Computational Complexity

Algorithm analysis

- **Correctness**
  - Testing
  - Proofs of correctness
- **Efficiency**
  - How to define?
  - **Asymptotic complexity** - how running times scales as function of size of input

Proving Programs Correct

- Usually in the form of an inductive proof
- Example: summing an array

```
int sum(int v[], int n)
{
   if (n==0) return 0;
   else return v[n-1]+sum(v,n-1);
}
```

Theorem: sum(v,n) correctly returns sum of 1st n elements of array v for any n.
Basis Step: Program is correct for n=0; returns 0.
Inductive Hypothesis (n=k): Assume sum(v,k) returns sum of first k elements of v.
Inductive Step (n=k+1): sum(v,k+1) returns v[k]+sum(v,k), which is the same of the first k+1 elements of v.

Defining efficiency

- Asymptotic Complexity - how running time scales as function of size of input

- Why is this a reasonable definition?

  - Many kinds of small problems can be solved in practice by almost any approach

  - E.g., exhaustive enumeration of possible solutions

- Want to focus efficiency concerns on larger problems

- Definition is independent of any possible advances in computer technology

Defining efficiency

- Asymptotic Complexity - how running time scales as function of size of input
- What is “size”?
    - Often: length (in characters) of input
    - Sometimes: value of input (if input is a number)
- Which inputs?
    - Worst case
    - Best case

Average case analysis

- More realistic analysis, first attempt:
    - Assume inputs are randomly distributed according to some “realistic” distribution D
    - Compute expected running time

$$E(T,n) = \sum_{x \in \mathrm{Inputs}(n)} \mathrm{Prob}_{\Delta}(x)\,\mathrm{RunTime}(x)$$

- Drawbacks
    - Often hard to define realistic random distributions
    - Usually hard to perform math

Amortized analysis

- Instead of a single input, consider a sequence of inputs
    - Choose worst possible sequence
- Determine average running time on this sequence
- Advantages
    - Often less pessimistic than simple worst-case analysis
    - Guaranteed results - no assumed distribution
    - Usually mathematically easier than average case analysis

Comparing runtimes

- Program A is asymptotically less efficient than program B iff
  - the runtime of A dominates the runtime of B, as the size of the input goes to infinity

$$\left(\frac{\mathrm{RunTime}(A,n)}{\mathrm{RunTime}(B,n)}\right) \to \infty \quad \text{as} \quad n \to \infty$$

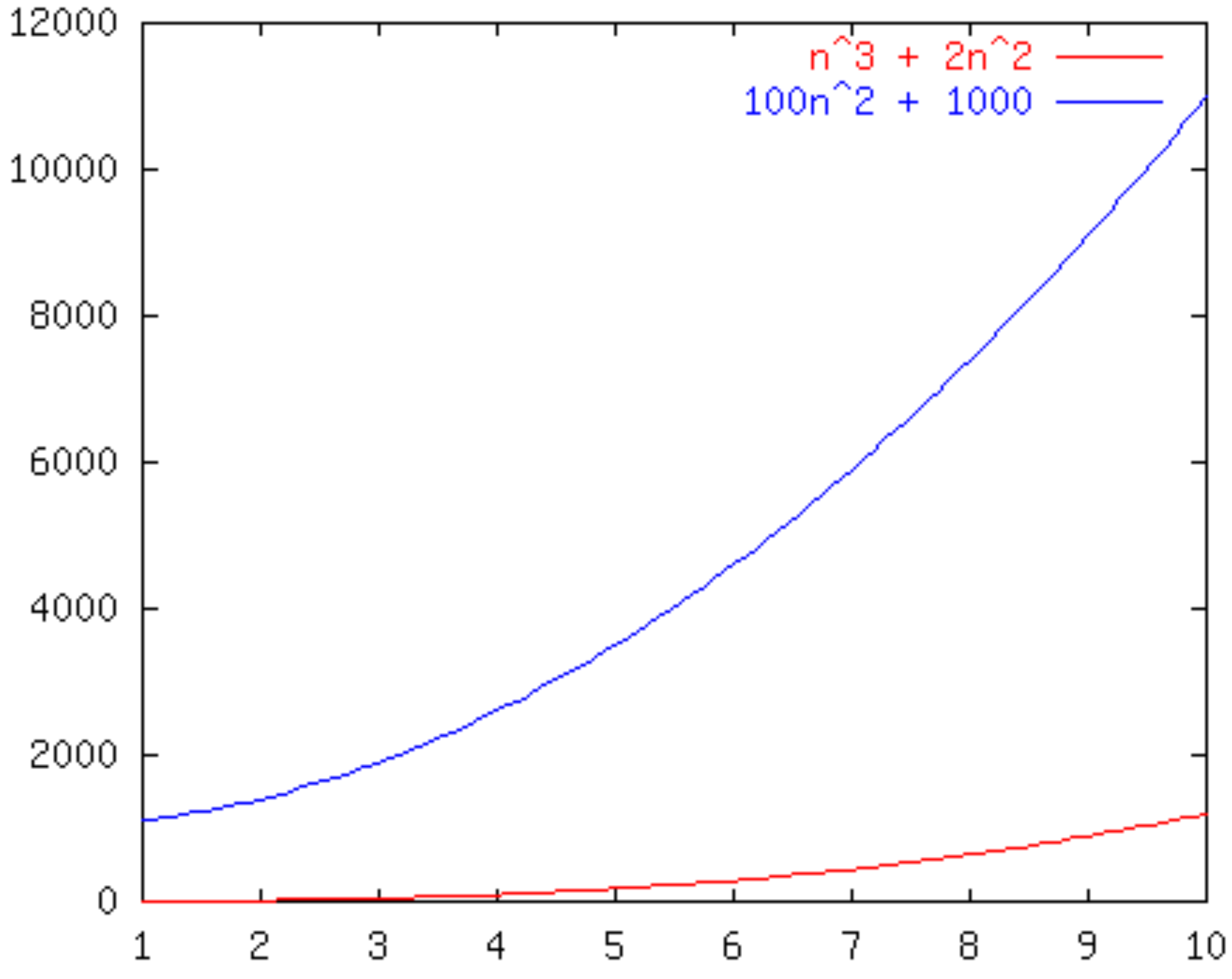
- Note: RunTime can be “worst case”, “best case”, “average case”, “amortized case”

