PROGRAM COMPLEXITY

MSCI 240: Algorithms & Data Structures

lecture summary

introduction to program complexity

- linear search
- best, worst, average case

analytical approach

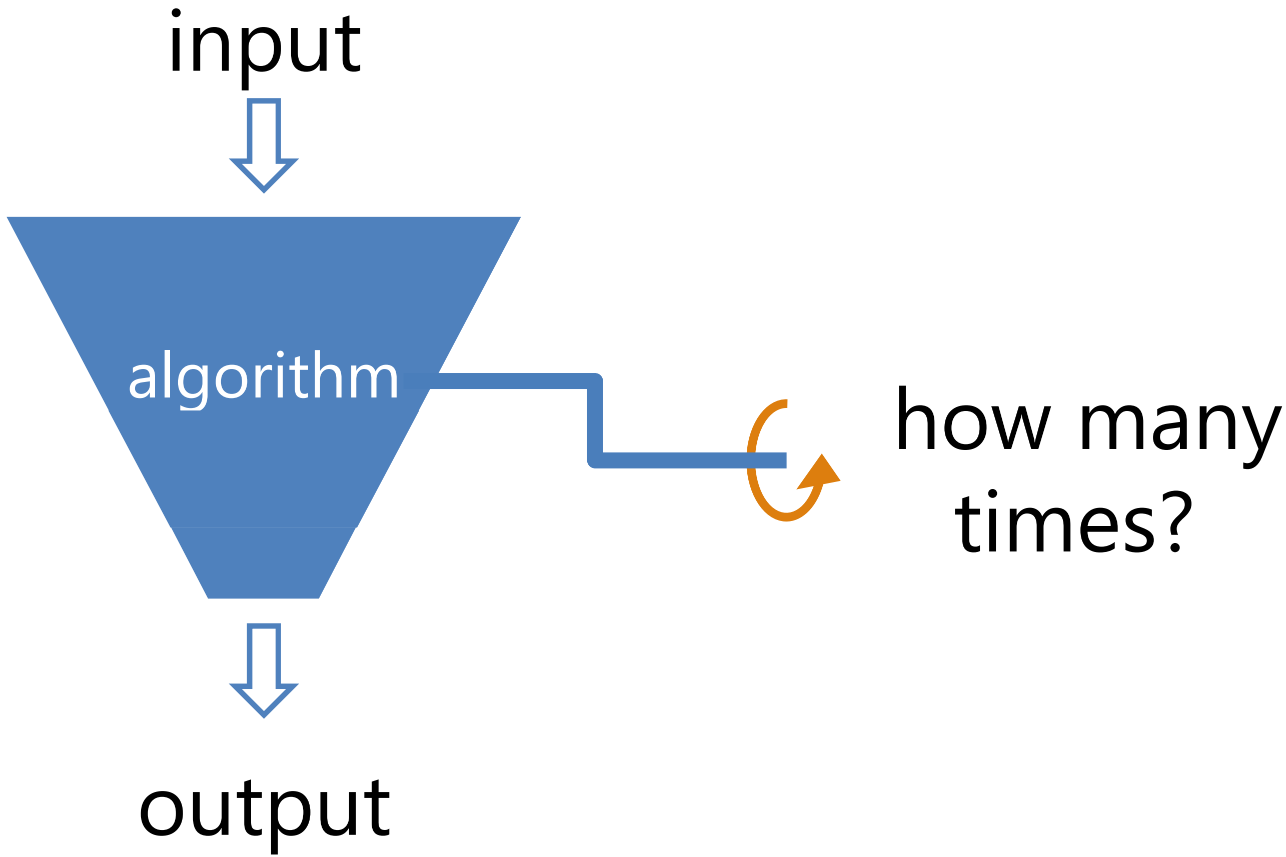
- calculating running time
- order of growth

program complexity

Topic	Building Java Programs	Algorithms (Sedgewick)
classes, ADTs	chapter 8	1.2
arrays	chapter 7	
ArrayList<T>	chapter 10	1.3
Stack/Queue	chapter 14, (11)	1.3
LinkedList	chapter 16	1.3
Complexity		1.4
Searching	chapter 13	pp. 46-47
Sorting		chapter 2.1-2.3
Recursion	chapter 12	1.1 (p. 25)
BSTs	chapter 17	chapter 3.1-3.2
Dictionaries	chapter 18.1	chapter 3.4
Graphs	N/A (Wikipedia good)	chapter 4.1
Heaps/Priority Queues	chapter 18.2	chapter 2.4

fill in the blank:

how much _________ will this program take to run?


