Introduction to Data Structure

- Program = Algorithm + Data Structure.
- All of you have programmed; thus have already been exposed to algorithms and data structure.
- Perhaps you didn't see them as separate entities;
- Perhaps you saw data structures as simple programming constructs (provided by STL--standard template library).
- However, data structures are quite distinct from algorithms, and very important in their own right.

Objectives

- The main focus of this course is to introduce you to a systematic study of algorithms and data structure.
- The two guiding principles of the course are: abstraction and formal analysis.
- Abstraction: We focus on topics that are broadly applicable to a variety of problems.
- Analysis: We want a formal way to compare two objects (data structures or algorithms).
- In particular, we will worry about "always correct"-ness, and worst-case bounds on time and memory (space).

Course Outline

- Introduction
- Structure and applications
- Object-oriented programming: class and programming
- Linear data structures ---

  sequential storage, stacks, queues
- Linked data structure

  linked lists; operations on lists -- insertion, deletion, merge,
  split, etc; list heads, circular lists, doubly-linked lists
- Trees

  a. binary trees

  b. Huffman trees

  c. AVL trees

  d. path, nodes, branches, deletion, insertion of nodes

  e. traversal of binary trees

  f. application of trees
- Searching
- Sorting
- Graphs
- Algorithm analysis

Algorithm Analysis

- Foundations of Algorithm Analysis and Data Structures.
- Analysis:
  - How to predict an algorithm’s performance
  - How well an algorithm scales up
  - How to compare different algorithms for a problem
- Data Structures
  - How to efficiently store, access, manage data
  - Data structures effect algorithm’s performance

Example Algorithms

Two algorithms for computing the Factorial

Which one is better?

```
int factorial (int n) {
        if (n <= 1)    return 1;
        else   return n * factorial(n-1);
    }
```

```
int factorial (int n) {
        if (n<=1)    return 1;
        else {
                fact = 1;
                for (k=2; k<=n; k++)
                        fact *= k;
                return fact;
        }
    }
```

Examples of famous algorithms

- Newton's root finding
- Fast Fourier Transform
- Compression (Huffman, Lempel-Ziv, GIF, MPEG)
- DES, RSA encryption
- Simplex algorithm for linear programming
- Shortest Path Algorithms (Dijkstra, Bellman-Ford)
- Error correcting codes (CDs, DVDs)
- TCP congestion control, IP routing
- Pattern matching (Genomics)
- Search Engines

Role of Algorithms in Modern World

- Enormous amount of data
  - E-commerce (Amazon, Ebay)
  - Network traffic (telecom billing, monitoring)
  - Database transactions (Sales, inventory)
  - Scientific measurements (astrophysics, geology)
  - Sensor networks. RFID tags
  - Bioinformatics (genome, protein bank)

Max Subsequence Problem

- Given a sequence of integers A1, A2, …, An, find the maximum possible value of a subsequence Ai, …, Aj.
- Numbers can be negative.
- You want a contiguous chunk with largest sum.

- Example: -2, 11, -4, 13, -5, -2
- The answer is 20 (subseq. A2 through A4).

- We will discuss 4 different algorithms, with time complexities $O(n^3)$, $O(n^2)$, $O(n \log n)$, and $O(n)$.
- With $n = 10^6$, algorithm 1 may take > 10 years; algorithm 4 will take a fraction of a second!

Algorithm 1 for Max Subsequence Sum

- Given $A_1,\ldots,A_n$, find the maximum value of $A_i+A_{i+1}+\cdots+A_j$
  0 if the max value is negative

```
int maxSum = 0;

for( int i = 0; i < a.size( ); i++ )
for( int j = i; j < a.size( ); j++ )
{
        int thisSum = 0;
        for( int k = i; k <= j; k++ )
                thisSum += a[ k ];
        if( thisSum > maxSum )
                maxSum   = thisSum;
}
return maxSum;
```

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

- Time complexity: $O(n^3)$

Algorithm 2

- Idea: Given sum from i to j-1, we can compute the sum from i to j in constant time.
- This eliminates one nested loop, and reduces the running time to $O(n^2)$.

```
into maxSum = 0;

for( int i = 0; i < a.size( ); i++ )
        int thisSum = 0;
        for( int j = i; j < a.size( ); j++ )
        {
             thisSum += a[ j ];
             if( thisSum > maxSum )
                  maxSum   = thisSum;
        }
return maxSum;
```

Why Efficient Algorithms Matter

- Suppose $N = 10^6$
- A PC can read/process N records in 1 sec.
- But if some algorithm does N*N computation, then it takes 1M seconds = 11 days!!!

