Chapter 2. Time Complexity Analysis, Counts & Growth Functions

Problem instances

- An *instance* is the actual data for which the problem needs to be solved.
- We use the terms *instance* and *input* interchangeably.
- *Problem:* Sort list of records.

  Instances:

  (1, 10, 5)

  (1, 2, 3, 4, 1000, 27)
- Time complexity analysis is done in terms of input size

Size Examples

- Search and sort
  - Size = *n* number of records in the list, each of which is of the same size (c bits)
- Graphs problems
  - Size = (|V| + |E|)
  - |V|: number of nodes
  - |E|: number of edges
- Matrix problems
  - Size = r*c
  - r: number of rows
  - c: number of columns

Exceptions: Number problems

- Problems where the number of bits is not constant but may vary depending on input
- Examples:
  - Recall Fibonacci number
  - Factorial of 10, 10^6, 10^{15}
  - Operations (e.g., add and multiplication) of large numbers where a number is expressed using several words
  - For these problems we should use the formal definition – time complexity with respect to the number of bits used to express input

Efficiency

- The efficiency of an algorithm depends on the quantity of resources it requires

- Usually we compare algorithms based on their *time*
  - Sometimes also based on the *space* they need.

- The time required by an algorithm depends on the instance *size* and its *data*

Example: Sequential search

- Problem: *Find a search key in a list of records*
- Algorithm: Sequential search
  - Main idea: Compare search key to all keys until a match is found or list is exhausted
- Time depends on the size of the list *n* and the data stored in a list

Time Complexity Analysis

- **Best Case: The smallest amount of time needed to run any instance of a given size**
- **Worst Case: The largest amount of time needed to run any instance of a given size**
- **Average Case: the expected time required by an instance of a given size**

Time Complexity Analysis

- If the *best*, *worst* and *average* "times" of some algorithms are identical, we have ***every case time analysis*****.**

e.g., array addition, matrix multiplication, etc.

- Usually, the best, worst and average time of a algorithm are different.

Time Analysis for Sequential search

- Worst-case: if the search key x is the last item in the array or if x is not in the array.

  $$W(n) = n$$

- Best-case: if x is matched with the first key in array S, which means x=S[1], regardless of array size n

  $$B(n) = 1$$

Time Analysis for Sequential search

- Average-case: If the probability that x is in the k^{th} array slot is 1/n:

$$A(n) = \sum_{k=1}^{n} (k \times \tfrac{1}{n}) = \tfrac{1}{n} \times \sum_{k=1}^{n} k = \tfrac{1}{n} \times \tfrac{n(n+1)}{2} = \tfrac{n+1}{2}$$

Question: What is A(n) if x could be outside the array?

Worse case time analysis

- Most commonly used time complexity analysis
- *Because:*
  - Easier to compute than average case
  - Maximum time needed as a function of instance size
  - More useful than the best case

Worst case time analysis

- Drawbacks of comparing algorithms based on their worst case time:
  - An algorithm could be superior on average than another, although the worst case time complexity is not superior.
  - For some algorithms a worst case instance is very unlikely to occur in practice.

Evaluation of runtime through experiments

- Challenges
  - Algorithm must be fully *implemented*
  - To compare run time we need to use the *same hardware* and *software* environments
  - Different *coding style* of different individuals'

- Is there any better way?

Requirements for time complexity analysis

- *Independence*
- *A priori*
- *Large instances*
- *Growth rate classes*

Independence Requirement

- Time complexity analysis must be *independent* of:
  - *The hardware of a computer*
  - *The programming language used for pseudo code*
  - *The programmer that wrote the code*

A Priori Requirement

- Analysis should be a priori; that is, it should be done *before* implementing the algorithm

- Derived for any algorithm expressed in high level description or pseudo code

Large Instance Requirement

- Algorithms running efficiently on small instances may run very slowly with large instance sizes

- Analysis must capture algorithm behavior when problem instances are large
  - For example, linear search may not be efficient when the list size $n = 1{,}000{,}000$

Growth Rate Classes Requirement

- Time complexity analysis must classify algorithms into:
  - Ordered classes so that all algorithms in a single class are considered to have the same efficiency
  - If class A “is better than“ class B, then all algorithms that belong to A are considered more efficient than all algorithms in class B

Growth rate classes

- Growth rate classes are derived from instruction counts
- Time analysis partitions algorithms into general equivalence classes such as:
  - Logarithmic,
  - Linear,
  - Quadratic,
  - Cubic,
  - Polynomial,
  - Exponential, etc.

Comparing an nlogn to an n^2 algorithm

- An nlogn algorithm is always more efficient for *large* instances

- Pete is a programmer for a super computer. The computer executes 100 million instructions per second.
  His implementation of Insertion Sort requires $2n^2$ computer instructions to sort n numbers.

- Joe has a PC which executes 1 million instructions per second. Joe's sloppy implementation of Merge Sort requires $75n \lg n$ computer instructions to sort n numbers.

Who sorts 50 numbers faster?

Super Pete:

(2 $(50)^2$ instructions) / (10^8 instructions/sec)

$\approx$ 0.00005 seconds

Average Joe:

(75 *50 lg (50) instructions) / (10^6 instructions/sec)

$\approx$ 0.000353 seconds

Who sorts a million numbers faster?

Super Pete:

$(2\ (10^6)^2$ instructions$)\ /\ (\ 10^8$ instructions/sec$)$

= 20,000 seconds

≈ **5.56 hours**

Average Joe:

$(75 * 10^6\ \lg (10^6)$ instructions$)/\ (10^6$ instructions/sec$)$

= 1494.8 seconds ≈ **25 minutes**

Insertion sort

```
for i = 2 to n
    for (k = i; k > 1 and a[k] < a[k-1];
            k--)
            swap (a[k], a[k-1])
```

→ ***invariant: a[1..i] is sorted***

Worst-case time complexity in terms of number of comparisons:

In the inner "for" loop, for a given i, the comparison is done at most i-1 times

In total: $$\sum_{i=2}^{n}(i-1) = \frac{n(n-1)}{2}$$

$$W(n) = \frac{n(n-1)}{2} \in \Theta(n^2)$$

Sorting algorithm animations

- Read textbook chapters on sorting

- Good animations are available at http://www.sorting-algorithms.com/

Example: Binary search (Recursive)

```
Index Binsearch(index low, index high)
{
   index mid;

   if (low > high)  return 0;

  else
  {
     mid = floor[(low+high)/2];

     if (x == S[mid])  return mid;
     else if (x < S[mid])  return Binsearch(low, mid-1);
     else  return  Binsearch(mid+1, high);
  }
}
```

Worst-case Time complexity:

$W(n) = W(n/2) + 1$
$W(1) = 1$

$\rightarrow$ $W(n) = \lg n + 1$

Topics

- Instruction count for statements
- Methods
- Examples

Instruction counts

- Provide rough estimates of actual number of instructions executed
- Depend on:
  - Language used to describe algorithm
  - Programmer's style
  - Method used to derive count
- Could be quite different from actual counts
- Algorithm with count=2n, may not be faster than one with count=5n.

Computing Instruction Counts

- Given a (non-recursive) algorithm expressed in pseudo code we explain how to:
  - Assign counts to high level statements
  - Describe methods for deriving an instruction count
  - Compute counts for several examples

Counts for High Level Statements

- Assignment
- loop condition
- for loop
  - for loop control
  - for loop body
- while loop
  - while loop control
  - while loop body
- if

Note: The counts we use are estimates; The goal is to derive a correct growth function

***Assignment* Statement**

1. A= B*C-D/F

- $\text{Count}_1 = 1$
- In reality? At least 4

Note: When numbers B, C, D, F are very large (a number can't be stored in a single word), algorithms that deal with large numbers will be used and the count will depend on the number of digits needed to store the large numbers.

Loop condition

1. (i < n) && (!found)

- $Count_1 = 1$

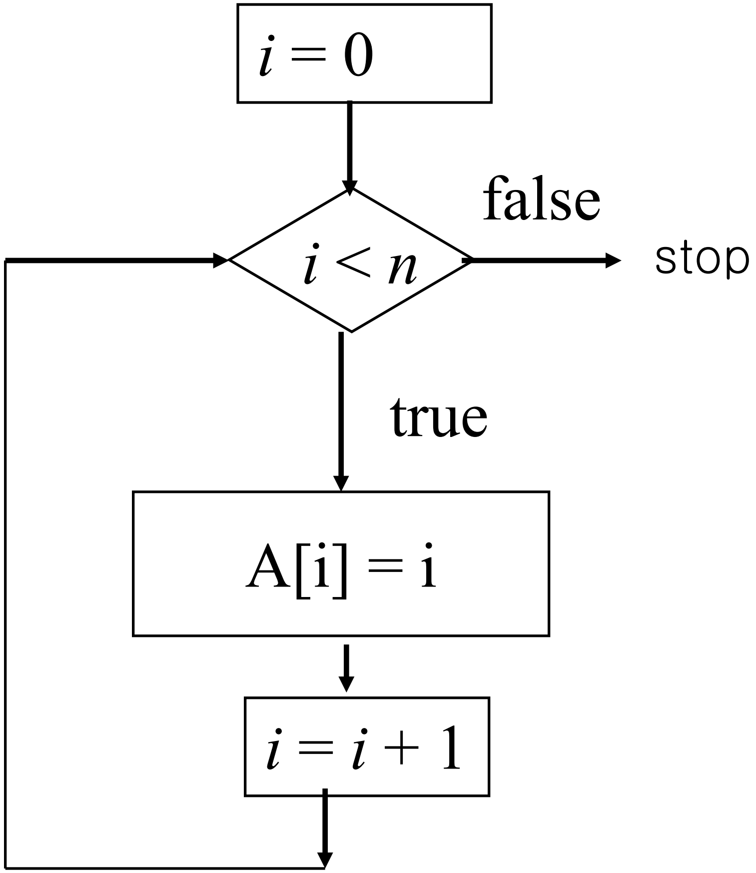
Note: if loop condition invokes a function, count of the function must be used

