# 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

# Mileage Example

Problem:

John drives his car, how much gas does he use?

# 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

trying on a shirt

calling a friend for fashion advice

Linear

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

ex: finding particular element in array (sequential search)

trying on all your shirts

calling all your n friends for fashion advice

# 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 shirts (n) with all your ties (n)

having conference calls with each pair of n friends

Polynomial

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

ex: looking for maximum substrings in array

trying on all combinations of k separates types of apparels (n of each)

having conferences calls with each k-tuple of n friends

# Growth Functions (cont)

Exponential

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

ex: constructing all possible orders of array elements

Logarithmic

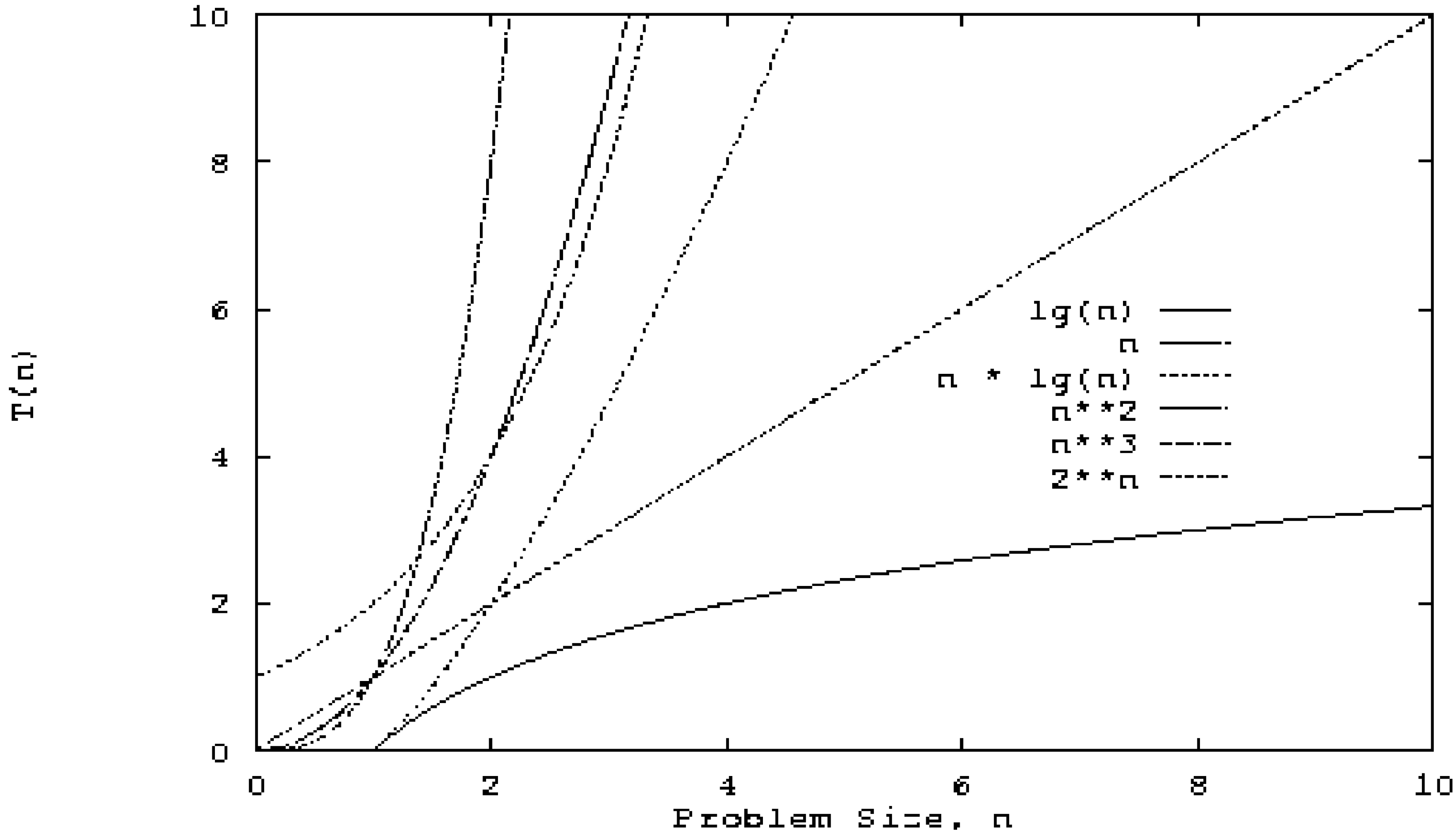
$T(n) = \log n$ [ + possible lower order terms]

ex: finding a particular array element (binary search)