- 100 City Traveling Salesman Problem.
  - A supercomputer checking 100 billion tours/sec still requires 10^{100} years!

- Fast factoring algorithms can break encryption schemes. Algorithms research determines what is safe code length. (> 100 digits)

How to Measure Algorithm Performance

- What metric should be used to judge algorithms?
  - Length of the program (lines of code)
  - Ease of programming (bugs, maintenance)
  - Memory required
  - ❑ Running time

- **Running time is the dominant standard.**
  - Quantifiable and easy to compare
  - Often the critical bottleneck

Abstraction

- An algorithm may run differently depending on:
  - the hardware platform (PC, Cray, Sun)
  - the programming language (C, Java, C++)
  - the programmer (you, me, Bill Joy)
- While different in detail, all hardware and prog models are equivalent in some sense: **Turing machines**.
- It suffices to count basic operations.
- Crude but valuable measure of algorithm's performance *as a function of input size*.

Average, Best, and Worst-Case

- On which input instances should the algorithm's performance be judged?

- Average case:
  - Real world distributions difficult to predict
- Best case:
  - Seems unrealistic
- Worst case:
  - Gives an absolute guarantee
  - **We will use the worst-case measure.**

Examples

- Vector addition **Z = A+B**

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

$T(n) = c\, n$

- Vector (inner) multiplication **z =A*B**

```
z = 0;
for (int i=0; i<n; i++)
    z = z + A[i]*B[i];
```

$T(n) = c' + c_1 n$

Examples

- Vector (outer) multiplication $Z = A*B^T$

```
for (int i=0; i<n; i++)
    for (int j=0; j<n; j++)
        Z[i,j] = A[i] * B[j];
```

$T(n) = c_2 n^2;$

- A program does all the above

$T(n) = c_0 + c_1 n + c_2 n^2;$

Simplifying the Bound

- $T(n) = c_k n^k + c_{k-1} n^{k-1} + c_{k-2} n^{k-2} + \ldots + c_1 n + c_o$
  - too complicated
  - too many terms
  - Difficult to compare two expressions, each with 10 or 20 terms
- Do we really need that many terms?

Simplifications

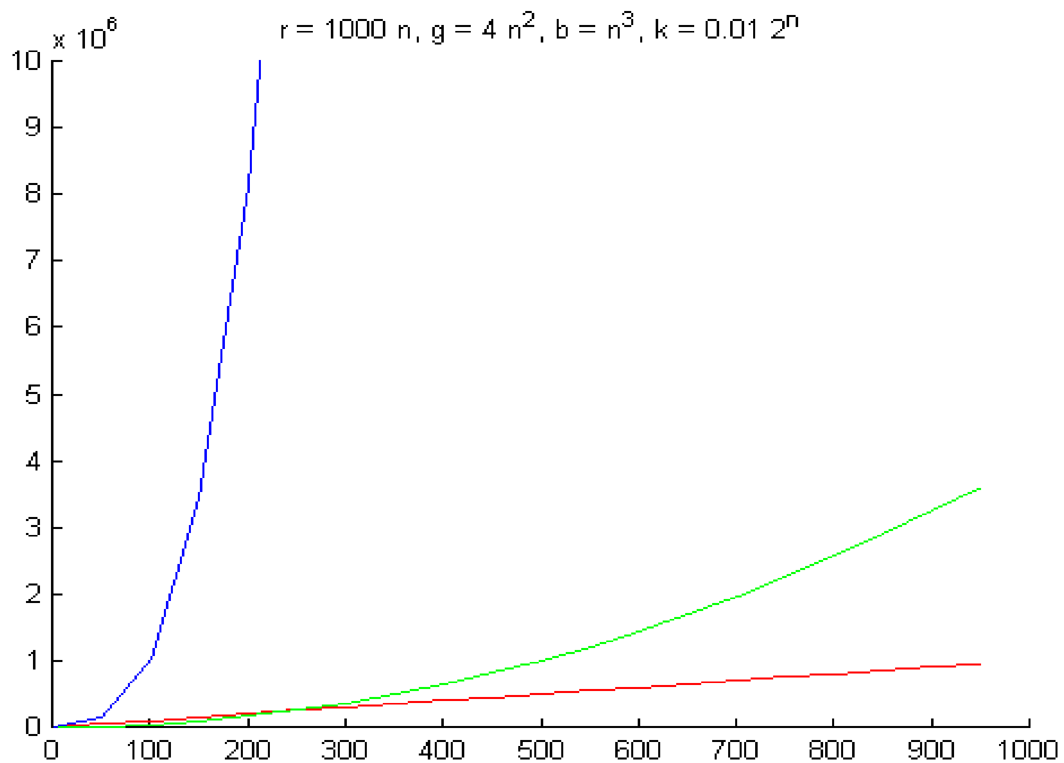
- Keep just one term!
  - the fastest growing term (dominates the runtime)
- No constant coefficients are kept
  - Constant coefficients affected by machines, languages, etc.