for loop body

1. for (i=0; i < n; i++)
2. A[i] = i

Count$_2$ = 1

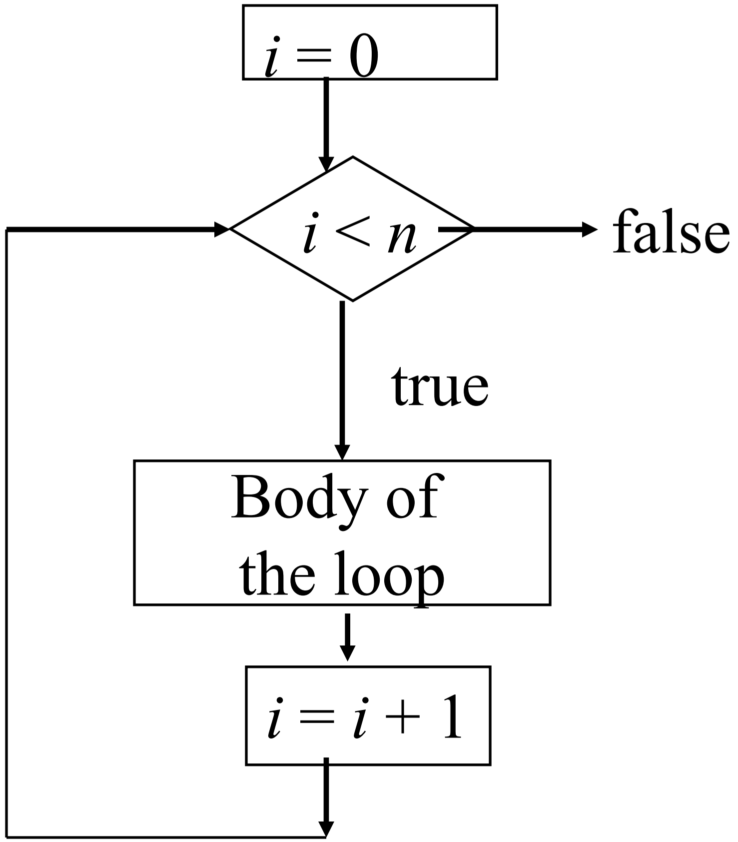
Count$_{1(2)}$ = $\sum_{i=0}^{n-1} \text{Count}_2$



for loop control

1. for (I = 0; i < n; i++)
2. <body>

Count = number of times loop condition is executed (assuming loop condition has a count of 1)



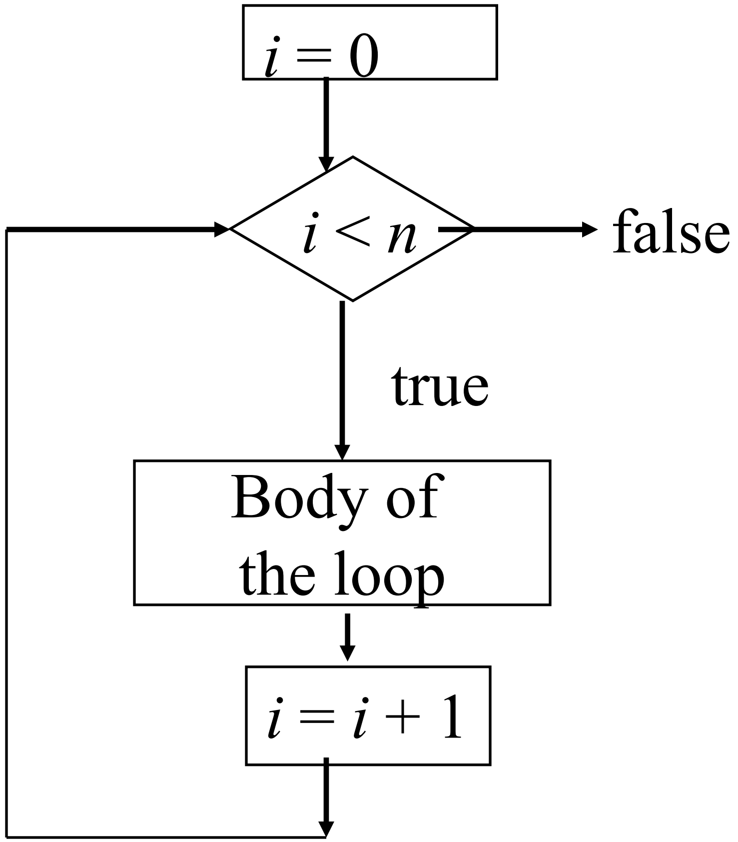
for loop control

1. for (i=0; i < n; i++)
2. <body>

Count$_1$ = number times loop condition i < n is executed

= $n + 1$

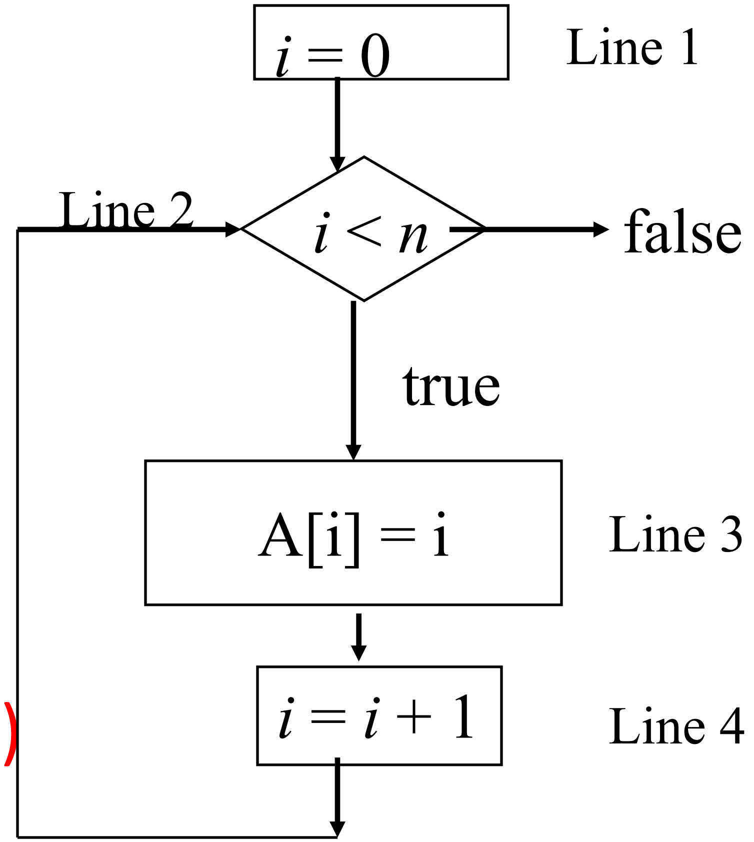
Note: last time condition is checked when i = n and (i < n) evaluates to false



while loop control

```
1.  i = 0
2.  while (i < n){
3.      A[i] = i
4.      i = i + 1}
```

Count = number of times loop condition is executed (assuming loop condition has a count of 1)



while loop control

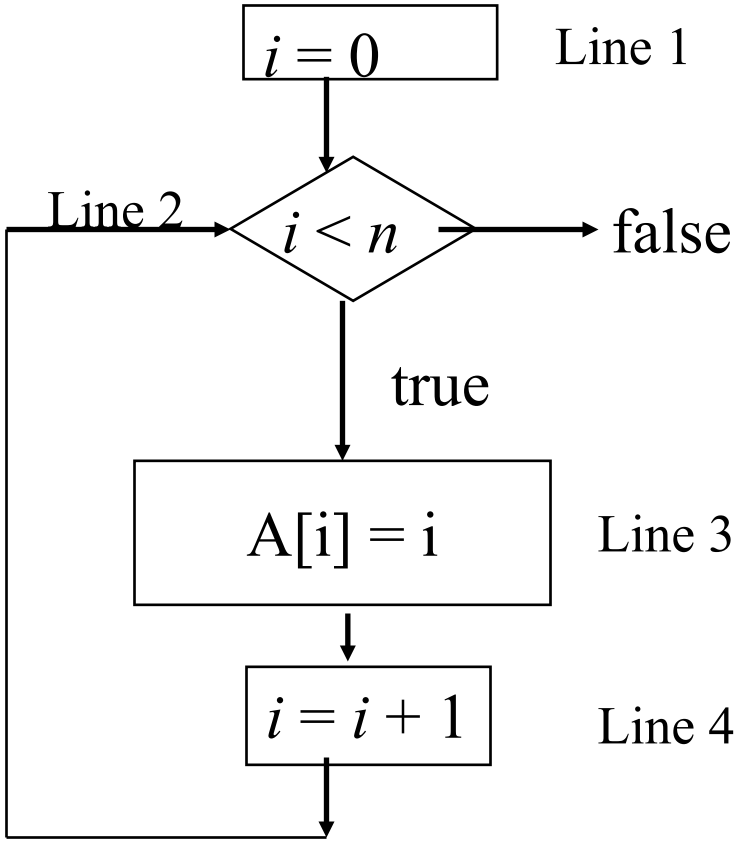
```
1.  i = 0
2.  while (i < n){
3.      A[i] = i
4.      i = i + 1}
```

Count$_2$ = number of times

loop condition

(i < n) is executed

= n + 1



If statement

```
Line 1: if (i == 0)
Line 2:   statement
        else
Line 3:   statement
```

For worst case analysis, how many counts are there for Count_{if}?

$$\text{Count}_{\text{if}} = 1 + \max\{\text{count2}, \text{count3}\}$$

Method 1: Sum Line Counts

- Derive a count for each line of code taking into account of all nested loops
- Compute total by adding line counts

Method 2: Barometer Operation

- A “barometer instruction” is selected
- Count = number of times that barometer instruction is executed.
- Search algorithms:
  - barometer instruction (x == L[j]?).
- Sort algorithms:
  - barometer instruction (L[i] <= L[j]?).

