CS 206
Data Structures

Asymptotic Analysis

Complexity

- How many resources will it take to solve a problem of a given size?
  - time
  - space
- Expressed as a function of problem size (beyond some minimum size)
  - how do requirements grow as size grows?
- Problem size
  - number of elements to be handled
  - size of thing to be operated on

The Goal of Asymptotic Analysis

- How to analyze the running time (aka computational complexity) of an algorithm in a theoretical model.
- Using a theoretical model allows us to ignore the effects of
  - Which computer are we using?
  - How good is our compiler at optimization
- We define the running time of an algorithm with input size n as T (n) and examine the rate of growth of T (n) as n grows larger and larger.

Growth Functions

- Constant

  T(n) = c

  ex: getting array element at known location

  any simple C++ statement (e.g. assignment)

- Linear

  T(n) = cn [+ possible lower order terms]

  ex: finding particular element in array of size n (i.e. sequential search)

  trying on all of your n shirts

Growth Functions (cont.)

- Quadratic

  $T(n) = cn^2$ [+ possible lower order terms]

  ex: sorting all the elements in an array (using bubble sort)

  trying all your n shirts with all your n pants

- Polynomial

  $T(n) = cn^k$ [+ possible lower order terms]

  ex: finding the largest element of a k-dimensional array

  looking for maximum substrings in array

Growth Functions (cont.)

- Exponential

  $T(n) = c^n$ [+ possible lower order terms]

  ex: constructing all possible orders of array elements

  Towers of Hanoi (2^n)

  Recursively calculating n^{th} Fibonacci number (2^n)