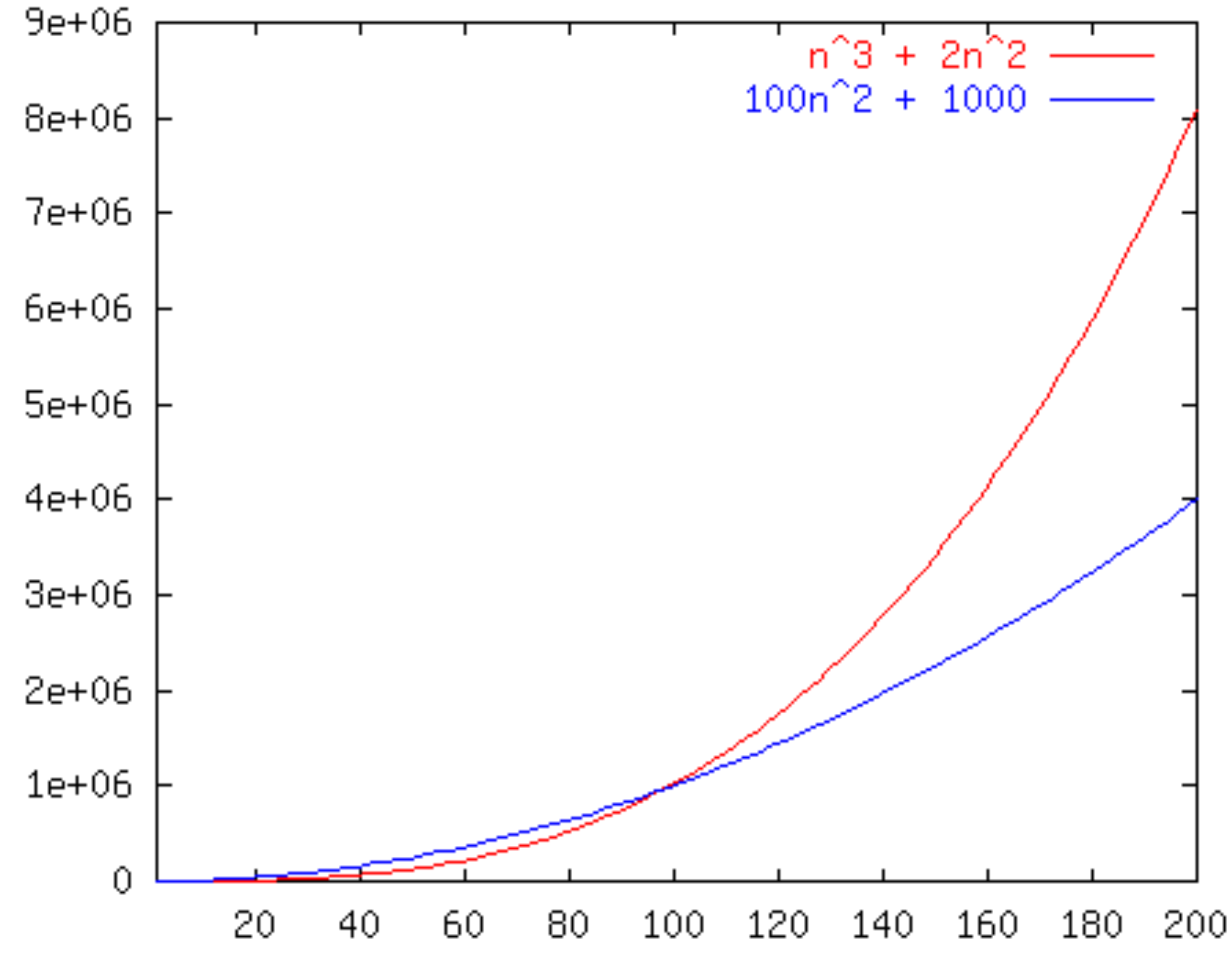
Which Function Dominates?

$n^3 + 2n^2$	$100n^2 + 1000$
$n^{0.1}$	$\log n$
$n + 100n^{0.1}$	$2n + 10 \log n$
$5n^5$	$n!$
$n^{-15}2^n/100$	$1000n^{15}$
$8^{2\log n}$	$3n^7 + 7n$

Race I

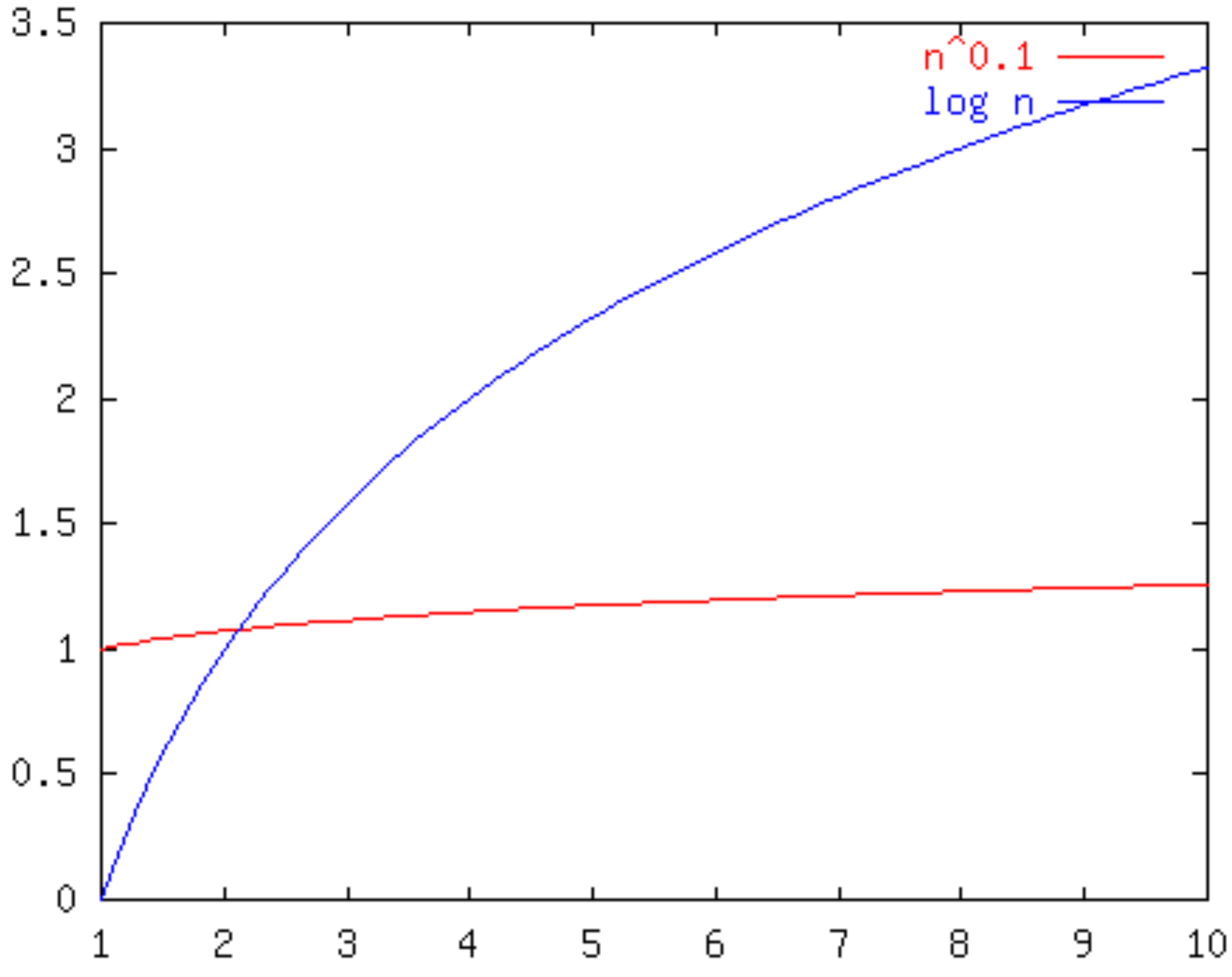
$\mathtt{n^3 + 2n^2}$ vs. $\mathtt{100n^2 + 1000}$

