Unit 1.3 Analysis II

Algorithms

EE/NTHU

Mar. 7, 2018

Asymptotic Notations

- Computational complexities are usually denoted using the following notations.

Definition 1.3.1. Big O.

The function $f(n) = \mathcal{O}(g(n))$ if and only if there are positive constants c and n_0 such that $f(n) \leq c \cdot g(n)$ for all n, $n \geq n_0$.

- Examples
  - $3n + 2 = \mathcal{O}(n)$,
  - $1000n^2 + 100n - 6 = \mathcal{O}(n^2)$,
  - $6 \cdot 2^n + n^2 = \mathcal{O}(2^n)$.
- $\mathcal{O}(1)$ means the complexity is constant.
- $\mathcal{O}(n)$ is called linear.
- $\mathcal{O}(n^2)$ is called quadratic.
- $\mathcal{O}(n^3)$ is called cubic.
- $\mathcal{O}(2^n)$ is called exponential.
- The following complexities are seen more often: $\mathcal{O}(1)$, $\mathcal{O}(\lg n)$, $\mathcal{O}(n)$, $\mathcal{O}(n \lg n)$, $\mathcal{O}(n^2)$, $\mathcal{O}(n^3)$, $\mathcal{O}(2^n)$.

Asymptotic Notations, II

Theorem 1.3.2.

If $f(n) = a_m n^m + \cdots + a_1 n + a_0$, then $f(n) = \mathcal{O}(n^m)$.

Proof.

$$
\begin{aligned}
f(n) \leq & \sum_{i=0}^{m} |a_i| n^i \\
= & n^m \sum_{i=0}^{m} |a_i| n^{i-m} \\
\leq & n^m \sum_{i=0}^{m} |a_i| \qquad \text{for } n \geq 1.
\end{aligned}
$$

Therefore, $f(n) \leq cn^m$ for $n \geq 1$ and $c = \sum |a_i|$,
and by definition, $f(n) = \mathcal{O}(n^m)$. □

Asymptotic Notations, III

Definition 1.3.3. Omega.

The function $f(n) = \Omega(g(n))$ if and only if there are positive constants c and n_0 such that $f(n) \geq c \cdot g(n)$ for all n, $n \geq n_0$.

- $f(n)$ is bounded below by $g(n)$.
- Example
  - $3n + 2 \geq 3n$ for $n \geq 0$, thus $3n + 2 = \Omega(n)$,
  - $10n^2 + 4n + 2 \geq 10n^2$ for $n \geq 0$, thus $10n^2 + 4n + 2 = \Omega(n^2)$,
  - $6 \cdot 2^n + n^2 = \Omega(2^n)$.
- Note that $10n^2 + 4n + 2 = \Omega(n)$ as well, but it is less informative to write so.
- Thus, we usually take the highest order $g(n)$ in this notation.

Theorem 1.3.4.

If $f(n) = a_m n^m + \cdots + a_1 n + a_0$ and $a_m > 0$, then $f(n) = \Omega(n^m)$.

Asymptotic Notations, IV

Definition 1.3.5. Theta.

The function $f(n) = \Theta(g(n))$ if and only if there are positive constants c_1, c_2 and n_0 such that $c_1 \cdot g(n) \le f(n) \le c_2 \cdot g(n)$ for all $n \ge n_0$.

- $f(n) = \Theta(g(n))$ if and only if $g(n)$ is both an upper and lower bound on $f(n)$.
- Example
  - $3n + 2 = \Theta(n)$,
  - $10n^2 + 4n + 2 = \Theta(n^2)$,
  - $6 \cdot 2^n + n^2 = \Theta(2^n)$.
  - $10 \lg n + 4 = \Theta(\lg n)$.

Theorem 1.3.6.

Given two functions $f(n)$ and $g(n)$, we have $f(n) = \Theta(g(n))$ if and only if $f(n) = \mathcal{O}(g(n))$ and $f(n) = \Omega(g(n))$.

Theorem 1.3.7.

If $f(n) = a_m n^m + \cdots + a_1 n + a_0$ and $a_m > 0$, then $f(n) = \Theta(n^m)$.

Asymptotic Notations, V

Definition 1.3.8. Little o.