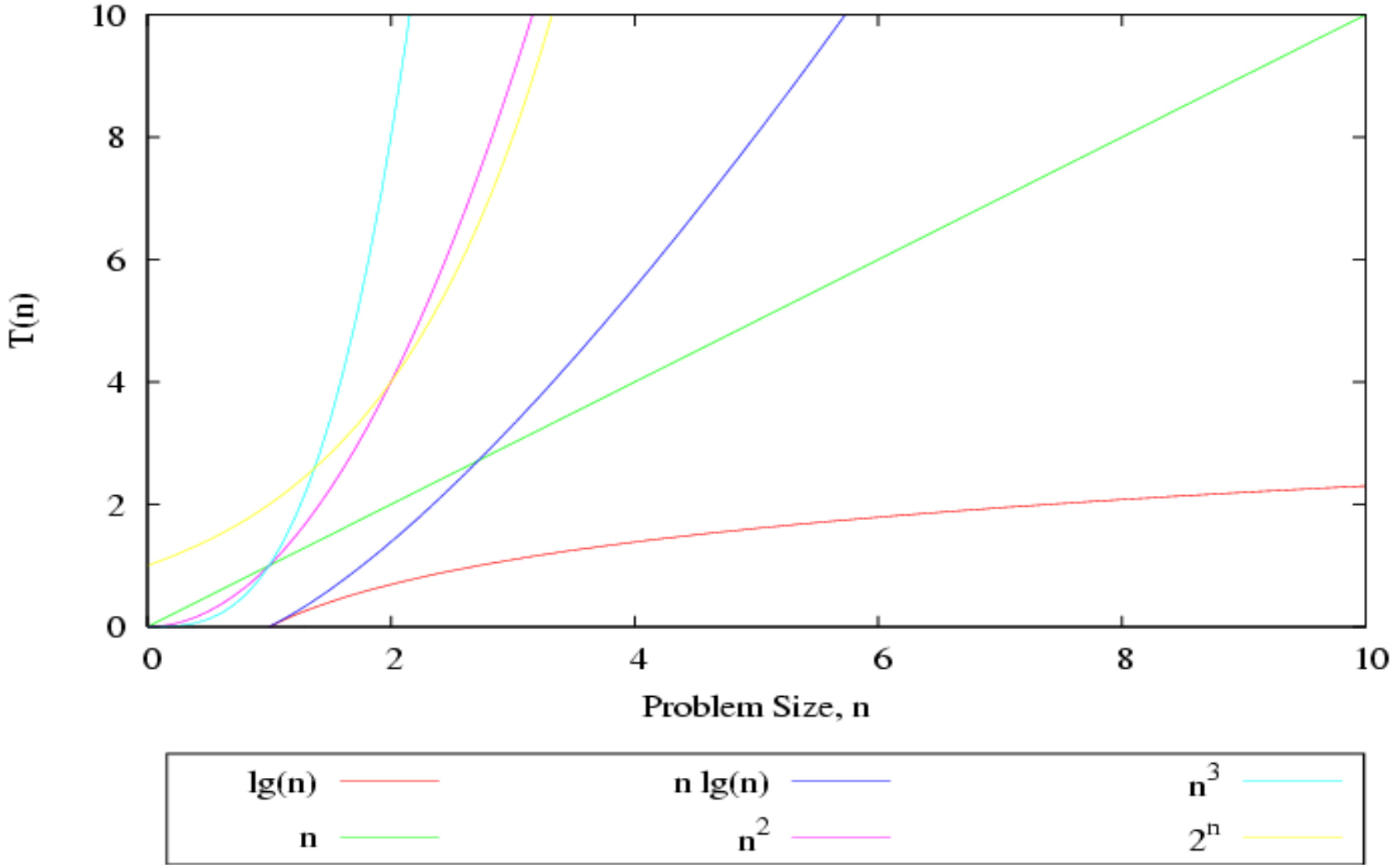
- Logarithmic

  $T(n) = \lg n$ [+ possible lower order terms]

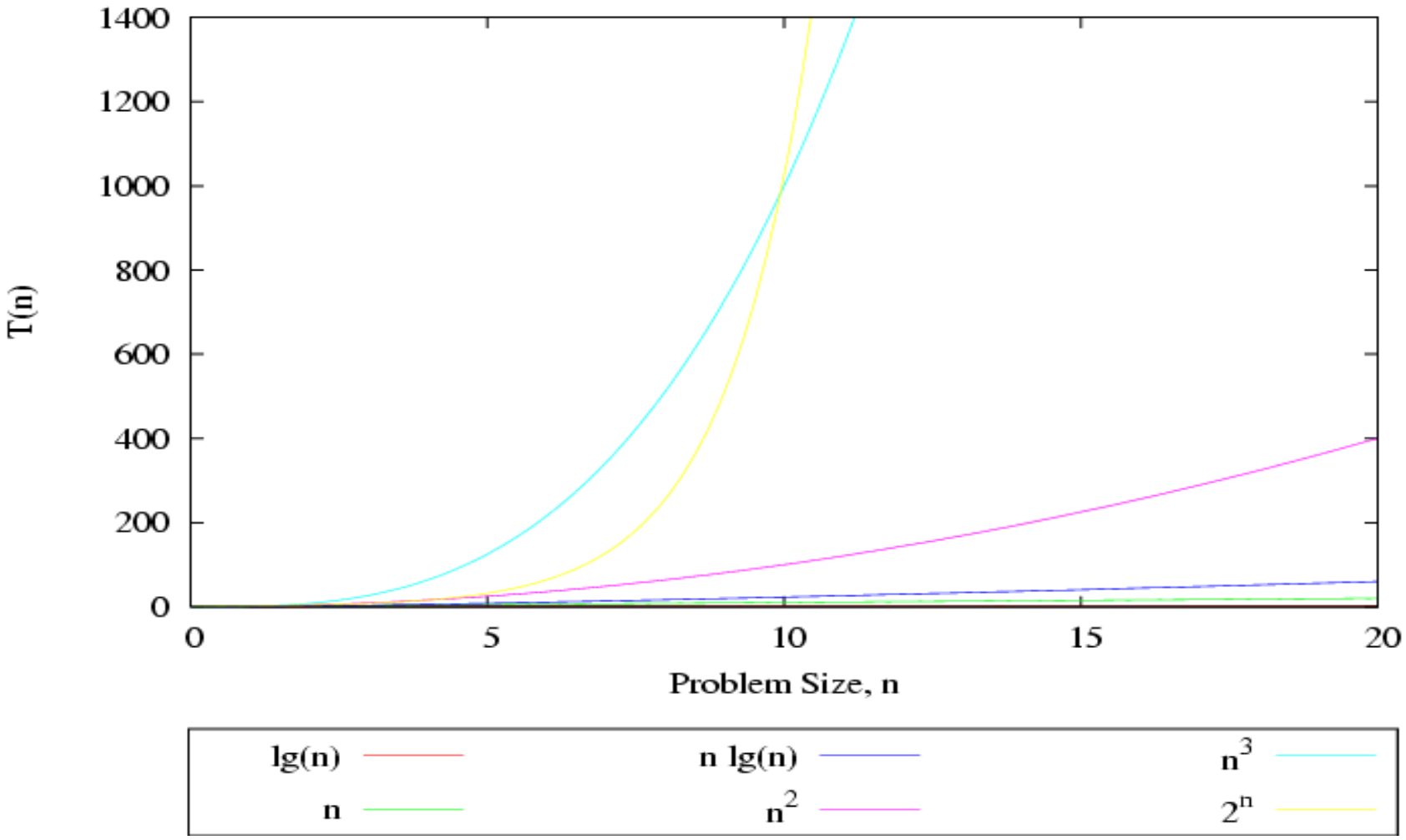
  ex: finding a particular array element (binary search)

  any algorithm that continually divides a problem in half

A Graph of Growth Functions



Expanded Scale



Asymptotic Analysis

- How does the time (or space) requirement grow as the problem size grows really, really large?
  - We are interested in “order of magnitude” growth rate.
  - We are usually not concerned with constant multipliers. For instance, if the running time of an algorithm is proportional to (let’s suppose) the square of the number of input items, i.e. T(n) is $c*n^2$, we won’t (usually) be concerned with the specific value of c.
  - Lower order terms don’t matter.

