Lecture Two- Algorithm Analysis

Data Structure and Algorithm Analysis

Algorithm analysis

- Studies computing resource requirements of different algorithms
- Computing Resources
  - Running time (Most precious)
  - Memory usage
  - Communication bandwidth etc
- Why need algorithm analysis ?
  - Writing a working program is not good enough
    - The program may be inefficient!
    - If the program is run on a large data set, then the running time becomes an issue
- Goal is to pick up an efficient algorithm for the problem at hand

Reasons to perform analyze algorithms

- It enables us to:
  - Predict performance of algorithms
  - Compare algorithms.
  - Provide guarantees on running time/space of algorithms
  - Understand theoretical basis.
- Primary practical reason: **avoid performance bugs.**
  - client gets poor performance because programmer did not understand performance characteristics

How to Measure Efficiency/performance?

- Two approaches to measure algorithms efficiency/performance
  - Empirical
    - Implement the algorithms and
    - Trying them on different instances of input
    - Use/plot actual clock time to pick one
  - Theoretical/Asymptotic Analysis
    - Determine quantity of resource required mathematically needed by each algorithms

Example- Empirical

Input size → **Actual clock time**

N	time (seconds) †
250	0.0
500	0.0
1,000	0.1
2,000	0.8
4,000	6.4
8,000	51.1
16,000	?

Drawbacks of empirical methods

- It is difficult to use actual clock because clock time varies based on
  - Specific processor speed
  - Current processor load
  - Specific data for a particular run of the program
    - Input size
    - Input properties
  - Programming language (C++, java, python …)
  - The programmer (You, Me, Billgate …)
  - Operating environment/platform (PC, sun, smartphone etc)
- Therefore, it is quite machine dependent

Machine independent analysis

- Critical resources:
  - Time, Space (disk, RAM), Programmer's effort, Ease of use (user's effort).
- Factors affecting running time:
  - System dependent effects.
    - Hardware: CPU, memory, cache, …
    - Software: compiler, interpreter, garbage collector, …
    - System: operating system, network, other apps, …
  - System independent effects
    - Algorithm.
    - Input data/ Problem size

Machine independent analysis…

- For most algorithms, running time depends on "size" of the input.
  - Size is often the number of inputs processed
  - Example:- in searching problem, size is the no of items to be sorted
- Running time is expressed as T(n) for some function T on input size n.

Machine independent analysis

- Efficiency of an algorithm is measured in terms of the number of basic operations it performs.
  - Not based on actual time-clock
- We assume that every basic operation takes constant time.
  - Arbitrary time
- Examples of Basic Operations:
  - Single Arithmetic Operation (Addition, Subtraction, Multiplication)
  - Assignment Operation
  - Single Input/Output Operation
  - Single Boolean Operation
  - Function Return
- We do not distinguish between the basic operations.
- Examples of Non-basic Operations are
  - Sorting, Searching.

Examples: Count of Basic Operations T(n)

- Sample Code

```
int count()
{
  Int k=0;
  cout<< "Enter an integer";
  cin>>n;
  for (i = 0;i < n;i++)
    k = k+1;
  return 0;
}
```

Examples: Count of Basic Operations T(n)

Sample Code

```
int count()
{
Int k=0;
cout<< "Enter an integer";
cin>>n;
for (i = 0;i < n;i++)
   k = k+1;
return 0;
}
```

Count of Basic Operations (Time Units)

- 1 for the assignment statement: int k=0
- 1 for the output statement.
- 1 for the input statement.
- In the for loop:
  - 1 assignment, n+1tests, and n increments.
  - n loops of 2 units for an assignment, and an addition.
- 1 for the return statement.

- $T(n) = 1+1+1+(1+n+1+n)+2n+1 = 4n+6$

Examples: Count of Basic Operations T(n)

```
int total(int n)
{
  Int sum=0;
  for (int i=1;i<=n;i++)
            sum=sum+i;
  return sum;
}
```

Examples: Count of Basic Operations T(n)

Sample Code

```
int total(int n)
{
Int sum=0;
for (inti=1;i<=n;i++)
        sum=sum+i;
return sum;
}
```

Count of Basic Operations (Time Units)

- 1 for the assignment statement: int sum=0
- In the for loop:
  - 1 assignment, n+1tests, and n increments.
  - n loops of 2 units for an assignment, and an addition.
- 1 for the return statement.