input
algorithm
how many
times?
output

example: linear (sequential) search

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
value	-4	2	7	10	15	20	22	25	30	36	42	50	56	68	85	92	103

i

input:

n numbers, $A = < a_1, a_2, \ldots, a_n >$

a value, v (e.g., 42 above)

output:

index i, such that $v = a_i$ (e.g., 10 above)

or, nil if $v \notin A$

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
value	-4	2	7	10	15	20	22	25	30	36	42	50	56	68	85	92	103

`i`

exercise: write pseudocode for linear search

exercise: write Java code for linear search

how many steps does it take to complete?

(a.k.a. how many elements does it need to examine?)

what does the number of steps depend on?

best case?

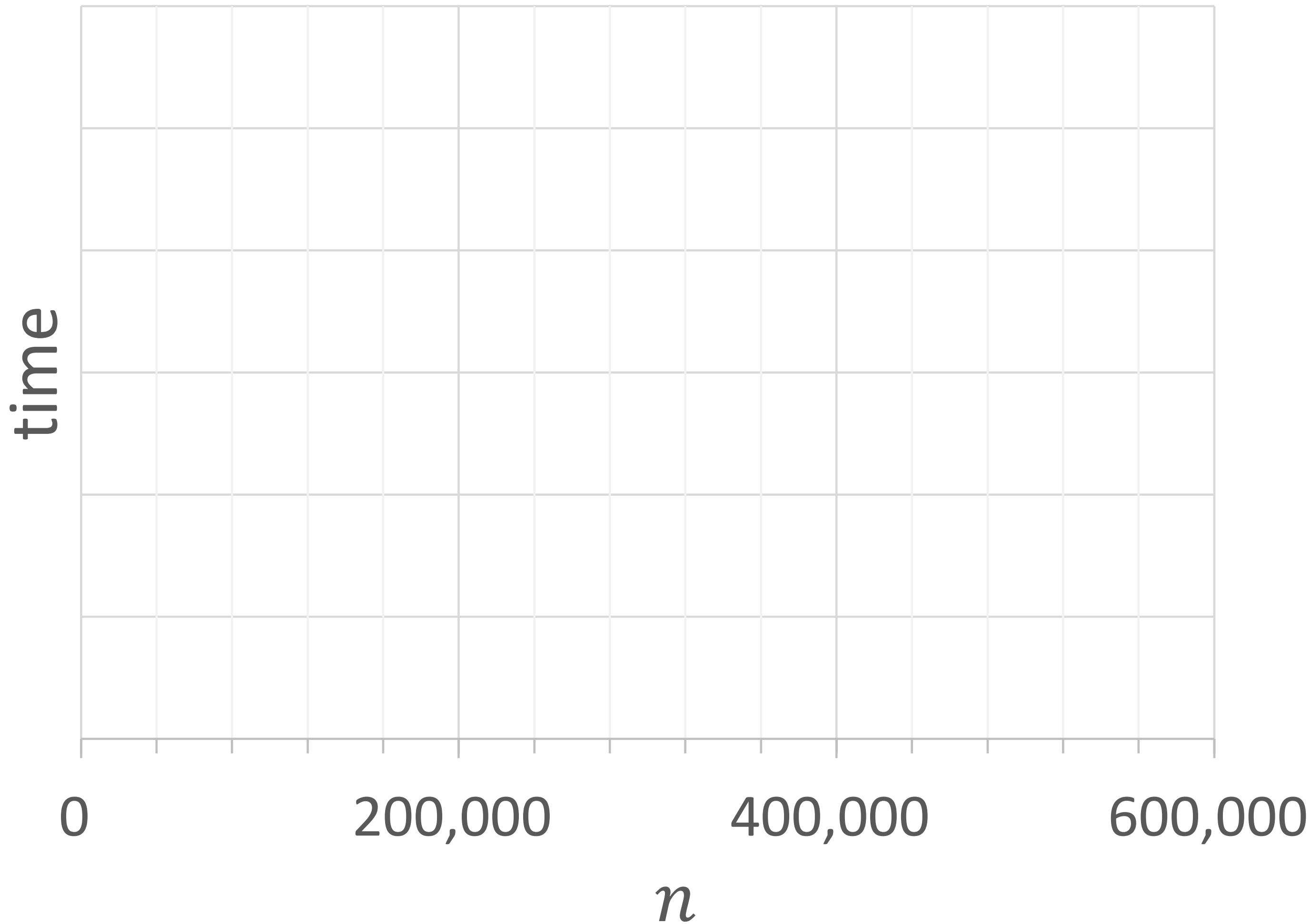
first element (1 step)

