Data Structures and Algorithms

Lecture 3: Algorithm Analysis

Motivation

- Purpose: Understanding the resouce requirements of an algorithm
  - Time
  - Memory
- Runing time analysis estimates the time required of an algorithm as a function of the input size. (upper and lower bounds)
- Usages:
  - Estimate growth rate as input grows.
  - Guide to choose between alternative algorithms.

An example

- ```
  int sum(int set[], int n) {
  int temsum, i;
  tempsum = 1;      /* step/execution 1 */
  for (i=0; i<n; i++)      /* step/execution n+1 */
      tempsum +=set[i]; /* step/execution n */
  return tempsum;       /* step/execution 1 */
  }
  ```
- Input size: <u>n</u> (number of array elements)
- Total number of steps: <u>2*n + 3</u>
  ```

Algorithm Efficiency

- There are often many approaches (algorithms) to solve a problem. How do we choose between them?

- As the cores of computer program design, there are two (sometimes conflicting) goals.
  1. To design an algorithm that is easy to understand, code, debug.
  2. To design an algorithm that makes efficient use of the computer's resources.

Algorithm Efficiency (cont.)

- Goal (1) is the concern of Software Engineering.
- Goal (2) is the concern of data structures and algorithm analysis.
- When **goal (2)** is important, how do we measure an algorithm's cost?

Analysis and measurements

- Performance measurement (execution time): **machine dependent**.
- Performance analysis: **machine independent**.
- How do we analyze a program independent of a machine?
  - Counting the number steps.

How to Measure Efficiency?

- Empirical comparison (run programs)
- It is difficult to be `fair' due to:
  - Time consuming, especially when there are many alternative algorithms for a problem
  - Depend on your programming skills
    - One program may be finely tuned, while the other is not
  - Depend on the computers running algorithms
    - e.g., CPU, workload, etc.
  - May vary for different test cases
    - One program may favor some test cases

How to Measure Efficiency? (cont.)

- Analytical method: asymptotic algorithm analysis
- Critical resources, factors affecting running time
  - Running time, space (memory or disk)

For most algorithms, running time depends on "size" of the input.

Running time is expressed as **T**(n) for some function **T** on input size n.

How to Measure Efficiency? (cont.)

- Primary consideration when estimation an algorithm's performance is the number of **basic operations** required by the algorithm to process an input of a certain **size**.
  - **Basic operations**
    - The time for performing a basic operation does not depend on particular inputs
    - E.g., operations for +, -, X, /
  - **Size**
    - The number of inputs processed

Random Access Machine (RAM)

- To analyze the efficiency, we need an abstract machine model
- RAM
  - Each simple operation takes 1 time step
  - Loops and subroutines are not simple operations
  - Each memory access takes one time step, no shortage of memory

What does “size” exactly mean?

- Number of inputs → strong
  - Strongly polynomial time
- Input length (binary encoded) → weak
  - (Weakly) polynomial time
  - Most commonly adopted definition
- Input magnitudes → even weaker
  - Pseudo-polynomial time

Growth rate

- Growth rate: A program with O(f(n)) is said to have growth rate of f(n). It shows how fast the running time grows when *n* increases.

Growth rates illustrated

	n=1	n=2	n=4	n=8	n=16	n=32
$O(1)$	1	1	1	1	1	1
$O(logn)$	0	1	2	3	4	5
$O(n)$	1	2	4	8	16	32
$O(nlogn)$	0	2	8	24	64	160
$O(n^2)$	1	4	16	64	256	1024
$O(n^3)$,	1	8	64	512	4096	32768
$O(2^n)$	2	4	16	235	65536	4294967296

Exponential growth

- Say that you have a problem that, for an input consisting of n items, can be solved by going through 2^n cases
- You use Deep Blue, that analyses *200* million cases per second
  - Input with *15* items, *163* microseconds
  - Input with *30* items, *5.36* seconds
  - Input with *50* items, more than two months
  - Input with *80* items, *191* million years

Examples of Growth Rate

- Example 1, find the largest value in an array