- $T(n) = 1+ (1+n+1+n)+2n+1 = 4n+4$

Examples: Count of Basic Operations T(n)

```
void func()
{
 Int x=0;
 Int i=0;
 Int j=1;
 cout<< "Enter an Integer value";
 cin>>n;
 while (i<n){
           x++;
           i++;
 }
 while (j<n)
 {
   j++;
 }
}
```

Examples: Count of Basic Operations T(n)

Sample Code

```
void func()
{
 Int x=0;
 Int i=0;
 Int j=1;
 cout<< "Enter an Integer value";
 cin>>n;
 while (i<n){
           x++;
           i++;
 }
 while (j<n)
 {
   j++;
 }
}
```

Count of Basic Operations (Time Units)

- 1 for the first assignment statement: x=0;
- 1 for the second assignment statement: i=0;
- 1 for the third assignment statement: j=1;
- 1 for the output statement.
- 1 for the input statement.
- In the first while loop:
  - n+1tests
  - n loops of 2 units for the two increment (addition) operations
- In the second while loop:
  - n tests
  - n-1 increments
- T (n) = 1+1+1+1+1+n+1+2n+n+n-1 = 5n+5

Examples: Count of Basic Operations T(n)

- Sample Code

```
int sum (int n)
{
int partial_sum= 0;
for (int i = 1; i <= n; i++)
partial_sum= partial_sum+ (i * i * i);
return partial_sum;
}
```

Examples: Count of Basic Operations T(n)

Sample code

```
int sum (int n)
{
int partial_sum= 0;
for (int i = 1; i <= n; i++)
partial_sum= partial_sum+ (i * i *
i);
return partial_sum;
}
```

Count of Basic Operations (Time Units)

- 1 for the assignment.
- 1 assignment, n+1tests, and n increments.
- n loops of 4 units for an assignment, an addition, and two multiplications.
- 1 for the return statement.

- $T(n) = 1+(1+n+1+n)+4n+1 = 6n+4$

Simplified Rules to Compute Time Units(Formal Method)

- for Loops:
  - In general, a for loop translates to a summation. The index and bounds of the summation are the same as the index and bounds of the for loop.

```
for ( int i = 1; i <= N; i++) {
    sum = sum+i;
}
```

$\sum_{i=1}^{N} 2 = 2N$

Simplified Rules to Compute Time Units

- Nested Loops:

```
for ( int i = i; i <= N; i++) {
    for ( int j = 1; j <= M; j++) {
        sum = sum+i+j;
    }
}
```

$$\sum_{i=1}^{N} \sum_{j=1}^{M} 3M = 3MN$$

Simplified Rules to Compute Time Units

- Consecutive Statements

```
for ( int i = 1; i <= N; i++) {
sum = sum+i;
}

for ( int i = 1; i <= N; i++) {
    for ( int j = 1; j <= N; j++) {
            sum = sum+i+j;
    }
}
```

$$|\sum_{i=1}^{N} 2| + |\sum_{i=1}^{N} \sum_{j=1}^{N} 3| = 2N + 3N^2$$

Simplified Rules to Compute Time Units

- Conditionals:
  - If (test) s1 else s2: Compute the maximum of the running time for s1 and s2.

```
if (test == 1) {
    for ( int i = 1; i <= N; i++) {
    sum = sum+i;
}}
Else
{
 for ( int i = 1; i <= N; i++) {
    for ( int j = 1; j <= N; j++) {
    sum = sum+i+j;
}}
```

$$\max\left(\sum_{i=1}^{N} 2, \sum_{i=1}^{N}\sum_{j=1}^{N} 3\right) =$$

$$\max\left(2N, 3N^2\right) = 3N^2$$

Example: Computation of Run-time

- Suppose we have hardware capable of executing 10^6 instructions per second. How long would it take to execute an algorithm whose complexity function was $T(n) = 2n^2$ on an input size of $n = 10^8$?