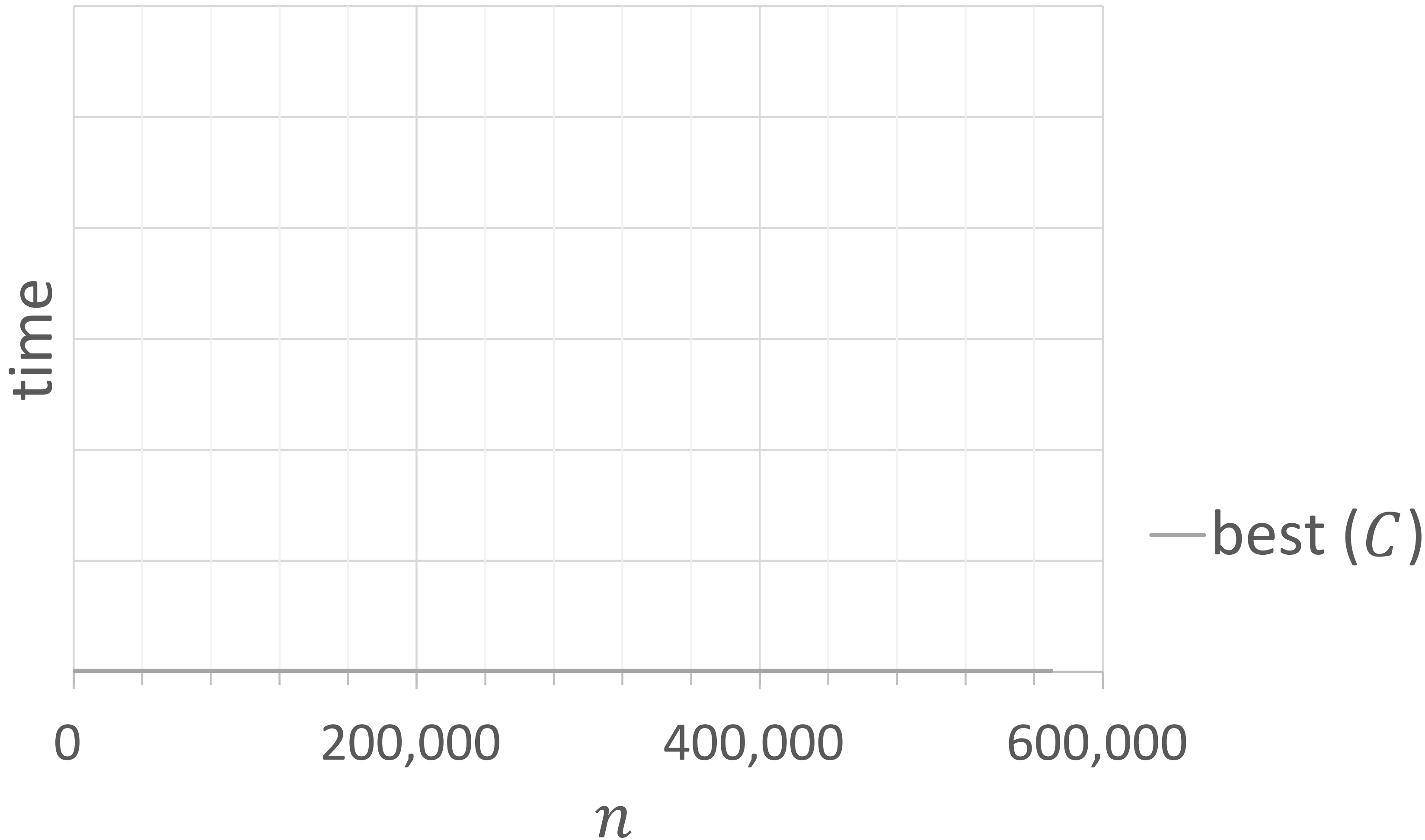
worst case?

