CMSC 341

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 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 ties

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)

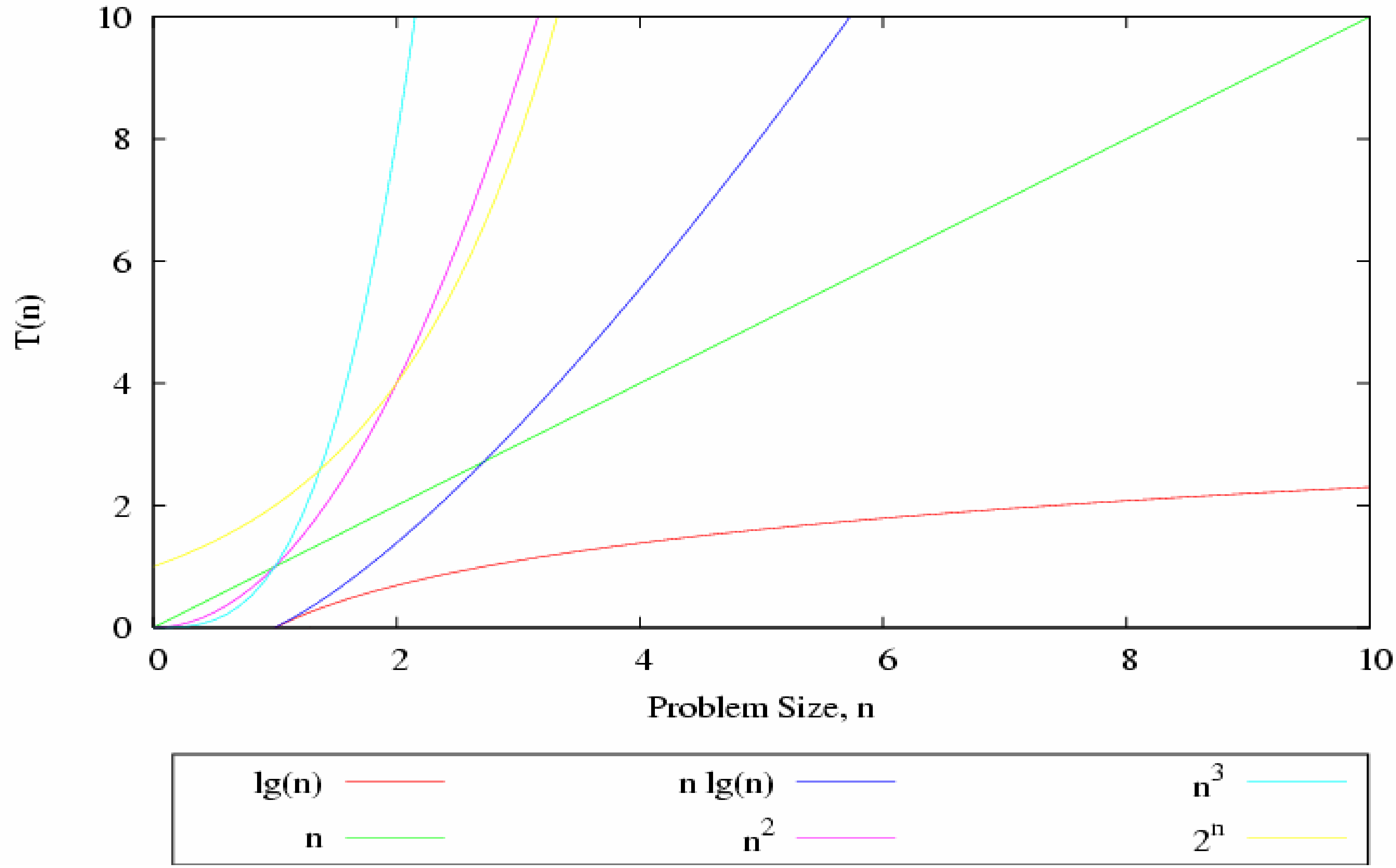
Logarithmic