Analysis Cases

- What particular input (of given size) gives worst/best/average complexity?

Best Case: If there is a permutation of the input data that minimizes the "run time efficiency", then that minimum is the best case run time efficiency

Worst Case: If there is a permutation of the input data that maximizes the "run time efficiency", then that maximum is the best case run time efficiency

Average case is the "run time efficiency" over all possible inputs.

- Mileage example: how much gas does it take to go 20 miles?
  - Worst case: all uphill
  - Best case: all downhill, just coast
  - Average case: "average terrain

Cases Example

- Consider sequential search on an unsorted array of length n, what is time complexity?
- Best case:
- Worst case:
- Average case:

Formal Definition of Big-Oh

- T(n) = O(f(n)) (read "T(n) is in Big-Oh of f(n)")
  if and only if $T(n) \leq cf(n)$ for some constants c, n_0 and $n \geq n_0$

This means that eventually (when $n \geq n_0$), T(n) is always less than or equal to c times f(n).

The growth rate of T(n) is less than or equal to that of f(n)

Loosely speaking, f(n) is an "upper bound" for T (n)

NOTE: if T(n) =O(f(n)), there are infinitely many pairs of c's and n_0's that satisfy the relationship. We only need to find one such pair for the relationship to hold.

Big-Oh Example

- Suppose we have an algorithm that reads N integers from a file and does something with each integer.
- The algorithm takes some constant amount of time for initialization (say 500 time units) and some constant amount of time to process each data element (say 10 time units).
- For this algorithm, we can say T(N) = 500 + 10N.
- The following graph shows T(N) plotted against N, the problem size and 20N.
- Note that the function N will ***never*** be larger than the function T(N), no matter how large N gets. But there are constants c_0 and n_0 such that T(N) <= c_0N when N >= n_0, namely c_0 = 20 and n_0 = 50.
- Therefore, we can say that T(N) is in ***O***(N).

Simplifying Assumptions

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

2. If $f(n) = O(kg(n))$ for any $k > 0$, then $f(n) = O(g(n))$

3. If $f_1(n) = O(g_1(n))$ and $f_2(n) = O(g_2(n))$,
   then $f_1(n) + f_2(n) = O(\max(g_1(n), g_2(n)))$

4. If $f_1(n) = O(g_1(n))$ and $f_2(n) = O(g_2(n))$,
   then $f_1(n) * f_2(n) = O(g_1(n) * g_2(n))$

Example

- Code:

```
a = b;
++sum;
int y = Mystery( 42 );
```

- Complexity:

Example

- Code:

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

- Complexity:

Example

- Code:

```
sum1 = 0;
for (i = 1; i <= n; i++)
    for (j = 1; j <= n; j++)
        sum1++;
```

- Complexity:

Example

- Code:

```
sum2 = 0;
for (i = 1; i <= n; i++)
    for (j = 1; j <= i; j++)
        sum2++;
```

- Complexity:

Example

- Code:

```
sum = 0;
for (j = 1; j <= n; j++)
    for (i = 1; i <= j; i++)
        sum++;
for (k = 0; k < n; k++)
    a[ k ] = k;
```

- Complexity:

Example

- Code:

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

- Complexity:

Example

- Using Horner's rule to convert a string to an integer

```
static int convertString(String key)
{
  int intValue = 0;
  // Horner's rule
  for (int i = 0; i < key.length(); i++)
   intValue = 37 * intValue + key.charAt(i);
  return intValue
}
```

Example

- Square each element of an N x N matrix
- Printing the first and last row of an N x N matrix
- Finding the smallest element in a sorted array of N integers
- Printing all permutations of N distinct elements

Space Complexity

- Does it matter?
- What determines space complexity?
- How can you reduce it?
- What tradeoffs are involved?

Little-Oh and Big-Theta

- In addition to Big-O, there are other definitions used when discussing the relative growth of functions

Big-Theta – $T(n) = \Theta(f(n))$ if $c_1 * f(n) \leq T(n) \leq c_2 * f(n)$

This means that f(n) is both an upper- and lower-bound for T(n)

In particular, if $T(n) = \Theta(f(n))$, then $T(n) = O(f(n))$

Little-Oh – $T(n) = o(f(n))$ if for all constants **c** there exist n_0 such that $T(n) < c*f(n)$.