The function $f(n) = o(g(n))$ if and only if

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0. \qquad (1.3.1)$$

- Example
  - $3n + 2 = o(n^2)$,
  - $3n + 2 = o(n \lg n)$,
  - $3n + 2 = o(n \lg \lg n)$,
  - $6 \cdot 2^n + n^2 = o(3^n)$,
  - $6 \cdot 2^n + n^2 = o(2^n \lg n)$,

Definition 1.3.9. Little omega.

The function $f(n) = \omega(g(n))$ if and only if

$$\lim_{n\to\infty} \frac{g(n)}{f(n)} = 0. \qquad (1.3.2)$$

Properties of Asymptotic Notations

- The following properties hold for asymptotic notations.
- Transitivity:

$$
\begin{array}{llll}
f(n) = \Theta(g(n)) & \text{and} & g(n) = \Theta(h(n)) & \text{then} \quad f(n) = \Theta(h(n)), \\
f(n) = \mathcal{O}(g(n)) & \text{and} & g(n) = \mathcal{O}(h(n)) & \text{then} \quad f(n) = \mathcal{O}(h(n)), \\
f(n) = \Omega(g(n)) & \text{and} & g(n) = \Omega(h(n)) & \text{then} \quad f(n) = \Omega(h(n)), \\
f(n) = o(g(n)) & \text{and} & g(n) = o(h(n)) & \text{then} \quad f(n) = o(h(n)), \\
f(n) = \omega(g(n)) & \text{and} & g(n) = \omega(h(n)) & \text{then} \quad f(n) = \omega(h(n)).
\end{array}
$$

- Reflexivity:

$$
\begin{aligned}
f(n) &= \Theta(f(n)), \\
f(n) &= \mathcal{O}(f(n)), \\
f(n) &= \Omega(f(n)).
\end{aligned}
$$

Properties of Asymptotic Notations, II

- Symmetry:

$$f(n) = \Theta(g(n)) \text{ if and only if } g(n) = \Theta(f(n)).$$

- Transpose symmetry:

$$
\begin{array}{lll}
f(n) = \mathcal{O}(g(n)) & \text{if and only if} & g(n) = \Omega(f(n)), \\
f(n) = o(g(n)) & \text{if and only if} & g(n) = \omega(f(n)).
\end{array}
$$

Definition 1.3.10. Asymptotic Comparisons.

Given two functions $f(n)$ and $g(n)$, $f(n)$ is asymptotically smaller than $g(n)$ if

$$f(n) = o(g(n)). \tag{1.3.3}$$

And $f(n)$ is asymptotically larger then $g(n)$ if

$$f(n) = \omega(g(n)). \tag{1.3.4}$$

Complexity in Asymptotic Notations

- These notations can be applied to asymptotic complexity analysis.

Statement	s/e	freq.	Total steps
1 **Algorithm** **Sum** (A, n)	0	–	0
2 // Simple summation.			
3 {	0	–	0
4 $Sum := 0$;	1	1	$\Theta(1)$
5 **for** $i := 1$ **to** n **do**	1	$n+1$	$\Theta(n)$
6 $Sum := Sum + A[i]$;	1	n	$\Theta(n)$
7 **return** Sum;	1	1	$\Theta(1)$
8 }	0	–	0
Total			$\Theta(n)$

- Some details in calculating the exact execution steps can be ignored using these notations.

Complexity in Asymptotic Notations, II

- Another example

Statement	s/e	freq.	total steps
1 **Algorithm** **MAdd** (A, B, C, m, n)	0	–	0
2 // $C := A + B$, all are $m \times n$ matrices.			
3 {	0	–	0
4 **for** $i := 1$ **to** m **do**	1	$\Theta(m)$	$\Theta(m)$
5 **for** $j := 1$ **to** n **do**	1	$\Theta(mn)$	$\Theta(mn)$
6 $C[i,j] := A[i,j] + B[i,j]$;	1	$\Theta(mn)$	$\Theta(mn)$
7 }	0	–	0
Total			$\Theta(mn)$

Power Function

- To calculate x^n, where $n \geq 0$ is an integer.

Algorithm 1.3.11. Power

```
Algorithm Pow1(x, n)
// Return x^n for an integer n ≥ 0.
{
      power := 1;
      for i := 1 to n do {
            power := power × x;
      }
      return power;
}
```

- This algorithm has computational complexity of $\mathcal{O}(n)$.

