Defining Efficiency

- *Asymptotic Complexity* - how running time scales as function of size of input
- Two problems:
  - What is the "input size" ?
  - How do we express the running time ? (The Big-O notation, and its associates)

Input Size

- Usually: length (in characters) of input
- Sometimes: number of "items" (Ex: numbers in an array, nodes in a graph)
- Input size is denoted **n**.
- Which inputs (because there are many inputs at a given size)?
  - **Worst case**: look at the input of size n that causes the longest running time; tells us how *good* an algorithm works
  - **Best case**: look at the input on which the algorithm is the fastest; it is rarely relevant
  - **Average case**: useful in practice, but there are technical problems here (next)

Input Size

Average Case Analysis

- Assume inputs are *randomly* distributed according to some "realistic" distribution Δ
- Compute expected running time

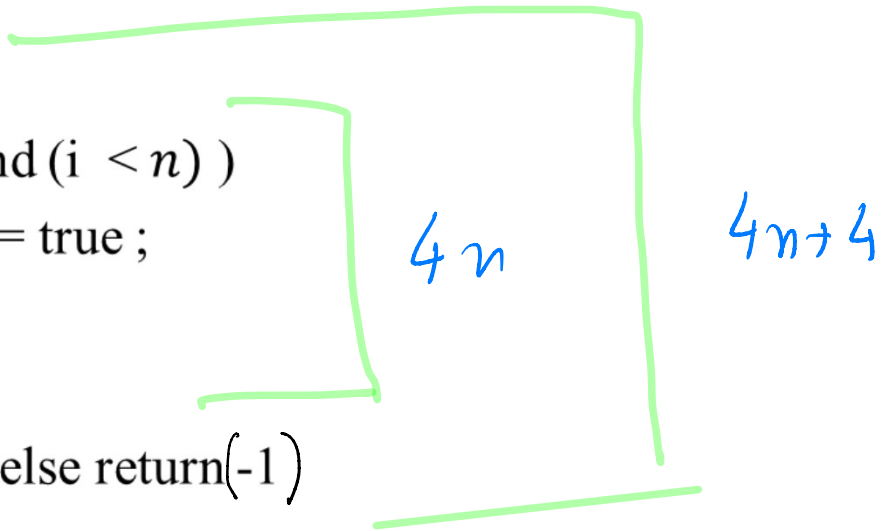
$$E(T,n) = \sum_{x \in \text{Inputs}(n)} \text{Prob}_{\Delta}(x)\,\text{RunTime}(x)$$

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

Input Size

- Example: the function: `find(x, v, n)(find v in the array x[1..n])`
- Input size: n (the length of the array)
- T(n) = "running time for size `n`"
- But T(n) needs clarification:
  - Worst case T(n): it runs in at most T(n) time for any `x,v`
  - Best case T(n): it takes at least T(n) time for any `x,v`
  - Average case T(n): average time over all `v` and `x`

- find (x, v, n) {
  found = false; i=0
  while ((! found) and (i < n))
    if (x[i]=v) found= true ;
    else i++;
  end-while
  if (found) return (i) else return(-1)
}



Worst-case: c × n , for some constant c.

Best case: c

Average case: n/2 (times some constant, assuming v is equally likely to be in any position, or not in x at all)

Amortized analysis

Typically for data structures.

- Instead of a single execution of the alg., consider a *sequence* of consecutive executions (in other words, a *history* of executions):
  - This is interesting when the running time on an execution depends on the result of previous ones (typically, this holds for operations on data structures.)
- Worst case analysis over the sequence of executions $op_1, op_2, \ldots, op_n$
- Determine average running time on this sequence

Amortized runtime $= [\ time(op_1) + time(op_2) + \ldots + time(op_n)\ ] / n$

Average over history

- Will illustrate in the course

Asymptotic notation (or Big O) (or ORDER NOTATION)

- Part of the basic vocabulary for discussing algorithms
- Big O – coarse enough to suppress small details that depend on programming implementation (style, language, compiler) and computer;

  - precise enough to understand scaling, to make meaningful comparisons between algorithms.

High-level idea:

they depend on implementation, rather than algorithm

not relevant for large inputs

suppress constant factors and lower–order terms

Recall Alg 1 for max contig. subsequence sum: run time

We simply say this is $O(n^3)$.

$\frac{7}{6}n^3 + \frac{11}{2}n^2 + \frac{22}{3}n + 5$

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:

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

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

Other notations: o(f), ω(f) - later

$\exists c_1, c_2, n'$

$T(n) \leq c_1 \cdot f(n)$

$\geq c_2 \cdot f(n)$

for all $n \geq n'$

Warning about the notation T(n) = O(f(n))

- The notation is not mathematically correct, but everyone uses it, so we'll use it too.
- What's wrong: the left hand side is a function, the right hand side is a set of functions.
- $O(f(n)) = \{ t : N \to N \mid \exists\, c, n_0 \;\; t(n) \leq c\, f(n) \; for \; all \; n \geq n_0 \}$
- It should be $T(n) \in O(f(n))$.
- Same problem with $\Omega(f(n))$, $\Theta(f(n))$, $o(f(n))$, $\omega(f(n))$.