- **Asymtotic behavior** (as *n* gets large) is determined entirely by the **leading** term.

  - Example. $T(n) = 10\, n^3 + n^2 + 40n + 800$
    - *If n = 1,000, then T(n) = 10,001,040,800*
    - error is 0.01% if we drop all but the n^3 term
  - In an assembly line the slowest worker determines the throughput rate

Simplification

- Drop the constant coefficient
  - Does not effect the relative order



Simplification

- The *faster* growing term (such as 2^n) *eventually* will outgrow the slower growing terms (e.g., 1000 n) no matter what their coefficients!

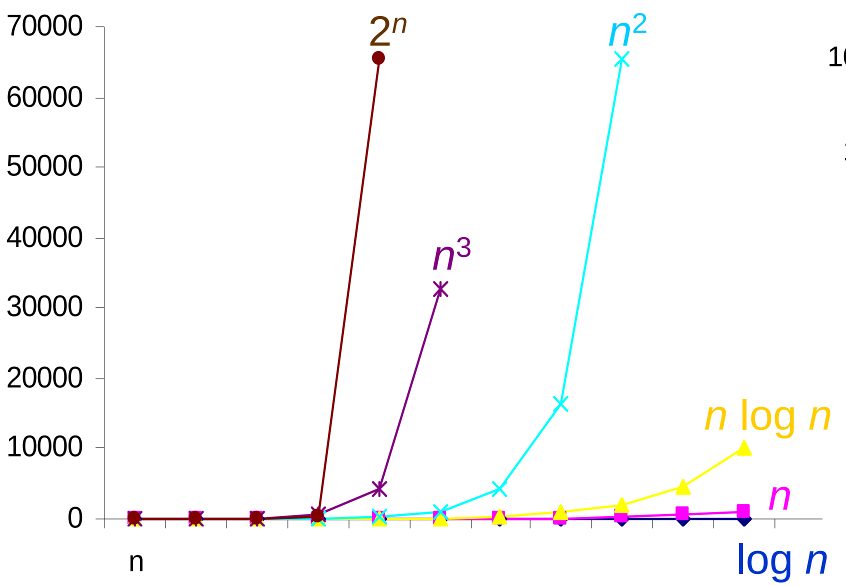
- Put another way, given a certain increase in allocated time, a higher order algorithm will not reap the benefit by solving much larger problem

