CSE 373: Data Structures and Algorithms

Lecture 5: Math Review/Asymptotic Analysis III

Growth rate terminology

- T(N) = O(f(N))
  - f(N) is an **upper bound** on T(N)
  - T(N) **grows no faster** than f(N)
- T(N) = Ω(g(N))
  - g(N) is a **lower bound** on T(N)
  - T(N) grows at least as fast as g(N)
- T(N) = Θ(g(N))
  - T(N) grows at the same rate as g(N)
- T(N) = o(h(N))
  - T(N) grows strictly slower than h(N)

More about asymptotics

- Fact: If f(N) = O(g(N)), then g(N) = Ω(f(N)).

- Proof: Suppose f(N) = O(g(N)).
  Then there exist constants c and n_0 such that
  f(N) $\leq$ c g(N) for all N $\geq$ n_0

  Then g(N) $\geq$ (1/c) f(N) for all N $\geq$ $n_{0,}$
  and so g(N) = Ω(f(N))

Facts about big-Oh

- If $T_1(N) = O(f(N))$ and $T_2(N) = O(g(N))$, then
  - $T_1(N) + T_2(N) = O(f(N) + g(N))$
  - $T_1(N) * T_2(N) = O(f(N) * g(N))$
- If $T(N)$ is a polynomial of degree *k*, then: $T(N) = \Theta(N^k)$
  - example: $17n^3 + 2n^2 + 4n + 1 = \Theta(n^3)$
- $\log^k N = O(N)$, for any constant *k*

Complexity cases

- **Worst-case complexity**: “most challenging” input of size n
- **Best-case complexity:** “easiest” input of size n
- **Average-case complexity**: random inputs of size n
- **Amortized complexity**: m “most challenging” *consecutive* inputs of size n, divided by m

Bounds vs. Cases

Two orthogonal axes:

- Bound
  - Upper bound (O, o)
  - Lower bound (Ω)
  - Asymptotically tight (Θ)
- Analysis Case
  - Worst Case (Adversary), $T_{worst}(n)$
  - Average Case, $T_{avg}(n)$
  - Best Case, $T_{best}(n)$
  - Amortized, $T_{amort}(n)$

One can estimate the bounds for any given case.

Example

`List.contains(Object o)`

- returns `true` if the list contains `o`; `false` otherwise
- Input size: n (the length of the `List`)
- $T(n)$ = "running time for size n"
- But $T(n)$ needs clarification:
  - Worst case $T(n)$: it runs in at most $T(n)$ time
  - Best case $T(n)$: it takes at least $T(n)$ time
  - Average case $T(n)$: average time

Complexity classes

- **complexity class**: A category of algorithm efficiency based on the algorithm's relationship to the input size N.

Class	Big-Oh	If you double N, ...	Example
constant	$O(1)$	unchanged	10ms
logarithmic	$O(\log_2 N)$	increases slightly	175ms
linear	$O(N)$	doubles	3.2 sec
log-linear	$O(N \log_2 N)$	slightly more than doubles	6 sec
quadratic	$O(N^2)$	quadruples	1 min 42 sec
cubic	$O(N^3)$	multiplies by 8	55 min
...	...	...	...
exponential	$O(2^N)$	multiplies drastically	$5 * 10^{61}$ years

Recursive programming

- A method in Java can call itself; if written that way, it is called a *recursive method*

- The code of a recursive method should be written to handle the problem in one of two ways:
  - **base case**: a simple case of the problem that can be answered directly; does not use recursion.
  - **recursive case**: a more complicated case of the problem, that isn't easy to answer directly, but can be expressed elegantly with recursion; makes a recursive call to help compute the overall answer

Recursive power function

- Defining powers recursively:

```
pow(x, 0) = 1
pow(x, y) = x * pow(x, y-1),    y > 0
```

```
// recursive implementation
public static int pow(int x, int y) {
    if (y == 0) {
        return 1;
    } else {
        return x * pow(x, y - 1);
    }
}
```

Searching and recursion

- Problem: Given a <u>sorted</u> array *a* of integers and an integer *i*, find the index of any occurrence of *i* if it appears in the array. If not, return -1.
  - We could solve this problem using a standard iterative search; starting at the beginning, and looking at each element until we find *i*
  - What is the runtime of an iterative search?

- However, in this case, the array is sorted, so does that help us solve this problem more intelligently? Can recursion also help us?