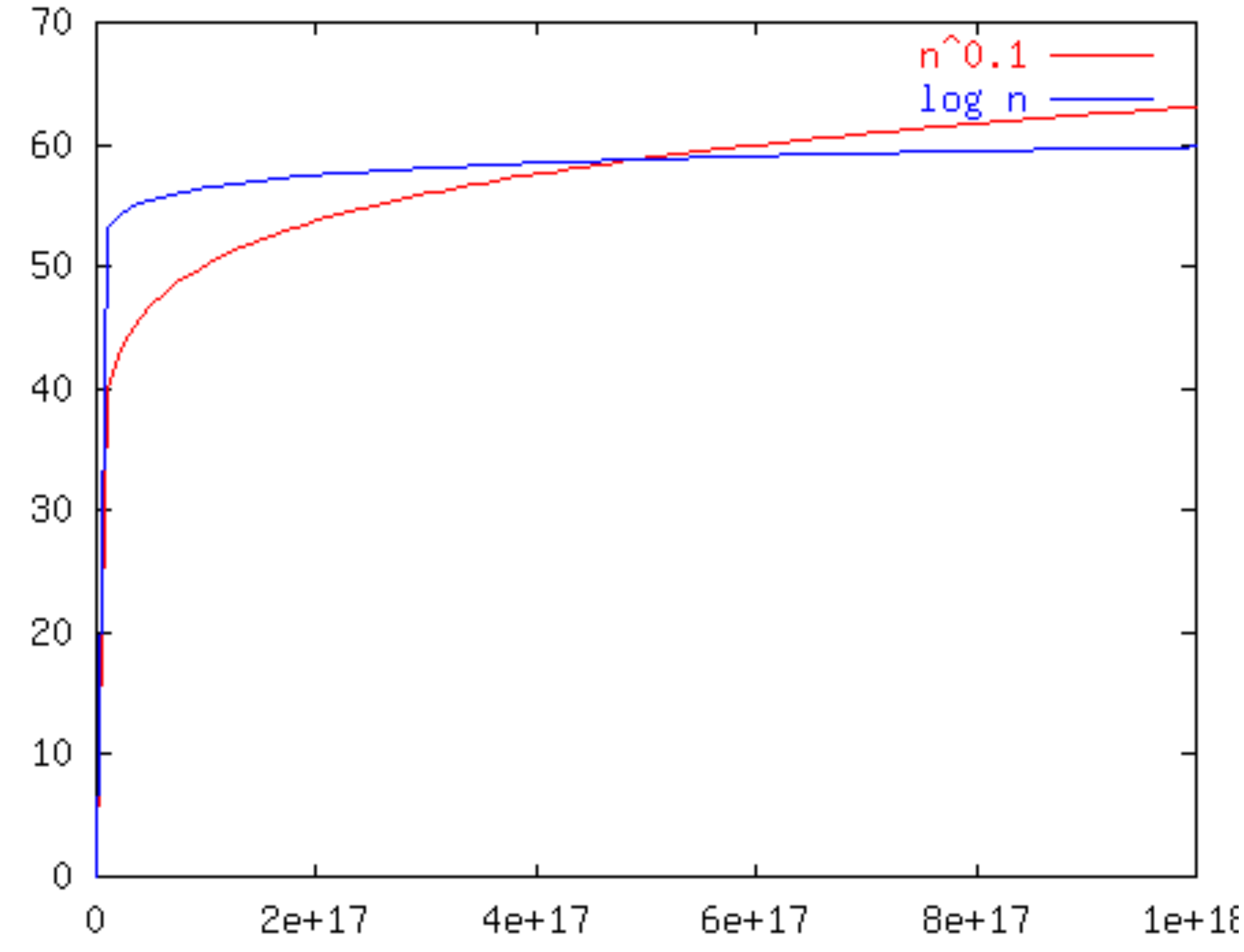




Race II

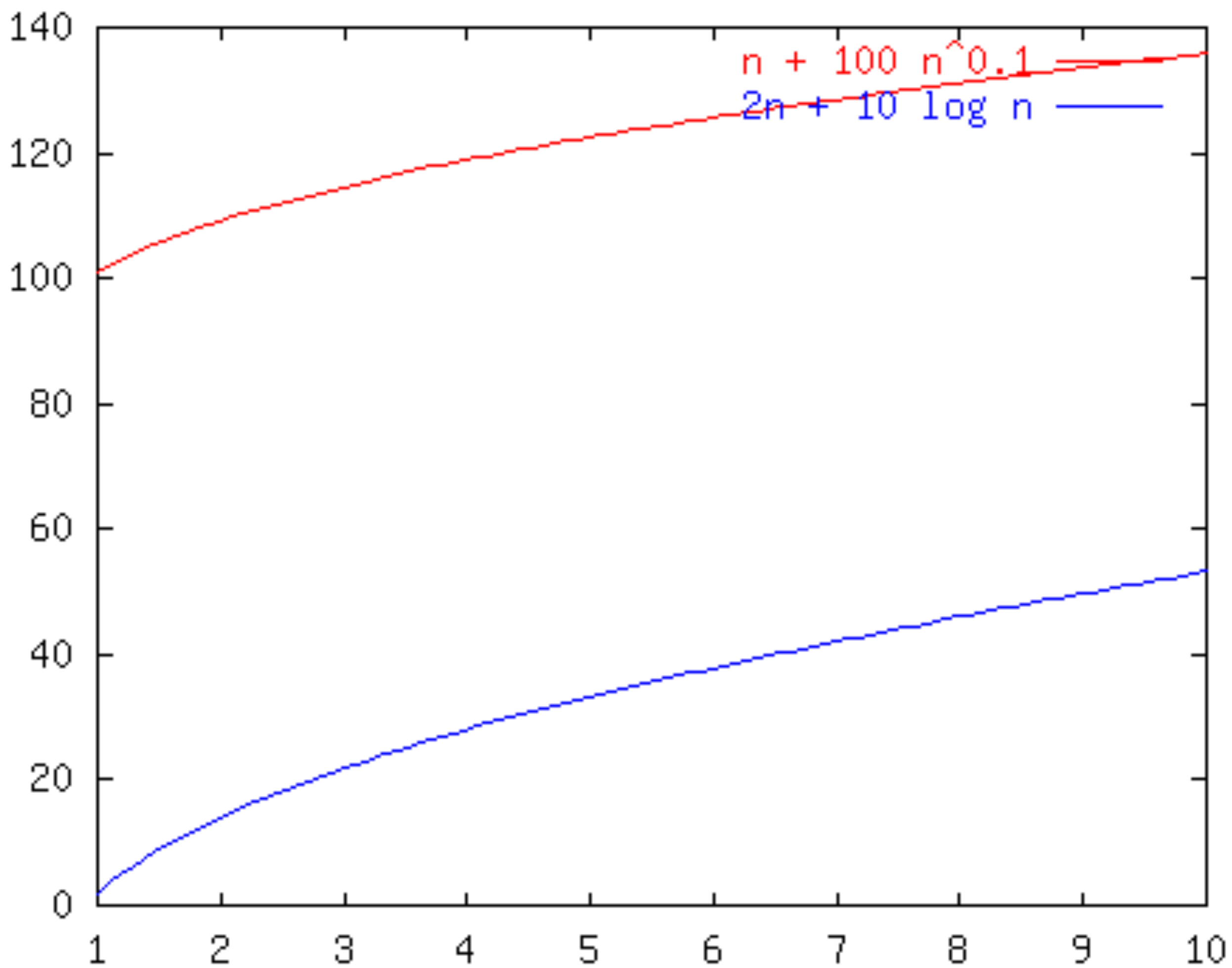
$n^{0.1}$ vs. $\log n$