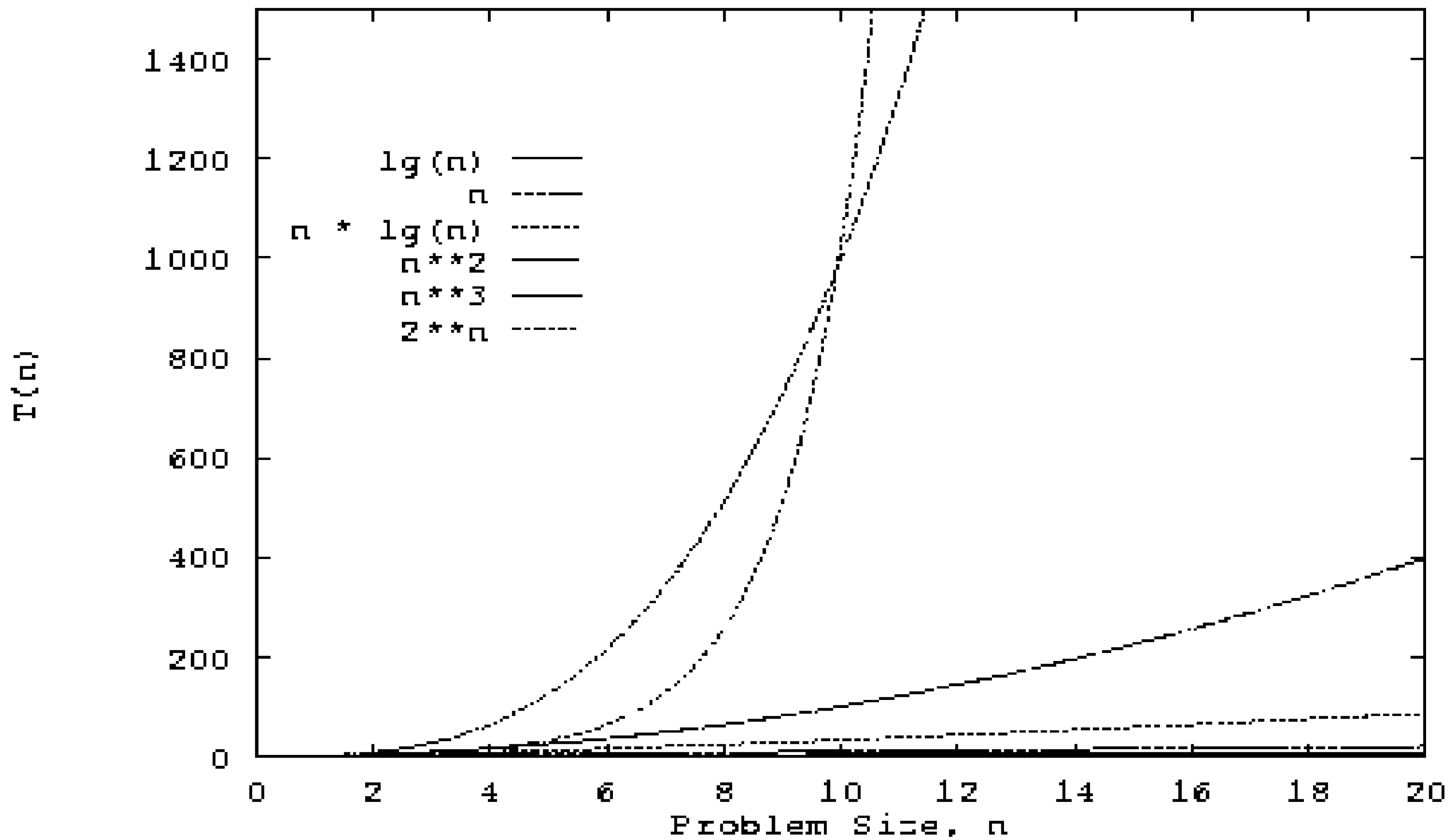
trying on all Garanimal combinations

getting fashion advice from n friends using phone tree

# A graph of Growth Functions



# Expanded Scale



# Asymptotic Analysis

What happens as problem size grows really, really large? (in the limit)

- constants don’t matter
- lower order terms don’t matter

# Analysis Cases

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

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 ).