Example: Computation of Run-time

- Suppose we have hardware capable of executing 10^6 instructions per second. How long would it take to execute an algorithm whose complexity function was $T(n) = 2n^2$ on an input size of $n = 10^8$?

The total number of operations to be performed would be $T(10^8)$:

$T(10^8) = 2*(10^8)^2 = 2*10^{16}$

The required number of seconds would be given by

$T(10^8)/10^6$ so:

Running time $= 2*10^{16}/10^6 = 2*10^{10}$

The number of seconds per day is 86,400 so this is about 231,480 days (634 years).

Types of Algorithm complexity analysis

- Best case.
  - Lower bound on cost.
  - Determined by “easiest” input.
  - Provides a goal for all inputs.
- Worst case.
  - Upper bound on cost.
  - Determined by “most difficult” input.
  - Provides a guarantee for all inputs.
- Average case. Expected cost for random input.
  - Need a model for “random” input.
  - Provides a way to predict performance.

Best, Worst and Average Cases

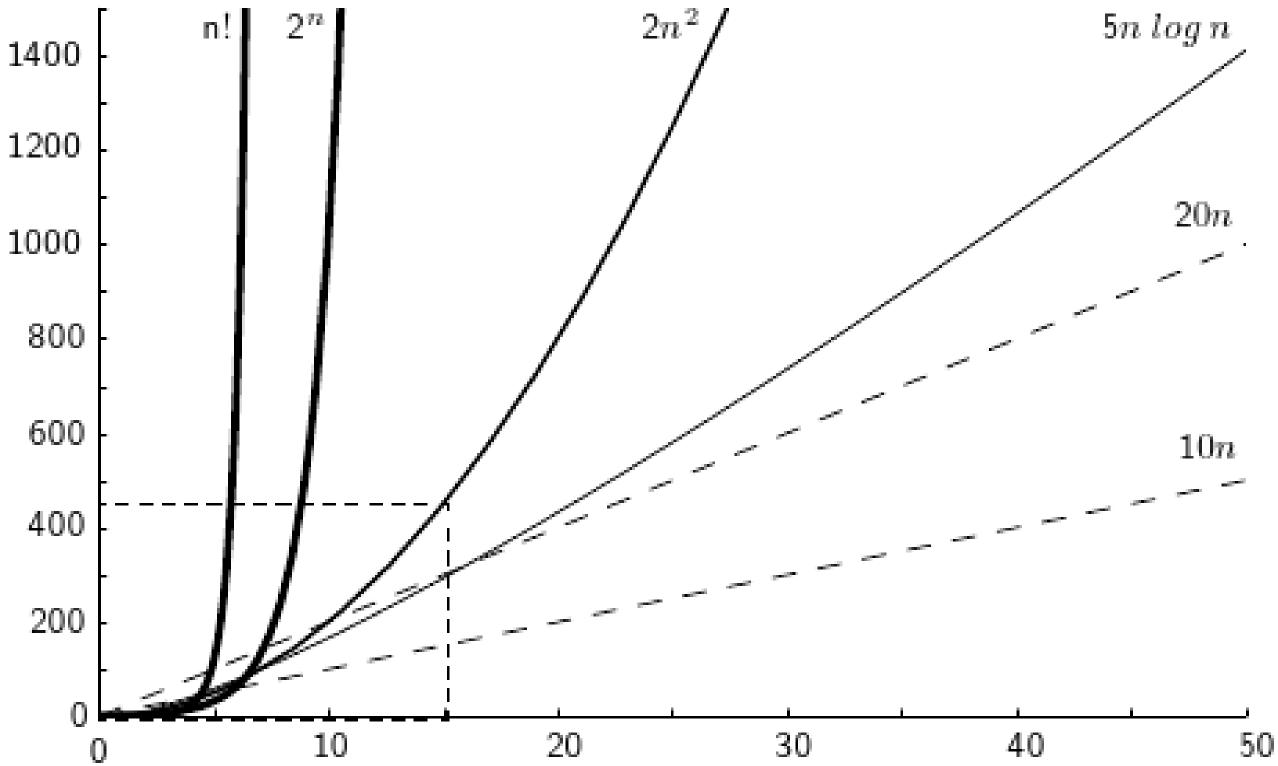
- Not all inputs of a given size take the same time.
- Sequential search for K in an array of n integers:
  - Begin at first element in array and look at each element in turn until K is found.
- Best Case: [Find at first position: 1 compare]
- Worst Case: [Find at last position: n compares]
- Average Case: [$(n + 1)/2$ compares]
- While average time seems to be the fairest measure, it may be difficult to determine.
  - Depends on distribution. Assumption for above analysis: Equally likely at any position.
- When is worst case time important?
  - algorithms for time-critical systems