Example 1: Method 1

```
for (i=0;  i<n;  i++ )
    A[i] = i
```

- Method 1

$count_1 = \quad n+1$

$count_{1(2)} = n*1 = n$

Total = $(n+1)+n = 2n+1$

Example 1: Method 2

- Method 2

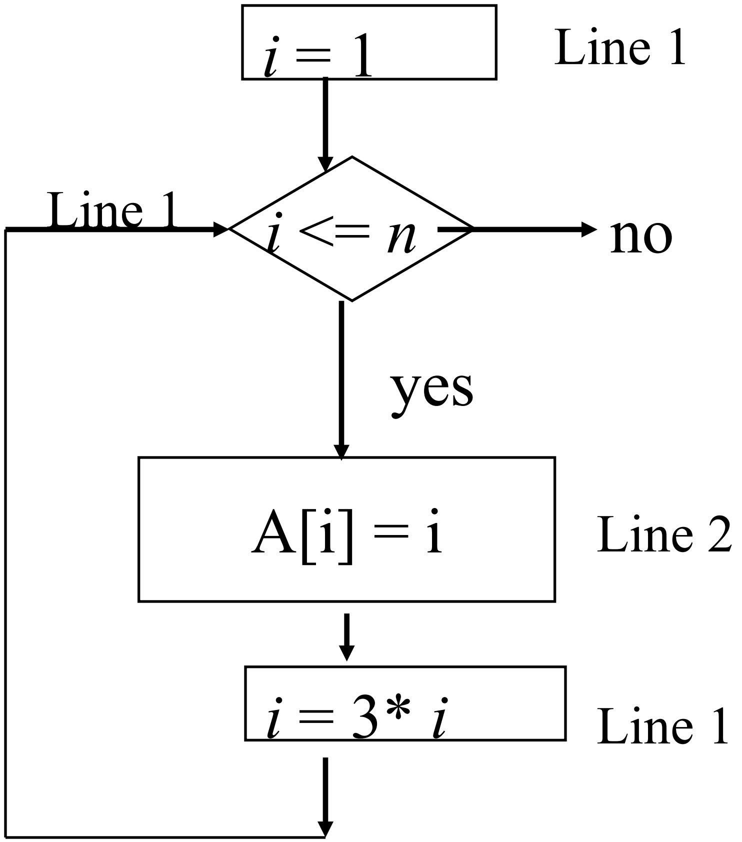
```
1.   for (i=0;  i<n;  i++ )
2.       A[i] = i + 1
```

- Barometer operation = + in body of loop

- $count_{1(+)} = n$

Example 2: What is $count_{1(2)}$?

```
1.for (i=1;i<=n;i=3*i)
2.    A[i] = i
```



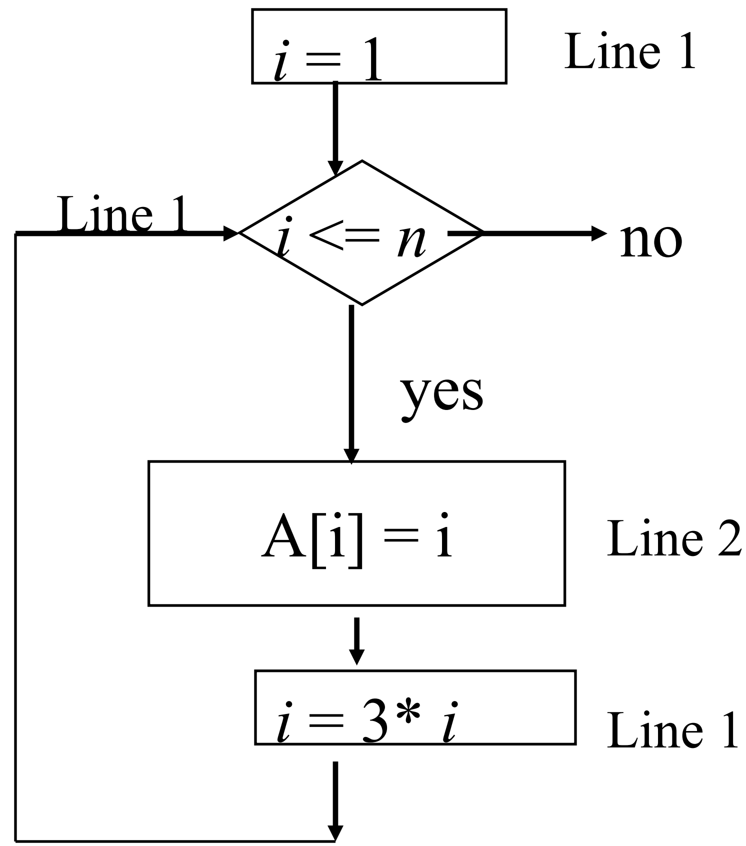
Example 2: What is $count_{1(2)}$?

```
1. for(i=1;i<=n; i=3*i)
2.    A[i] = i
```

For simplicity, $n = 3^k$ for some positive integer k.

Body of the loop executed for i = $1(=3^0), 3^1, 3^2, \ldots, 3^k$.

So $count_{1(2)}$= $\sum_{q=0}^{k} count_2 = k+1$



Since $k = \log_3 n$, it is executed $\log_3 n + 1$ times.

Example 3: Sequential Search

```
location=0
while (location<=n-1
       && L[location]! = x)
    location++
return location
```

- Barometer operation = (L[location]! = x?)
- Best case analysis

  x == L[0] and the count is 1

- Worst case analysis

  x = L[n-1] or x not in the list. Count is n.

Example 4:

```
1. x = 0
2. for (i=0; i<n; i++)
3.     for (j=0, j<m; j++)
4.         x = x + 1
```

Barometer is + in body of loop.

$count_{2(3(+))} = ?$

$$\sum_{i=0}^{n-1} count_{3(+)} = \sum_{i=0}^{n-1}\sum_{j=0}^{m-1} count_{+} =$$

$$= \sum_{i=0}^{n-1}\sum_{j=0}^{m-1} 1 = \sum_{i=0}^{n-1} m = m\sum_{i=0}^{n-1} 1 = mn$$

Example 5:

1. x=0
2. **for** (i=0; $i<n$; i++)
3.   **for** (j=0, $j<n^2$; j++)
4.   x = x + 1

- $\text{Count}_{2(3(+))} = ?$

Answer: $n * n^2 * 1$

Example 6:

```
Line 1: for (i=0; i<n; i++)
Line 2:      for (j=0, j<i; j++)
Line 3.            x = x + 1
```

Barometer operator = +

$\text{Count}_{1(2(+))}$= ?

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

Example 7:

1. **for** (i=0; i<n; i++)
2.   **for** (j=0, j<i; j++)
3.   **for** (k=0; k<=j; k++)
4.   x++;

count$_{1(2(3))}$= $$\sum_{i=0}^{n-1}\sum_{j=0}^{i-1}\sum_{k=0}^{j} 1 =$$

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

$$\frac{1}{6}(n-1)n(n+1)$$

Note: $1^2 + 2^2 + \ldots + n^2 = \frac{1}{6}n(n+1)(2n+1)$

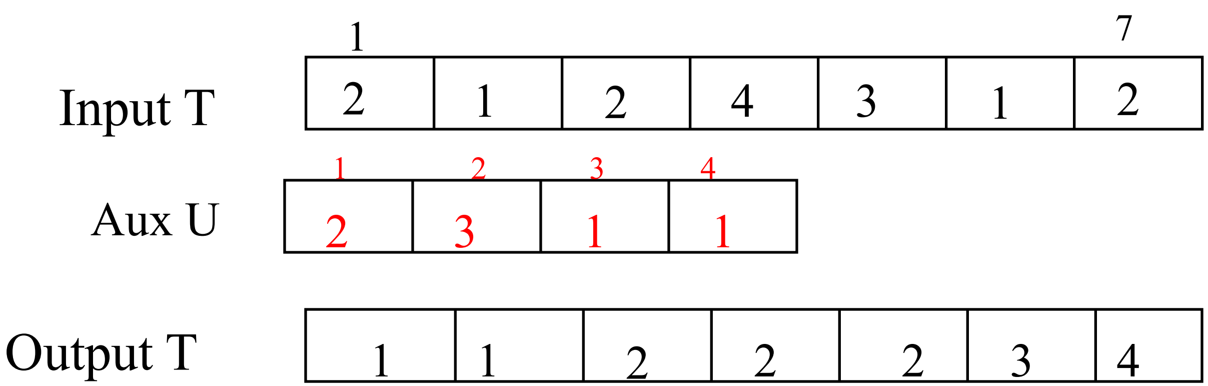
Example of Count: Counting sort

Input: Array *T* containing *n* keys in ranges 1.. *S*

Idea: (Similar to Histogram)