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

# 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).

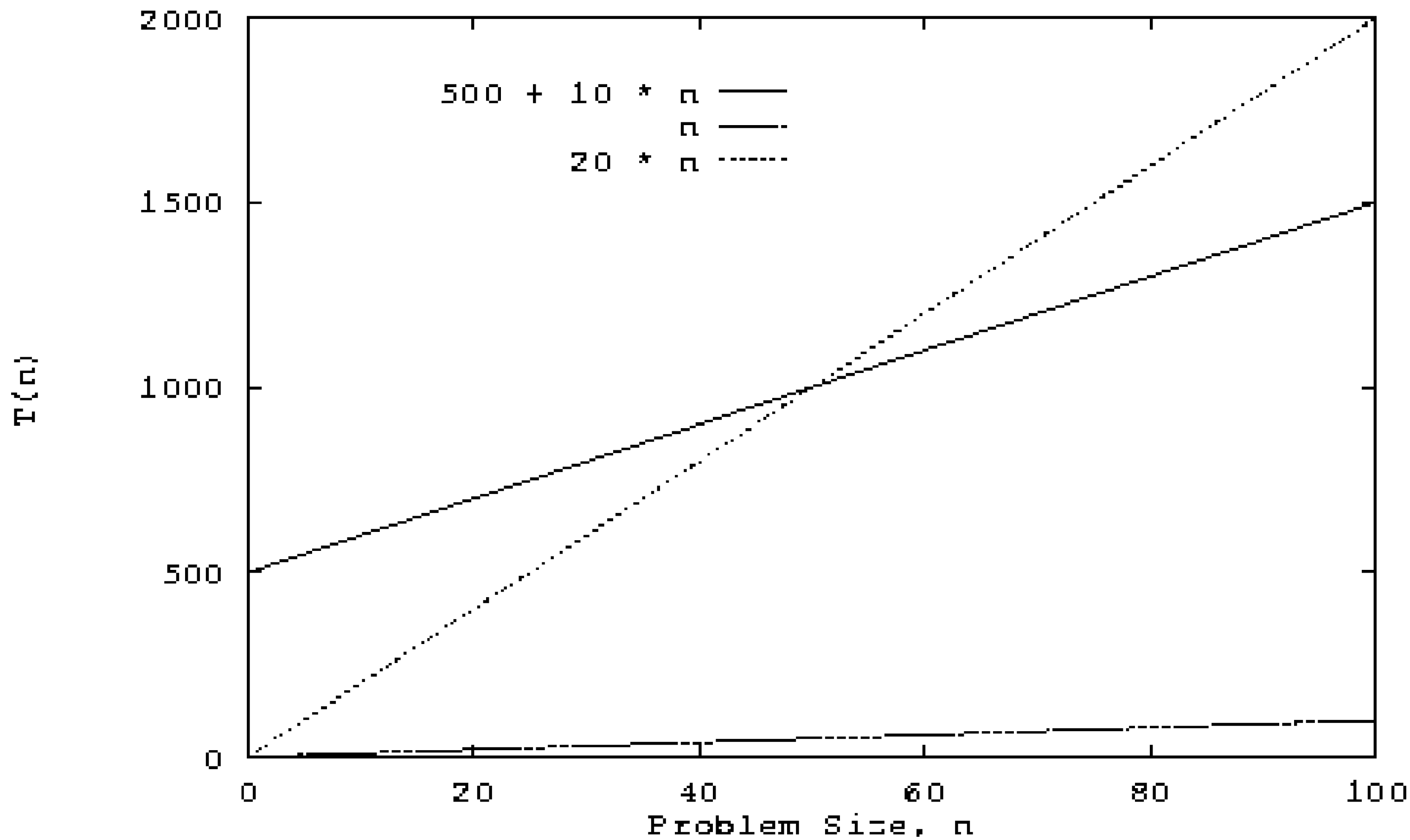
For this algorithm, we can say T( N ) = 500 + 10N.

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 ) <= $c_0$N 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



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

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

Code:

```
sum2 = 0;
for (k = 1; k <= n; k *= 2)
    for (j = 1; j <= k; j++)
        sum2++;
```

Complexity:

# 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

# Some Questions

1. Is upper bound the same as worst case?

2. What if there are multiple parameters?

   Ex: Rank order of p pixels in c colors

```
for (i = 0; i < c; i++)
      count[i] = 0;
for (i = 0; i < p; i++)
      count[value(i)]++;
sort(count)
```

# Space Complexity

Does it matter?

What determines space complexity?

How can you reduce it?

What tradeoffs are involved?