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

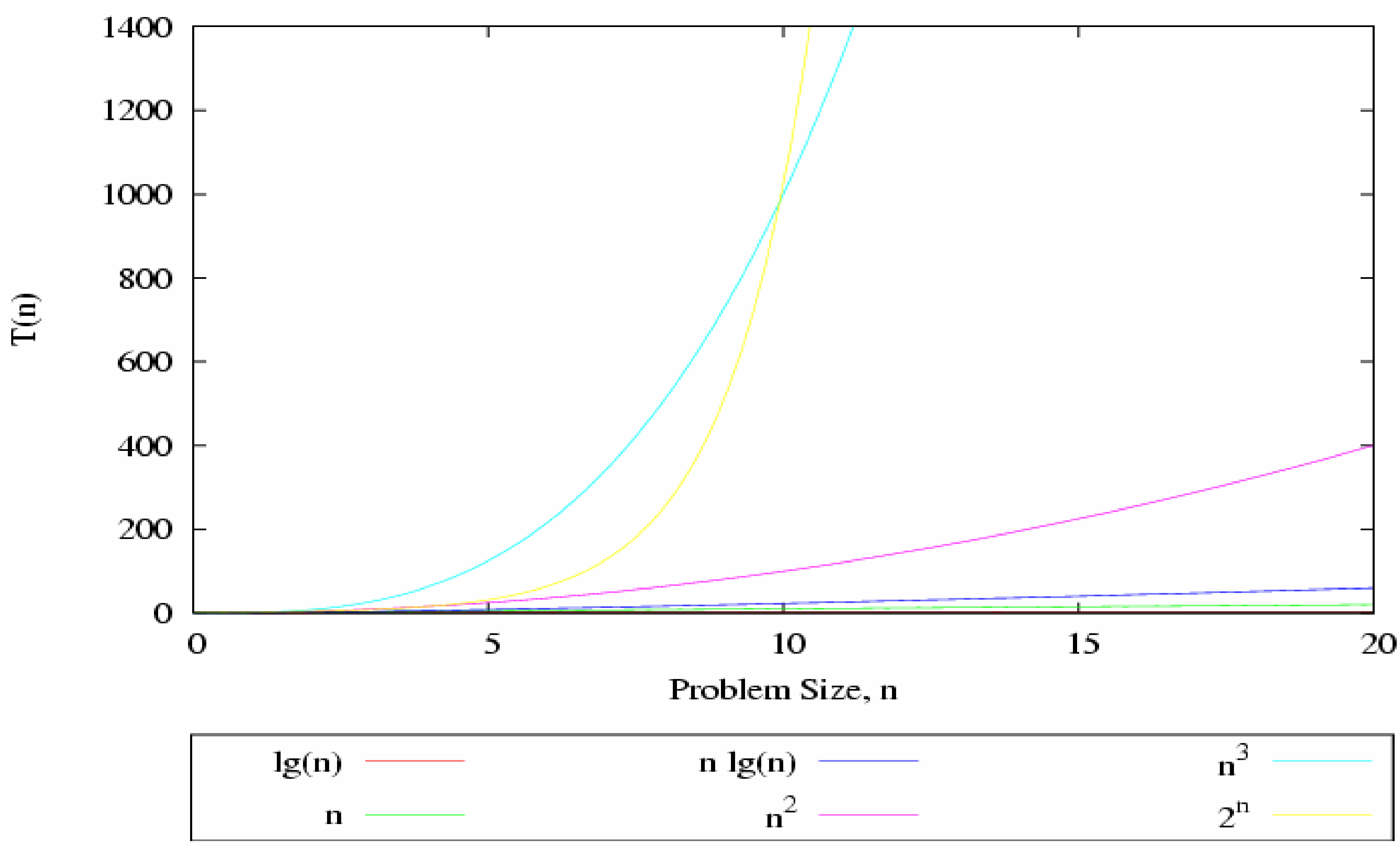
ex: finding a particular array element (binary search)

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:

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).

