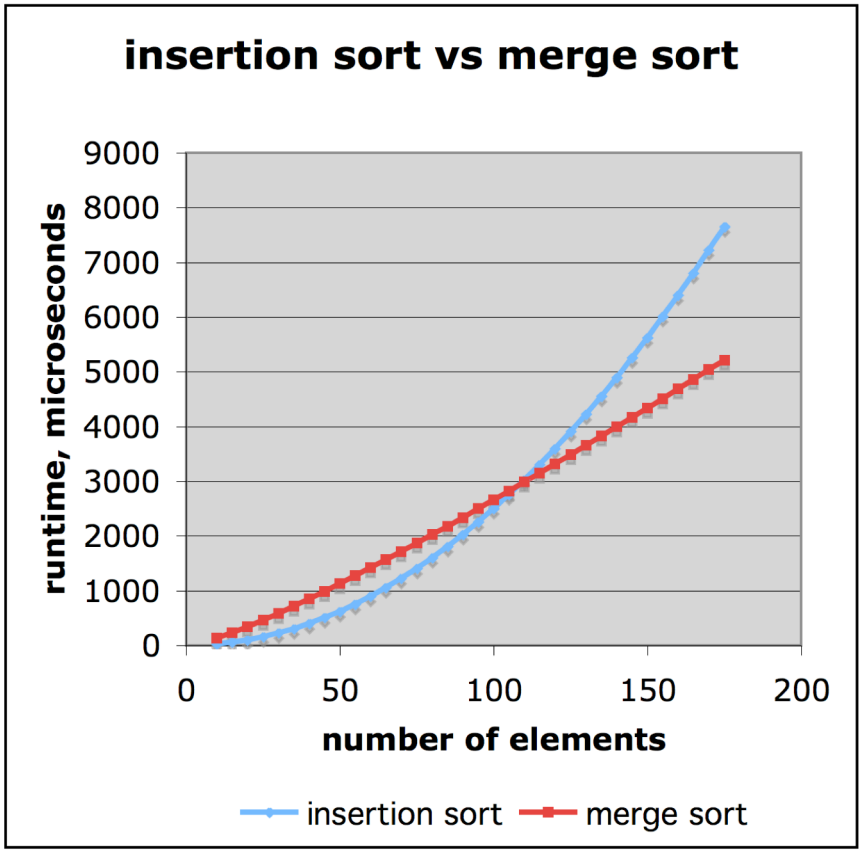


Image Courtesy of Yaacov Apelbaum

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 100 x as fast then it's 40 minutes versus less than 0.5 seconds

Runing time for 1-billion-steps-per-second computer

	$f(n)$						
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 μs	.15 μs	.9 μs	27 μs	810 μs	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

Element Uniqueness Problem (iterative)

```
bool isUniqueLoop(const vector<int>& arr, int start, int end) {
    if (start >= end) return true;
    for (int i = start; i < end; i++)
      for (int j = i+1; j <= end; j++)
        if (arr[i] == arr[j]) return false;
    return true;
  }
```

Element Uniqueness Problem (recursion)

```
bool isUnique(const vector<int>& arr, int start, int end) {
    if (start >= end) return true;
    if (!isUnique(arr, start, end-1))
      return false;
    if (!isUnique(arr, start+1, end))
      return false;
    return (arr[start] != arr[end]);
  }
```

Element Uniqueness Problem (sort)

```
bool isUniqueSort(const vector<int>& arr, int start, int end) {
    if (start >= end) return true;
    vector<int> buf(arr);                          // duplicate copy of arr
    sort(buf.begin()+start, buf.begin()+end); // sort the subarray
    for (int i = start; i < end; i++)          // check for duplicates
      if (buf[i] == buf[i+1]) return false;
    return true;
  }
```

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
- Generate data for performance-specific