BETTER FORMULATIONS

"T(n) is in O(f(n))"

or "T(n) is O(f(n))"

Example: Upper Bound

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

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

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

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

$$n^2 + 100n \leq 2n^2$$

$\Longleftrightarrow$ $100n \leq n^2$

$\Longleftrightarrow$ $100 \leq n$

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

Using a Different Pair of Constants

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

Let's try another c, namely c = 101.

$$n^2 + 100n \leq 101\,n^2$$

$$\Leftrightarrow \quad n + 100 \leq 101\,n \;(divide\ both\ sides\ by\ n)$$

$$\Leftrightarrow \quad 100 \leq 100n$$

$$\Leftrightarrow \quad 1 \leq n$$

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 $n^2 + 100n = \theta(n^2)$

INTUITIVE MEANING OF ORDER NOTATIONS

$T(n) = O(f(n)) \longmapsto T(n)$ "$\leq$" $f(n)$

$T(n) = \Omega(f(n)) \longmapsto T(n)$ "$\geq$" $f(n)$

$T(n) = \Theta(f(n)) \longmapsto T(n)$ "$=$" $f(n)$

$T(n) = o(f(n)) \longmapsto T(n)$ "$<$" $f(n)$

$T(n) = \omega(f(n)) \longmapsto T(n)$ "$>$" $f(n)$

Order notation is not symmetric

We say: $2n^2 + n = O(n^2)$

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

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

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

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

} ALL CORRECT

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

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

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

} ALL CORRECT

Common Functions and Their 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)
hyperexponential:	$O(2^{2^{2^{\cdots^2}}})$	(a tower of n exponentials

Fastest Growth

Other names:

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

Execution time comparison

$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(2^n) < O(n!) < O(2^{2^n})$

Size	**10**	**30**	**50**
O(log n)	0.00004 sec	0.00009 sec	0.00012 sec
O(n)	0.0001 sec	0.0003	0.0005
$O(n^2)$	0.01	0.09	0.25
$O(n^5)$	0.1	24.3	5.2 minutes
$O(2^n)$	0.001 sec	17.9 minutes	35.7 years
$O(3^n)$	0.059	6.5 years	$2*10^8$ centuries

An algorithm is **infeasible** if its running time is super-polynomial

- it takes too long to compute

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

Which Function Dominates?

	f(n) =	g(n) =
(1)	$n^3 + 2n^2$	$100n^2 + 1000$
(2)	$n^{0.1}$	$\log n$
(3)	$n + 100n^{0.1}$	$2n + 10 \log n$
(4)	$5n^5$	$n!$
(5)	$n^{-15}2^n/100$	$1000n^{15}$
(6)	$8^{2\log n}$	$3n^7 + 7n$

Question to class: is f = O(g) ? Is g = O(f) ?

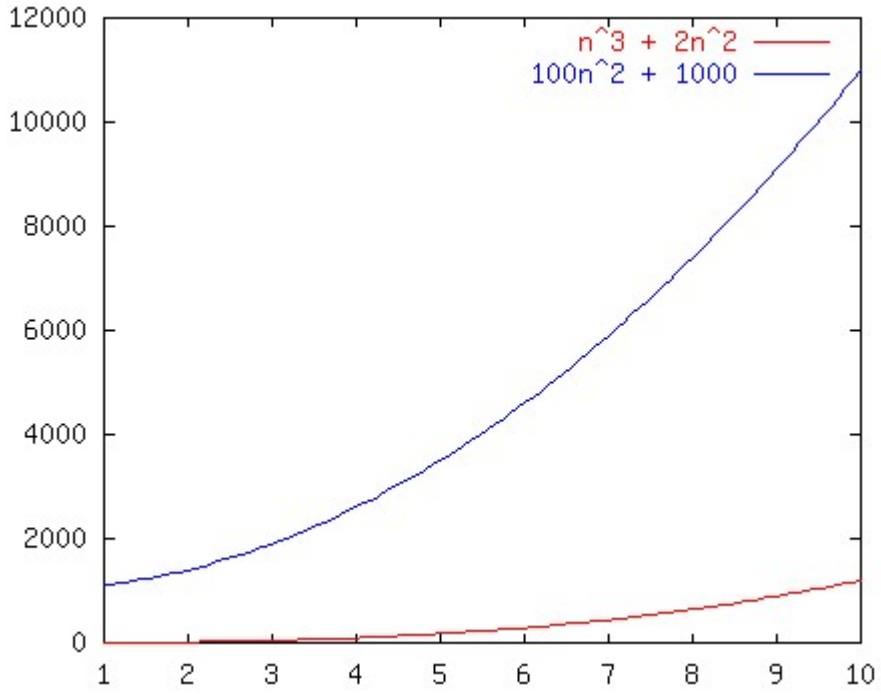
$$8^{2\log_2 n} = \left(2^3\right)^{2\log_2 n} = 2^{6\log_2 n}$$