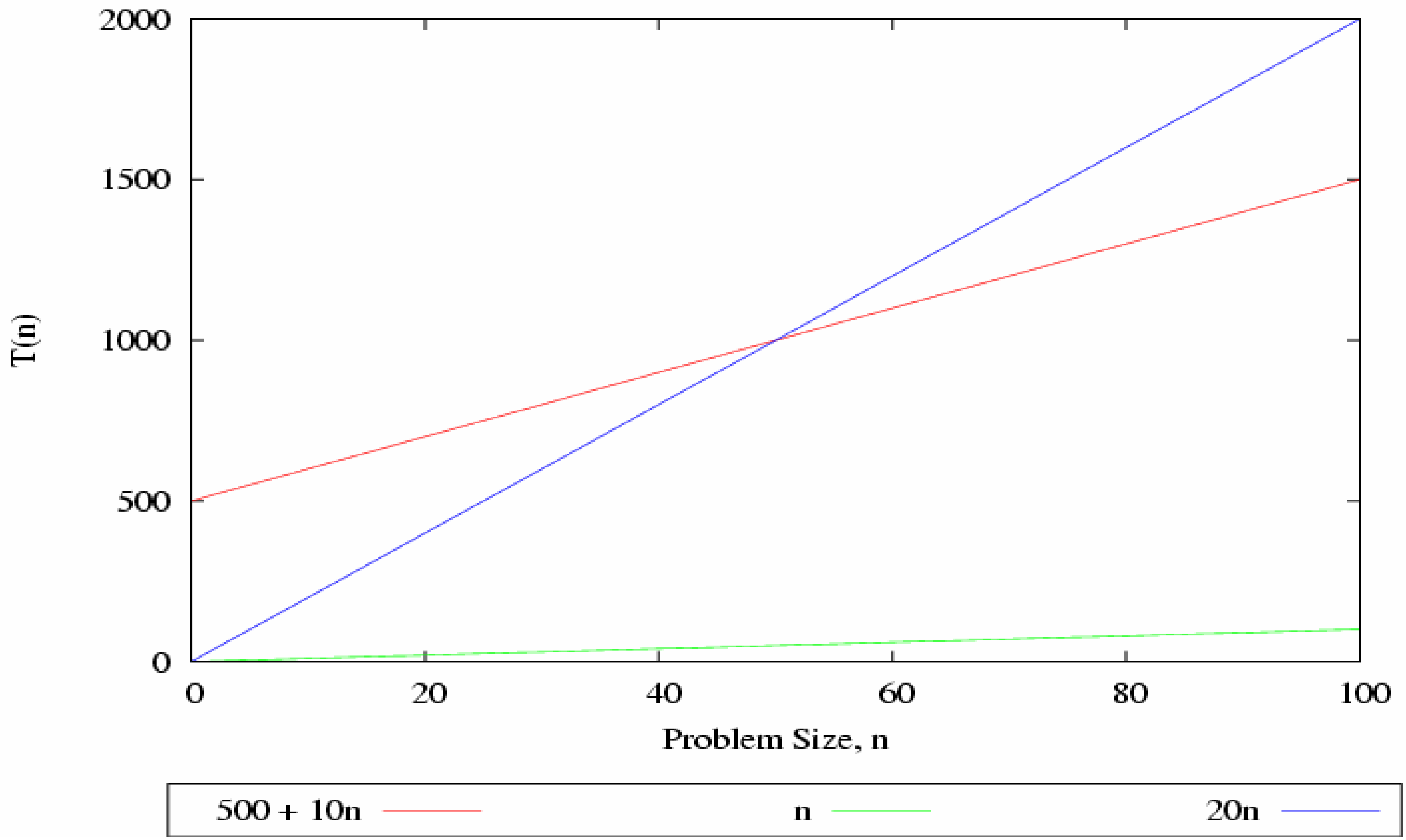
For this algorithm, we can say T(N) = 500 + 10N.

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).