Power Function, II

Algorithm 1.3.12. Power – Improved

```
Algorithm Pow2(x, n)
// Return x^n for an integer n ≥ 0.
{
      m := n; power := 1;
      while (m > 0) do {
            z := x;
            while (m mod 2 = 0) do {
                  m := m/2; z := z × z;
            }
            m := m − 1; power := power × z;
      }
      return power;
}
```

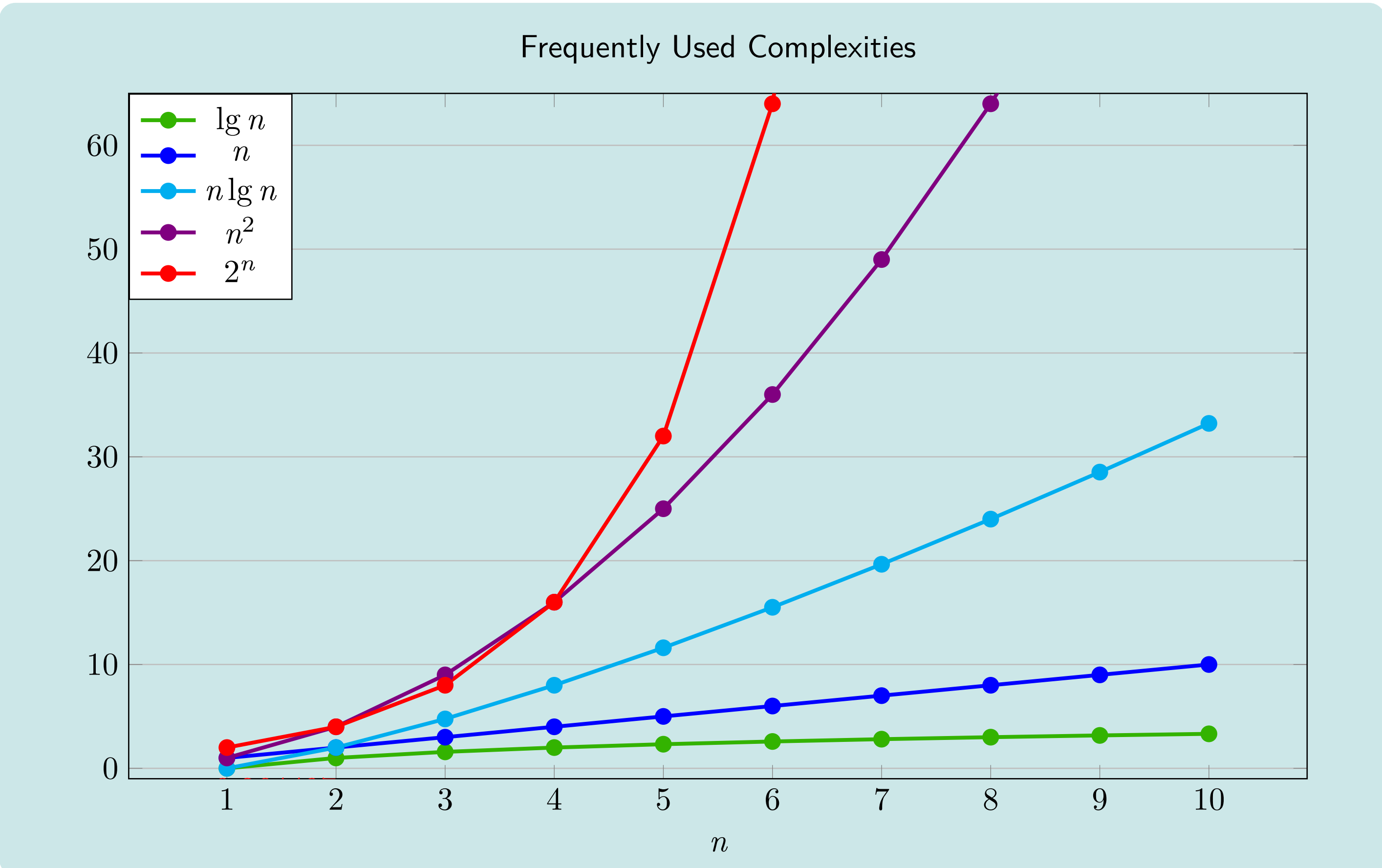
- This algorithm has computational complexity of $\mathcal{O}(\lg n)$.
- Asymptotic analysis enables comparison of different algorithms.