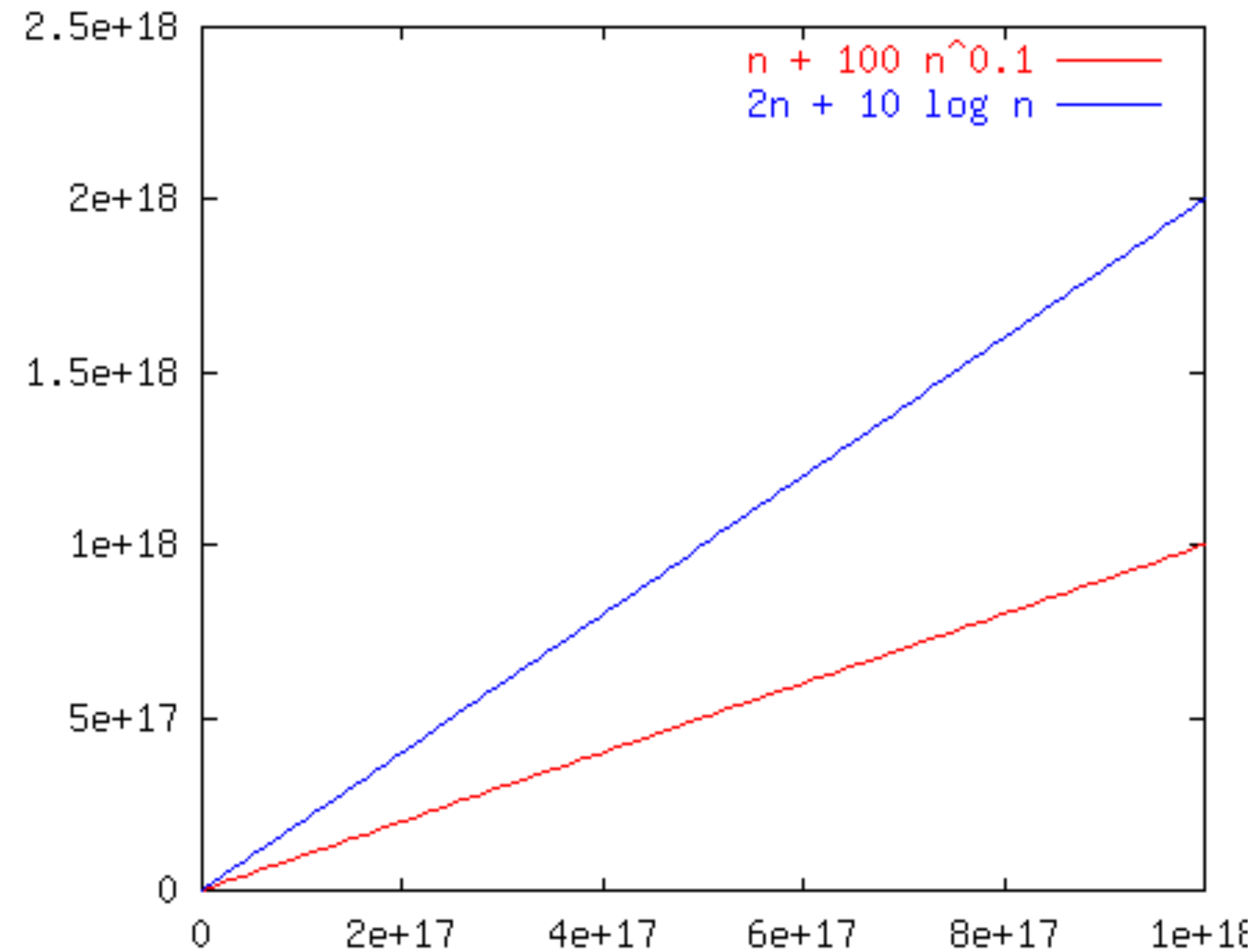




Race III

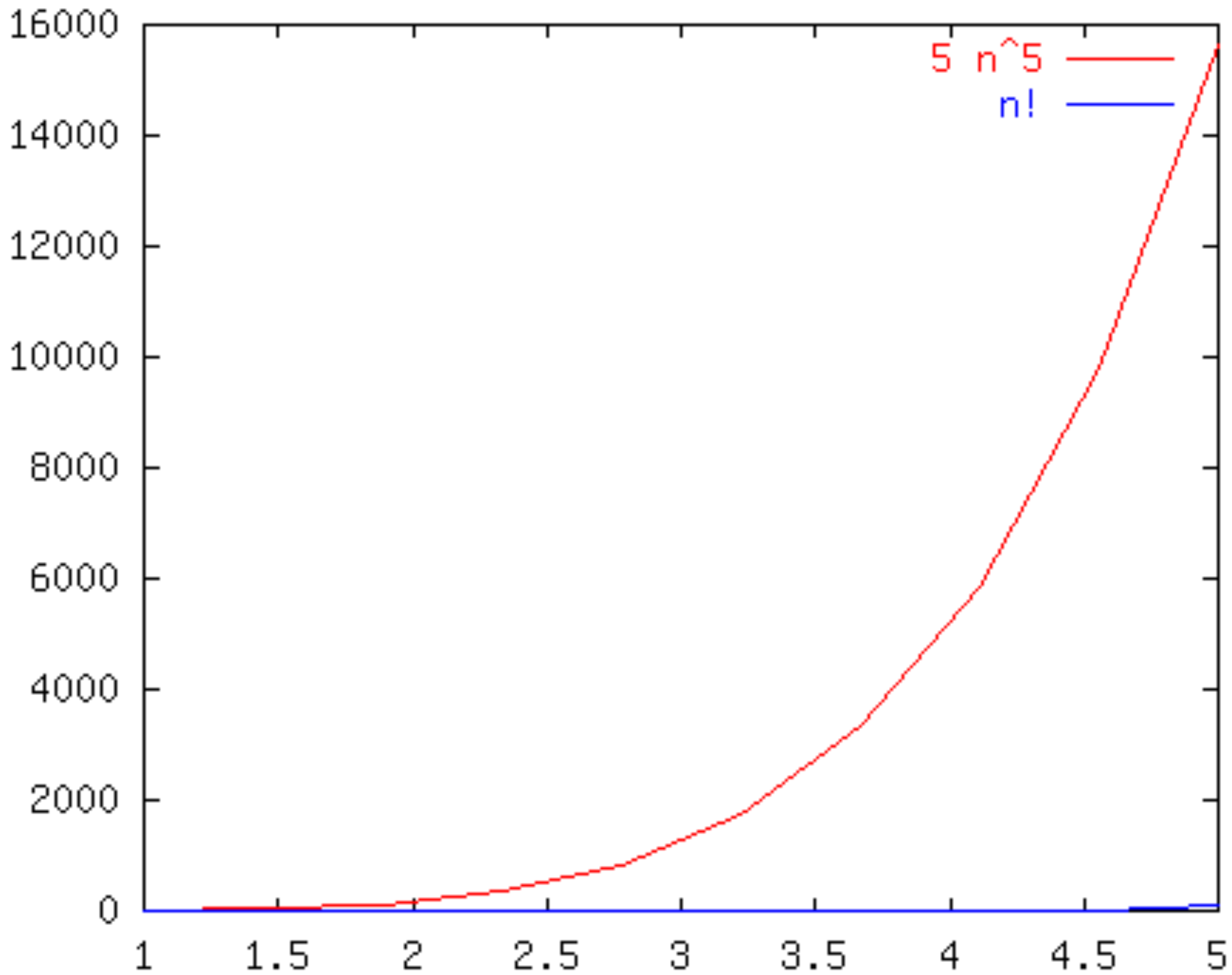
$\mathbf{n + 100n^{0.1}}$ vs. $\mathbf{2n + 10 \log n}$