1) Maintain the count of number of keys in an auxiliary array U

2) Use counts to overwrite array in place



Algorithm:

Counting-Sort (T, s)

1. for $i = 1$ to s //initialize U
2. $U[i] = 0$
3. for $j = 1$ to n // n = $length[T]$
4. $U[T[j]] = U[T[j]] + 1$ //Count keys
5. q = 1
6. for $j = 1$ to s //rewrite T
7. while U[j] > 0
8. $T[q] = j$
9. $U[j] = U[j] - 1$
10. q = q + 1

Count:

```
5.   q ← 1
6.   for j ← 1 to s     //rewrite T
7.        while U[j] > 0
8.             T[q] = j
9.             U[ j ] ← U[ j ] - 1
10.            q ← q+1
```

Barometer operation – in line 9

$$Count_{6(7(9))} = \sum_{j=1}^{s} Count_{7(9)} = \sum_{j=1}^{s} U[j] = n$$

n (not *n* + *s*)

Asymptotic Growth Rate

Asymptotic Running Time

- The running time of an algorithm as input size approaches infinity is called the *asymptotic running time*
- We study different notations for asymptotic efficiency.
- In particular, we study tight bounds, upper bounds and lower bounds.

Outline

- Why do we need the different sets?
- Definition of the sets O (Oh), Ω (Omega) and Θ (Theta), o (oh), ω (omega)
- Classifying examples:
  - Using the original definition
  - Using limits

The functions

- Let $f(n)$ and $g(n)$ be *asymptotically nonnegative* functions whose domains are the set of natural numbers N={0,1,2,…}.
- A function $g(n)$ is *asymptotically nonnegative*, if $g(n) \geq 0$ for all $n \geq n_0$ where $n_0 \in$ N

Big Oh

- Big "Oh" - asymptotic upper bound on the growth of an algorithm
- When do we use Big Oh?

1. To provide information on the maximum number of operations that an algorithm performs
   - Insertion sort is $O(n^2)$ in the worst case
     - This means that in the worst case it performs at most cn^2 operations where c is a positive constant
2. Theory of NP-completeness
   1. An algorithm is polynomial if it is $O(n^k)$ for some constant k
   2. P = NP if there is any polynomial time algorithm for any NP-complete problem

   Note: Theory of NP-completeness will be discussed much later in the semester

Definition of Big Oh

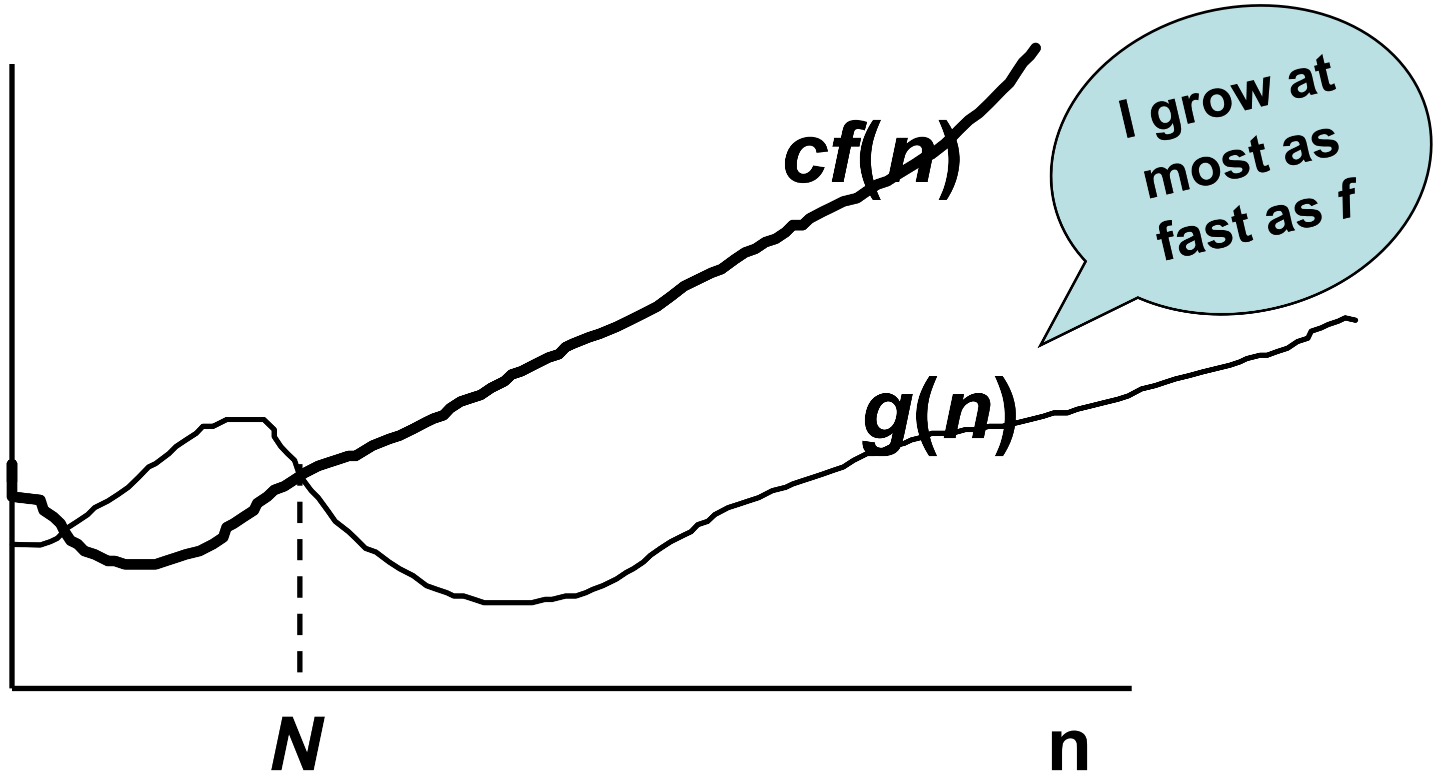
- $O(f(n))$ is the set of functions $g(n)$ such that:

  there exist positive constants c and N, for which

  $$0 \le g(n) \le cf(n) \text{ for all } n \ge N$$

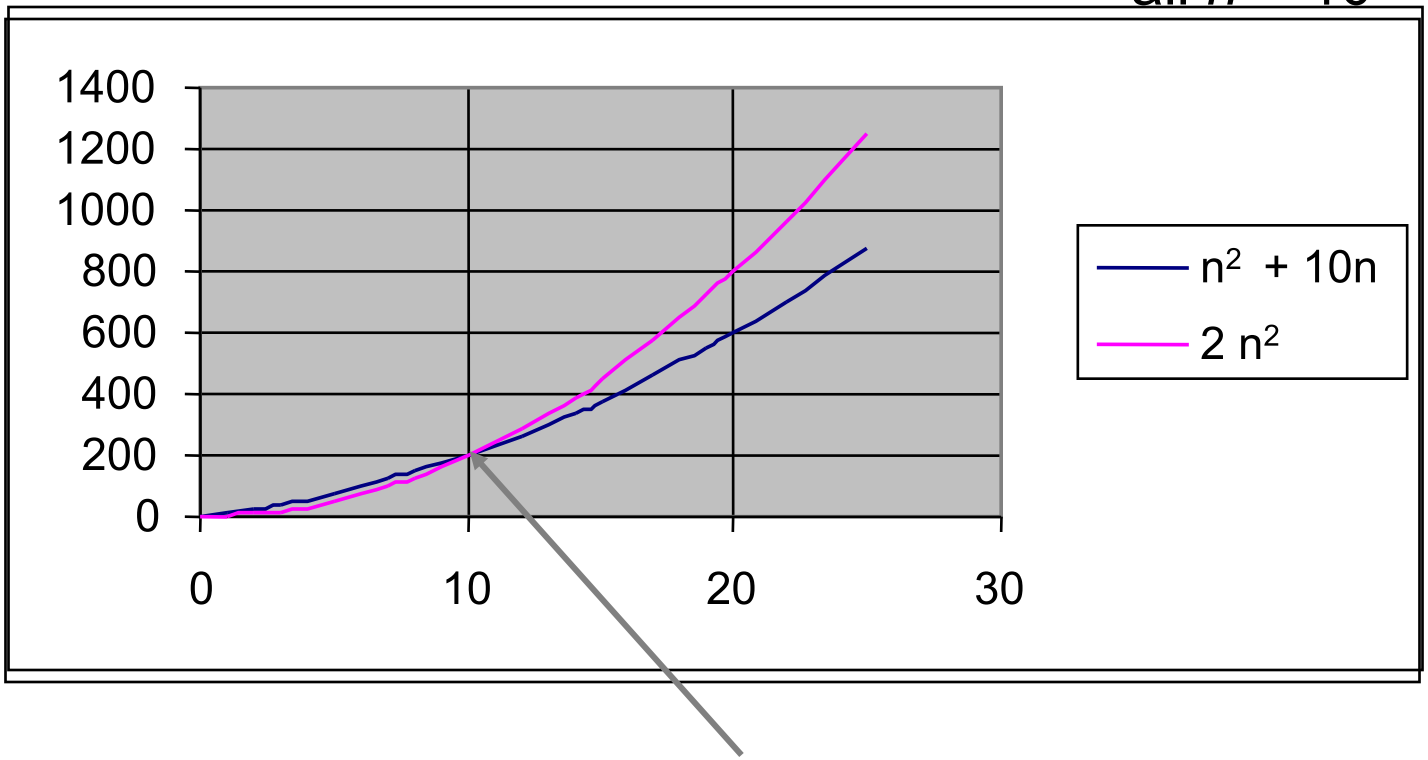
- g(n) is O(f(n)): $f(n)$ is an *asymptotically upper bound* for $g(n)$

$g(n) \in O(f(n))$



$n^2 + 10\,n \in O(n^2)$ Why?

take c = 2
N = 10
n^2+10n <=2n^2 for all n>=10



Does 5n+2 $\in$O(n)?