# Constants in Bounds

Theorem:

$O(cf(x)) = O(f(x))$

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, $T_1(n) + T_2(n) \le c_3 f(n) + c_3 g(n)$

$$\le 2c_3 \max(f(n), g(n))$$

$$\le c \max(f(n), g(n)) \quad \square$$

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

# 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 $n0 = \max(n_1, n_2)$
- 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 x, then $T(n) = O(n^x)$.

Proof:

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

# 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 to determine O ordering of two functions

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

# Polynomials of Logarithms in Bounds

Theorem:

$\lg^x n = O(n)$ 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

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

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

$$= \dots$$

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

$$= 0$$

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

# Relative Orders of Growth

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
    }
 }
```

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

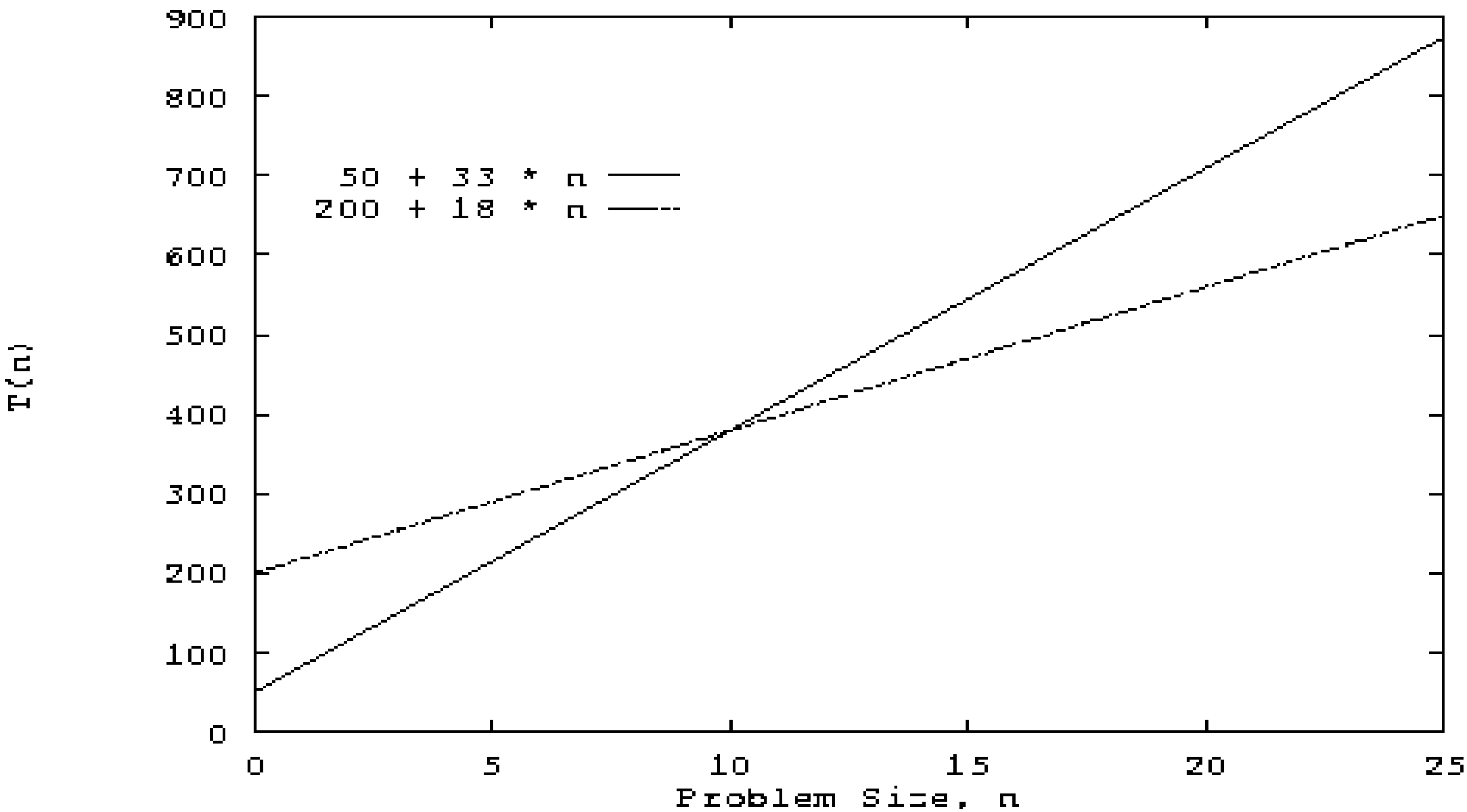
# Algorithm B

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
0
5
10
15
20
25
T(n)
Problem Size, n
50 + 33 * n
200 + 18 * n

# 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 x $10^{13}$ 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

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)