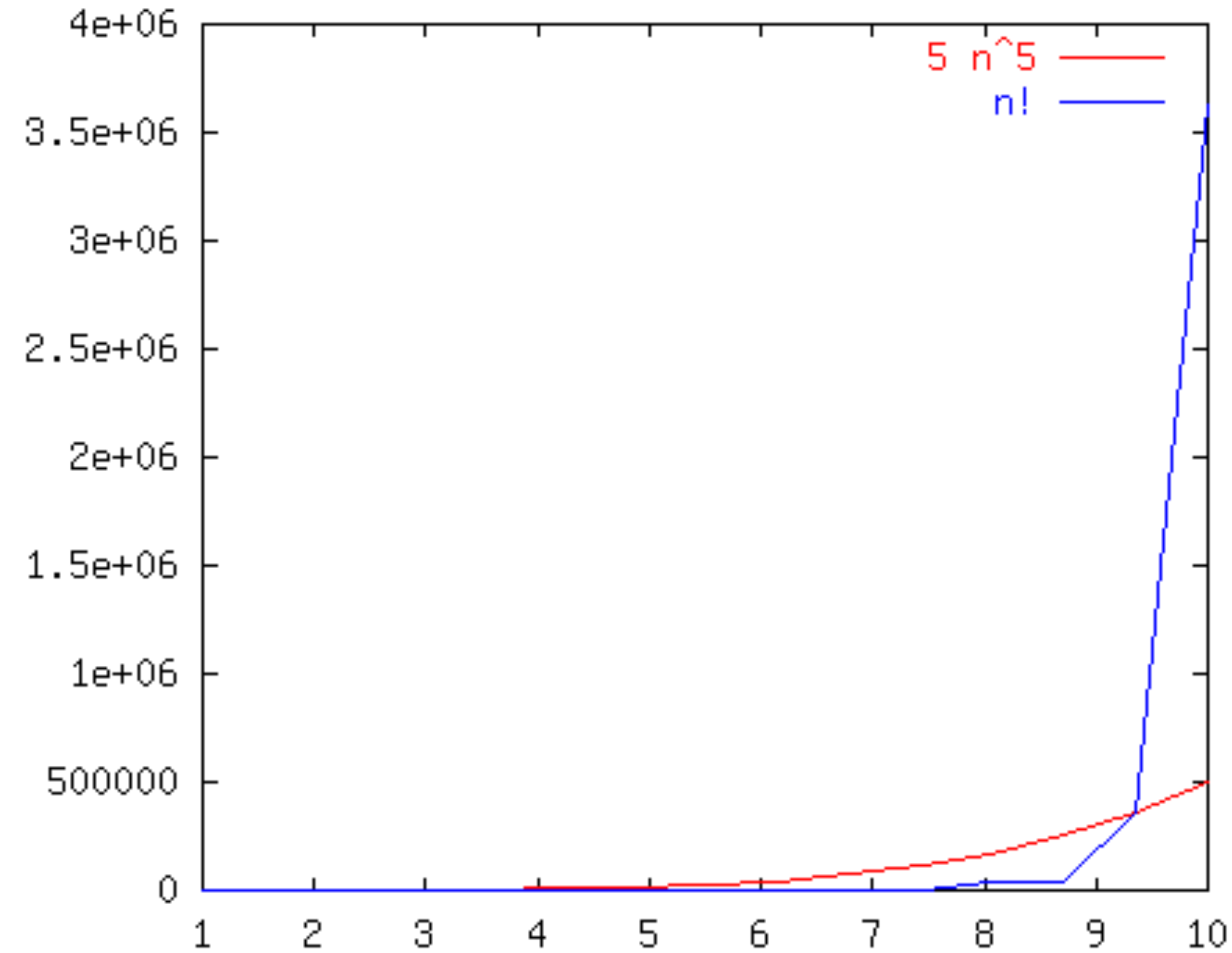




Race IV

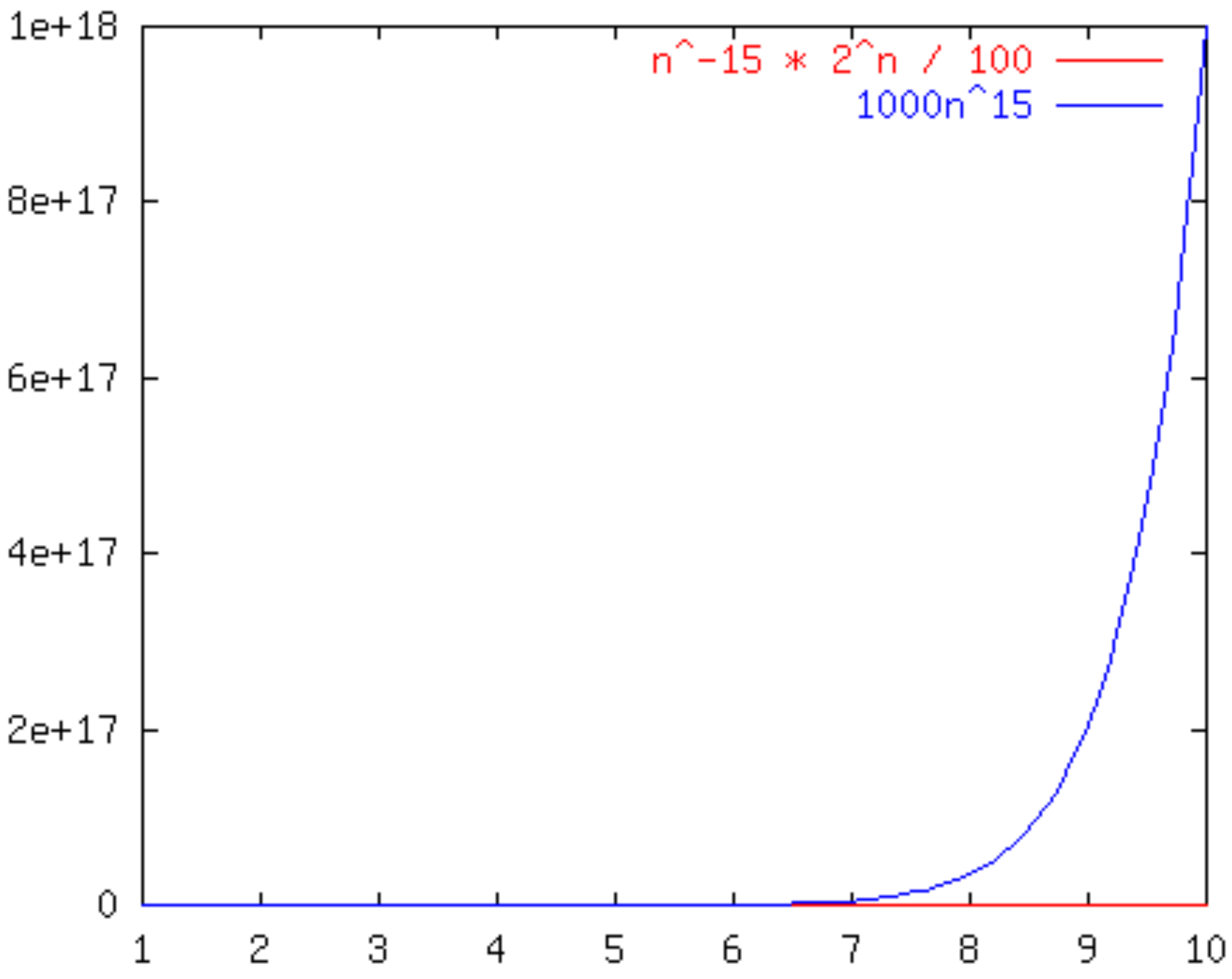
$\mathbf{5n^5}$ vs. $\mathbf{n!}$

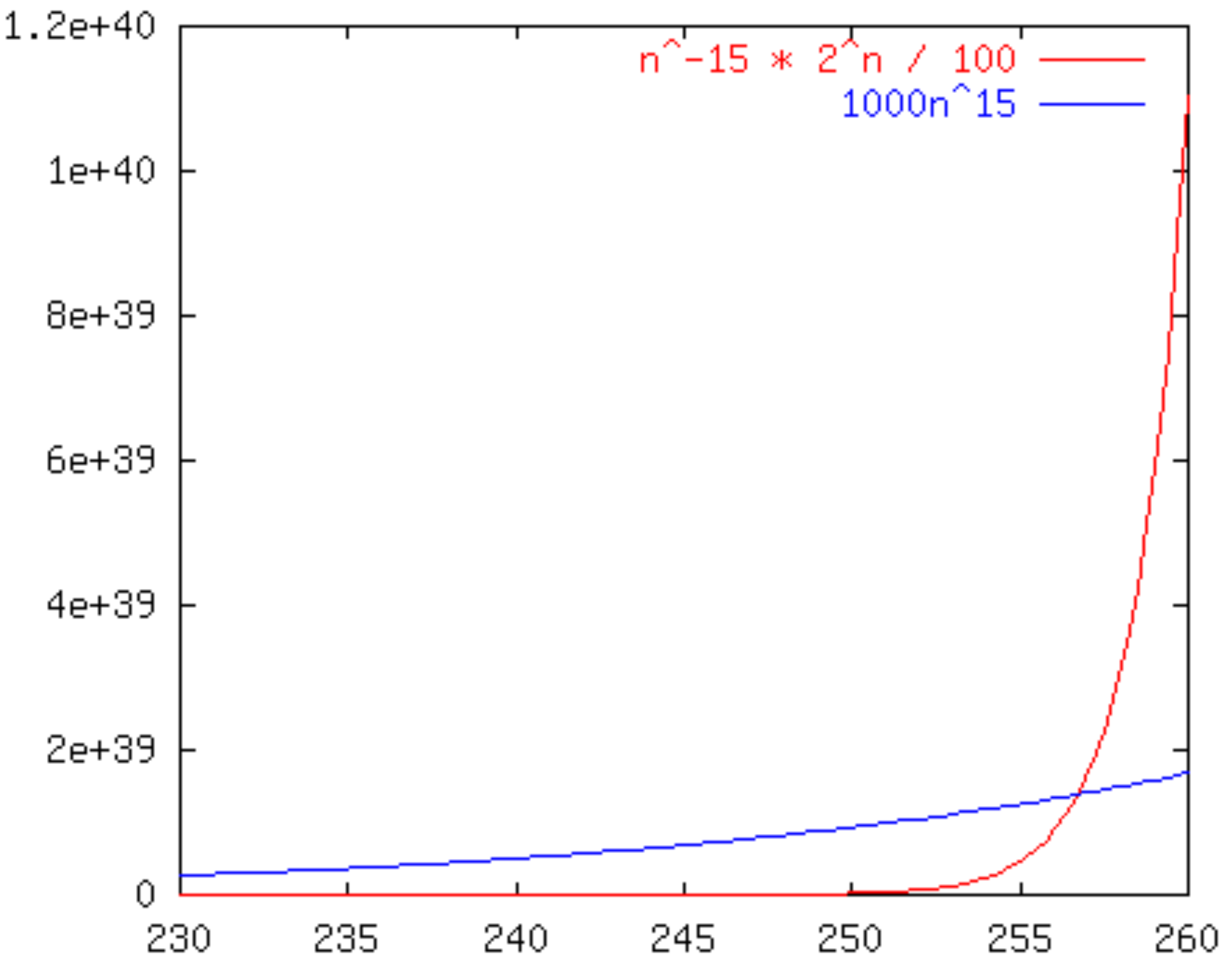




Race V

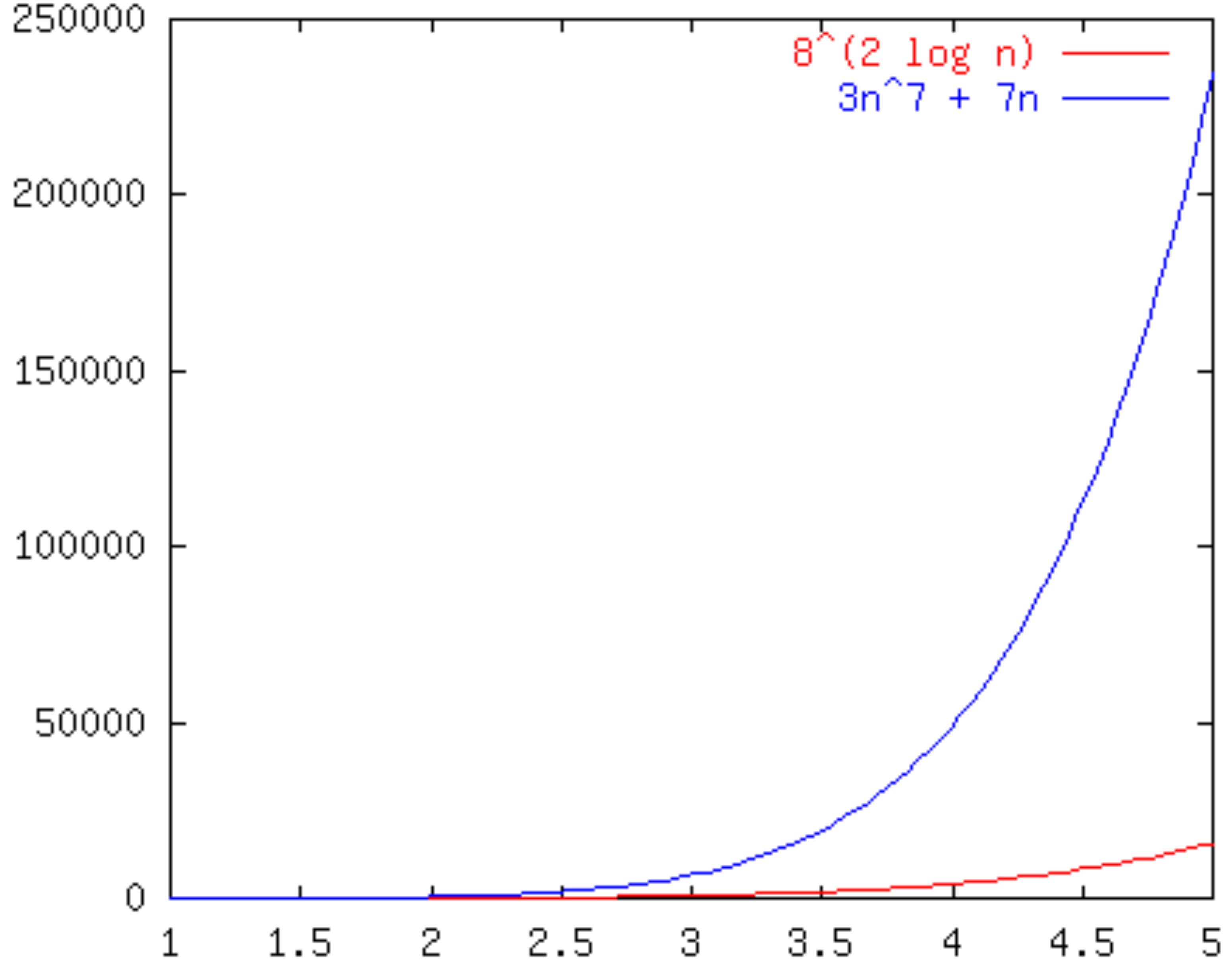
$n^{-15}2^n/100$ vs. $1000n^{15}$





Race VI

$8^{2\log(n)}$ vs. $3n^7 + 7n$



Order of Magnitude Notation (big O)

- Asymptotic Complexity - how running time scales as function of size of input
  - We usually only care about order of magnitude of scaling

- Why?
  - As we saw, some functions overwhelm other functions
    - So if running time is a sum of terms, can drop dominated terms
  - “True” constant factors depend on details of compiler and hardware
    - Might as well make constant factor 1

$16n^3 \log_8(10n^2) + 100n^2 = O(n^3 \log(n))$

- Eliminate low order terms
- Eliminate constant coefficients

$$16n^3 \log_8(10n^2) + 100n^2$$
$$\Rightarrow 16n^3 \log_8(10n^2)$$
$$\Rightarrow n^3 \log_8(10n^2)$$
$$\Rightarrow n^3 \left[\log_8(10) + \log_8(n^2)\right]$$
$$\Rightarrow n^3 \log_8(10) + n^3 \log_8(n^2)$$
$$\Rightarrow n^3 \log_8(n^2)$$
$$\Rightarrow n^3 2 \log_8(n)$$
$$\Rightarrow n^3 \log_8(n)$$
$$\Rightarrow n^3 \log_8(2) \log(n)$$
$$\Rightarrow n^3 \log(n)$$

Common Names

Slowest Growth

constant:	$O(1)$	
logarithmic:	$O(\log n)$	
linear:	$O(n)$	
log-linear:	$O(n \log n)$	
quadratic:	$O(n^2)$	
exponential:	$O(c^n)$	(c is a constant > 1)

Fastest Growth

superlinear:	$O(n^c)$	(c is a constant > 1)
polynomial:	$O(n^c)$	(c is a constant > 0)

How to determine the complexity of an algorithm ?

Formal Asymptotic Analysis

- In order to *prove* complexity results, we must make the notion of "order of magnitude" more precise

- Asymptotic bounds on runtime
  - Upper bound
  - Lower bound

Definition of Order Notation