Proof: From the definition of Big Oh, there must exist $c > 0$ and integer $N > 0$ such that $0 \le 5n+2 \le cn$ for all $n \ge N$.

Dividing both sides of the inequality by $n > 0$ we get:

$0 \le 5+2/n \le c$.

- $2/n$ (> 0) becomes smaller as n increases
- For instance, let $N = 2$ and $c = 6$

There are many choices here for c and N.

Is 5n+2 $\in$O(n)?

If we choose $N = 1$ then $5+2/n \leq 5+2/1= 7$. So any $c \geq 7$ works. Let's choose c = 7.

If we choose $c = 6$, then $0 \leq 5+2/n \leq 6$. So any $N \geq 2$ works. Choose $N = 2$.

In either case (we only need one!) , $c > 0$ and $N > 0$ such that $0 \leq 5n+2 \leq cn$ for all $n \geq N$. So the definition is satisfied and

$5n+2 \in O(n)$

Does $n^2 \in O(n)$? No.

We will prove by contradiction that the definition cannot be satisfied.

- Assume that $n^2 \in O(n)$. From the definition of Big Oh, there must exist $c > 0$ and integer $N > 0$ such that $0 \leq n^2 \leq cn$ for all $n \geq N$.
- Divide the inequality by $n > 0$ to get $0 \leq n \leq c$ for all $n \geq N$.
- $n \leq c$ cannot be true for any $n > \max\{c, N\}$. This contradicts the assumption. Thus, $n^2 \notin O(n)$.

Are they true? Why or why not?

- **$1{,}000{,}000\ n^2 \in O(n^2)$?**
- **True**

- **$(n - 1)n / 2 \in O(n^2)$?**
- **True**

- **$n / 2 \in O(n^2)$?**
- **True**

- **$\lg (n^2) \in O(\lg n)$?**
- **True**

- **$n^2 \in O(n)$?**
- **False**

Omega

Asymptotic lower bound on the growth of an algorithm or a problem

When do we use Omega?

1. To provide information on the minimum number of operations that an algorithm performs
   - Insertion sort is $\Omega(n)$ in the best case
     - This means that in the best case its instruction count is at least cn
   - It is $\Omega(n^2)$ in the worst case
     - This means that in the worst case its instruction count is at least cn^2

Omega (cont.)

2. To provide information on a class of algorithms that solve a problem

- Sorting algorithms based on comparisons of keys are Ω(nlgn) in the worst case
  - This means that all sort algorithms based only on comparisons of keys have to do at least *cn*lg*n* operations
- Any algorithm based only on comparisons of keys to find the maximum of *n* elements is Ω(n) in every case
  - This means that all algorithms only based on key comparisons to find maximum have to do at least *cn* operations

Supplementary topic: Why $\Omega(n\lg n)$ for sorting?

- n numbers to sort with no further information or assumption about them
- n! permutations: A decision tree (full binary tree) with n! leaf nodes
- One comparison has only two outcomes
- So, lg(n!) comparisons are required in the worst case
- n! is approximately equal to $(n/e)^n$

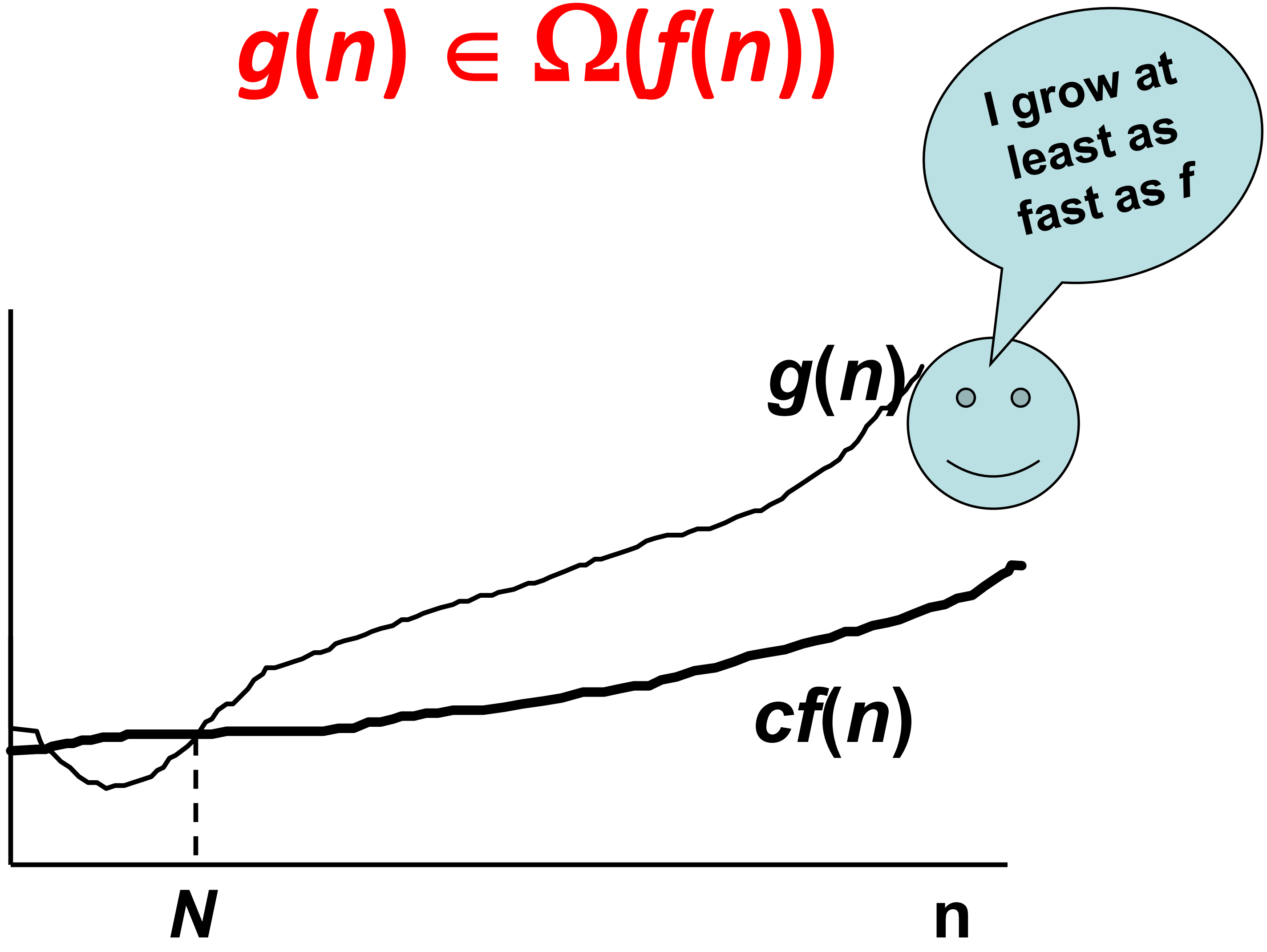
Definition of the set Omega

- $\Omega(f(n))$ is the set of functions $g(n)$ such that there exist positive constants c and N for which

  $$0 \le cf(n) \le g(n) \text{ for all } n \ge N$$

- g(n) = $\Omega(f(n))$: $f(n)$ is an *asymptotically lower bound* for $g(n)$

$g(n) \in \Omega(f(n))$



Is $5n\text{-}20 \in \Omega(n)$?

Proof: From the definition of Omega, there must exist c > 0 and integer N>0 such that $0 \leq cn \leq 5n\text{-}20$ for all $n \geq N$

Dividing the inequality by $n > 0$ we get: $0 \leq c \leq 5\text{-}20/n$ for all $n \geq N$.

$20/n \leq 20$, and $20/n$ becomes smaller as n grows.

There are many choices here for c and N.

Since $c > 0$, $5 - 20/n > 0$ and $N > 4$.

If we choose c=1, then $5 - 20/n \geq 1$ and $N \geq 5$ Choose $N = 5$.

If we choose c=4, then $5 - 20/n \geq 4$ and $N \geq 20$. Choose $N = 20$.

In either case (we only need one!) we have c>o and N>0 such that $0 \leq cn \leq 5n\text{-}20$ for all $n \geq N$. So $5n\text{-}20 \in \Omega(n)$.

Are they true?