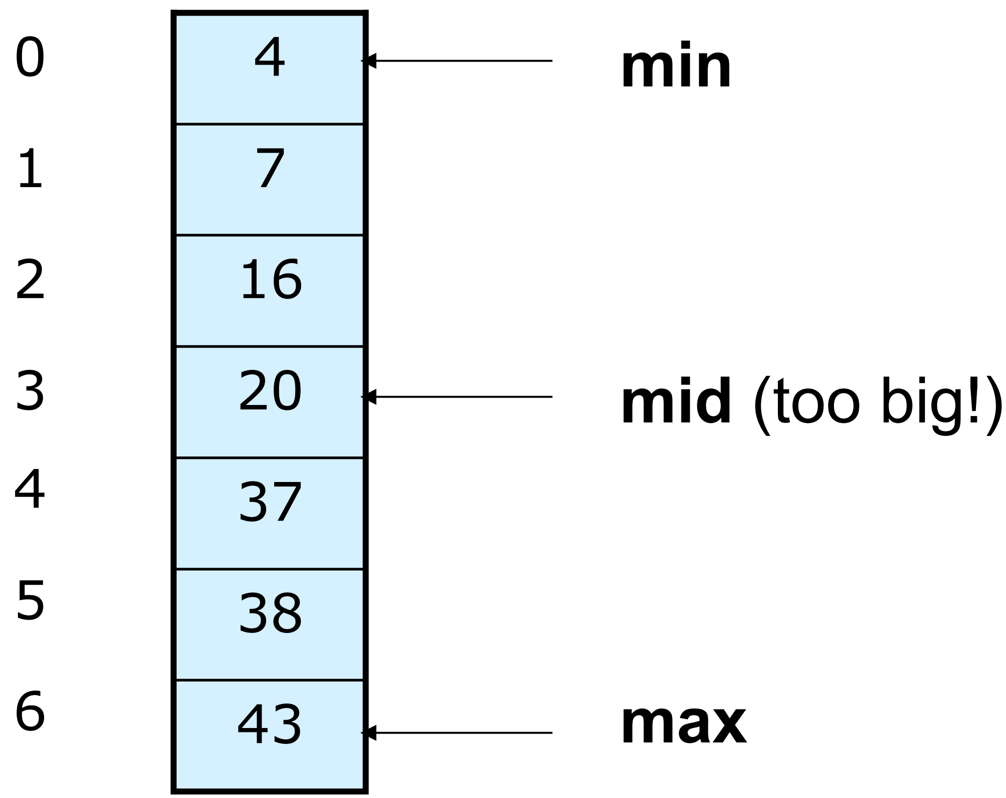
Binary search algorithm

- Algorithm idea: Start in the middle, and only search the portions of the array that might contain the element *i*. Eliminate half of the array from consideration at each step.
  – can be written iteratively, but is harder to get right

- called **binary search** because it chops the area to examine in half each time
  – implemented in Java as method `Arrays.binarySearch` in `java.util` package