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;
```

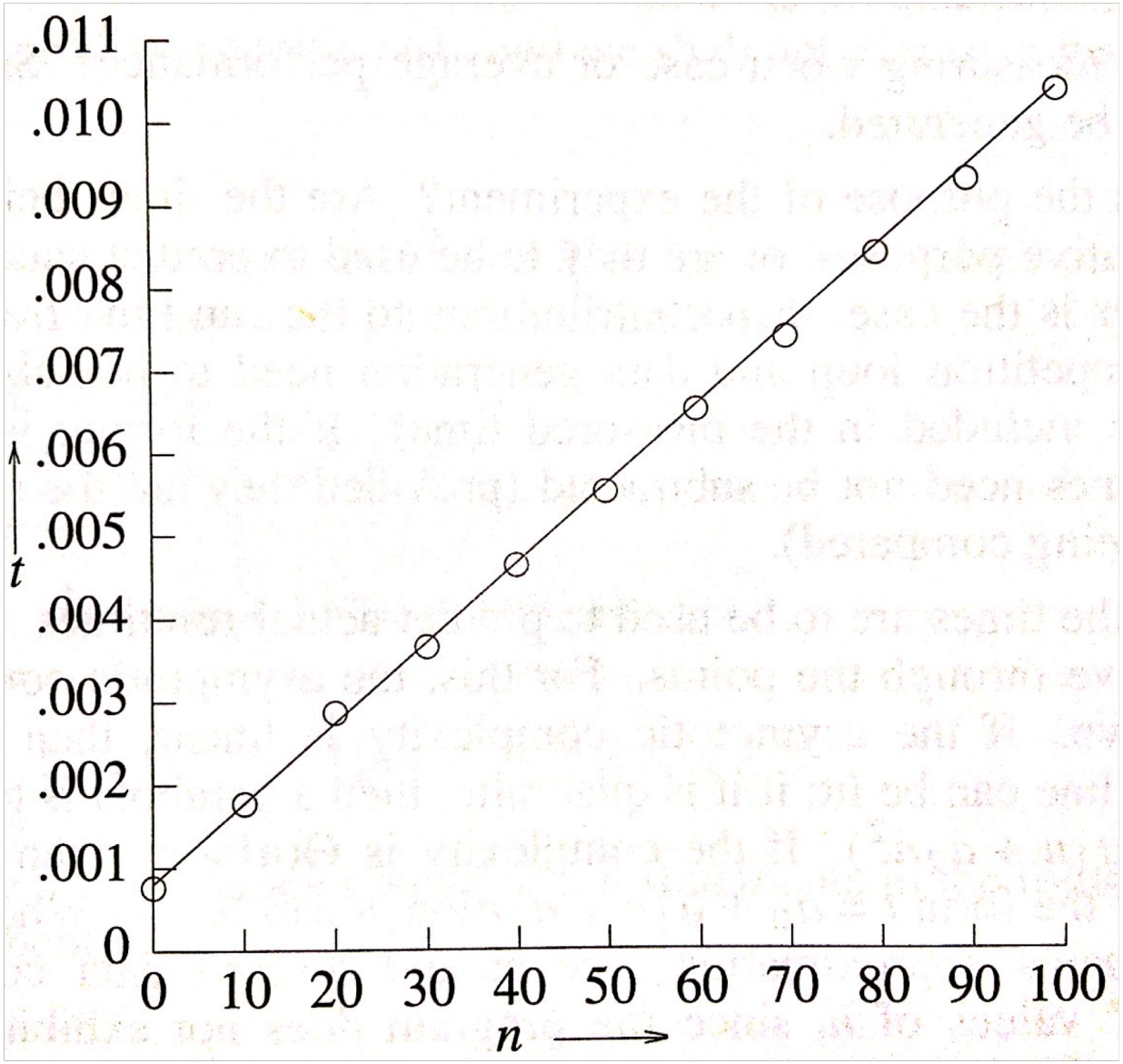
Multiple Runs: Problem?

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

```
void TimeSearch() {
   int a[1001], n[20];
   const long r[20] = { 300000, 300000, 200000, 200000, 100000, 100000, 100000, 80000,
        80000,50000, 50000, 25000,15000, 15000,10000, 7500, 7000, 6000, 5000, 5000 };

   for (int j = 1; j <= 1000; j++)
        a[j] = j;

   for (j = 0; j < 10; j++) {
       n[j] = 10 * j; n[j+10] = 100 * (j+1);
   }
  cout << " n totalTime runTime" << endl;

  for (j = 0; j < 20; j++) { // measure running time
      long start, stop;
      time(start); // start clock
      for (long b = 1; b <= r[j]; b++) {
           int k = seqsearch(a, n[j], 0);
      }
      time(stop); // stop clock
      long totalTime = stop - start;
      float runTime = (float)(totalTime)/(float)(r[j]);
      cout << " " << n[j] << " " << totalTime << " " << runTime << endl;
  }
  cout << "Times are in hundredths of a second." << endl;
}
```

n[ ] : data size

r[ ] : # of iterations to measure average running time



Questions?