```
// Find largest value
int largest(int array[], int n) {
  int currlarge = 0; // Largest value seen
  for (int i=0; i<n; i++) // For each val
    if (array[currlarge] < array[i])
      currlarge = i;        // Remember pos
  return currlarge;         // Return largest
}
```

c: the time for performing a comparison operation <, which varies for different computers

n: the number of < operations processed

T(n) = c n

Examples (cont.)

- Example 2: Assignment statement.

  $T(n) = c_1$

- Example 3:

```
sum = 0;
for (i=1; i<=n; i++)
  for (j=1; j<n; j++)
    sum++;
}
```

$T(n) = c_2 n^2$

The growth rate of a recursive algorithm

- Example 1: int Fact(int n){

```
 if (n ==0 ) return 1;
  return n * Fact(n-1);
}
```

- Denote by **T**(n) the time for computing Fact(n)
  - **T**(n) = **T**(n-1)+ c

    = **T**(n-2) + c + c = **T**(n-2) + 2c

    …

    =**T**(n-n) + nc= c(n+1)

Binary Search

Position	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Key	11	13	21	26	29	36	40	41	45	51	54	56	65	72	77	83

- How many elements are examined in the worst case?

Binary Search

```
// Return position of element in sorted
// array of size n with value K.
int binary(int array[], int l, int r, int K) {
   if( l==r ){
       if( array[r] == K ) return r;
       else                return -1; //not found
   }