T(N) vs. N vs. 20N


2000
1500
1000
500
0
T(n)
0
20
40
60
80
100
Problem Size, n
500 + 10n
n
20n

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:

```
sum1 = 0;
for (i = 1; i <= m; 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

```
int ConvertString(const string &key)
{
  int intValue = 0;

  // Horner's rule
  for (int i = 0; i < key.length(); i++)
     intValue = 37 * intValue + key[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?

Constants in Bounds

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

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, $T_1(n) \le c_1 f(n)$ for $n \ge n_1$ and $T_2(n) \le c_2 g(n)$ for $n \ge n_2$
- Let $n_0 = \max(n_1, n_2)$.
- Then, for $n \ge n_0$, $T_1(n) + T_2(n) \le c_1 f(n) + c_2 g(n)$
- Let $c_3 = \max(c_1, c_2)$.
- Then, $T_1(n) + T_2(n) \le c_3 f(n) + c_3 g(n)$

$$\begin{aligned} &\le 2c_3 \max(f(n), g(n)) \\ &\le c \max(f(n), g(n)) \\ &= O(\max(f(n), g(n))) \end{aligned}$$

Products in Bounds

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, $T(n) = O(n^k)$.

L'Hospital's Rule

Finding limit of ratio of functions as variable approaches ∞

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

Use this rule to prove other function growth relationships

$f(x) = O(g(x))$ if $\lim_{x\to\infty}\frac{f(x)}{g(x)}=0$

Polynomials of Logarithms in Bounds

Theorem:

$\lg^k n = O(n)$ for any positive constant k

Proof:

- Note that $\lg^k n$ means $(\lg n)^k$.
- Need to show $\lg^k n \le cn$ for $n \ge n_0$. Equivalently, can show $\lg n \le cn^{1/k}$
- Letting $a = 1/k$, we will show that $\lg n = O(n^a)$ for any positive constant a. Use L'Hospital's rule:

$$\lim_{n \to \infty} \frac{\lg n}{cn^a} = \lim_{n \to \infty} \frac{\frac{\lg e}{n}}{acn^{a-1}} = \lim_{n \to \infty} \frac{c_2}{n^a} = 0$$

Ex: $\lg^{1000000}(n) = O(n)$

Polynomials vs Exponentials in Bounds

Theorem:

$n^k = O(a^n)$ for $a > 1$

Proof:

– Use L'Hospital's rule

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

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

$$= \ldots$$

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

$$= 0$$

Ex: $n^{1000000} = O(1.00000001^n)$

Little-o 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 n0 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 $T(n) = O(f(n))$

Determining relative order of Growth

Given the definitions of Big-Theta and Little-o,

we can compare the relative growth of any two functions using limits. See text pages 43 – 45.

$$f(x) = o(g(x)) \text{ if } \lim_{x\to\infty}\frac{f(x)}{g(x)} = 0$$

By definition, if f(x) = o(g(x)), then f(x) = O(g(x)).

$$f(x) = \Theta(g(x)) \text{ if } \lim_{x\to\infty}\frac{f(x)}{g(x)} = c$$

for some constant $c > 0$.

By definition if f(x) = Θ(g(x)), then f(x) = O(g(x))

Determining relative order of Growth

Often times using limits is unnecessary as simple algebra will do.

For example, if $f(n) = n \log n$ and $g(n) = n^{1.5}$ then deciding which grows faster is the same as determining which of $f(n) = \log n$ and $g(n) = n^{0.5}$ grows faster (after dividing both functions by n), which is the same as determining which of $f(n) = \log^2 n$ and $g(n) = n$ grows faster (after squaring both functions). Since we know from previous theorems that n (linear functions) grows faster than any power of a log, we know that g(n) grows faster than 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