Order of Growth and Asymptotic Analysis

- Suppose an algorithm for processing a retail store's inventory takes:
  - 10,000 milliseconds to read the initial inventory from disk, and then
  - 10 milliseconds to process each transaction (items acquired or sold).
- Processing n transactions takes (10,000 + 10 n) milliseconds.
- Even though 10,000 >> 10, the "10 n" term will be more important if the number of transactions is very large.
- We also know that these coefficients will change if we buy a faster computer or disk drive, or use a different language or compiler.
  - we want to ignore constant factors (which get smaller and smaller as technology improves)
- In fact, we will not worry about the exact values, but will look at "broad classes" of values.

Growth rates

- The ***growth rate*** for an algorithm is the rate at which the cost of the algorithm grows as the size of its input grows.



Rate of Growth

- Consider the example of buying *elephants* and *goldfish:*

  Cost: cost_of_elephants + cost_of_goldfish

  Cost ~ cost_of_elephants (approximation)

  since the cost of the gold fish is insignificant when compared with cost of elephants

- Similarly, the low order terms in a function are relatively insignificant for **large** *n*

$$n^4 + 100n^2 + 10n + 50 \quad \sim \quad n^4$$

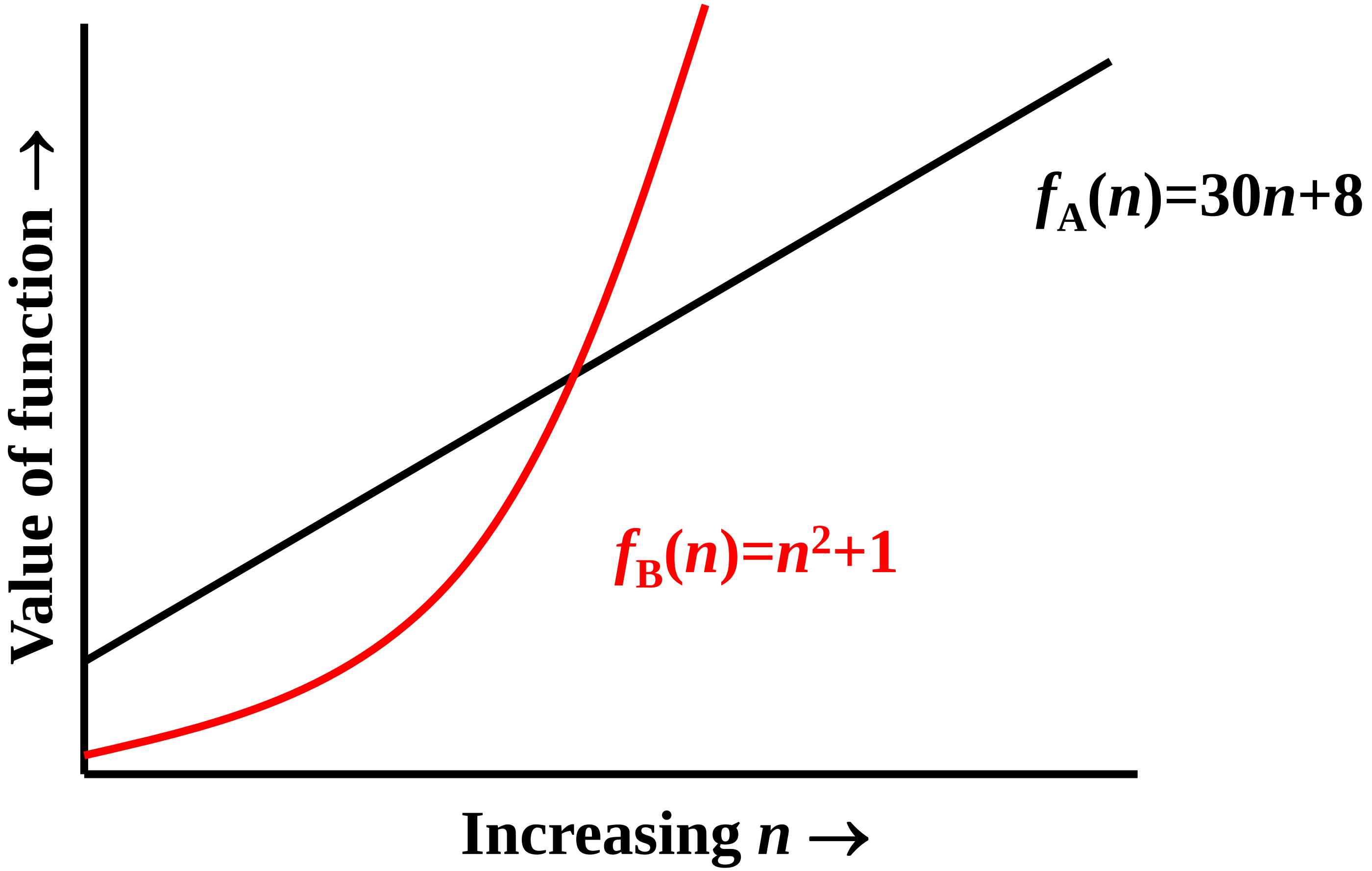
i.e., we say that $n^4 + 100n^2 + 10n + 50$ and n^4 have the same **rate of growth**