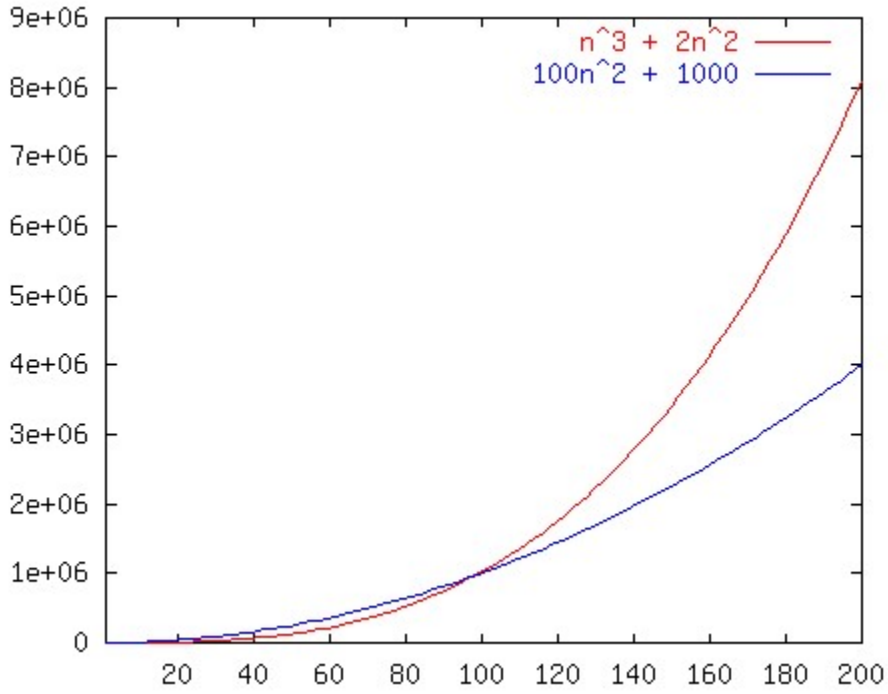
$$= 2^{\log_2 n^6} = n^6$$

$$2^{\log_2 a} = a$$

Race I

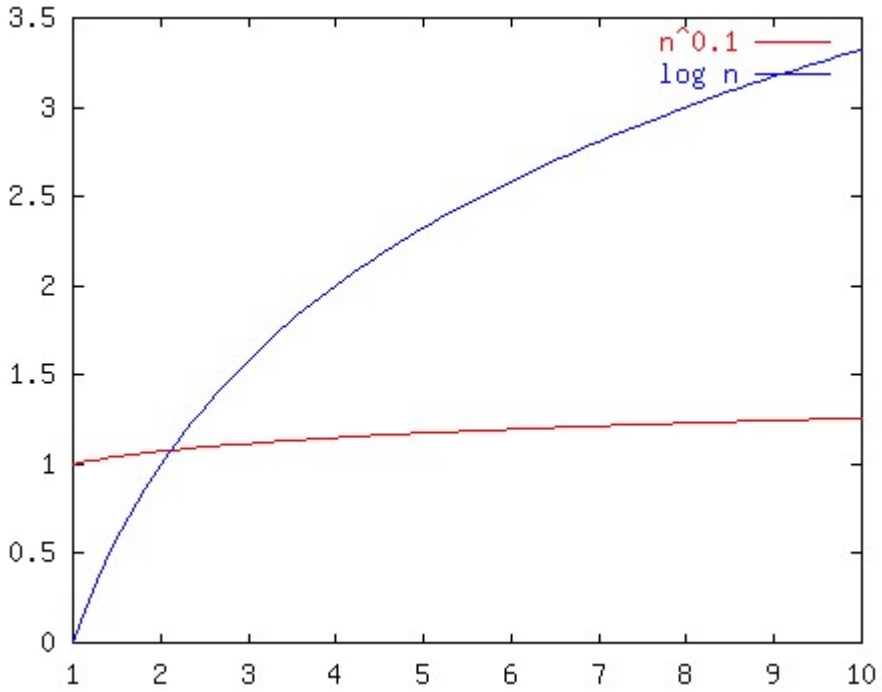
$f(n) = n^3+2n^2$ vs. **$g(n)=100n^2+1000$**