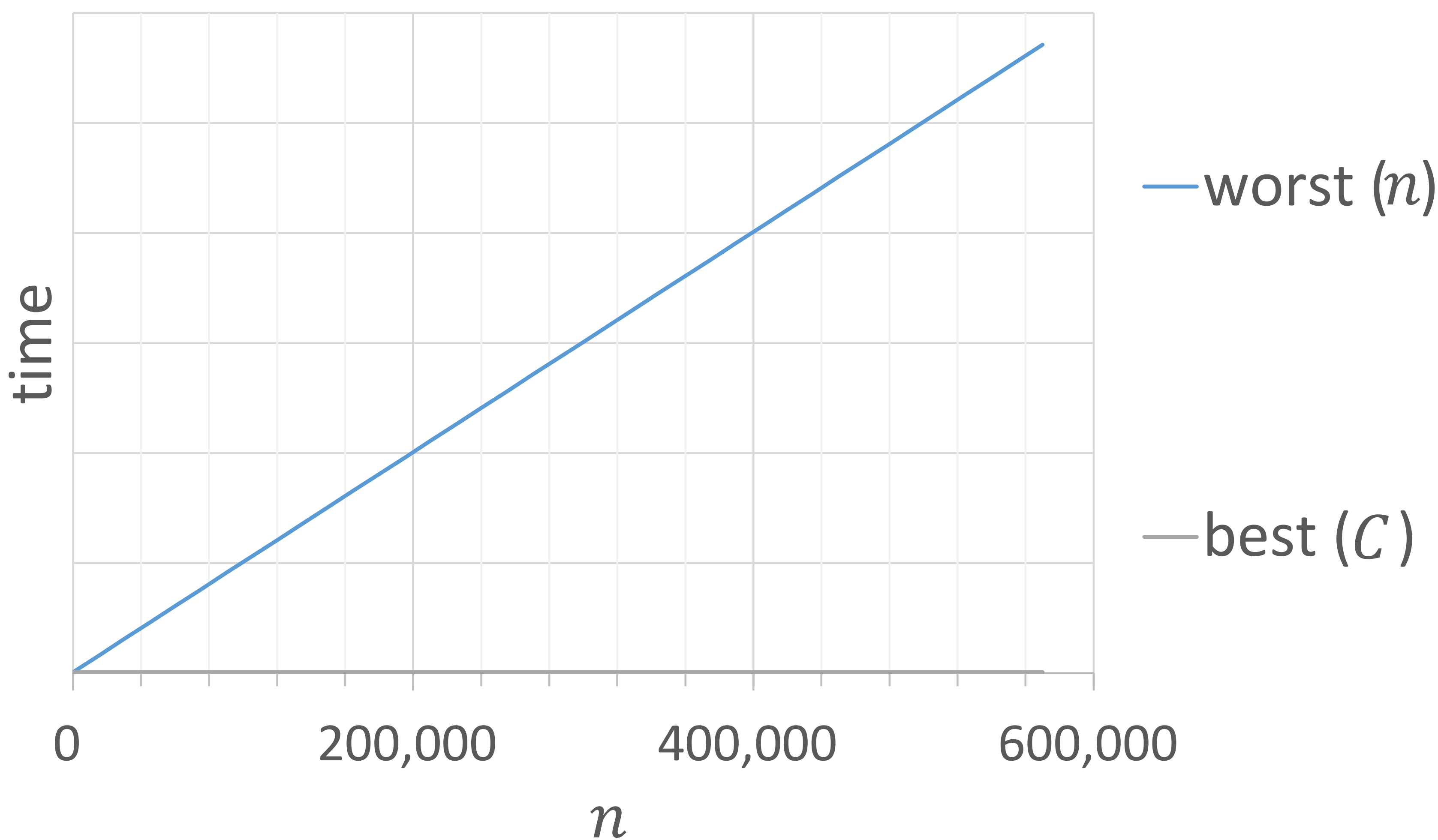
not found (n steps)