Comparing Algorithms

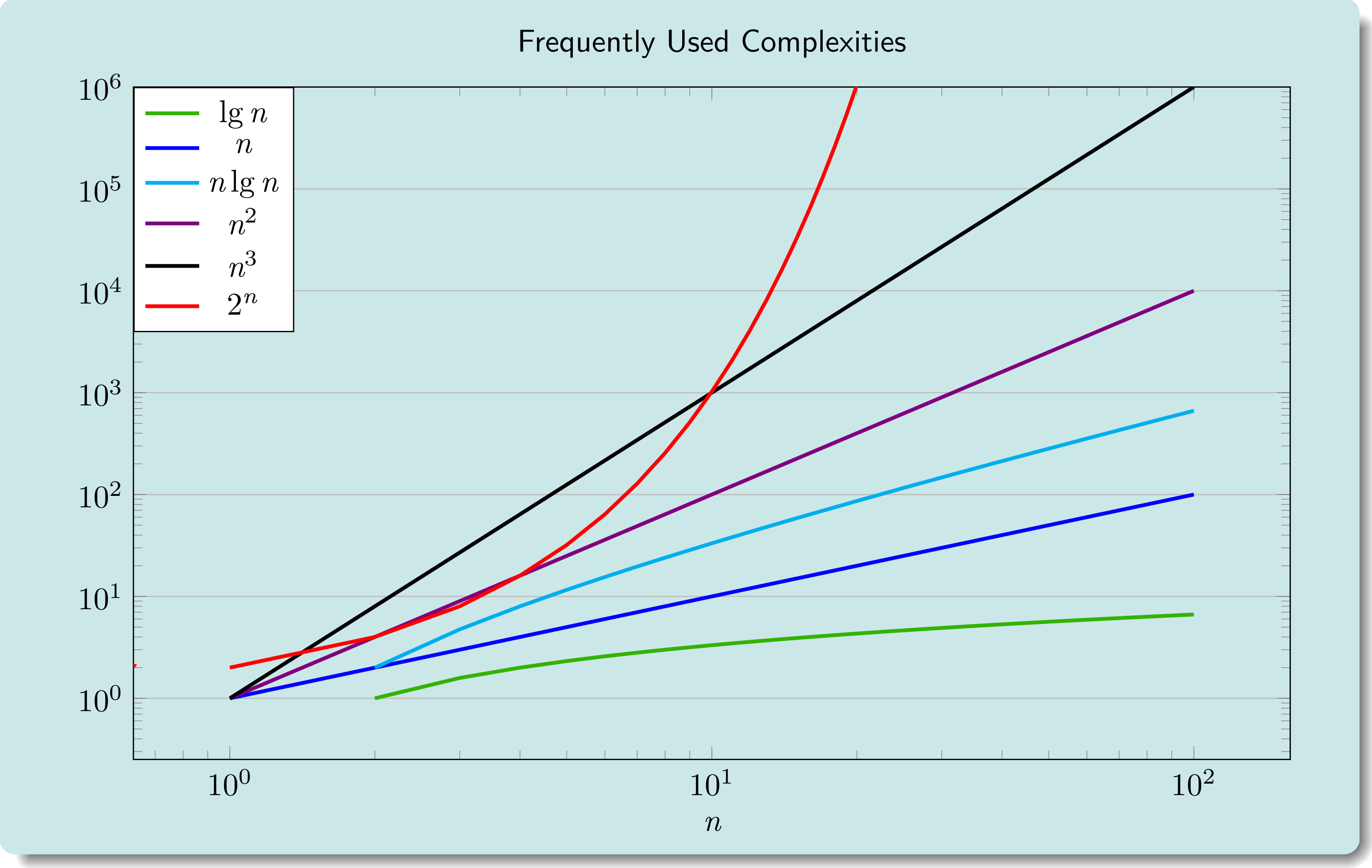
- It can be shown that `Pow1` algorithm has the asymptotic computational complexity of $\Theta(n)$,
  $\exists c_1, c_2, n_1$, such that $c_1 \cdot n \le t_{\text{Pow1}} \le c_2 \cdot n$ for $n \ge n_1$.
- While `Pow2` algorithm is $\Theta(\lg n)$,
  $\exists d_1, d_2, n_2$, such that $d_1 \cdot \lg n \le t_{\text{Pow2}} \le d_2 \cdot \lg n$ for $n \ge n_2$.
- Since $\lg n < n$ for $n \ge 1$, $t_{\text{Pow2}} < t_{\text{Pow1}}$ for $n > \max\{n_1, n_2\}$.
- Frequently used complexities