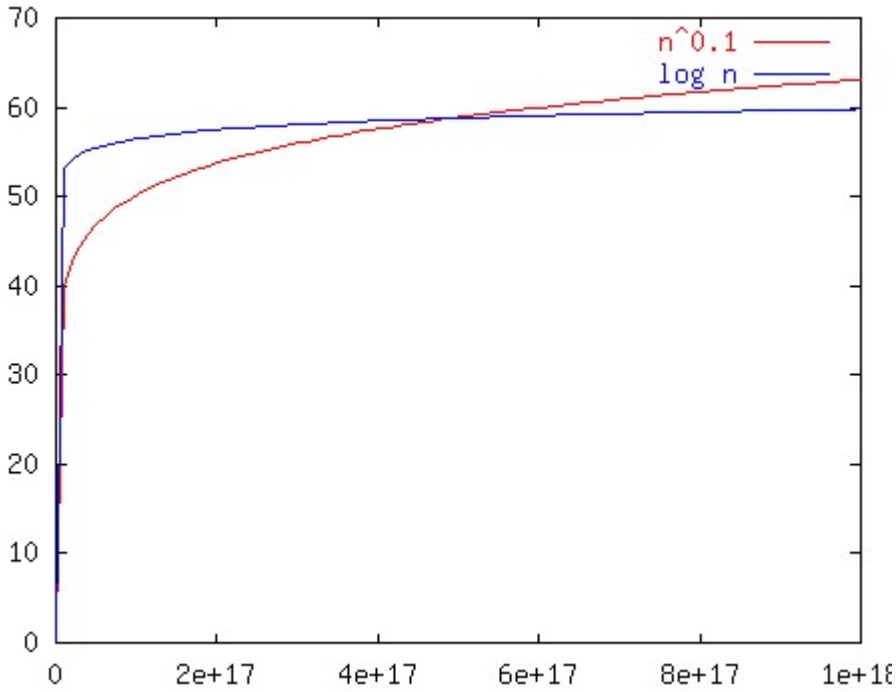




Race II

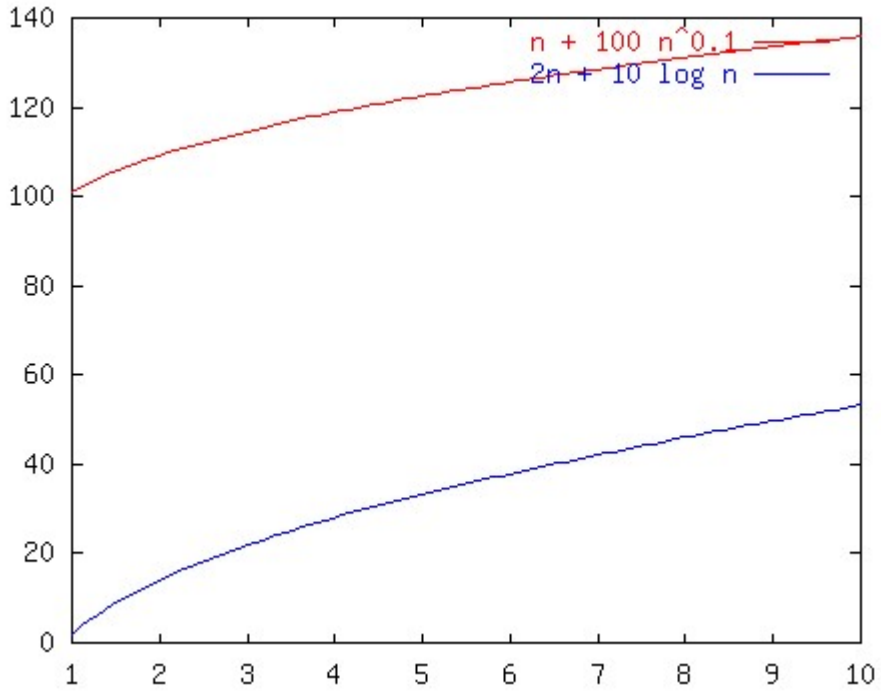
$n^{0.1}$ vs. $\log n$

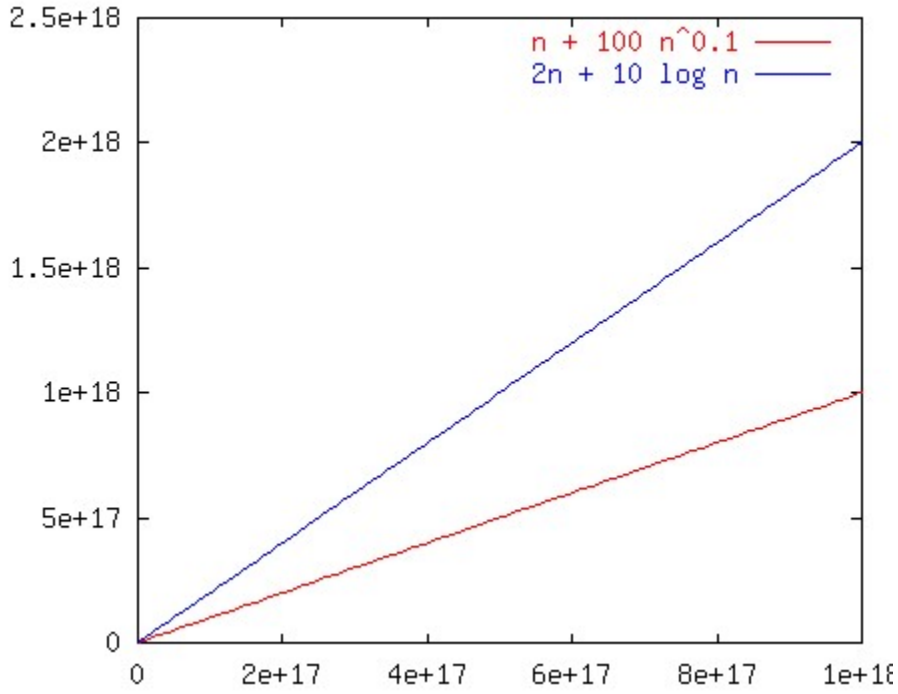




Race III

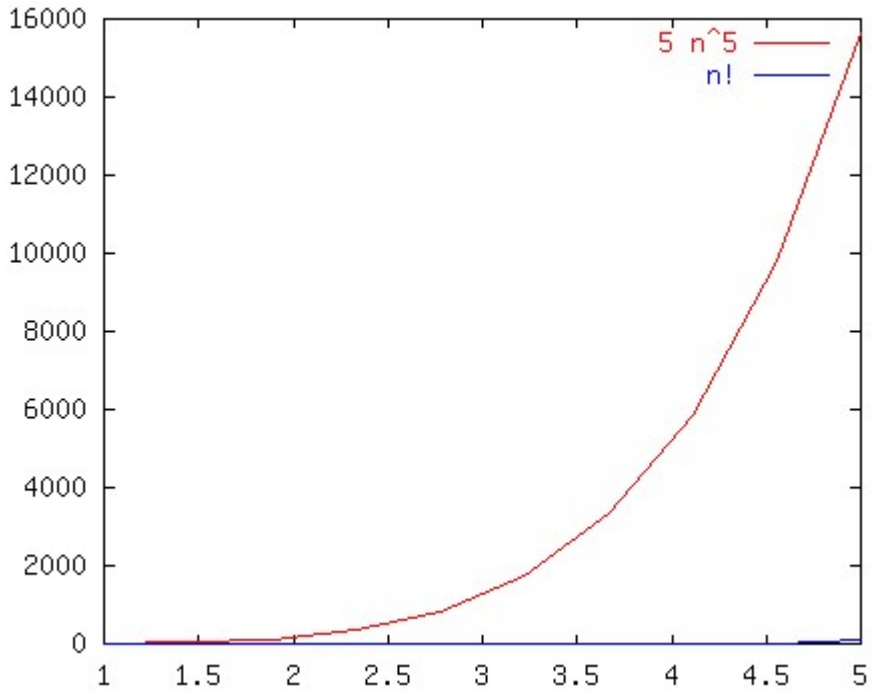
$n + 100n^{0.1}$ vs. **$2n + 10 \log n$**