Note that this is more stringent than the definition of Big-O and therefore if $T(n) = o(f(n))$ then $T(n) = O(f(n))$

Relative Orders of Growth

An Exercise

- n (linear)
- $\log^k n$ for $0 < k < 1$
- constant
- n^{1+k} for $k > 0$ (polynomial)
- 2^n (exponential)
- $n \log n$
- $\log^k n$ for $k > 1$
- n^k for $0 < k < 1$
- $\log n$

Relative Orders of Growth Answers

constant

$\log^k n$ for $0 < k < 1$

log n

$\log^k n$ for $k > 1$

n^k for $k < 1$

n (linear)

n log n

n^{1+k} for $k > 0$ (polynomial)

2^n (exponential)

Big-Oh is not the whole story

- Suppose you have a choice of two approaches to writing a program. Both approaches have the same asymptotic performance (for example, both are O(n lg(n)). Why select one over the other, they're both the same, right? They may not be the same. There is this small matter of the constant of proportionality.
- Suppose algorithms A and B have the same asymptotic performance, $T_A(n) = T_B(n) = O(g(n))$. Now suppose that A does 10 operations for each data item, but algorithm B only does 3. It is reasonable to expect B to be faster than A even though both have the same asymptotic performance. The reason is that asymptotic analysis ignores constants of proportionality.

- The following slides show a specific example.

Algorithm A

- Let's say that algorithm A is

```
{
   initialization                  // takes 50 units
   read in n elements into array A;  // 3 units/element
   for (i = 0; i < n; i++)
   {
        do operation1 on A[i];     // takes 10 units
        do operation2 on A[i];     // takes 5 units
        do operation3 on A[i];     // takes 15 units
   }
 }
```

$T_A(n) = 50 + 3n + (10 + 5 + 15)n = 50 + 33n$

Proofs of Rules

These are included only for completeness and are optional reading.

Constants in Bounds ("constants don't matter")

- Theorem:

  If T(x) = *O*(cf(x)), then T(x) = *O*(f(x))

- Proof:
  - T(x) = *O*(cf(x)) implies that there are constants c_0 and n_0 such that $T(x) \leq c_0(cf(x))$ when $x \geq n_0$
  - Therefore, $T(x) \leq c_1(f(x))$ when $x \geq n_0$ where $c_1 = c_0c$
  - Therefore, T(x) = *O*(f(x))

Sum in Bounds (the "sum rule")

- Theorem:

  Let $T_1(n) = O(f(n))$ and $T_2(n) = O(g(n))$.

  Then $T_1(n) + T_2(n) = O(\max(f(n), g(n)))$.

- Proof:
  - From the definition of O,
    $T_1(n) \leq c_1 f(n)$ for $n \geq n_1$ and $T_2(n) \leq c_2 g(n)$ for $n \geq n_2$
  - Let $n_0 = \max(n_1, n_2)$.
  - Then, for $n \geq n_0$, $T_1(n) + T_2(n) \leq c_1 f(n) + c_2 g(n)$
  - Let $c_3 = \max(c_1, c_2)$.
  - Then, $T_1(n) + T_2(n) \leq c_3 f(n) + c_3 g(n) \leq 2c_3 \max(f(n), g(n)) \leq c \max(f(n), g(n))$

    $= O(\max(f(n), g(n)))$

Products in Bounds ("the product rule")

- Theorem:

  Let $T_1(n) = O(f(n))$ and $T_2(n) = O(g(n))$.

  Then $T_1(n) * T_2(n) = O(f(n) * g(n))$.

- Proof:
  - Since $T_1(n) = O(f(n))$, then $T_1(n) \leq c_1 f(n)$ when $n \geq n_1$
  - Since $T_2(n) = O(g(n))$, then $T_2(n) \leq c_2 g(n)$ when $n \geq n_2$
  - Hence $T_1(n) * T_2(n) \leq c_1 * c_2 * f(n) * g(n)$ when $n \geq n_0$ where $n_0 = \max(n_1, n_2)$
  - And $T_1(n) * T_2(n) \leq c * f(n) * g(n)$ when $n \geq n_0$ where $n_0 = \max(n_1, n_2)$ and $c = c_1 * c_2$
  - Therefore, by definition, $T_1(n) * T_2(n) = O(f(n) * g(n))$.

Polynomials in Bounds

- Theorem:

  If $T(n)$ is a polynomial of degree k, then $T(n) = O(n^k)$.

- Proof:
  - $T(n) = n^k + n^{k-1} + \ldots + c$ is a polynomial of degree k.
  - By the sum rule, the largest term dominates.
  - Therefore, $T(n) = O(n^k)$.