   int m = (l+r)/2; // Check middle
   if (K <= array[m]) // Left half
       return binary( array, l, m, K);
   else               // Right half
       return binary( array, m+1, r, K);
}
```

The growth rate of a recursive algorithm (cont.)

- Binary search algorithm
- $\mathbf{T}(n) = c + \mathbf{T}(n/2)$

$$
\begin{aligned}
&= c + c + \mathbf{T}(n/4) \\
&= 2c + \mathbf{T}(n/2^2) \\
&= 3c + \mathbf{T}(n/2^3) \\
&\ldots \\
&= c \log n + \mathbf{T}(n/2^{\log n}) \\
&= c \log n + \mathbf{T}(n/n) \\
&= c (\log n + 1)
\end{aligned}
$$

The growth rate of a recursive algorithm (cont.)

- Hanoi Puzzle

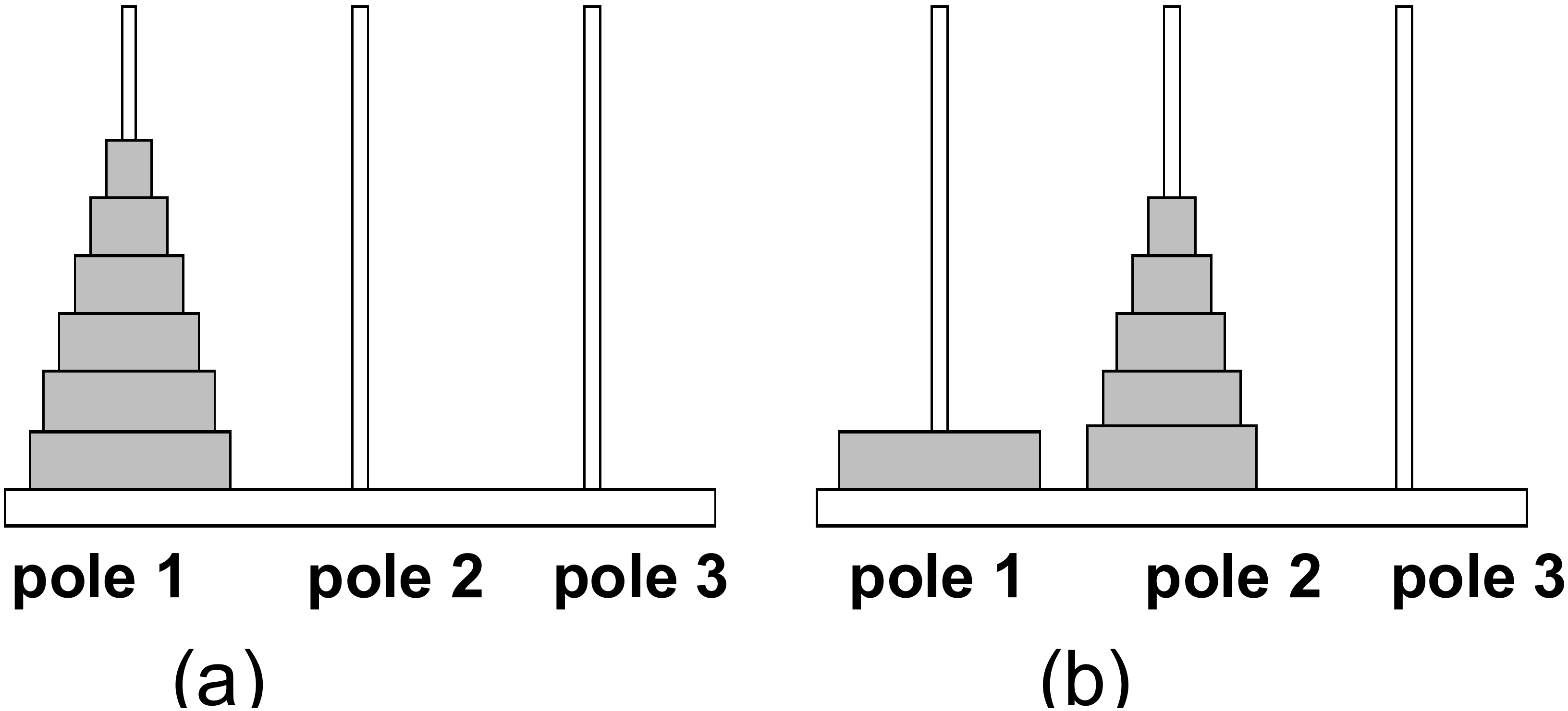


Figure 2.2 Towers of Hanoi example. (a) The initial conditions for a problem with six rings. (b) A necessary intermediate step on the road to a solution.

The growth rate of a recursive algorithm (cont.)

//moves *n* rings from pole *s* to pole *f* with the help of pole *t*

- void Hanoi(int n, int s, int f, int t){

  if(n == 1) printf("move ring 1 from poles %d to %d\n", s, f);

  else{

  // move the upmost *n-1* rings in pole *s* to pole *t* with the help of pole *f*

  Hanoi(n-1, s, t, f);

  printf("move ring %d from %d pole to %d pole\n", n, s, f);

  // moves the n-1 rings in pole *t* to pole *f* with the help of pole *s*

  Hanoi(n-1, t, f, s);

  }

}

The growth rate of a recursive algorithm (cont.)

- Denote by **T**(n) the running time of Hanoi
- **T**(n)=**T**(n-1) + c + **T**(n-1)

$$= \mathbf{2T}(n-1) + c$$

$$=2(2\mathbf{T}(n-2)+c) + c$$

$$=2^2\mathbf{T}(n-2)+2c+c$$

$$=2^3\mathbf{T}(n-3)+ 2^2c + 2c+c$$

…

$$=2^n\mathbf{T}(n-n)+ 2^{n-1}c +\ldots+ 2^2c + 2c+c$$

$$=2^{n-1}c +\ldots+ 2^2c + 2c+c, \text{ as } \mathbf{T}(0)=0$$

$$=\mathbf{(2^n-1)c}$$

The growth rate of a recursive algorithm (cont.)

- The steps for analyzing the growth rate of a recursive algorithm
  - Derive the recurrence relation of **T**(n)
    - E.g., **T**(n)=**T**(n-1)+c for the factorial function and **T**(n)=c+**T**(n/2) for the binary search algorithm
  - Solve the recurrence relation **T**(n)
    - see the relation with T(n-1) and T(n-2), or T(n/2) and T(n/4), etc, e.g., T(n-1)=T(n-2)+c
    - Expand T(n) with substitute
    - Expand T(n) until the base case of T(0) or T(1)
    - Sum up some terms

The Master Method

- A "cookbook" method for estimating the growth rate of a recursive algorithm
  - The CLRS book (3rd edition), Sections 4.5

Theorem 4.1 (Master theorem)

Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the nonnegative integers by the recurrence

$$T(n) = aT(n/b) + f(n) \, ,$$

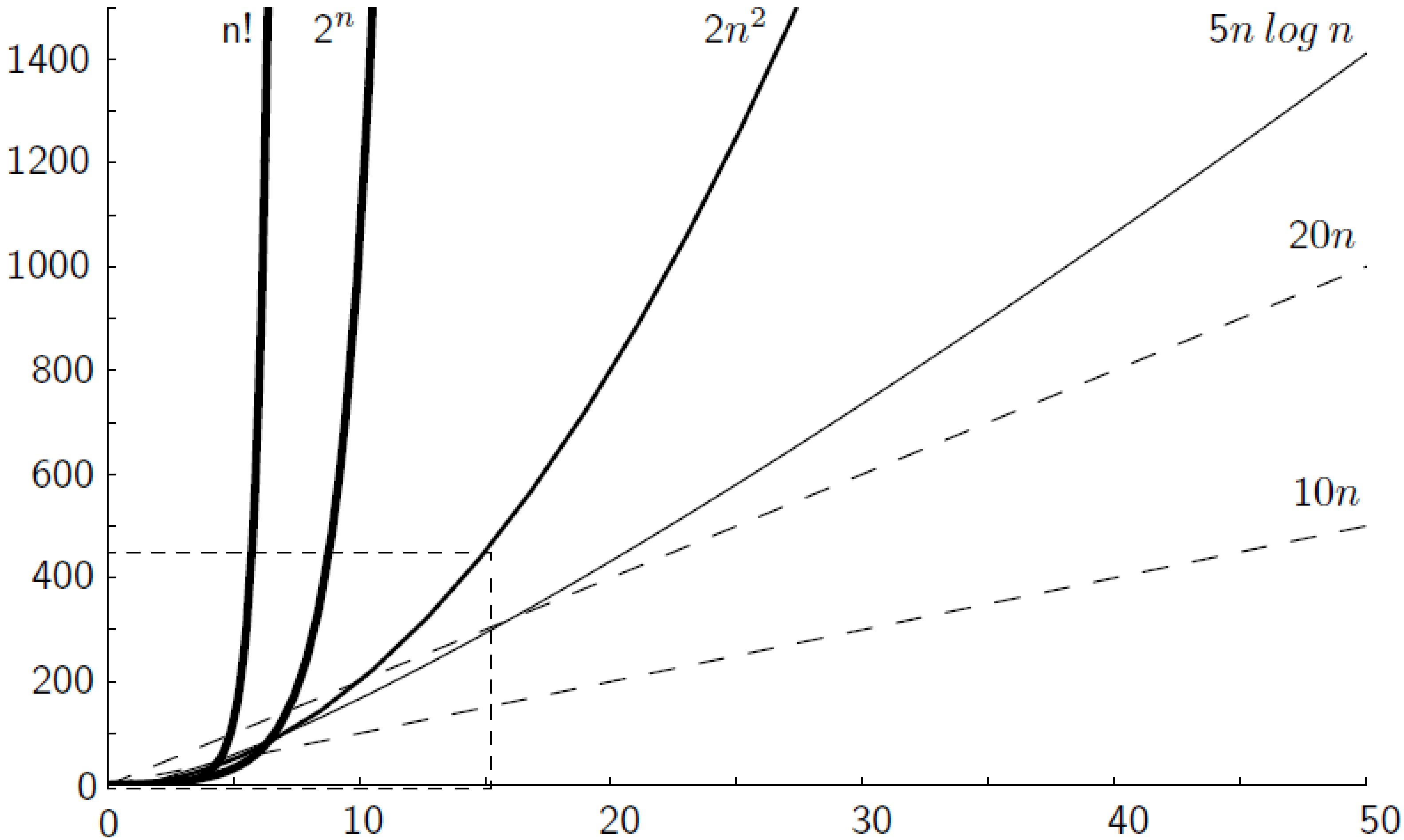