$\lg n$	n	$n \lg n$	n^2	n^3	2^n
0	1	0	1	1	2
1	2	2	4	8	4
2	4	8	16	64	16
3	8	24	64	512	256
4	16	64	256	4,096	65,536
5	32	160	1,024	32,768	4,294,967,296

Comparing Algorithms, II



Comparing Algorithms, III



Comparing Algorithms, IV

n	t(n)	t($n \lg n$)	t(n^2)	t(n^3)	t(n^4)	t(n^{10})	t(2^n)
10	0.01 μs	0.0332 μs	0.1 μs	1 μs	10 μs	10 s	1.02 μs
20	0.02 μs	0.0864 μs	0.4 μs	8 μs	160 μs	2.84 h	1.05 ms
30	0.03 μs	0.147 μs	0.9 μs	27 μs	810 μs	6.83 d	1.07 s
40	0.04 μs	0.213 μs	1.6 μs	64 μs	2.56 ms	121 d	18.3 m
50	0.05 μs	0.282 μs	2.5 μs	125 μs	6.25 ms	3.1 y	13 d
100	0.1 μs	0.664 μs	10 μs	1 ms	100 ms	3171 y	4.02×10^{13} y
10^3	1 μs	9.97 μs	1 ms	1 s	16.7 m	3.17×10^{13} y	3.4×10^{284} y
10^4	10 μs	133 μs	100 ms	16.7 m	116 d	3.17×10^{23} y	
10^5	100 μs	1.66 ms	10 s	11.6 d	3171 y	3.17×10^{33} y	
10^6	1 ms	19.9 ms	16.7 m	31.7 y	3.17×10^7 y	3.17×10^{43} y	

Units: μs: 10^{-6} seconds; ms: 10^{-3} seconds; s: seconds; m: minutes; h: hours; d: days; y: years.

- Assuming 10^9 operations per second can be performed
- Higher complexity algorithms can not handle large amount of data.
- Improving computer operations has limited benefits.
- Algorithm's complexity is of critical importance for practical programming.

Performance Measurement

- The implemented algorithm can be measured on a computer.
- Run time (CPU time) is the focus.
  - Compilation time is ignored.
- Algorithms with short run time should be repeated a number of times for more accurate run time measurement.
- Example algorithm to be measured.

Algorithm 1.3.13. Sequential Search

```
Algorithm SeqSearch(A, x, n)
// Search for x in A[1 : n]. A[0] is used as additional space.
{
    i := n; A[0] := x;
    while (A[i] ≠ x) do i := i − 1 ; // Search backward.
    return i; // If not found, return 0.
}
```


Performance Measurement, II

Algorithm 1.3.14. Measuring Search Time

```
Algorithm TimeSearch()
// To measure search CPU time with repetitions.
{
    R[20] := { 2e7,2e7,1.5e7,1e7,1e7,1e7,5e6,5e6,5e6,5e6, // #Repetition
          5e6,5e6,5e6,5e6,5e6,5e6,2.5e6,2.5e6,2.5e6,2.5e6 } ;
    for j := 1 to 1000 do A[j] := j; // Init A[ ] = {1, 2, 3, . . . , 1000}.
    for j := 1 to 10 do { // Init N[ ] = {0, 10, 20, . . . , 90, 100, 200, . . . , 1000}
          N[j] := 10× (j − 1) ;
          N[j + 10] := 100× j;
    }
    for j := 2 to 20 do { // Set n to be N[2 : 20]
          h := GetTime() ;
          for i := 1 to R[j] do // Repeat R[j] times for each n.
                k := SeqSearch(A, 0, N[j]) ;
          h1 := GetTime() ;
          t1 := h1 − h; t := t1/R[j] ;
          write (N[j], t1, t) ; // Write total and average execution times.
    }
}
```