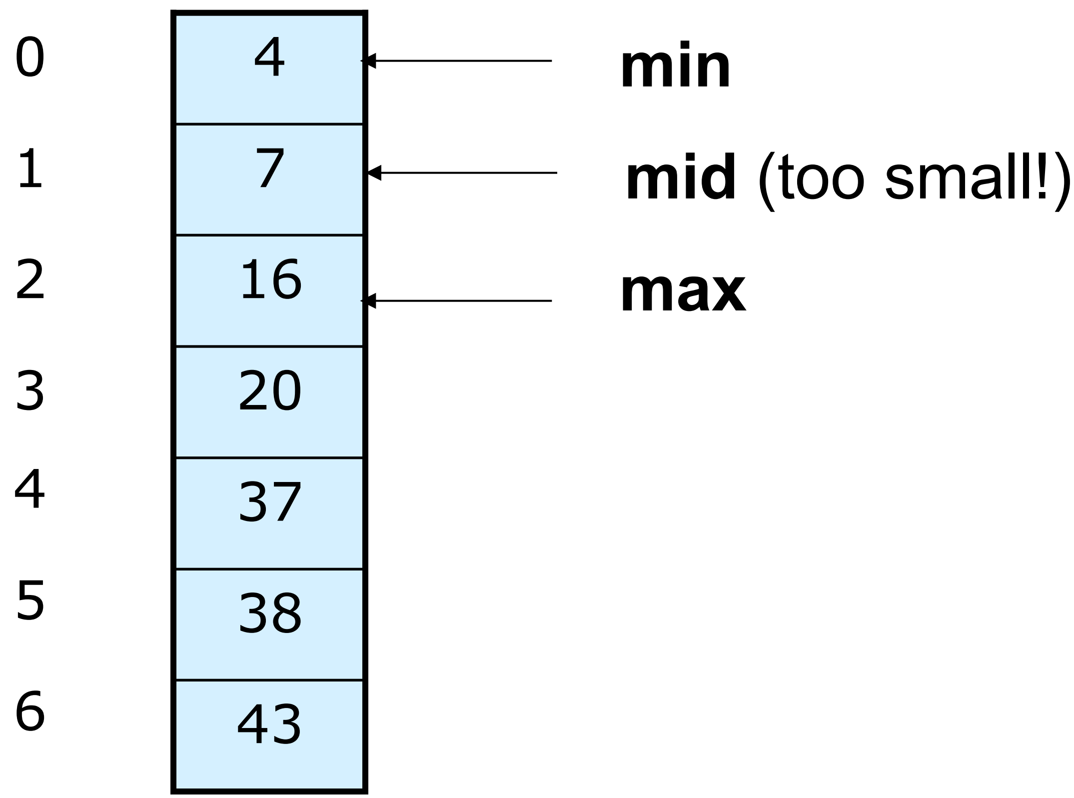
Binary search example

i = 16



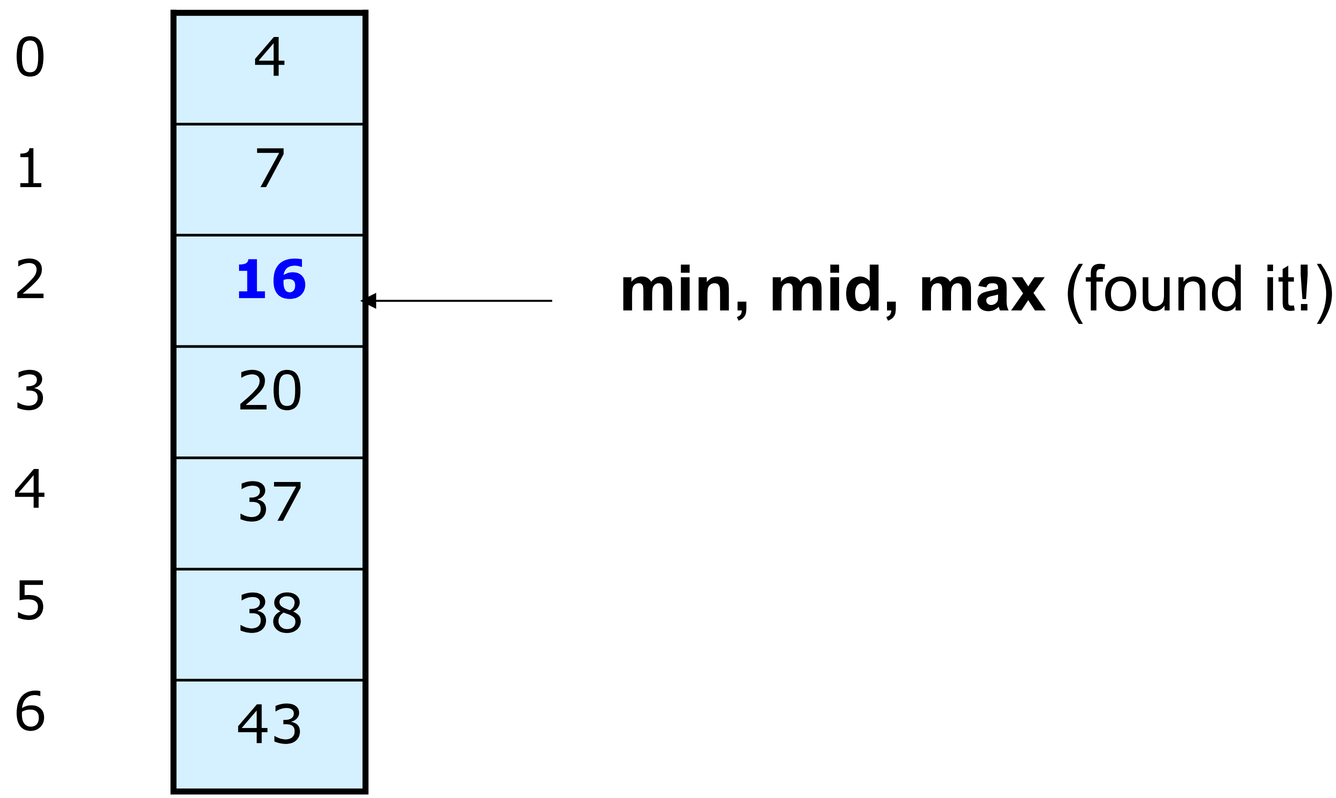
Binary search example

i = 16



Binary search example

i = 16



Binary search pseudocode

```
binary search array a for value i:
   if all elements have been searched,
      result is -1.
   examine middle element a[mid].
   if a[mid] equals i,
      result is mid.
   if a[mid] is greater than i,
      binary search left half of a for i.
   if a[mid] is less than i,
      binary search right half of a for i.
```

Runtime of binary search

- How do we analyze the runtime of binary search and recursive functions in general?

- binary search either exits immediately,
  when input size <= 1 or value found (base case),
  or executes itself on 1/2 as large an input (rec. case)
  - T(1) = c
  - T(2) = T(1) + c
  - T(4) = T(2) + c
  - T(8) = T(4) + c
  - ...
  - T(n) = T(n/2) + c

- How many times does this division in half take place?

Divide-and-conquer

- **divide-and-conquer algorithm**: a means for solving a problem that first separates the main problem into 2 or more smaller problems, then solves each of the smaller problems, then uses those sub-solutions to solve the original problem
  - 1: "divide" the problem up into pieces
  - 2: "conquer" each smaller piece
  - 3: (if necessary) combine the pieces at the end to produce the overall solution

  - binary search is one such algorithm

Recurrences, in brief

- How can we prove the runtime of binary search?

- Let's call the runtime for a given input size *n*, T(*n*). At each step of the binary search, we do a constant number *c* of operations, and then we run the same algorithm on 1/2 the original amount of input. Therefore:

  – $T(n) = T(n/2) + c$
  – $T(1) = c$

- Since T is used to define itself, this is called a **recurrence relation**.