where we interpret n/b to mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then $T(n)$ has the following asymptotic bounds:

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \lg n)$.
3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$. ■

Glossary

- growth rate
  - The rate at which the **cost** of an algorithm grows as **the size of inputs** grows
- linear growth rate / linear time cost
  - $T(n) = cn$
- quadratic growth rate
  - $T(n) = cn^2$
- exponential growth rate
  - $T(n) = 2^n$

Growth Rates Comparison



Faster Computer or Algorithm?

What happens when we buy a computer 10 times faster?

T(n)	n	n'	Change	n'/n
$10n$	1,000	10,000	$n' = 10n$	10
$20n$	500	5,000	$n' = 10n$	10
$5n \log n$	250	1,842	$\sqrt{10}\, n \leq n' < 10n$	7.37
$2n^2$	70	223	$n' = \sqrt{10}n$	3.16
2^n	13	16	$n' = n + 3$	-----

Best, Worst, Average Cases

- Not all inputs of a given size take the same time to run.
- Sequential search for K in an array of n integers:
  - Begin at first element in array and look at each element in turn until K is found
- Best case: Find at first position. Cost is 1 compare
- Worst case: Find at last position. Cost is n compares
- Average case: $(n+1)/2$ compares

Which Analysis to Use?

- Best case analysis is too optimistic
- While average time appears to be the fairest measure, it may be difficult to determine.
  - require knowledge of the distribution of inputs
- When is the worst case time important?
  - Give an upper bound on the running time
    - Important for real-time algorithms
  - Worst case running time usually is in the order of average case running time, with only a few times longer

Asymptotic Analysis: Big-Oh

- Definition: For **T**(n) a non-negatively valued function, **T**(n) is **in the set O(*f*(*n*))** if there exist two positive constants ***c*** **and** $\boldsymbol{n_0}$ such that **T**(n) <= $cf(n)$ for all $n > n_0$.
- Usage: The algorithm is in O(n^2) in [best, average, worst] case.
- Meaning: For all data sets big enough (i.e., $n>n_0$), the algorithm always executes in **less than *cf*(*n*)** steps in [best, average, worst] case.

Big-Oh Notation (cont)

- Big-Oh notation indicates an upper bound on a growth rate
- Example 1: If $\mathbf{T}(n) = 3n^2$ then $\mathbf{T}(n)$ is in $O(n^2)$.
- Example 2: If $\mathbf{T}(n) = 3n^2$ then $\mathbf{T}(n)$ is in $O(n^3)$.
- Use the tightest upper bound:
  - While $\mathbf{T}(n) = 3n^2$ is in $O(n^3)$, we prefer $O(n^2)$.

Big-Oh Examples

- Definition does not require upper bound to be tight, though we would prefer as tight as possible