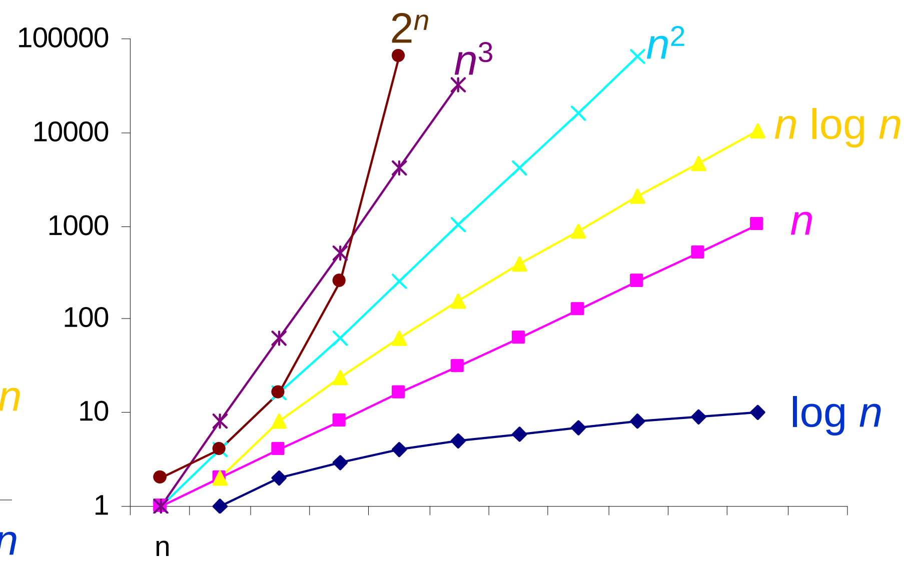
Complexity and Tractability

	$T(n)$						
n	n	$n \log n$	n^2	n^3	n^4	n^{10}	2^n
10	.01µs	.03µs	.1µs	1µs	10µs	10s	1µs
20	.02µs	.09µs	.4µs	8µs	160µs	2.84h	1ms
30	.03µs	.15µs	.9µs	27µs	810µs	6.83d	1s
40	.04µs	.21µs	1.6µs	64µs	2.56ms	121d	18m
50	.05µs	.28µs	2.5µs	125µs	6.25ms	3.1y	13d
100	.1µs	.66µs	10µs	1ms	100ms	3171y	4×10^{13}y
10^3	1µs	9.96µs	1ms	1s	16.67m	3.17×10^{13}y	32×10^{283}y
10^4	10µs	130µs	100ms	16.67m	115.7d	3.17×10^{23}y	
10^5	100µs	1.66ms	10s	11.57d	3171y	3.17×10^{33}y	
10^6	1ms	19.92ms	16.67m	31.71y	3.17×10^{7}y	3.17×10^{43}y	

Assume the computer does 1 billion ops per sec.

$\log n$	n	$n \log n$	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	4096	65,536
5	32	160	1,024	32,768	4,294,967,296





Another View

- More resources (time and/or processing power) translate into large problems solved if complexity is low

T(n)	Problem size solved in 10^3 sec	Problem size solved in 10^4 sec	Increase in Problem size
100n	10	100	10
1000n	1	10	10
$5n^2$	14	45	3.2
N^3	10	22	2.2
2^n	10	13	1.3

Asymptotics

T(n)	keep one	drop coef
$3n^2+4n+1$	$3\ n^2$	n^2
$101\ n^2+102$	$101\ n^2$	n^2
$15\ n^2+6n$	$15\ n^2$	n^2
$a\ n^2+bn+c$	$a\ n^2$	n^2

- They all have the same "growth" rate

Caveats

- Follow the spirit, not the letter
  - a 100n algorithm is more expensive than n^2 algorithm when $n < 100$
- Other considerations:
  - a program used only a few times
  - a program run on small data sets
  - ease of coding, porting, maintenance
  - memory requirements

Asymptotic Notations

- Big-O, "bounded above by": $T(n) = O(f(n))$
  - For some c and N, $T(n) \leq c \cdot f(n)$ whenever $n > N$.

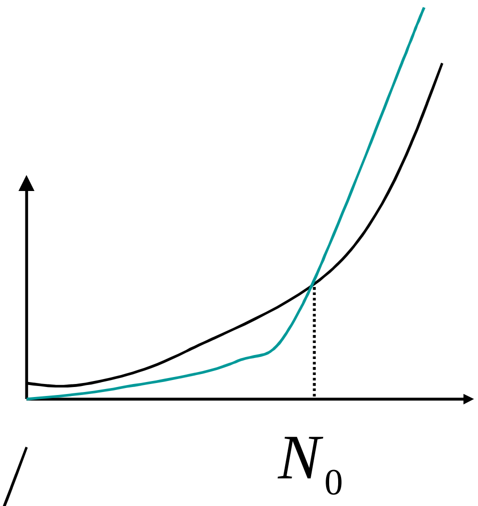
- Big-Omega, "bounded below by": $T(n) = \Omega(f(n))$
  - For some $c>0$ and N, $T(n) \geq c \cdot f(n)$ whenever $n > N$.
  - Same as $f(n) = O(T(n))$.