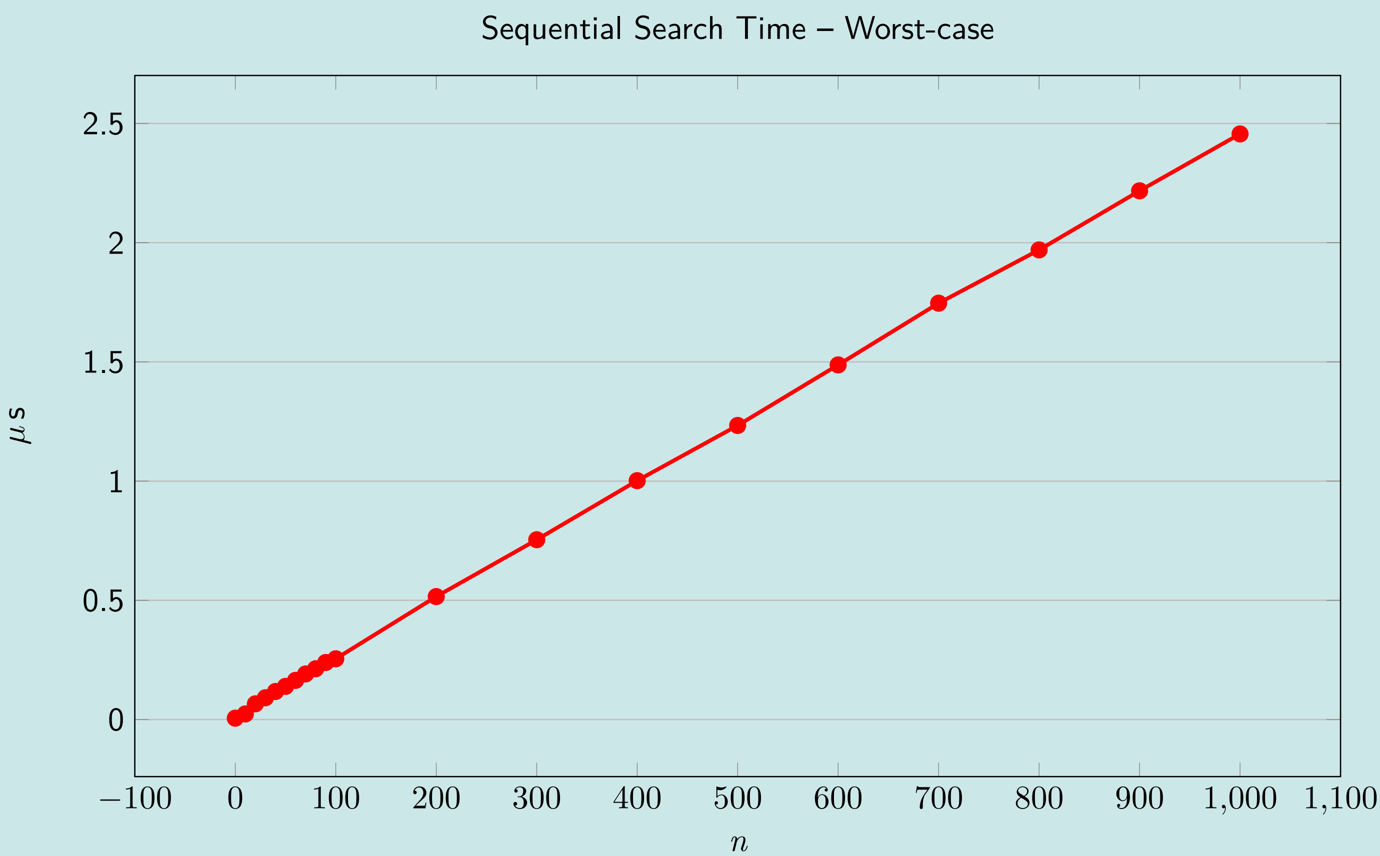
Performance Measurement, III

- The following function can get time of the day in seconds on linux systems.