- Upper bound: $T(n) = O(f(n))$ Big-O

  Exist constants c and n' such that

  $T(n) \leq c\, f(n)$ for all $n \geq n'$

- Lower bound: $T(n) = \Omega(g(n))$ Omega

  Exist constants c and n' such that

  $T(n) \geq c\, g(n)$ for all $n \geq n'$

- Tight bound: $T(n) = \theta(f(n))$ Theta

  When both hold:

  $T(n) = O(f(n))$

  $T(n) = \Omega(f(n))$

Example: Upper Bound

Claim: $n^2 + 100n = O(n^2)$

Proof: Must find c, n' such that for all $n > n'$,

$$n^2 + 100n \le cn^2$$

Let's try setting $c = 2$. Then

$$n^2 + 100n \le 2n^2$$

$$100n \le n^2$$

$$100 \le n$$

So we can set $n' = 100$ and reverse the steps above.

Using a Different Pair of Constants

Claim: $n^2 + 100n = O(n^2)$

Proof: Must find c, n' such that for all $n > n'$,

$$n^2 + 100n \le cn^2$$

Let's try setting $c = 101$. Then

$$n^2 + 100n \le 100n^2$$

$$n + 100 \le 101n \quad \text{(divide both sides by n)}$$

$$100 \le 100n$$

$$1 \le n$$

So we can set $n' = 1$ and reverse the steps above.

Example: Lower Bound

Claim: $n^2 + 100n = \Omega(n^2)$

Proof: Must find c, n' such that for all $n > n'$,

$$n^2 + 100n \geq cn^2$$

Let's try setting $c = 1$. Then

$$n^2 + 100n \geq n^2$$

$$n \geq 0$$

So we can set $n' = 0$ and reverse the steps above.

Thus we can also conclude $n^2 + 100n = \theta(n^2)$

Conventions of Order Notation

Order notation is not symmetric: write $2n^2 + n = O(n^2)$

but never $O(n^2) = 2n^2 + n$

The expression $O(f(n)) = O(g(n))$ is equivalent to

$f(n) = O(g(n))$

The expression $\Omega(f(n)) = \Omega(g(n))$ is equivalent to

$f(n) = \Omega(g(n))$

The right-hand side is a "cruder" version of the left:

$18n^2 = O(n^2) = O(n^3) = O(2^n)$

$18n^2 = \Omega(n^2) = \Omega(n \log n) = \Omega(n)$

Upper/Lower vs. Worst/Best

- Worst case upper bound is $f(n)$