- **1,000,000 $n^2 \in \Omega(n^2)$ why /why not?**
  - **true**
- **$(n - 1)n / 2 \in \Omega(n^2)$ why /why not?**
  - **true**

- **$n / 2 \in \Omega(n^2)$ why /why not?**
  - **(false)**
- **$\lg(n^2) \in \Omega(\lg n)$ why /why not?**
  - **(true)**
- **$n^2 \in \Omega(n)$ why /why not?**
  - **(true)**

Reminder of Important Policies

- Grading:
  - Relative but final (will take curve)
  - “A” for Top 10 Students
  - Minimum 60 in each exam to pass
- Projects: C/C++ in Linux
- Academic Honesty
  - Zero on the first violation
  - F on the second violation
- Refer to syllabus for more details

Theta

- Asymptotic tight bound on the growth rate of an algorithm
  - Insertion sort is $\Theta(n^2)$ in the worst and average cases
    - This means that in the worst case and average cases insertion sort performs cn^2 operations
  - Binary search is $\Theta(\lg n)$ in the worst and average cases
    - This means that, in the worst case and average cases, binary search performs $c\lg n$ operations

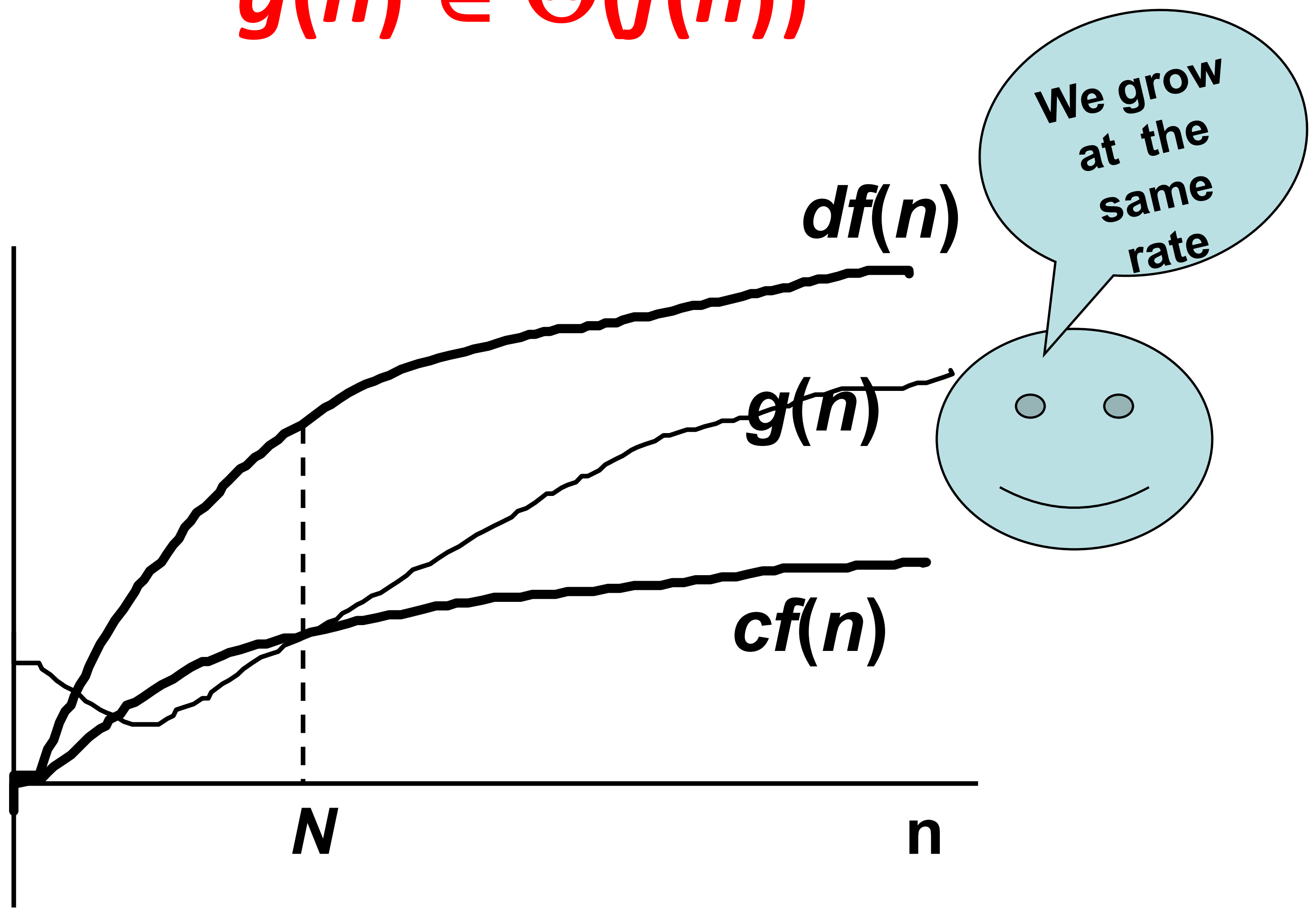
Definition of Theta

- $\Theta(f(n))$ is the set of functions g(n) such that there exist positive constants c, d, and N for which

  $0 \leq cf(n) \leq g(n) \leq df(n)$ for all $n \geq N$

- g(n) is $\Theta(f(n))$: f(n) is an asymptotic tight bound for g(n)

$g(n) \in \Theta(f(n))$



$$\text{Does } \frac{1}{2}n^2 - 3n = \Theta(n^2)?$$

- We use the last definition and show:

1.

$$\frac{1}{2}n^2 - 3n = O(n^2)$$

2.

$$\frac{1}{2}n^2 - 3n = \Omega(n^2)$$

Does $\frac{1}{2}n^2 - 3n = O(n^2)$?

From the definition there must exist $c > 0$, and $N > 0$ such that

$$0 \le \frac{1}{2}n^2 - 3n \le cn^2 \text{ for all } n \ge N.$$

Dividing the inequality by $n^2 > 0$ we get:

$$0 \le \frac{1}{2} - \frac{3}{n} \le c \text{ for all } n \ge N.$$

Clearly any $c \ge 1/2$ can be chosen

Choose $c = 1/2$.

$0 \le \frac{1}{2} - \frac{3}{n} \le \frac{1}{2}$ for all $N \ge 6$. Choose $N = 6$

Does $\frac{1}{2}n^2 - 3n = \Omega(n^2)$?

There must exist $c > 0$ and $N > 0$ such that

$0 \leq cn^2 \leq \frac{1}{2}n^2 - 3n$ for all $n \geq N$

Dividing by $n^2 > 0$ we get

$$0 \leq c \leq \frac{1}{2} - \frac{3}{n}.$$

Since $c > 0$, $0 < \frac{1}{2} - \frac{3}{N}$ and $N > 6$.

Since $3/n > 0$ for finite n, $c < 1/2$. Choose $c = 1/4$.

$\frac{1}{4} \leq \frac{1}{2} - \frac{3}{n}$ for all $n \geq 12$.

So $c = 1/4$ and $N = 12$.

More Θ

- **1,000,000 $n^2 \in \Theta(n^2)$ why /why not?**
  - **True**
- **$(n - 1)n / 2 \in \Theta(n^2)$ why /why not?**
  - **True**
- **$n / 2 \in \Theta(n^2)$ why /why not?**
  - **False**
- **$\lg (n^2) \in \Theta (\lg n)$ why /why not?**
  - **True**
- **$n^2 \in \Theta(n)$ why /why not?**
  - **False**

small o

- $o(f(n))$ is the set of functions $g(n)$ which satisfy the following condition:
- *g(n)* is *o(f(n))*: For *every* positive real constant c, there exists a positive integer *N*, for which

$$g(n) \leq cf(n) \text{ for all } n \geq N$$

small o

- Little “oh” - used to denote an upper bound that is not asymptotically tight.
  - n is in $o(n^3)$
  - n is **not** in $o(n)$

small omega

- $\omega(f(n))$ is the set of functions $g(n)$ which satisfy the following condition:
- $g(n)$ is $\omega(f(n))$: For *every* positive real constant c, there exists a positive integer N, for which

  $$g(n) \geq cf(n) \text{ for all } n \geq N$$

small omega: ω

- Little "oh" - used to denote an upper bound that is not asymptotically tight.
  - n^3 is in ω(n)
  - n is **<u>not</u>** in $\omega(n)$