More Examples: $f_B(n)=n^2+1 \sim n^2$

- $f_A(n)=30n+8 \sim n$

Visualizing Orders of Growth

- On a graph, as you go to the right, a faster growing function eventually becomes larger...



Asymptotic analysis

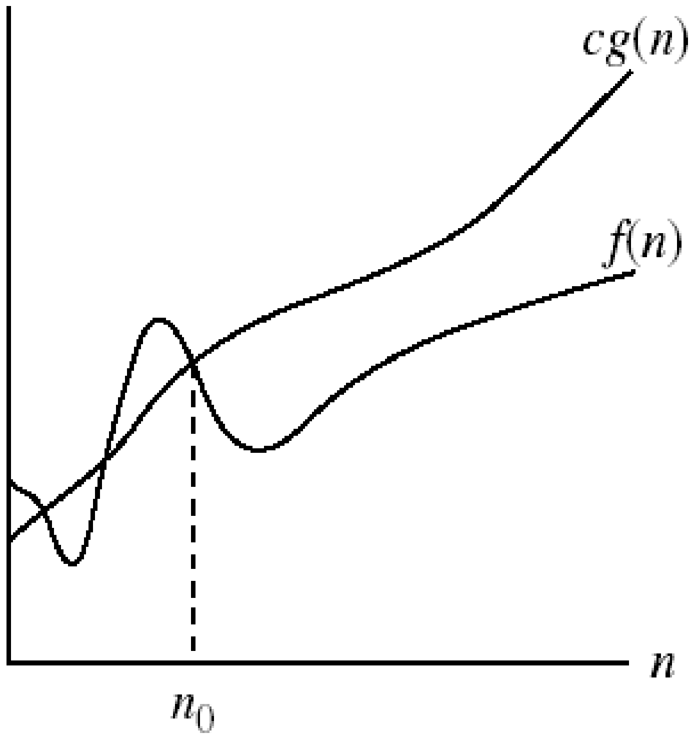
- Refers to the study of an algorithm as the input size "gets big" or reaches a limit.
- To compare two algorithms with running times $f(n)$ and $g(n)$, we need a **rough measure** that characterizes **how fast each function grows- growth rate.**
  - *Ignore constants [especially when input size very large]*
  - *But constants may have impact on small input size*
- Several notations are used to describe the running-time equation for an algorithm.
  - Big-Oh (O), Little-Oh (o)
  - Big-Omega (Ω), Little-Omega(ω)
  - Theta Notation()

Big-Oh Notation

- Definition
  - For f(n) a non-negatively valued function, f(n) is in set O(g(n)) if there exist two positive constants c and n_0 such that $f(n) \leq cg(n)$ 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 > n0), the algorithm always executes in less than cg (n) steps [in best, average or worst case].