- **Example 1**: What is Big-Oh of $T(n) = 3n+3$
  - Let $f(n) = n$, $c = 6$ and $n_0 = 1$;
    $T(n) = O(f(n)) = O(n)$ because $3n+3 \le 6f(n)$ if $n \ge 1$
  - Let $f(n) = n$, $c = 4$ and $n_0 = 3$;
    $T(n) = O(f(n)) = O(n)$ because $3n+3 \le 4f(n)$ if $n \ge 3$
  - Let $f(n) = n^2$, $c = 1$ and $n_0 = 5$;
    $T(n) = O(f(n)) = O(n^2)$ because $3n+3 \le (f(n))^2$ if $n \ge 5$
- We certainly prefer $O(n)$.

Big-Oh Examples

- **Example 2**: Finding value *X* in an array (average cost).
- How to identify constants c and n_0 ?

- $\mathbf{T}(n) = c_s n/2$.
  - For all values of $n > 1$, $c_s n/2 \le c_s n$.

  Therefore, by the definition, $\mathbf{T}(n)$ is in $O(n)$ for $n_0 = 1$ and $c = c_s$.

Big-Oh Examples

- **Example 3**: $\mathbf{T}(n) = c_1 n^2 + c_2 n$ in average case.

$c_1 n^2 + c_2 n <= c_1 n^2 + c_2 n^2 <= (c_1 + c_2) n^2$ for all $n > 1$.

$\mathbf{T}(n) <= cn^2$ for $c = c_1 + c_2$ and $n_0 = 1$.

Therefore, $\mathbf{T}(n)$ is in $O(n^2)$ by the definition.

- **Example 4**: $\mathbf{T}(n) = c$. We say this is in $O(1)$.

Rules for Big-Oh

- If $T(n) = O(c\ f(n))$ for a constant c, then $T(n) = O(f(n))$
- If $T_1(n) = O(f(n))$ and $T_2(n)=O(g(n))$ then $T_1(n) + T_2(n) = O(\max(f(n), g(n)))$
- If $T_1(n) = O(f(n))$ and $T_2(n)=O(g(n))$ then $T_1(n) * T_2(n) = O(f(n) * g(n))$
- If $T(n) = a_m n^k + a_{m-1} n^{k-1} + \ldots + a_1 n + a_0$ then $T(n) = O(n^k)$
- Thus
  - Lower-order terms can be ignored.
  - Constants can be thrown away.

More about Big-Oh notation

- Asymptotic: Big-Oh is meaningful only when n is sufficiently large

  $n \geq n_0$ means that we only care about large size problems.

- Growth rate: A program with O(f(n)) is said to have growth rate of f(n). It shows how fast the running time grows when n increases.

Typical bounds (Big-Oh functions)

- Typical bounds in an increasing order of growth rate
- Function Name

  $O(1)$, 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

How do we use Big-Oh?

- Programs can be evaluated by comparing their Big-Oh functions with the constants of proportionality neglected. For example,
  - $T_1(n) = 10000\ n$ and $T_2(n) = 9\ n$. The time complexity of $T_1(n)$ is equal to the time complexity of $T_2(n)$.
- The common Big-Oh functions provide a "yardstick" for classifying different algorithms.
- Algorithms of the same Big-Oh can be considered as equally good.
- A program with $O(\log n)$ is better than one with $O(n)$.

Nested loops

- Running time of a loop equals running time of the code within the loop times the number of iterations.
- Nested Loops: analyze inside out