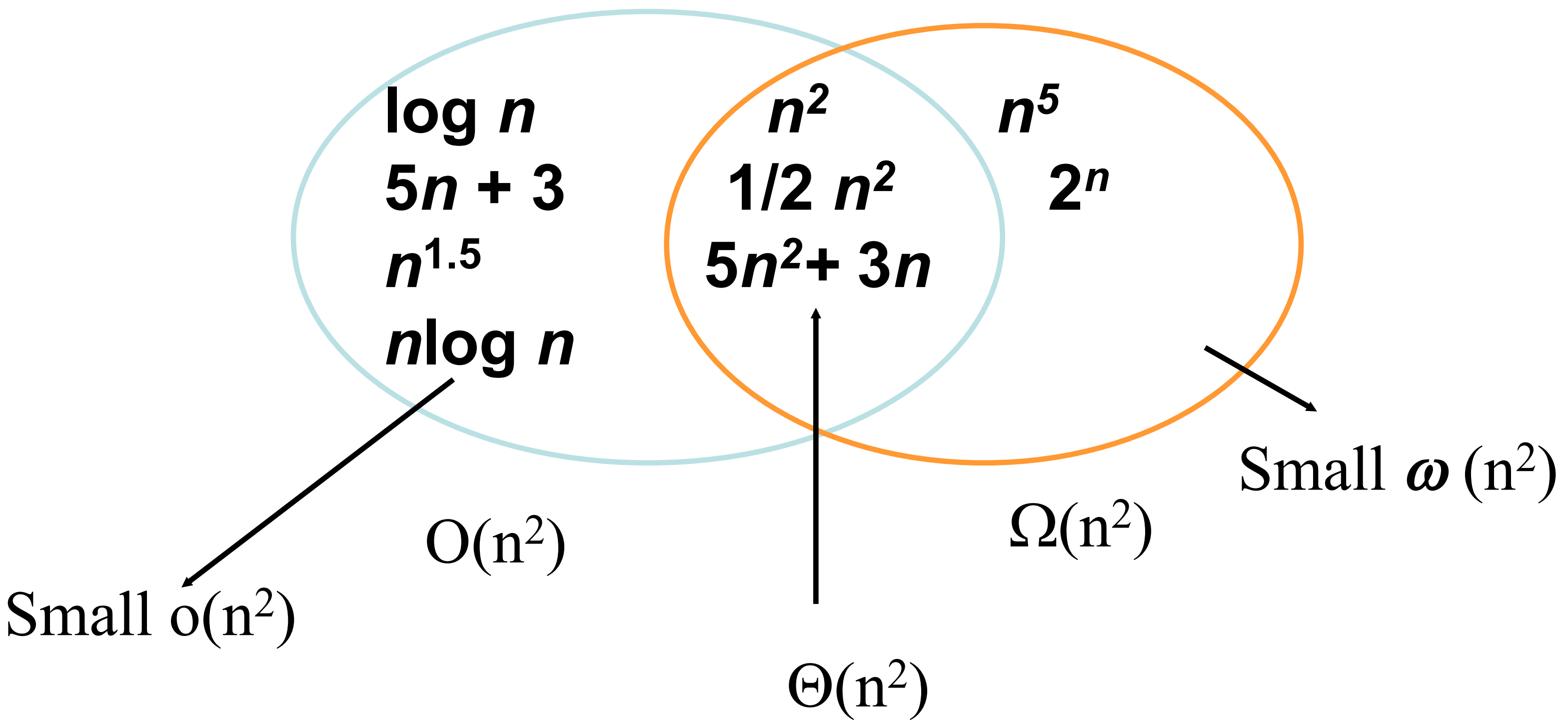
small omega and small o

- ***$g(n) \in \omega(f(n))$ if and only if***
  $f(n) \in o(g(n))$

- ***Example: $g(n) = n^2$, $f(n) = n$. Observe that***
  ***n^2 = ω*(n) and n = o(n^2).**

Comprehensive Example

$$\Theta(f(n)) = O(f(n)) \bigcap \Omega(f(n))$$



Limits can be used to determine Order

$$\text{if } \lim_{n\to\infty} f(n)/g(n) = \begin{cases} c \text{ then } f(n) = \Theta(g(n)) \text{ if } c > 0 \\ 0 \text{ then } f(n) = o(g(n)) \\ \infty \text{ then } f(n) = \omega(g(n)) \end{cases}$$

- We can use this method if the limit exists

Example using limits

$$5n^3 + 3n \in \omega(n^2)$$

$$\lim_{n\to\infty} \frac{5n^3 + 3n}{n^2} = \lim_{n\to\infty} \frac{5n^3}{n^2} + \lim_{n\to\infty} \frac{3n}{n^2} = \infty$$

L'Hopital's Rule

If *f*(x) and *g*(x) are both differentiable with derivatives *f'*(x) and *g'*(x), respectively, and if

$$\lim_{x\to\infty} g(x) = \lim_{x\to\infty} f(x) = \infty \text{ then}$$

$$\lim_{x\to\infty} \frac{f(x)}{g(x)} = \lim_{x\to\infty} \frac{f'(x)}{g'(x)}$$

whenever the limit on the right exists

Example using limits

$10n^3 - 3n \in \Theta(n^3)$ since,

$$\lim_{n\to\infty}\frac{10n^3 - 3n}{n^3} = \lim_{n\to\infty}\frac{10n^3}{n^3} - \lim_{n\to\infty}\frac{3n}{n^3} = 10$$

$n\log_e n \in o(n^2)$ since,

$$\lim_{n\to\infty}\frac{n\log_e n}{n^2} = \lim_{n\to\infty}\frac{\log_e n}{n} = ? \quad \text{Use L'Hopital's Rule:}$$

$$\lim_{n\to\infty}\frac{(\log_e n)'}{(n)'} = \lim_{n\to\infty}\frac{1/n}{1} = 0$$

$$y = \log_a x$$

$$y^{(k)} = \frac{(-1)^{k-1}(k-1)!}{x^k \ln a}$$ (k^{th} order differentiation of y)

Example using limit

$$\lg n \in o(n)$$

$$\lg n = \frac{\ln n}{\ln 2} \text{ and } (\lg n)' = \left(\frac{\ln n}{\ln 2}\right)' = \frac{1}{n \ln 2}$$

$$\lim_{n\to\infty} \frac{\lg n}{n} = \lim_{n\to\infty} \frac{(\lg n)'}{n'} = \lim_{n\to\infty} \frac{1}{n \ln 2} = 0$$

Comparing ln *n* with n^k ($k > 0$)

- Using limits we get:

$$\lim_{n\to\infty}\frac{\ln n}{n^k} = \lim_{n\to\infty}\frac{1}{kn^k} = 0$$

- So $\ln n = o(n^k)$ for any $k > 0$
- When the exponent k is very small, we need to look at very large values of n to see that $n^k > \ln n$

Example using limits

$n^k \in o(2^n)$ where k is a positive integer

$2^n = e^{n \ln 2}$

$(2^n)' = (e^{n \ln 2})' = \ln 2 e^{n \ln 2} = \ln 2 (2^n)$

$Note: (e^x)' = x'(e^x)$

$$\lim_{n \to \infty} \frac{n^k}{2^n} = \lim_{n \to \infty} \frac{kn^{k-1}}{2^n \ln 2} = \lim_{n \to \infty} \frac{k(k-1)n^{k-2}}{2^n \ln^2 2} = \ldots =$$

$$= \lim_{n \to \infty} \frac{k!}{2^n \ln^k 2} = 0$$

Summary: Quick Check vs. Proof

- *For a quick check, use analogy:*
  - $f(n) = O(g(n)) \quad \approx \quad f(n) \leq g(n)$
  - $f(n) = \Omega(g(n)) \quad \approx \quad f(n) \geq g(n)$
  - $f(n) = \Theta(g(n)) \quad \approx \quad f(n) = g(n)$
  - $f(n) = o(g(n)) \quad \approx \quad f(n) < g(n)$
  - $f(n) = \omega(g(n)) \quad \approx \quad f(n) > g(n)$
- *To prove, use formal definitions discussed before*

Transitivity:

If $f(n) = \Theta(g(n))$ and $g(n) = \Theta(h(n))$ then $f(n) = \Theta(h(n))$.

If $f(n) = O(g(n))$ and $g(n) = O(h(n))$ then $f(n) = O(h(n))$.

If $f(n) = \Omega(g(n))$ and $g(n) = \Omega(h(n))$ then $f(n) = \Omega(h(n))$.

If $f(n) = o(g(n))$ and $g(n) = o(h(n))$ then $f(n) = o(h(n))$.

If $f(n) = \omega(g(n))$ and $g(n) = \omega(h(n))$ then $f(n) = \omega(h(n))$

Reflexivity:

- $f(n) = \Theta(f(n))$.
- $f(n) = O(f(n))$.
- $f(n) = \Omega(f(n))$.
- "o" is not reflexive
- "ω" is not reflexive

- Example: f(n) = n
- Other examples?

Symmetry and Transpose symmetry

- Symmetry:

  $f(n) = \Theta(g(n))$ if and only if $g(n) = \Theta(f(n))$

- Transpose symmetry:

  $f(n) = O(g(n))$ if and only if $g(n) = \Omega(f(n))$

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

- Examples?

Order of Algorithm

- <u>Property</u>
  - Complexity Categories:

$\theta(\lg n)$ $\theta(n)$ $\theta(n \lg n)$ $\theta(n^2)$ $\theta(n^j)$ $\theta(n^k)$ $\theta(a^n)$ $\theta(b^n)$ $\theta(n!)$

Where $k>j>2$ and $b>a>1$. If a complexity function $g(n)$ is in a category that is to the left of the category containing $f(n)$, then $g(n) \in o(f(n))$

Values for $\log_{10} n$ *and* $n^{0.01}$

		0.01	
	n	log n	$n^{.01}$
	1	0	1
	1.00E+10	10	1.258925
	1.00E+100	100	10
	1.00E+200	200	100
	1.00E+230	230	199.5262
	1.00E+240	240	251.1886

Values for $\log_{10} n$ *and* $n^{0.001}$

n	$\log n$	$n^{.001}$
1	0	1
1.00E+10	10	1.023293
1.00E+100	100	1.258925
1E+1000	1000	10
1E+2000	2000	100
1E+3000	3000	1000
1E+4000	4000	10000

Lower-order terms and constants

- Lower order terms of a function do not matter since lower-order terms are dominated by the higher order term
- Constants (multiplied by highest order term) do not matter, since they do not affect the asymptotic growth rate
- All logarithms with base b >1 belong to Θ(lg n), since

$$\log_b n = \frac{\lg n}{\lg b} = c \lg n \text{ where } c \text{ is a constant}$$