Solving recurrences

- **Master Theorem**:
  A recurrence written in the form
  T(*n*) = *a* * T(*n* / b) + f(*n*)

  (where f(*n*) is a function that is O(n^k) for some power *k*)
  has a solution such that

$$T(n) = \begin{cases} O(n^{\log_b a}), & a > b^k \\ O(n^k \log n), & a = b^k \\ O(n^k), & a < b^k \end{cases}$$

- This form of recurrence is very common for divide-and-conquer algorithms

Runtime of binary search

- Binary search is of the correct format:
  $T(n) = a * T(n / b) + f(n)$
  - $T(n) = T(n/2) + c$
  - $T(1) = c$
  - $f(n) = c = O(1) = O(n^0)$ ... therefore $k = 0$
  - $a = 1$, $b = 2$
- $1 = 2^0$, therefore:
  $T(n) = O(n^0 \log n) =$ **O(log n)**
- (recurrences not needed for our exams)

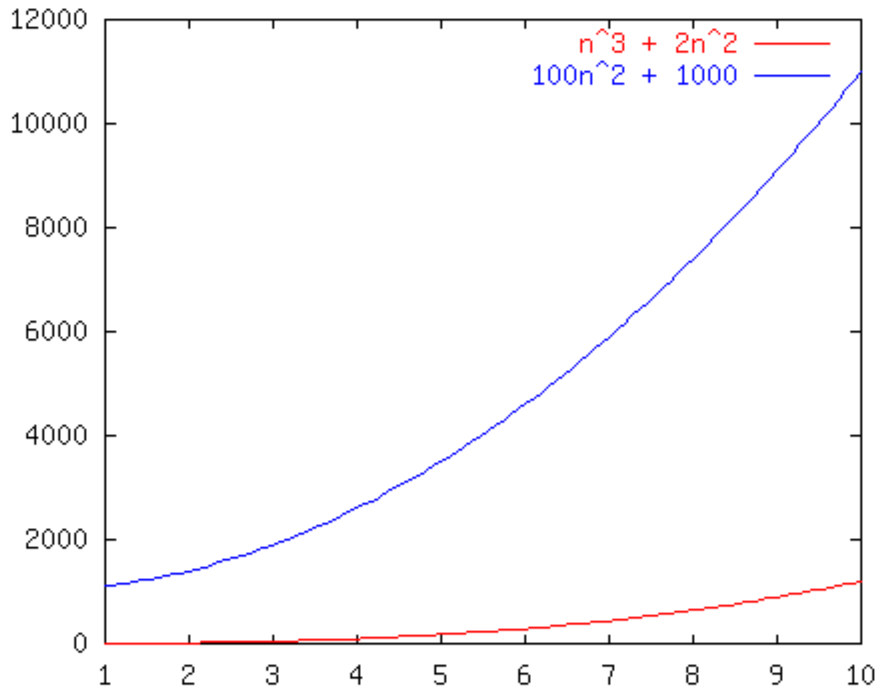
Which Function Dominates?

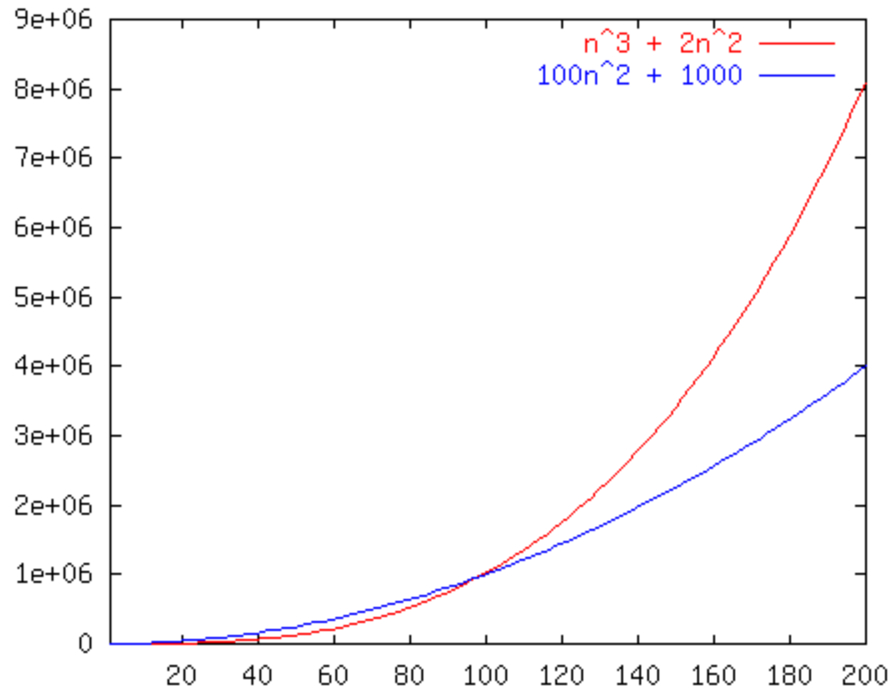
f(n) =	g(n) =
$n^3 + 2n^2$	$100n^2 + 1000$
$n^{0.1}$	$\log n$
$n + 100n^{0.1}$	$2n + 10 \log n$
$5n^5$	$n!$
$n^{-15}2^n/100$	$1000n^{15}$
$8^{2\log n}$	$3n^7 + 7n$

What we are asking is: is $f = O(g)$? Is $g = O(f)$?