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 per 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
    }
}
```

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

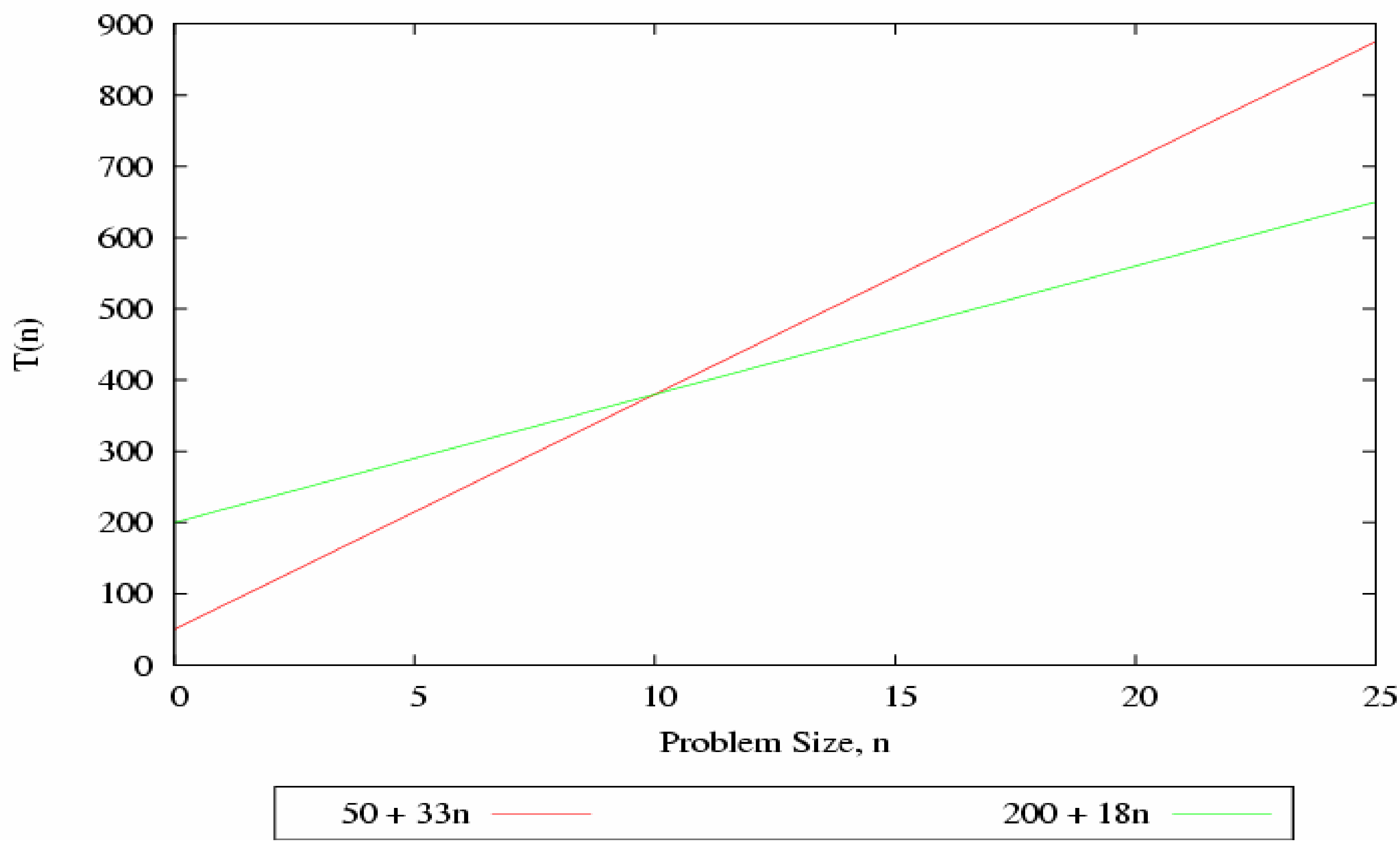
Algorithm B

Let's now say that algorithm B is