- Big-Theta, "bounded above and below": $T(n) = \Theta(f(n))$
  - $T(n) = O(f(n))$ and also $T(n) = \Omega(f(n))$

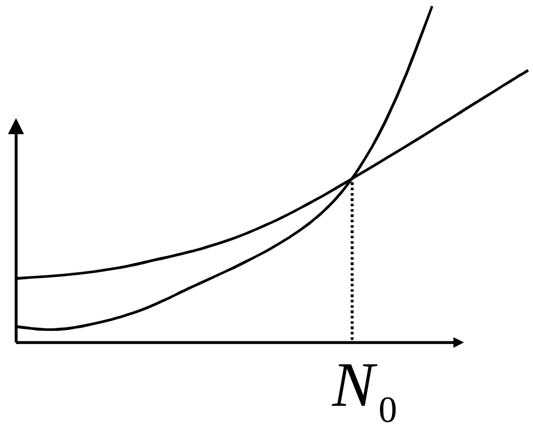
- Little-o, "strictly bounded above": $T(n) = o(f(n))$
  - $T(n)/f(n) \to 0$ as $n \to \infty$

By Pictures

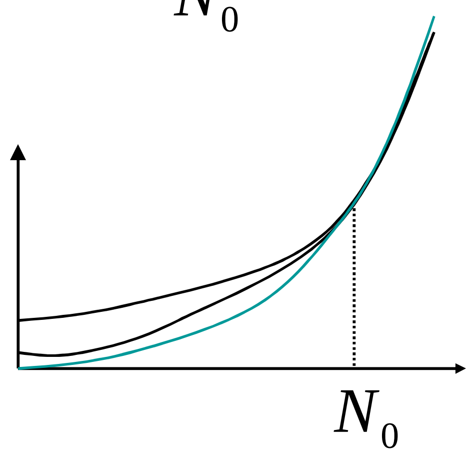
- Big-Oh (most commonly used)
  – bounded above



- Big-Omega
  – bounded below



- Big-Theta
  – exactly
- Small-o
  – not as expensive as ...



Example

$$T(n) = n^3 + 2n^2$$

$O(?)$	$\Omega(?)$
∞	0
n^{10}	n
n^5	n^2
n^3	n^3

Examples

$f(n)$	Asymptomic
c	$\Theta(1)$
$\Sigma_{i=1}^{k} c_i n^i$	$\Theta(n^k)$
$\Sigma_{i=1}^{n} i$	$\Theta(n^2)$
$\Sigma_{i=1}^{n} i^2$	$\Theta(n^3)$
$\Sigma_{i=1}^{n} i^k$	$\Theta(n^{k+1})$
$\Sigma_{i=0}^{n} r^i$	$\Theta(r^n)$
$n!$	$\Theta(n(n/e)^n)$
$\Sigma_{i=1}^{n} 1/i$	$\Theta(\log n)$

Summary (Why O(n)?)

- $T(n) = c_k\, n^k + c_{k-1}\, n^{k-1} + c_{k-2}\, n^{k-2} + \ldots + c_1\, n + c_o$
- Too complicated
- $O(n^k)$
  - a single term with constant coefficient dropped
- Much simpler, extra terms and coefficients *do not matter* asymptotically
- Other criteria hard to quantify

Example

```
for (i=1; i<n; i++)
    if A(i) > maxVal then
        maxVal= A(i);
        maxPos= i;
```

Asymptotic Complexity: O(n)

Example

```
for (i=1; i<n-1; i++)
    for (j=n; j>= i+1;  j--)
        if (A(j-1) > A(j)) then
            temp = A(j-1);
            A(j-1) = A(j);
            A(j) = tmp;
        endif
    endfor
endfor
```

- Asymptotic Complexity is $O(n^2)$

Run Time for Recursive Programs

- T(n) is defined recursively in terms of T(k), k<n
- The recurrence relations allow T(n) to be “unwound” recursively into some base cases (e.g., T(0) or T(1)).
- Examples:
  - Factorial
  - Hanoi towers

Example: Factorial

```
int factorial (int n) {
   if (n<=1)   return 1;
   else  return n * factorial(n-1);
}
```

factorial (n) = n*n-1*n-2* … *1

$$
\begin{aligned}
& T(n) \\
& = T(n-1) + d \\
& = T(n-2) + d + d \\
& = T(n-3) + d + d + d \\
& = ... \\
& = T(1) + (n-1) * d \\
& = c + (n-1) * d \\
& = O(n)
\end{aligned}
$$

n   * factorial(n-1)   T(n)

n-1   * factorial(n-2)   T(n-1)

n-2   * factorial(n-3)   T(n-2)

…

2   *factorial(1)   T(1)

Example: Factorial (cont.)

```
int factorial1(int n) {
  if (n<=1)  return 1;
  else {
    fact = 1;
    for (k=2;k<=n;k++)
        fact *= k;
    return fact;
  }
}
```

$O(1)$

$O(n)$

$O(1)$

- Both algorithms are O(n)