Race I

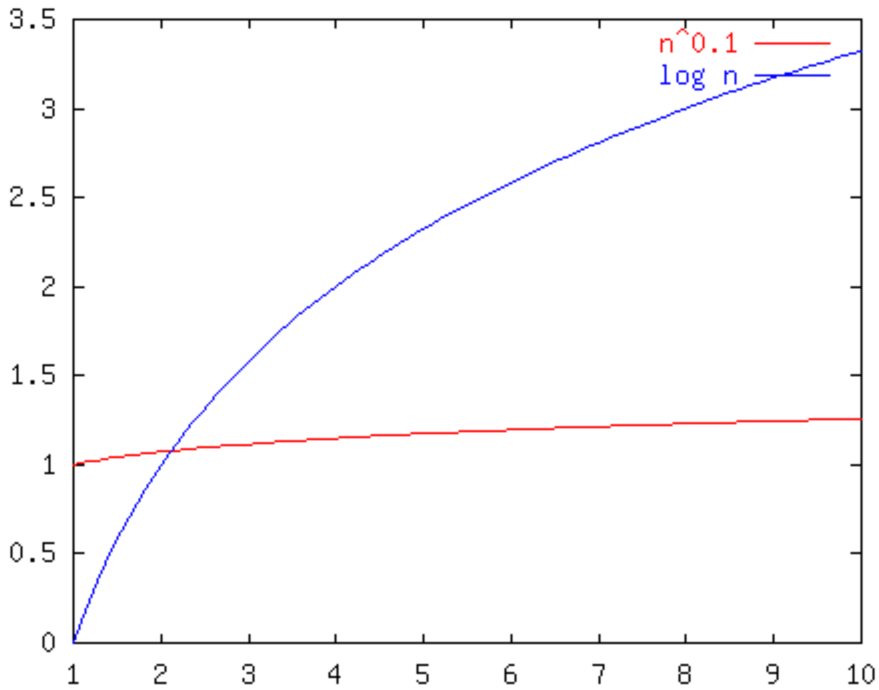
$f(n) = n^3 + 2n^2$ vs. $g(n) = 100n^2 + 1000$