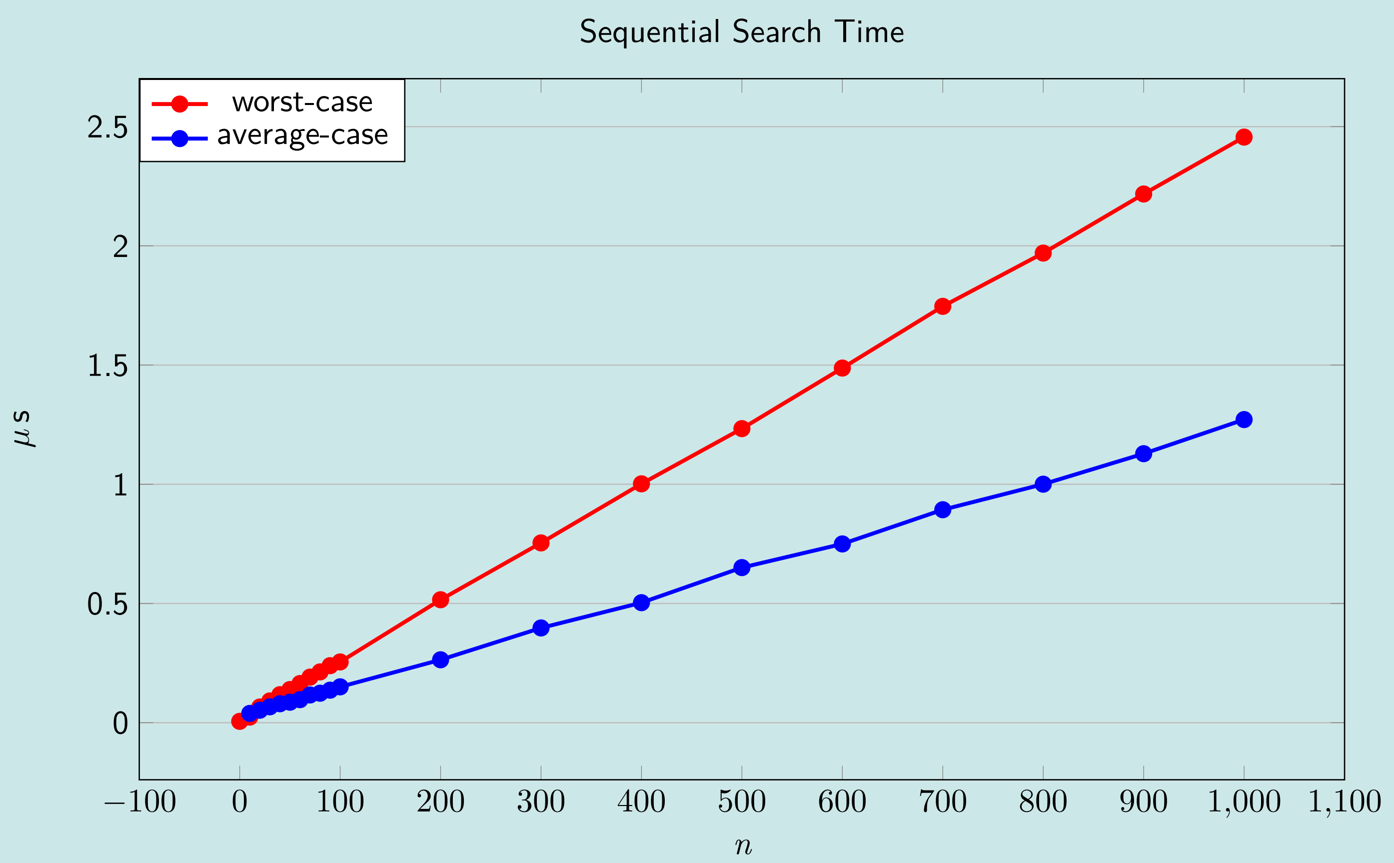
Function 1.3.15. Get Time of Day

```c
#include <sys/time.h>

double GetTime(void)
{
    struct timeval tv;

    gettimeofday(&tv,NULL);
    return tv.tv_sec+1e-6*tv.tv_usec;
}
```


Performance Measurement, IV



Performance Measurement, V



Performance Measurement, VI

- To measure the performance of an algorithm, the following factors should be considered
  - What is the resolution of the system clock?
  - What should be the number of repetitions for a meaningful measurement?
  - To measure worst-case or average-case performance?
  - For comparing two algorithms or to get the asymptotic complexity?
    - If the overhead in generating the test case should be deducted?
  - For asymptotic analysis, least square fit for larger values of n should be used to get the complexity.
- Worst-case analysis should generate test cases that for each n the maximum amount of CPU time will be taken.
  - Can be approximated by using random test cases and take the maximum of the run time given an n.
- Average-case analysis should generate all possible test cases and then take the average.
  - Similar random-input-test-case approach can be taken for a quick approximation.
- Best-case analysis is the minimum execution given the size n input.
- We are more interested in worst-case and average-case performance.

Summary

- Asymptotic notations.
  - $\mathcal{O}(f(n))$, $\Omega(f(n))$, $\Theta(f(n))$, $o(f(n))$, $\omega(f(n))$
- Some practical complexities.
- Performance measurement.
  - Worst-case performance
  - Average-case performance
  - Best-case performance