```
{
   initialization                          // takes 200 units
   read in n elements into array A;        // 3 units per element
   for (i = 0; i < n; i++)
   {
       do operation1 on A[i];              // takes 10 units
       do operation2 on A[i];              /takes 5 units
   }
}
```

$T_B(n) = 200 + 3n + (10 + 5)n = 200 + 18n$

$T_A(n)$ vs. $T_B(n)$


900
800
700
600
500
400
300
200
100
0
T(n)
0
5
10
15
20
25
Problem Size, n
50 + 33n
200 + 18n

A concrete example

The following table shows how long it would take to perform T(n) steps on a computer that does 1 billion steps/second. Note that a microsecond is a millionth of a second and a millisecond is a thousandth of a second.

N	T(n) = n	T(n) = nlgn	$T(n) = n^2$	$T(n) = n^3$	$Tn = 2^n$
5	0.005 microsec	0.01 microsec	0.03 microsec	0.13 microsec	0.03 microsec
10	0.01 microsec	0.03 microsec	0.1 microsec	1 microsec	1 microsec
20	0.02 microsec	0.09 microsec	0.4 microsec	8 microsec	1 millisec
50	0.05 microsec	0.28 microsec	2.5 microsec	125 microsec	13 days
100	0.1 microsec	0.66 microsec	10 microsec	1 millisec	4×10^{13} years

Notice that when n >= 50, the computation time for $T(n) = 2^n$ has started to become too large to be practical. This is most certainly true when n >= 100. Even if we were to increase the speed of the machine a million-fold, 2^n for n = 100 would be 40,000,000 years, a bit longer than you might want to wait for an answer.

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)

Amortized Analysis

Sometimes the worst-case running time of an operation does not accurately capture the worst-case running time of a *sequence* of operations.

What is the worst-case running time of the vector's push_back( ) method that places a new element at the end of the vector?

The idea of amortized analysis is to determine the average running time of the worst case.

Amortized Example – push_back( )

What is the running time for vector.push_back(X)?

In the worst case, there is no room in the vector for X. The vector then doubles its current size, copies the existing elements into the new vector, then places X in the next available slot. This operation is O(N) where N is the current number of elements in the vector.

But this doubling happens very infrequently. (how often?)

If there is room in the vector for X, then it is just placed in the next available slot in the vector and no doubling is required. This operation is O(1) – constant time

To discuss the running time of push_back( ) it makes more sense to look at a long sequence of push_back( ) operations.

A sequence of N push_back( ) operations can always be done in O(N), so we say the amortized running time of per push_back( )operation is O(N) / N = O(1) or constant time.

We are willing to perform a very slow operation (doubling the vector size) very infrequently in exchange for frequently having very fast operations.

Amortized Analysis Example

What is the average number of bits that are changed when a binary number is incremented by 1?

For example, suppose we increment 01100100.

We will change just 1 bit to get 0110010**1**.

Incrementing again produces 011001**10**, but this time 2 bits were changed.

Some increments will be “expensive”, others “cheap”.

How can we get an average? We do this by looking at a sequence of increments.

When we compute the total number of bits that change with n increments, divide that total by n, the result will be the average number of bits that change with an increment.

The table on the next slide shows the bits that change as we increment a binary number.(changed bits are shown in **red**).

Analysis

2^4	2^3	2^2	2^1	2^0	**Total bits changed**
0	0	0	0	**0**	**Start =0**
0	0	0	0	***1***	**1**
0	0	0	***1***	***0***	**3**
0	0	0	1	**1**	**4**
0	0	**1**	**0**	**0**	**7**
0	0	1	0	**1**	**8**
0	0	1	**1**	**0**	**10**
0	0	1	1	**1**	**11**
0	**1**	**0**	**0**	**0**	**15**

We see that bit position 2^0 changes every time we increment. Position 2^1 every other time (1/2 of the increments), and bit position 2^J changes each $1/2^J$ increments. We can total up the number of bits that change:

Analysis, continued

The total number of bits that are changed by incrementing n times is:

$$\sum_{j=0}^{\lfloor \lg(n) \rfloor} \lfloor n/2^j \rfloor$$

We can simplify the summation:

$$\sum_{j=0}^{\lfloor \lg(n) \rfloor} \lfloor n/2^j \rfloor \quad < \quad n * \sum_{j=0}^{\infty} (1/2)^j \quad = 2n$$

When we perform ***n*** increments, the total number of bit changes is <= 2n.

The *average* number of bits that will be flipped is 2n/n = 2. So the *amortized cost* of each increment is constant, or O(1).