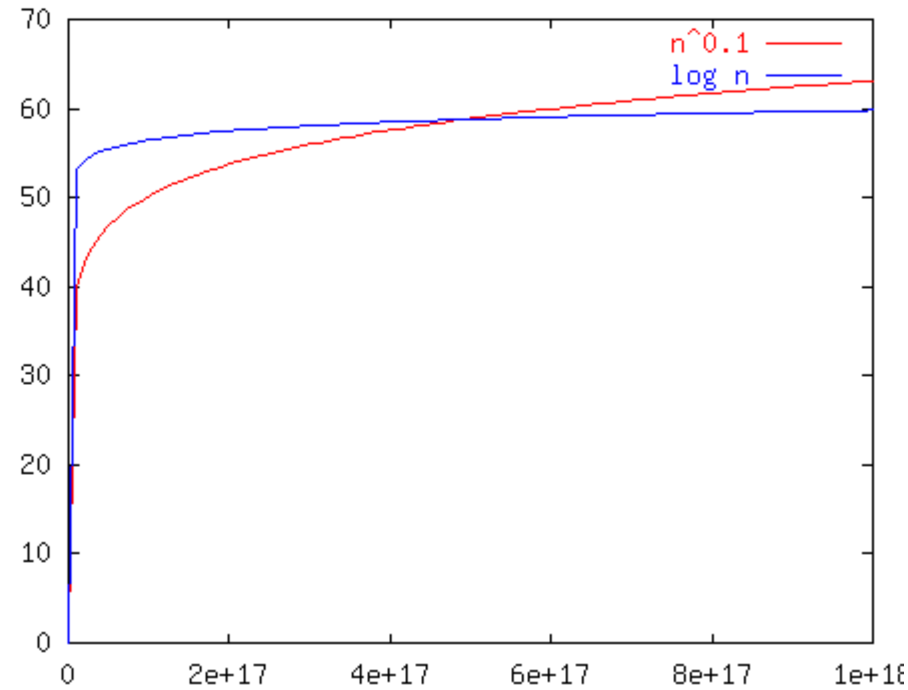




Race II

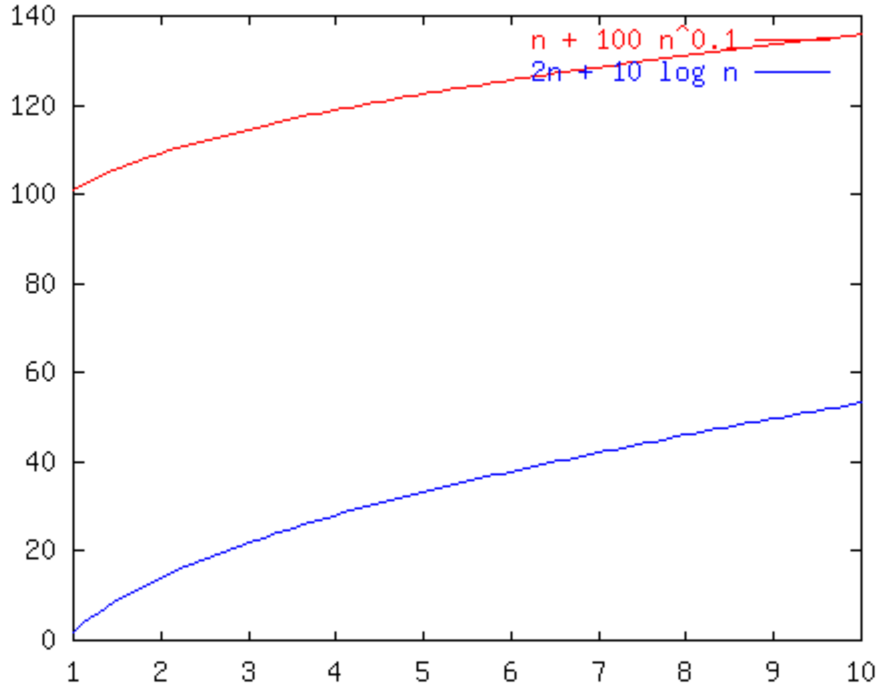
$n^{0.1}$ vs. **log n**

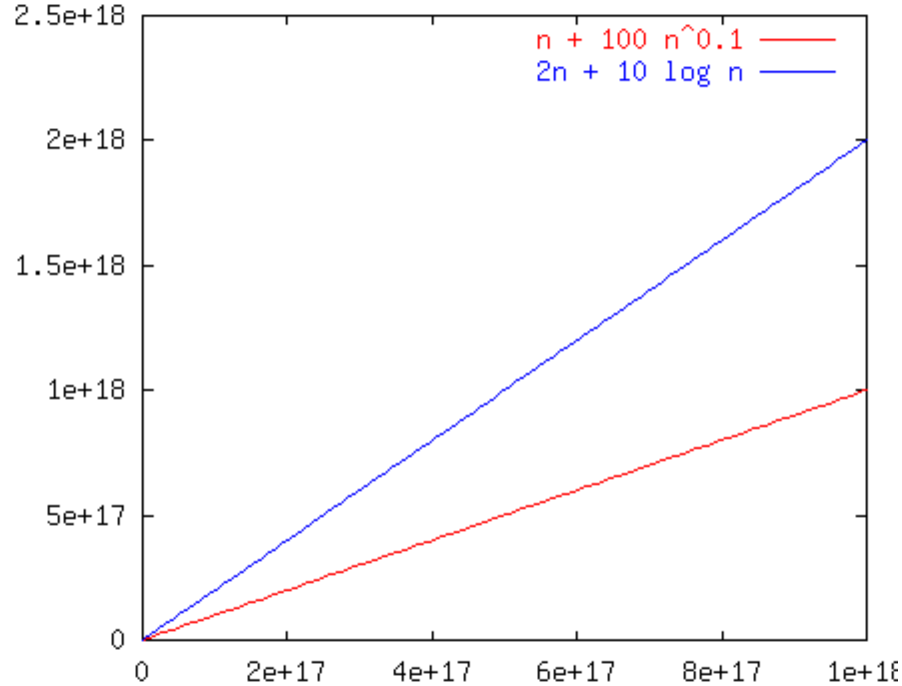




Race III

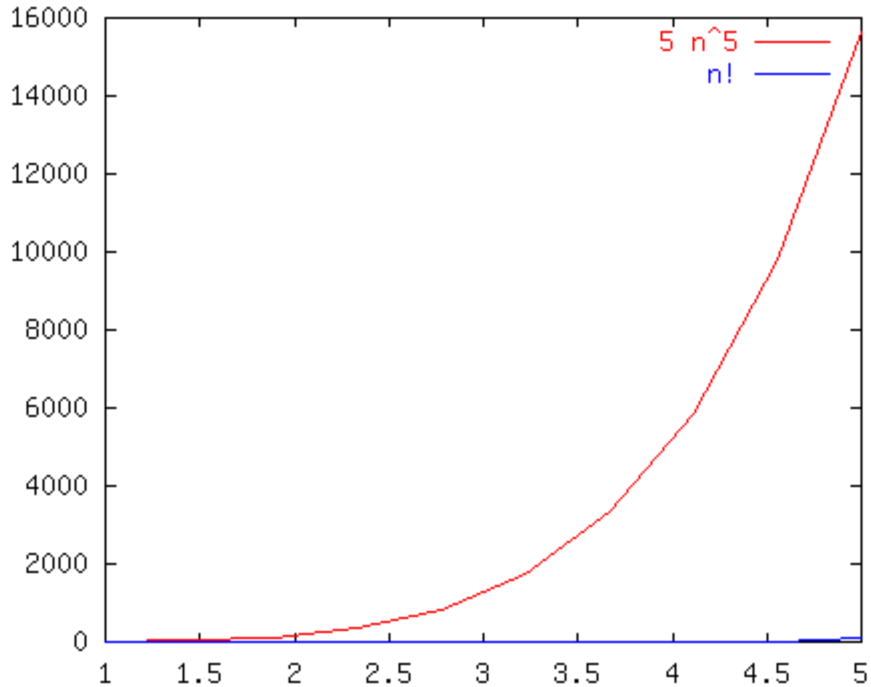
$\mathtt{n + 100n^{0.1}}$ vs. $\mathtt{2n + 10 \log n}$