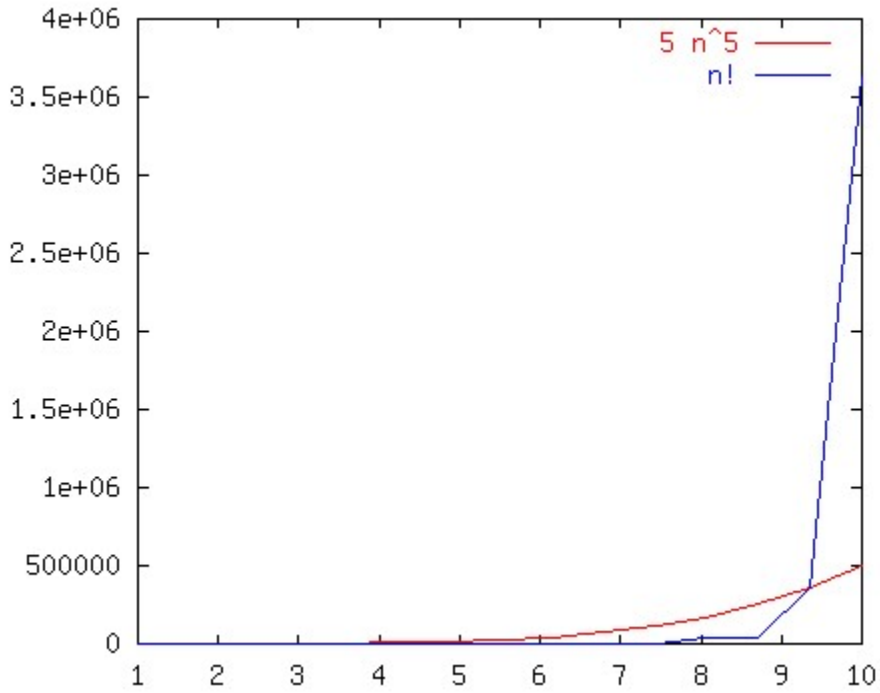




Race IV

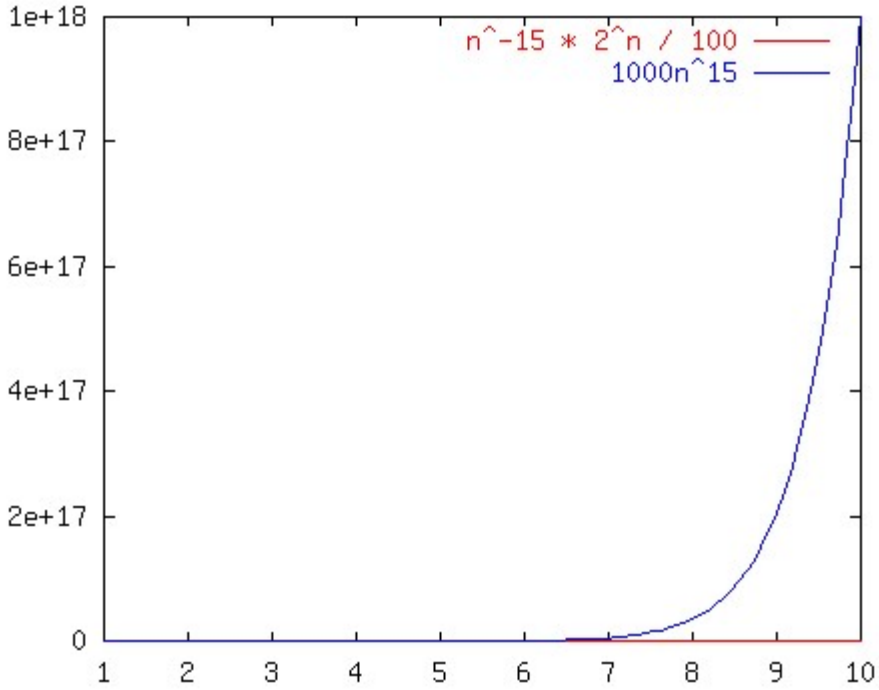
$5n^5$ vs. **$n!$**

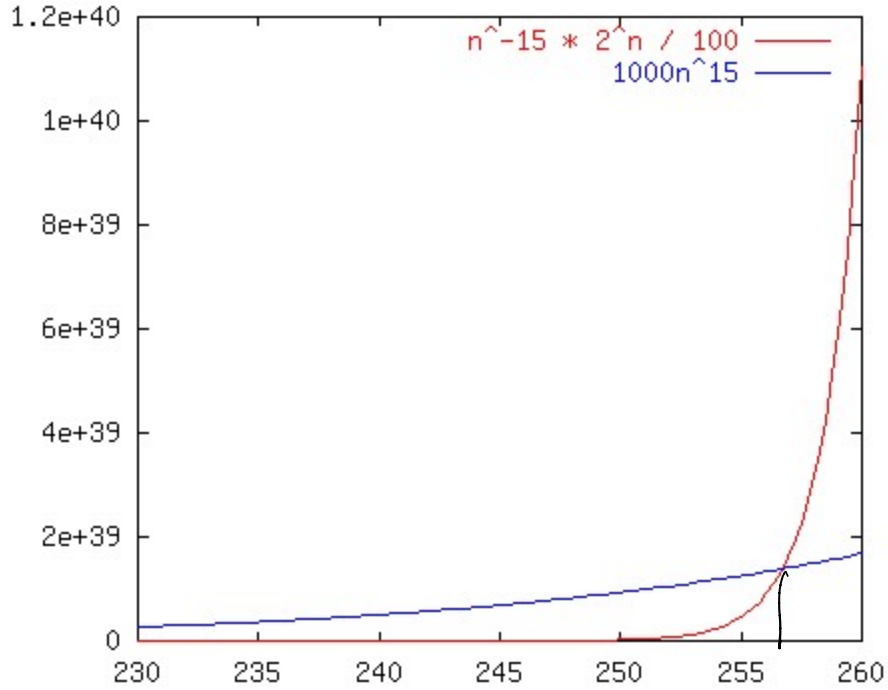




Race V

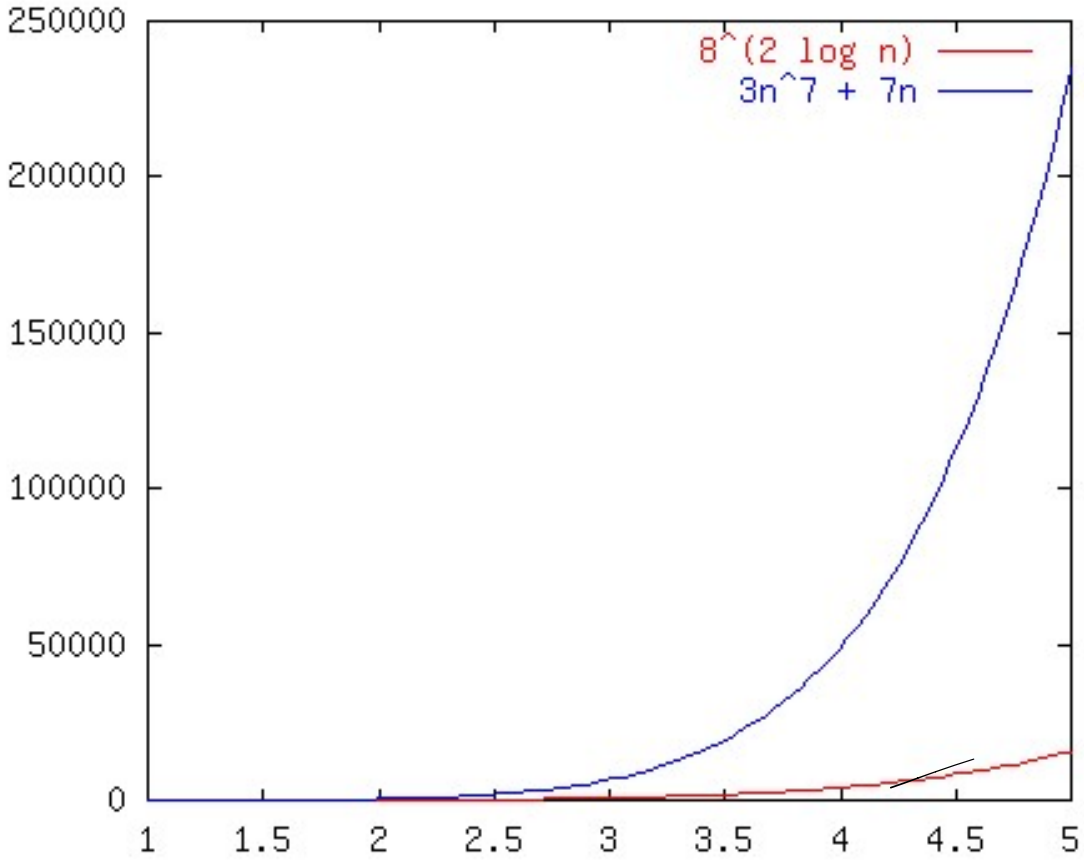
$n^{-15}2^n/100$ vs. $1000n^{15}$





Race VI

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



The big-O arithmetic

Rule 1 (transitivity):

If $T(n) = O(f(n))$ and $f(n) = O(g(n))$ then $T(n) = O(g(n))$

Proof:

Given $T(n) \leq C1 \cdot f(n)$ for $n > n_0'$

$f(n) \leq C2 \cdot g(n)$ for all $n > n_0''$

Substitute

$f(n)$ with $C2 \cdot g(n)$

we have: $T(n) \leq C1 \cdot C2 \cdot g(n)$ for all $n > \max(n_0', n_0'')$

$= C3 \cdot g(n)$ where $C3 = C1 \cdot C2$

Conclusion: $T(n) = O(g(n))$

The big-O arithmetic

Rule 2 (drop the multiplicative constant):

c f(n) = O(f(n)), for any constant c > 0.

Example: $17n^3 = O(n^3)$.

5 = O(1).

Cont. big-O Arithmetic

Rule 3 (max rule: in a sum drop the smaller term):

[redacted]

$T_1(n) + T_2(n) = O(\max(T_1(n), T_2(n)))$

Example: $T_1(n) = 2n^2+4n$

$T_2(n) = 4n^3+2n$

$T_1(n)+T_2(n) = O(4n^3+2n) = O(n^3)$

$T_1(n) = O(f(n))$, $T_2(n) = O(g(n))$

(b) $T_1(n) \times T_2(n) = O(f(n) \times g(n))$

Example: $T_1(n) = 3n^2$ $\quad f(n) = O(n^2)$

$T_2(n) = 4n^5$ $\quad g(n) = O(n^5)$

$T_1(n) \times T_2(n) = 3n^2 \times 4n^5 = O(n^7)$

PROOF OF SUM RULE

$$T_1(n) + T_2(n) = O(\max(T_1(n), T_2(n)))$$

$$T_1(n) + T_2(n) \leq 2 \cdot \max(T_1(n), T_2(n))$$

DONE!