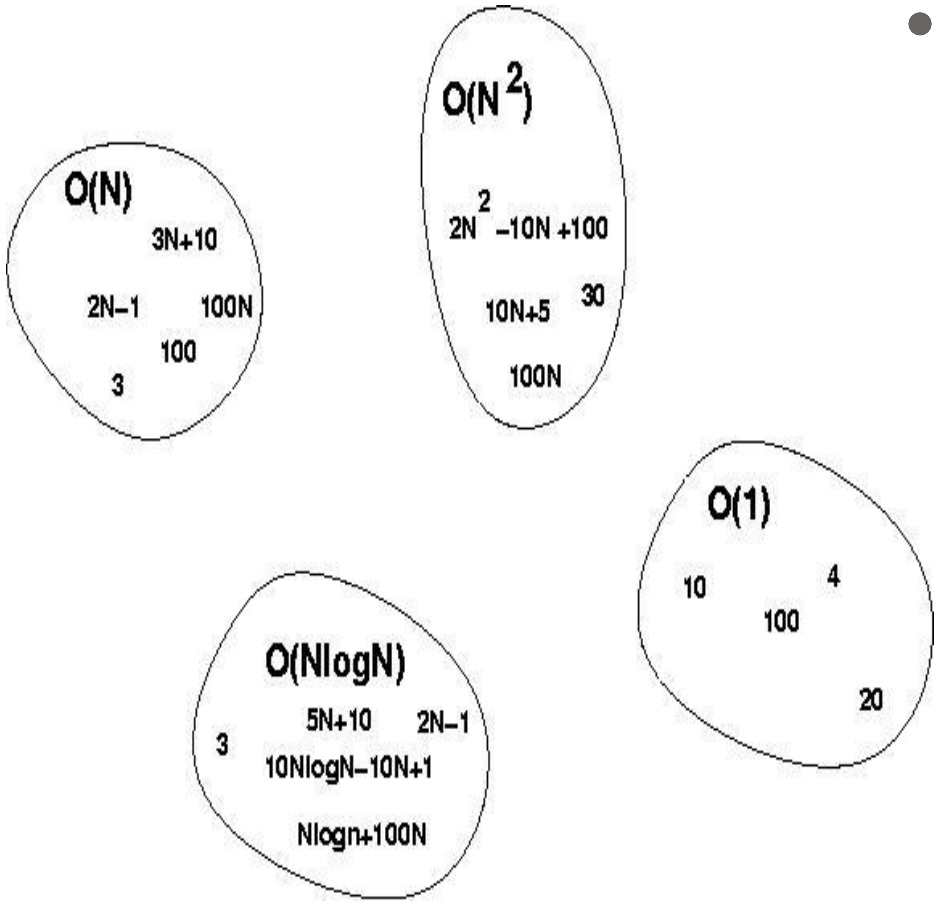
Big-Oh Notation - Visually

$$O(g(n)) = \{f(n) : \text{ there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \le f(n) \le cg(n) \text{ for all } n \ge n_0\}.$$



$g(n)$ is an ***asymptotic upper bound*** for $f(n)$.

Big-O Visualization



- *O*(*g*(*n*)) is the set of functions with smaller or same order of growth as *f*(*n*)
  - Wish tightest upper bound:
  - While $T(n) = 3n^2$ is in $O(n^3)$, we prefer $O(n^2)$.
  - Because, it provides more information to say $O(n^2)$ than $O(n^3)$

Big-O

- Demonstrating that a function f(n) is in big-O of a function g(n) requires that we find specific constants c and n_o for which the inequality holds.