```
1  for (i=0; i <n; i++)
2      for (j = 0; j< n; j++)
3          k++
```

- Running time of lines 2-3: $O(n)$
- Running time of lines 1-3: $O(n^2)$

Consecutive statements

- For a sequence S1, S2, .., Sk of statements, running time is maximum of running times of individual statements

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

- Running time is: $O(n^2)$

Conditional statements

- The running time of

  If (cond) S1

  else S2

  is running time of *cond* plus the max of running times of S1 and S2.

More nested loops

```
1 int k = 0;
2 for (i=0; i<n; i++)
3     for (j=i; j<n; j++)
4         k++
```

- Running time of lines 3-4: n-i
- Running time of lines 1-4:

$$\sum_{i=0}^{n-1}(n-i) = n(n+1)/2 = O(n^2)$$

More nested loops

```
int k = 0;
for (i=1; i<n; i*= 2)
    for (j=1; j<n; j++)
        k++
```

- Running time of inner loop: O(n)
- What about the outer loop?
- In *m*-th iteration, value of i is 2^{m-1}
- Suppose $2^{q-1} < n \le 2^q$, then outer loop is executed q times.
- Running time is O(n log n). Why?

A more intricate example

```
int k = 0;
for (i=1; i<n; i*= 2)
    for (j=1; j<i; j++)
        k++
```

- Running time of inner loop: O(i)
- Suppose $2^{q-1} < n \le 2^q$, then the total running time:

  $1 + 2 + 4 + \ldots.+2^{q-1} = 2^q - 1$
- Running time is O(n).

A Common Misunderstanding

- "The best case for my algorithm is n=1 because that is the fastest." WRONG!
  - Big-oh refers to a growth rate as n grows to ∞.
  - Best case is defined as which input of size n is cheapest among all inputs of size n.
  - Analyze the growth rate for best/average/worst cases, e.g., $\boldsymbol{T}(n)=2n^2+3n+6$, then obtain the upper bound for the growth rate, e.g., $\boldsymbol{T}(n)=O(2n^2)$

Lower Bounds

- To give better performance estimates, we may also want to give lower bounds on growth rates
- Definition (omega): $T(n) = \Omega(f(n))$

  if there exist some constants c and n_0 such that $T(n) \geq cf(n)$ for all $n \geq n_0$

"Exact" bounds

- Definition (Theta): $T(n) = \Theta(f(n))$ if and only if $T(n) = O(f(n))$ and $T(n) = \Omega(f(n))$.
- An algorithm is $\Theta(f(n))$ means that f(n) is a tight bound (as good as possible) on its running time.
  - On all inputs of size n, time is $\le f(n)$
  - On all inputs of size n, time is $\ge f(n)$