Cont. of big-O rules

Rule 4: If T(n) is a polynomial of degree k, then $T(n) = O(n^k)$

where n is the variable and k is the highest power

Example: $n^5+4n^4+8n^2+9 = O(n^5)$

$7n^3+2n+19 = O(n^3)$

Remarks:

- $2n^2 = O(n^3)$ and $2n^2 = O(n^2)$ are both correct, but $O(n^2)$ is more accurate
- The following notations are to be avoided:
  - $O(2n^2)$
  - $O(5)$
  - $O(n^2+10n)$
- The big-O notation is an upper bound

Big-O – by taking limits

Determine the relative growth rates f(n) vs. g(n) by computing

$$\lim_{n \to \infty} f(n)/g(n)$$

(1) if the limit is 0

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

more precisely: $f(n) = o(g(n))$

(2) if the limit is $C \neq 0$

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

more precisely $f(n) = \Theta(g(n))$

(3) if the limit is ∞

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

more precisely: $f(n) = \omega(g(n))$

$T(n)$ is $O(f(n))$:

there are constants c and n' (threshold) s.t.

$$T(n) \leq c \cdot f(n), \text{ for all } n \geq n'$$

intuitive meaning

$T(n)$ "$\leq$" $f(n)$

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

$\geq$ $\rightarrow$ $T(n)$ "$\geq$" $f(n)$

$T(n)$ is $\Theta(f(n))$

$\rightarrow$ $T(n)$ "$=$" $f(n)$.

$T(n)$ is $O(f(n))$ ⟶ $T(n)$ '$<$' $f(n)$

$$\frac{T(n)}{f(n)} \xrightarrow[n \to \infty]{} 0$$

$T(n)$ is $\omega(f(n))$ ⟶ $T(n)$ '$>$' $f(n)$.

$$\frac{T(n)}{f(n)} \xrightarrow[n \to \infty]{} \infty$$

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

- Eliminate low order terms
- Eliminate constant coefficients

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

General Rules for Computing Time Complexity

(1) O(1) if the execution time does not depend on the input data

(2) Apply Addition if the code segments are placed consecutively

(3) Apply Multiplication if the code segments are nested

(4) O(log n) if the number of executions is reduced in half repeatedly

(5) O(2^n) if the number of executions is doubled repeatedly

(6) Always consider the worst case

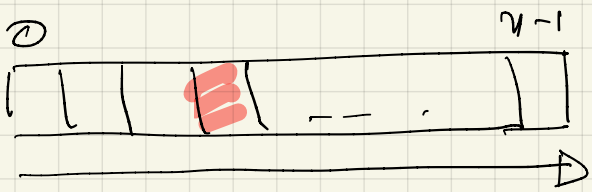
COMMON CODE FRAGMENTS and RUNTIMES

Ex. 1 $a = b + c \longrightarrow \Theta(1)$

Ex. 2

```
int[] a = new int[n]; sum = 0;
for (int i=0 ; i<n ; i++)
    sum = sum + a[i]
```

$\Theta(n)$

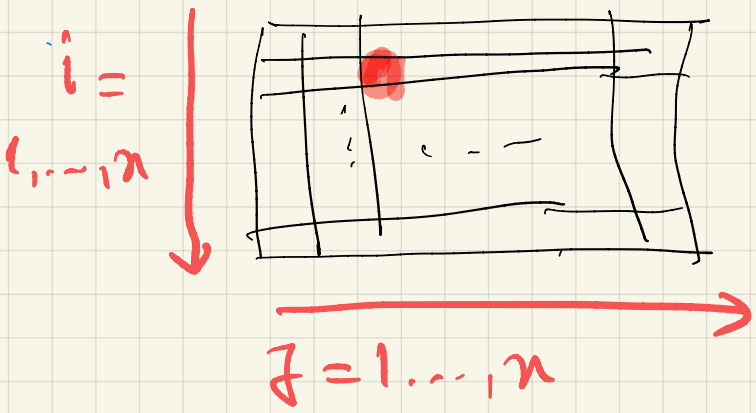


Ex. 3

```
int [][] a = new int[n][n]; sum = 0;
for (int i=1 ; i≤n ; i++)
    for (int j=1 ; j≤n ; j++)
        sum = sum + a[i][j]
```

$n \cdot (cn) = c \cdot n^2 = \Theta(n^2)$

$c \cdot n$



Ex. 4 `int [][] a = new int[n][n]; sum=0;`

```
for ( int i=1; i≤n; i++ )
    for( int j=i+1; j≤n; j++)
        sum = sum + a[i][j]
```

→ $C(n-i)$

$$\sum_{i=1}^{n} C\cdot(n-i) = C\cdot((n-1)+\cdots+1+0) = C\frac{(n-1)\cdot n}{2} = \Theta(n^2).$$

Ex. 5 `int [][][] a = new int [n][n][n]; sum=0;`

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

n (innermost) · n^2 (outer two) · n^3 (all three)

$\Theta(n^3)$

Ex. 6

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

RUNTIME : $\Theta(n^3)$.

inner loop: $k = 0, 1, 2, \ldots, j-1$

no. of iterations : j , const no of oper per iter

time: $\Theta(j)$

middle loop : $j = 0, 1, 2, \ldots, i-1$

TIME: $\sum_{j=0}^{i-1} c \cdot j = c \cdot (0+1+\cdots+(i-1))$

$= c \cdot \frac{i(i-1)}{2} = \Theta(i^2)$

outer loop: $i = 0, 1, \ldots, n-1$

TIME: $\sum c' \cdot i^2 = c' (0^2+1^2+\cdots+(n-1)^2)$

$= \frac{(n-1) \cdot n (2n-1)}{6} = \Theta(n^3)$.

Ex. 7

```
i = n;
while (i > 1)  i = i/2;
```

$O(\log n)$ → actually $\Theta(\log n)$.