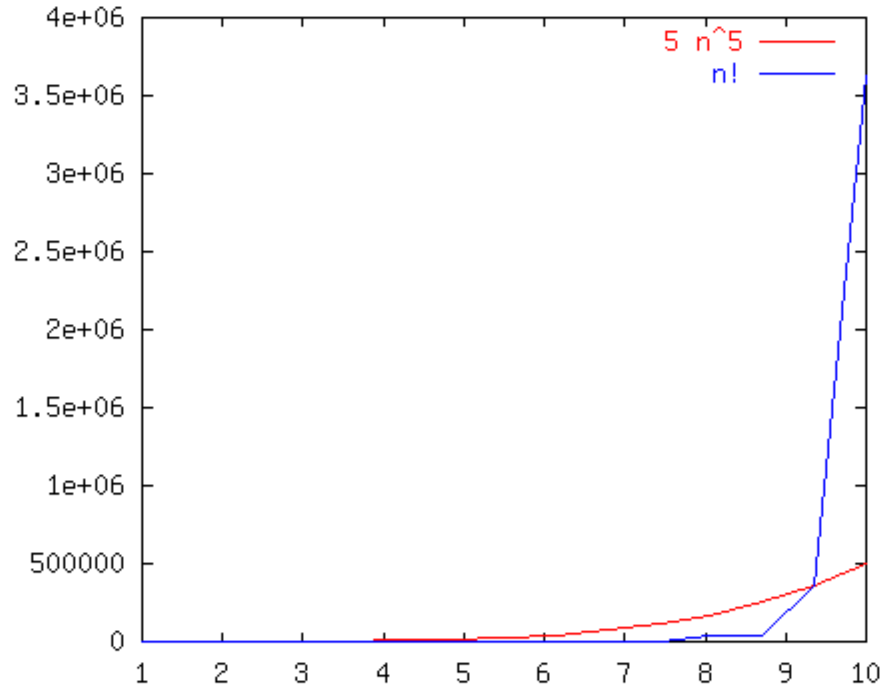




Race IV

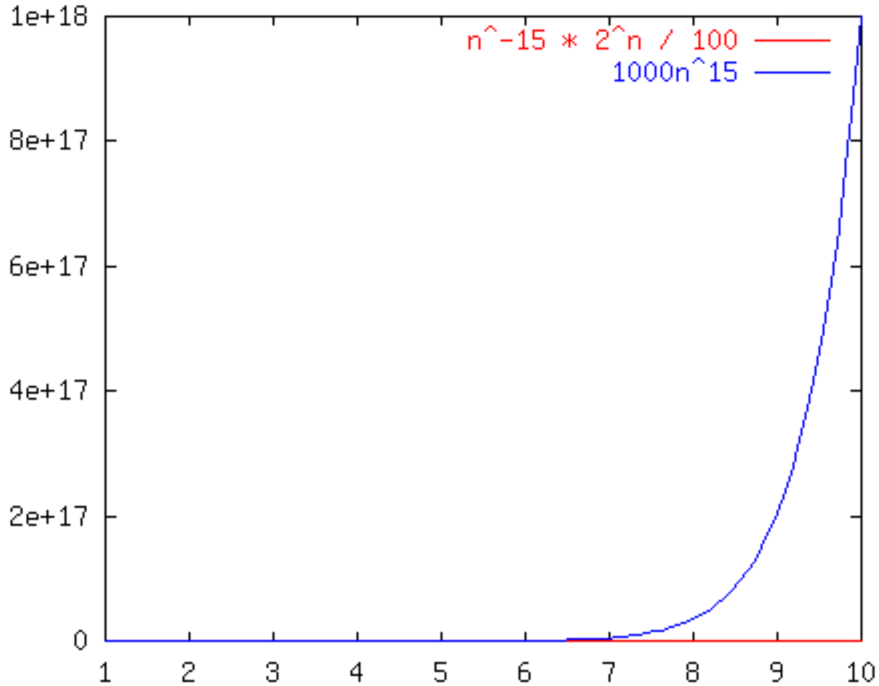
$5n^5$ vs. $n!$

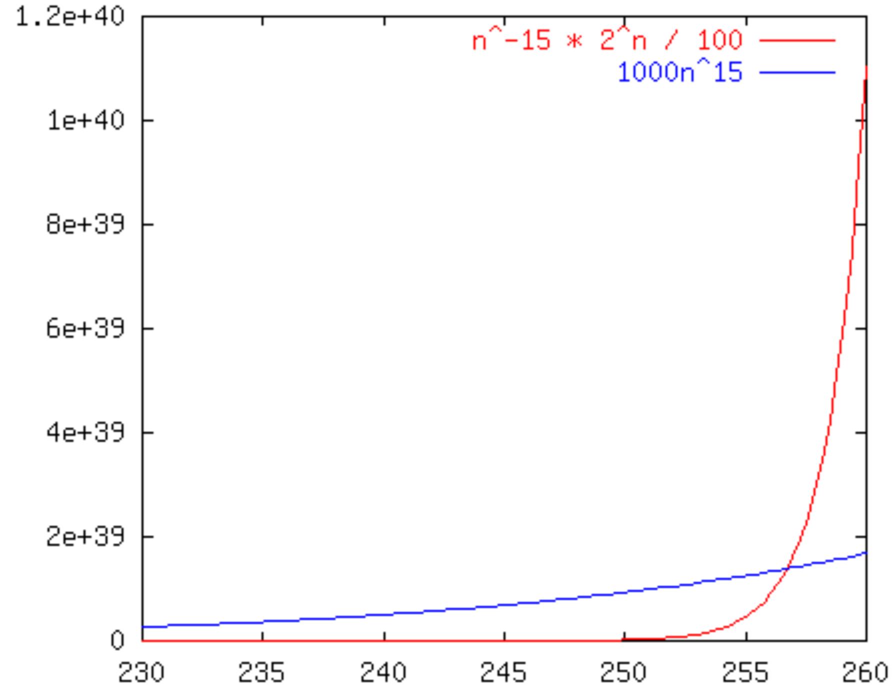




Race V

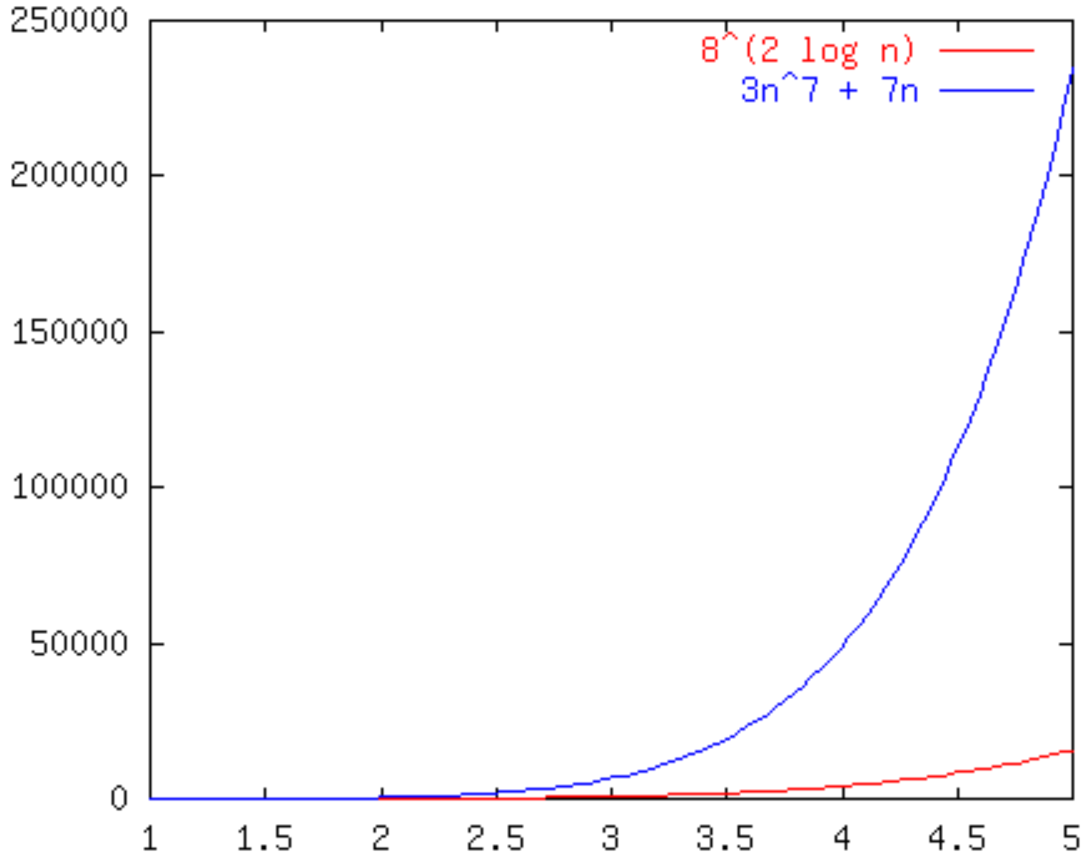
$n^{-15}2^n/100$ vs. $1000n^{15}$





Race VI

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



A Note on Notation

You'll see...

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

and people often say...

$g(n)$ is $O(f(n))$.

These really mean

$g(n) \in O(f(n))$.

That is, $O(f(n))$ represents a set or class of functions.