```
int k = 0;
for (i=1; i<n; i*=2)
    for (j=1;j<n; j++)
        k++
```

This program is $O(n^2)$ but not $\Omega(n^2)$; it is $\Theta(n \log n)$

Big-Omega

- Definition: For **T**(n) a non-negatively valued function, **T**(n) is in the set $\Omega(g(n))$ if there exist two positive constants c and n_0 such that **T**(n) >= $c*g(n)$ for all $n > n_0$.
- **Lower bound** on a growth rate

- Meaning: For all data sets big enough (i.e., $n > n_0$), the algorithm always executes in more than $c*g(n)$ steps.

Big-Omega Example

- **T**(n) = $c_1n^2 + c_2n$.

  $c_1n^2 + c_2n >= c_1n^2$ for all $n > 1$.
  T(n) >= cn^2 for $c = c_1$ and $n_0 = 1$.

Therefore, **T**(n) is in $\Omega(n^2)$ by the definition.

- **T**(n) in $\Omega(n)$ as **T**(n) >= c_2n for $n >= 1$
- We want the greatest lower bound.

Theta Notation

- When big-Oh and Ω meet, we indicate this by using Θ (big-Theta) notation.
- Definition: An algorithm is said to be $\Theta(h(n))$ if it is in $O(h(n))$ and it is in $\Omega(h(n))$.
- $\mathbf{T}(n) = c_1 n^2 + c_2 n$.
  - $\mathbf{T}(n) = \Theta(n^2)$ as $\mathbf{T}(n)$ in $O(n^2)$ and $\mathbf{T}(n)$ in $\Omega(n^2)$
- For **T***(n) given by an algebraic equation*, we always give a Θ analysis

Theta Notation (cont.)

- We may not have $\Theta(n)$ for some $\mathbf{T}(n)$
- Example

$$\mathbf{T}(n) = \begin{cases} n & \text{for all odd } n >= 1 \\ n^2 & \text{for all even } n >= 1 \end{cases}$$

- Upper bound
  - $\mathbf{T}(n)$ in $O(n^2)$
- Lower bound
  - $\mathbf{T}(n)$ in $\Omega(n)$
- big-Oh and Ω do not meet

An Alternative Definiton for Ω

- **T**(n) is in Ω *($g(n)$)* if there exists a positive constant c such that **T**(n)>=$cg(n)$ for an infinite number of values for n.

- Using this definition, **T**(n) is in $\Omega(n^2)$ for the example in the previous slide.

- Caveat: Not a lower bound for the function, but for a "subsequence"

A Common Misunderstanding

- Confusing worst case with upper bound, and best case with lower bound
- Worst case refers to the worst input from among the choices for possible inputs of a given size.
- Upper bound refers to a growth rate, and the rate may be for the worst case, average case, or the best case

Simplifying Rules

1. If $f(n)$ is in $O(g(n))$ and $g(n)$ is in $O(h(n))$, then $f(n)$ is in $O(h(n))$.
   a. If **T**(n) in O(n), then T(n) in $O(n^2)$
2. If $f(n)$ is in $O(kg(n))$ for any constant $k > 0$, then $f(n)$ is in $O(g(n))$.
   a. Ignore constants
3. If $f_1(n)$ is in $O(g_1(n))$ and $f_2(n)$ is in $O(g_2(n))$, then $(f_1 + f_2)(n)$ is in $O(\max(g_1(n), g_2(n)))$.
   a. Drop low order terms, e.g.$\mathbf{T}(n) = n^2 + n$ is in $O(n^2)$
4. If $f_1(n)$ is in $O(g_1(n))$ and $f_2(n)$ is in $O(g_2(n))$ then $f_1(n)f_2(n)$ is in $O(g_1(n)g_2(n))$.
   a) Useful for analyzing loops

Running Time Examples (1)

- Example 1: `a = b;`

This assignment takes constant time, so it is $\Theta(1)$.

- Example 2:

```
sum = 0;
for (i=1; i<=n; i++)
   sum += n;
```

T(n) = $\Theta(n)$

Running Time Examples (2)

- Example 3:

// take time $\Theta(1)$
```
sum = 0;
```
// take time $\Sigma i = \Theta(n^2)$
```
for (i=1; i<=n; i++)
   for (j=1; j<=i; j++)
      sum++;
```
// take time $\Theta(n)$
```
for (k=0; k<n; k++)
   A[k] = k;
```

- **T**(n) = $\Theta(1)+\Theta(n^2)+\Theta(n) = \Theta(n^2)$
  - Drop low order terms

Running Time Examples (3)

- Example 4:

```
sum1 = 0;
```
// takes time $n^2 = \Theta(n^2)$
```
for (i=1; i<=n; i++)
   for (j=1; j<=n; j++)
      sum1++;

sum2 = 0;
```
// takes time Σi = $n(n+1)/2 = \Theta(n^2)$
```
for (i=1; i<=n; i++)
   for (j=1; j<=i; j++)
      sum2++;
```

Running Time Examples (4)

- Example 5:

```
sum1 = 0;
for (k=1; k<=n; k*=2)
   for (j=1; j<=n; j++)
      sum1++;
```

  - Each inner loop takes time $\Theta(n)$
  - How many inner loops?
    - log n
  - $\Theta(n \log n)$.

- Example 6:

```
sum2 = 0;
for (k=1; k<=n; k*=2)
   for (j=1; j<=k; j++)
      sum2++;
```

  - Each inner loop takes k basic operations
  - Total time:

$$1+2+4+8+\ldots+n/2+n$$
$$=\Sigma 2^k \text{ for } k = 0 \text{ to } \log n$$
$$=2n-1=\Theta(n)$$

Other Control Statements

- `while` loop: Analyze like a `for` loop.
- `if` statement: Take greater complexity of `then`/`else` clauses.
- `switch` statement: Take complexity of most expensive case.
- Subroutine call: Complexity of the subroutine.

Analyzing Problems

- Upper bound: Upper bound of the best **known** algorithm.
  - e.g., O(n log n) for known sorting algorithms
- Lower bound: Lower bound for **every possible** algorithm.
- It is useful to see whether an algorithm is good enough

Analyzing Problems: Example

- Common misunderstanding: No distinction between upper/lower bound when you know the exact running time.

- Example of imperfect knowledge: Sorting

1. Cost of I/O: $\Omega(n)$.
2. Bubble or insertion sort: $O(n^2)$.
3. A better sort (Quicksort, Mergesort, Heapsort, etc.): $O(n \log n)$.
4. We prove later that sorting is $\Omega(n \log n)$.

Multiple Parameters

- Compute the rank ordering for all C pixel values in a picture of P pixels.

```
for (i=0; i<C; i++)   // Initialize count
  count[i] = 0;
for (i=0; i<P; i++)   // Look at all pixels
  count[value(i)]++; // Increment count
sort(count);          // Sort pixel counts
```

If we use P as the measure, then time is $\Theta(P)$.

- More accurate is $\Theta(P + C \log C)$.

Space Bounds

- Space bounds can also be analyzed with asymptotic complexity analysis.

- Time: Algorithm
- Space: Data Structure

Space/Time Tradeoff Principle

- One can often reduce time if one is willing to sacrifice space, or vice versa.
  - Encoding or packing information
    - Boolean flags
  - Table lookup
    - Fibonacci calculation
- Disk-based Space/Time Tradeoff Principle: The smaller you make the disk storage requirements, the faster your program will run.
  - Disk is about 1,000 times slower than memory

Summary: lower vs. upper bounds

- This section gives some ideas on how to analyze the complexity of programs.
- We have focused on worst case analysis.
- Upper bound $O(f(n))$ means that for sufficiently large inputs, running time $T(n)$ is bounded by a multiple of $f(n)$.
- Lower bound $\Omega(f(n))$ means that for sufficiently large n, there is at least one input of size n such that running time is at least a fraction of $f(n)$
- We also touch the "exact" bound $\Theta(f(n))$.

Summary: algorithms vs. Problems

- Running time analysis establishes bounds for individual algorithms.
- Upper bound O(f(n)) for *a problem*: there is some O(f(n)) algorithms to solve the problem.
- Lower bound Ω(f(n)) for *a problem*: every algorithm to solve the problem is Ω(f(n)).
- They different from the lower and upper bound of an algorithm.

Conclusion

- Growth rate of an algorithm
- The worst, average, and best cases
- The upper and low bounds on a growth rate
  - Big O, big Ω, big Θ
  - Consider only the most important term
  - Ignore low order terms
- The cost of an algorithm vs. the cost of a problem

Homework 1

- See course webpage
- Deadline: midnight before next lecture
- Submit to: cs_scu@foxmail.com
- File name format:
  - CS311_Hw1_yourID_yourLastName.doc (or .pdf)