  - Guarantee that run time is no more than $c\,f(n)$
- Best case upper bound is $f(n)$
  - If you are lucky, run time is no more than $c\,f(n)$
- Worst case lower bound is $g(n)$
  - If you are unlucky, run time is at least $c\,g(n)$
- Best case lower bound is $g(n)$
  - Guarantee that run time is at least $c\,g(n)$

Analyzing Code

- primitive operations
- consecutive statements
- function calls
- conditionals
- loops
- recursive functions

Conditionals

- Conditional
  `if` $\boldsymbol{C}$ **`then`** $\boldsymbol{S_1}$ **`else`** $\boldsymbol{S_2}$

- Suppose you are doing a O( ) analysis
  Time(C) + Max(Time(S1),Time(S2))
  or Time(C)+Time(S1)+Time(S2)
  or …

- Suppose you are doing a Ω( ) analysis
  Time(C) + Min(Time(S1),Time(S2))
  or Time(C)
  or …

Nested Loops

```
for i = 1 to n do
  for j = 1 to n do
    sum = sum + 1
```

Nested Loops

```
for i = 1 to n do
   for j = 1 to n do
      sum = sum + 1
```

$$\sum_{i=1}^{n}\sum_{j=1}^{n} 1 = \sum_{i=1}^{n} n = n^2$$

Nested Dependent Loops

```
for i = 1 to n do
  for j = i to n do
    sum = sum + 1
```

Nested Dependent Loops

```
for i = 1 to n do
  for j = i to n do
    sum = sum + 1
```

$$\sum_{i=1}^{n} \sum_{j=i}^{n} 1 = \quad ?$$

Nested Dependent Loops

```
for i = 1 to n do
  for j = i to n do
    sum = sum + 1
```

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

$$= \sum_{i=1}^{n} n - \sum_{i=1}^{n} i + \sum_{i=1}^{n} 1 = n^2 - \sum_{i=1}^{n} i + n$$

Arithmetic Series

$$S(N) = 1 + 2 + \ldots + N = \sum_{i=1}^{N} i = ?$$

- Note that: S(1) = 1, S(2) = 3, S(3) = 6, S(4) = 10, …
- Hypothesis: S(N) = N(N+1)/2

Prove by induction

- Base case: for N = 1, S(N) = 1(2)/2 = 1
- Assume true for N = k
- Suppose N = k+1.
- S(k+1) = S(k) + (k+1)
  = k(k+1)/2 + (k+1)
  = (k+1)(k/2 + 1)
  = (k+1)(k+2)/2.