General Rules

- We say a function $f(n)$ is polynomially bounded if $f(n) = O(n^k)$ for some positive constant k
- We say a function $f(n)$ is polylogarithmic bounded if $f(n) = O(\lg^k n)$ for some positive constant k
- Exponential functions
  - grow faster than positive polynomial functions
- Polynomial functions
  - grow faster than polylogarithmic functions

Asymptotic Growth Rate Part II (Advanced)

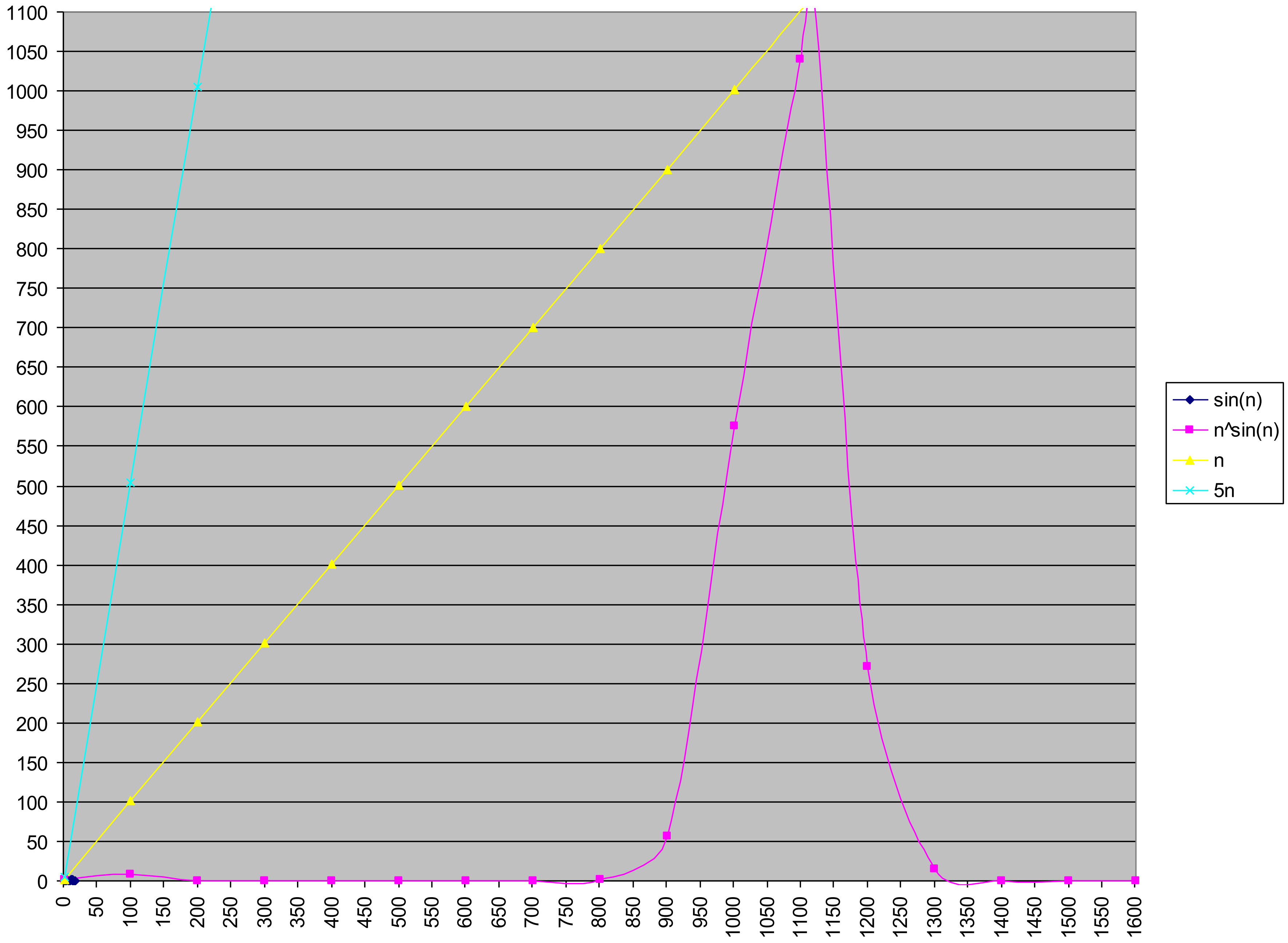
More properties

- The following slides show
  - Examples in which a pair of functions are not comparable in terms of asymptotic notation

Are n and $n^{\sin n}$ comparable with respect to growth rate? yes

$\sin n$	$n^{\sin n}$
Increases from 0 to 1	Increases from 1 to n
Decreases from 1 to 0	Decreases from n to 1
Decreases from 0 to –1	Decreases from 1 to 1/n
Increases from –1 to 0	Increases from 1/n to 1

Clearly $n^{\sin n} = O(n)$, but $n^{\sin n} \neq \Omega(n)$

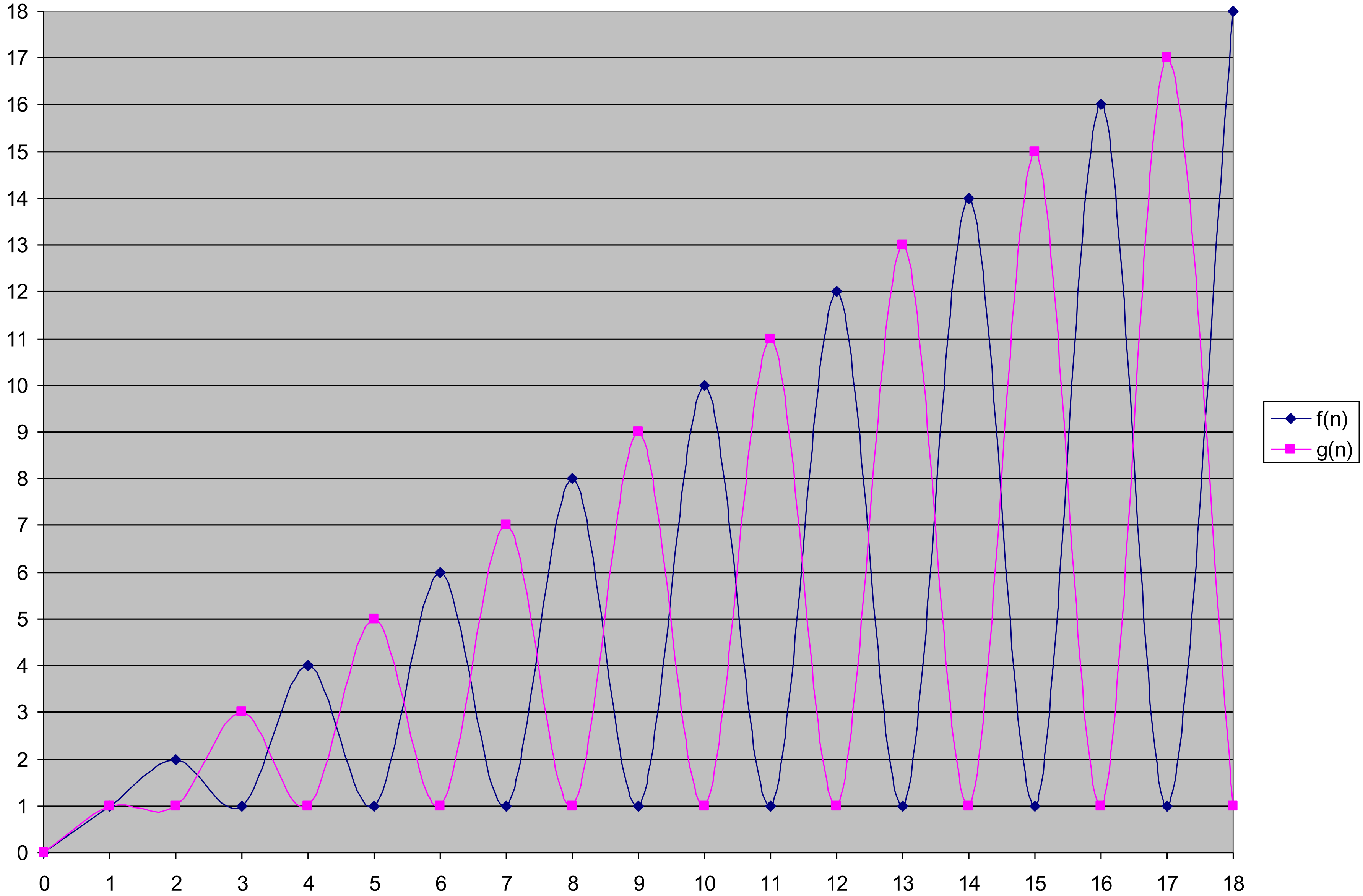

1100
1050
1000
950
900
850
800
750
700
650
600
550
500
450
400
350
300
250
200
150
100
50
0
0
50
100
150
200
250
300
350
400
450
500
550
600
650
700
750
800
850
900
950
1000
1050
1100
1150
1200
1250
1300
1350
1400
1450
1500
1550
1600
sin(n)
n^sin(n)
n
5n

Another example

The following functions are not asymptotically comparable:

$$f(n) = \begin{cases} n \text{ for even } n \\ 1 \ \text{ for odd } n \end{cases} \qquad g(n) = \begin{cases} 1 \text{ for even } n \\ n \ \text{ for odd } n \end{cases}$$

$$f(n) \notin O(g(n)), \text{ and } f(n) \notin \Omega(g(n)),$$



18
17
16
15
14
13
12
11
10
9
8
7
6
5
4
3
2
1
0
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
f(n)
g(n)