- The following points are facts that you can use for Big-Oh problems:
  - $1 <= n$ for all $n >= 1$
  - $n <= n^2$ for all $n >= 1$
  - $2^n <= n!$ for all $n >= 4$
  - $\log_2 n <= n$ for all $n >= 2$
  - $n <= n\log_2 n$ for all $n >= 2$

Examples

- $f(n) = 10n + 5$ and $g(n) = n$. Show that $f(n)$ is in $O(g(n))$.
  - To show that $f(n)$ is $O(g(n))$ we must show constants c and n_o such that
    - $f(n) <= c.g(n)$ for all $n >= n_o$
    - $10n + 5 <= c.n$ for all $n >= n_o$
    - Try $c = 15$. Then we need to show that $10n + 5 <= 15n$
    - Solving for n we get: $5 < 5n$ or $1 <= n$.
    - So $f(n) =10n + 5 <= 15.g(n)$ for all $n >= 1$.
    - ($c = 15$, $n_o = 1$).

Examples

- $2n^2 = O(n^3)$: $2n^2 \le cn^3 \Rightarrow 2 \le cn \Rightarrow c = 1$ and $n_0 = 2$
- $n^2 = O(n^2)$: $n^2 \le cn^2 \Rightarrow c \ge 1 \Rightarrow c = 1$ and $n_0 = 1$
- $1000n^2+1000n = O(n^2)$:

$1000n^2+1000n \le 1000n^2+ n^2 = 1001n^2 \Rightarrow c=1001$ and $n_0 = 1000$

- $n = O(n^2)$: $n \le cn^2 \Rightarrow cn \ge 1 \Rightarrow c = 1$ and $n_0 = 1$

More Examples

- Show that $30n+8$ is $O(n)$.
  - Show $\exists c, n_0$: $30n+8 \le cn$, $\forall n > n_0$.

    - Let $c=31$, $n_0=8$.
    - Assume $n > n_0 = 8$. Then
    - $cn = 31n = 30n + n > 30n+8$,
    - So $30n+8 < cn$.

No Uniqueness

- There is no unique set of values for n_0 and c in proving the asymptotic bounds

- Prove that $100n + 5 = O(n^2)$
  - $100n + 5 \le 100n + n = 101n \le 101n^2$

    for all $n \ge 5$

    $n_0 = 5$ and $c = 101$ is a solution
  - $100n + 5 \le 100n + 5n = 105n \le 105n^2$

    for all $n \ge 1$

    $n_0 = 1$ and $c = 105$ is also a solution

- Must find **SOME** constants c and n_0 that satisfy the asymptotic notation relation

Order of common functions

Notation	Name	Example
O(1)	Constant	Adding two numbers, c=a+b
O(log n)	Logarithmic	Finding an item in a sorted array with a binary search or a search tree (best case)
O(n)	Linear	Finding an item in an unsorted list or a malformed tree (worst case); adding two n-digit numbers
O(nlogn)	Linearithmic	Performing a Fast Fourier transform; heap sort, quick sort (best case), or merge sort
$O(n^2)$	Quadratic	Multiplying two n-digit numbers by a simple algorithm; adding two n×n matrices; bubble sort (worst case or naive implementation), shell sort, quick sort (worst case), or insertion sort

Some properties of Big-O

- Constant factors are may be ignored
  - For all $k>0$, kf is $O(f)$
- The growth rate of a sum of terms is the growth rate of its fastest growing term.
  - Ex, $an^3 + bn^2$ is $O(n^3)$
- The growth rate of a polynomial is given by the growth rate of its leading term
  - If f is a polynomial of degree d, then f is $O(n^d)$

Implication of Big-Oh notation

- We use Big-Oh notation to say how slowly code might run as its input grows.
- Suppose we know that our algorithm uses at most $O(f(n))$ basic steps for any n inputs, and n is sufficiently large, then we know that our algorithm will terminate after executing at most constant times $f(n)$ basic steps.
- We know that a basic step takes a constant time in a machine.
- Hence, our algorithm will terminate in a constant times $f(n)$ units of time, for all large n.

Other notations

- Reading Assignments

End of Lecture 2

Next Lecture:-Simple Sorting and Searching Algorithms