Other Important Series

- Sum of squares: $$\sum_{i=1}^{N} i^2 = \frac{N(N+1)(2N+1)}{6} \approx \frac{N^3}{3} \text{ for large N}$$
- Sum of exponents: $$\sum_{i=1}^{N} i^k \approx \frac{N^{k+1}}{|k+1|} \text{ for large N and k} \neq \text{-1}$$
- Geometric series: $$\sum_{i=0}^{N} A^i = \frac{A^{N+1}-1}{A-1}$$
- Novel series:
  - Reduce to known series, or prove inductively

Nested Dependent Loops

```
for i = 1 to n do
  for j = i to n do
    sum = sum + 1
```

$$n^2 - \sum_{i=1}^{n} i + n = n^2 - \frac{n(n+1)}{2} + n$$

$$= n(n+1) - \frac{n(n+1)}{2} = \frac{n(n+1)}{2}$$

$$= n^2/2 + n/2 = \theta(n^2)$$

Linear Search Analysis

```
void lfind(int x, int a[], int n)
{   for (i=0; i<n; i++)
        if (a[i] == x)
            return;   }
```

- Best case, tight analysis:

- Worst case, tight analysis:

Iterated Linear Search Analysis

```
for (i=0; i<n; i++) a[i] = i;
for (i=0; i<n; i++) lfind(i,a,n);
```

- Easy worst-case upper-bound: $nO(n) = O(n^2)$
- Worst-case tight analysis:
  - Just multiplying worst case by n does not justify answer, since each time lfind is called i is specified

$$\sum_{i=1}^{n}\sum_{j=1}^{i}1 = \sum_{i=1}^{n} i = \frac{n(n+1)}{2} = \theta\left(n^2\right)$$

Analyzing Recursive Programs

1. Express the running time $T(n)$ as a recursive equation

2. Solve the recursive equation
   - For an upper-bound analysis, you can optionally simplify the equation to something larger
   - For a lower-bound analysis, you can optionally simplify the equation to something smaller

Binary Search

```
void bfind(int x, int a[], int n)
{    m = n / 2;
     if (x == a[m]) return;
     if (x < a[m])
         bfind(x, a, m);
     else
         bfind(x, &a[m+1], n-m-1); }
```

What is the worst-case upper bound?

Binary Search

```
void bfind(int x, int a[], int n)
{    m = n / 2;
     if (n <= 1) return;
     if (x == a[m]) return;
     if (x < a[m])
          bfind(x, a, m);
     else
          bfind(x, &a[m+1], n-m-1); }
```

Okay, let's *prove* it is $\theta(\log n)$…

Binary Search

```
void bfind(int x, int a[], int n)
{   m = n / 2;
    if (n <= 1) return;
    if (x == a[m]) return;
    if (x < a[m])
        bfind(x, a, m);
    else
        bfind(x, &a[m+1], n-m-1); }
```

Introduce some constants…

b = time needed for base case

c = time needed to get ready to do a recursive call

Running time is thus: $T(1) = b$

$$T(n) = T(n/2) + c$$

Binary Search Analysis

One sub-problem, half as large

Equation: $T(1) \leq b$

$T(n) \leq T(n/2) + c$ for $n>1$

Solution:

Solving Recursive Equations by Repeated Substitution

- Somewhat "informal", but intuitively clear and straightforward

$$T(n) = T(n/2) + c \qquad \text{substitute for T(n/2)}$$

$$T(n) = T(n/4) + c + c$$

$$T(n) = T(n/4) + c + c \qquad \text{substitute for T(n/4)}$$

$$T(n) = T(n/8) + c + c + c$$

$$T(n) = T(n/2^k) + kc \qquad \text{"inductive leap"}$$

$$T(n) = T(n/2^{\log n}) + c\log n \qquad \text{choose k=log n}$$

$$T(n) = T(n/n) + c\log n$$

$$= T(1) + c\log n = b + c\log n = \theta(\log n)$$

Solving Recursive Equations by Induction

- Repeated substitution and telescoping construct the solution
- If you know the closed form solution, you can validate it by ordinary induction
- For the induction, may want to increase n by a multiple (2n) rather than by n+1

Inductive Proof

$T(1) = b + c\log 1 = b$ base case

Assume $T(n) = b + c\log n$ hypothesis

$T(2n) = T(n) + c$ definition of T(n)

$T(2n) = (b + c\log n) + c$ by induction hypothesis

$T(2n) = b + c((\log n) + 1)$

$T(2n) = b + c((\log n) + (\log 2))$

$T(2n) = b + c\log(2n)$ Q.E.D.

Thus: $T(n) = \theta(\log n)$

Example: Sum of Integer Queue

```
sum_queue(Q){
   if (Q.length() == 0 ) return 0;
   else return Q.deque() +
                  sum_queue(Q);   }
```

- One subproblem
- Linear reduction in size (decrease by 1)

Equation: $T(0) = b$

$T(n) = c + T(n-1)$ for $n>0$

Lower Bound Analysis: Recursive Fibonacci

```
int Fib(n){
  if (n == 0 or n == 1) return 1 ;
  else return Fib(n - 1) + Fib(n - 2); }
```

- *Lower* bound analysis $\Omega(n)$
- Instead of =, equations will use ≥
  T(n) ≥ Some expression
- Will simplify math by throwing out terms on the right-hand side

Analysis by Repeated Subsitution

$T(0) = T(1) = a$	base case
$T(n) = b + T(n-1) + T(n-2)$	recursive case
$T(n) \geq b + 2T(n-2)$	simplify to smaller quantity
$T(n) \geq b + 2(b + 2T(n-2-2))$	substitute
$T(n) \geq 3b + 4T(n-4))$	simplify
$T(n) \geq 3b + 4(b + 2T(n-4-2))$	substitute
$T(n) \geq 7b + 8T(n-6))$	simplify
$T(n) \geq 7b + 8(b + 2T(n-6-2))$	substitute
$T(n) \geq 15b + 16T(n-8)$	simplify
$T(n) \geq (2^k - 1)b + 2^k T(n-2k)$	inductive leap
$T(n) \geq (2^{n/2} - 1)b + 2^{n/2} T(n - 2(n/2))$	choose k=(n/2)
$T(n) \geq 2^{n/2}(b+a) - b$	simplify
$T(n) = \Omega(2^{n/2})$	Note: this is not the same as $\Omega(2^n)$!!!