$n = 32 \longrightarrow 16 \longrightarrow 8 \longrightarrow 4 \longrightarrow 2 \longrightarrow 1$

$= 2^5$ (1, 2, 3, 4, 5)

$5 = \log_2 32$

If n is a power of 2

$n = 2^k \longrightarrow 2^{k-1} \longrightarrow 2^{k-2} \longrightarrow \dots \longrightarrow 2^0 =$ **STOP**

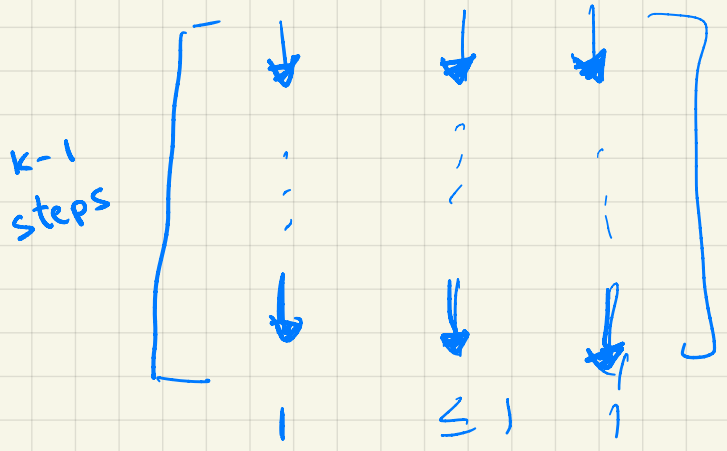
no of divisions : k

$k = \log_2 n.$

If n is not a power of 2

Then $2^{k-1} < n < 2^k \longrightarrow k-1 < \log n < k$

So $k = \lceil \log n \rceil \simeq \log n$



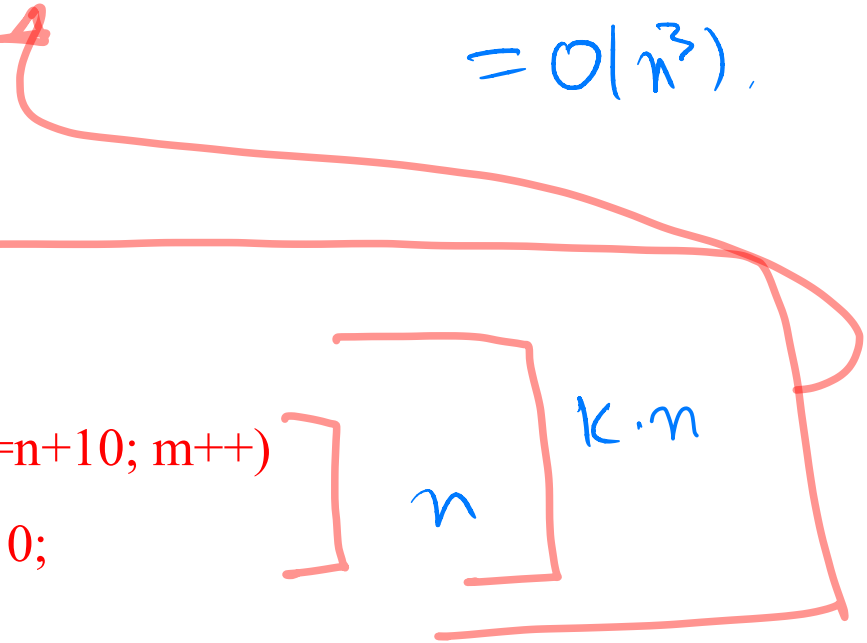
So: $k-1 \le$ no. of steps $\le k$

$\Rightarrow$ no. of steps $= \Theta(k) = \Theta(\log n)$

Big-O examples

$\sum_{k=1}^{n} k \cdot n = n(1+2+3+\dots+n) = n \cdot \frac{n(n+1)}{2}$

$= O(n^3)$.



(1)
```
cin >> n;
for (k=1; k<=n; k++)
        for(j=1; j<=k; j++)
                for(m=1; m<=n+10; m++)
                        a = 10;
```

Time complexity is $O(n^3)$

(2)
```
cin >> n
for(j=0; j  <  myfunc(n); j++)
        cout << j
        for(k=1; k<= n; k++)
                cout << j << k;
```

Time complexity is: (a) $O(n^2)$ if myfunc(n) returns n

(b) O(nlogn) if myfunc(n) returns log n

Big-O examples(2)

(3)

```
cin >> n; total = 0
for(j=1; j <n; j++)
        cin >> x
        total = x + total
        if((total %2) == 1)
                cout << total
        else
                for(k=1; k<=j; k++)
                cout << k;
```

Time complexity: $O(n^2)$

Big-O examples(3)

(4) float f(float n)

```
{ sum =0;
  for(j=1; j<n; j++)
        sum = sum +j;
  return sum;}
```

Time complexity: O(n)

(5) float g(float n)

```
{ sum = 0
  for(j=0; j<n; j++)  sum = sum +f(j)
}
```

Time complexity: $O(n^2)$

Big-O examples(4)

(6)

```
float h(int n)
    {for (j=0; j<n; j++)
          cout << g(n) + f(n); }
```

Time complexity: $O(n^3)$

(7)

```
        int j,n,k;  // Now what?
        cin >> n;   power =1;
         for(j=1; j< n; j++)
                    power = power * 2
                    for(k=1; k< power; k++)
                              cout << k
```

$2+4+\cdots+2^{n-1} = 2^n - 2 = O(2^n)$

power: $2, 4, 8, 16, \ldots, 2^{n-1}$

power

Time complexity: $O(2^n)$

Big-O examples(5)

(8)

```
cin >> n;
x =1
while (x < n)
        x = 2*x
```

Time Complexity: O(log n)