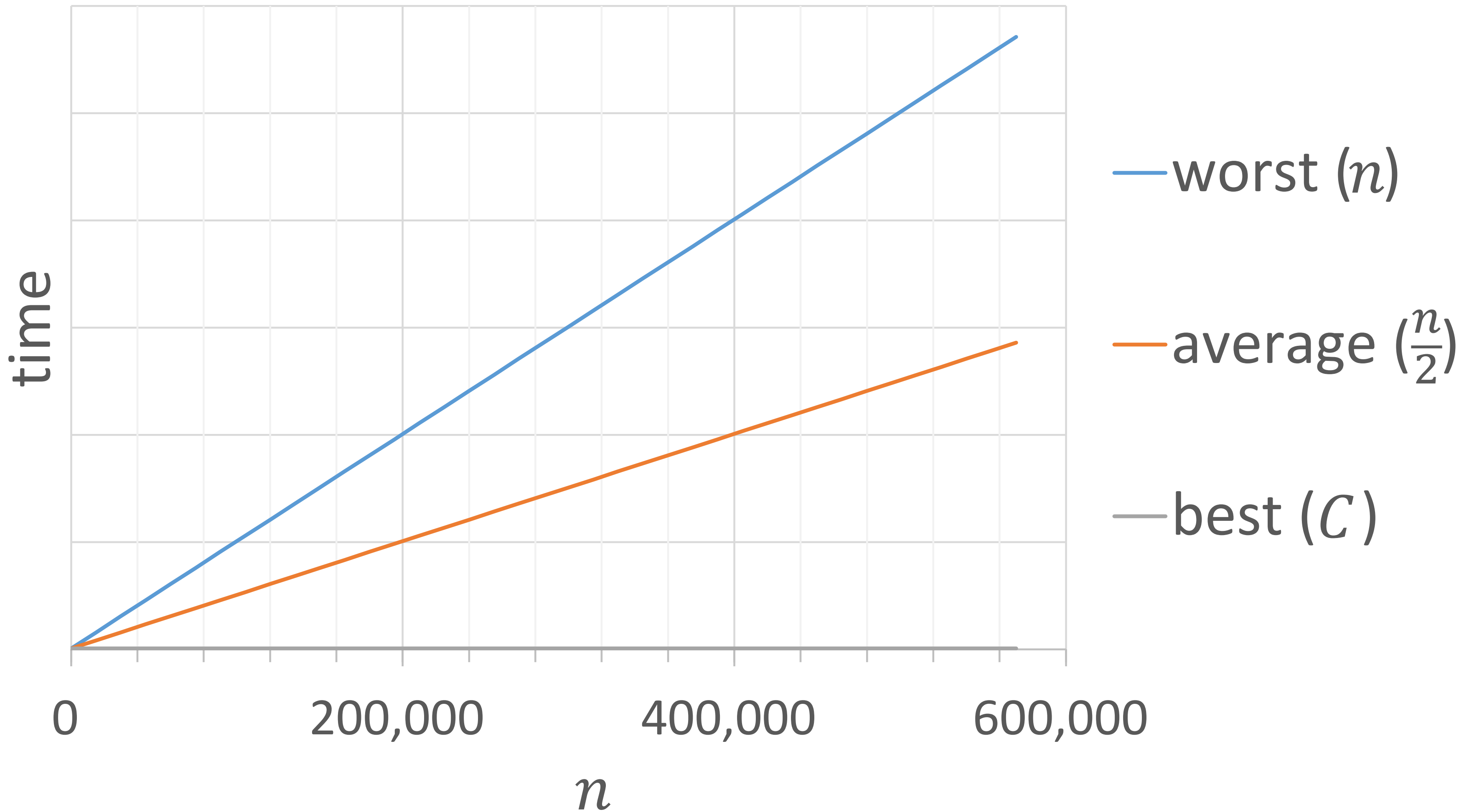
average case?

half way through the list ($\frac{n}{2}$ steps)


time
0
200,000
400,000
600,000
n


time
best (C)
0
200,000
400,000
600,000
n


time
worst (n)
best (C)
0
200,000
400,000
600,000
n


time
0
200,000
400,000
600,000
n
worst (n)
average ($\frac{n}{2}$)
best (C)

analytical approach

runtime efficiency (13.2)

efficiency: a measure of the use of computing resources by code

- can be relative to speed (time), memory (space), etc.
- most commonly refers to run time

assume the following:

- any single Java statement takes the same amount of time to run
- a method call's runtime is measured by the total of the statements inside the method's body
- a loop's runtime, if the loop repeats n times, is n times the runtime of the statements in its body

RAM model

assume all basic operations have equal cost

assume data locality has no (or negligible) influence

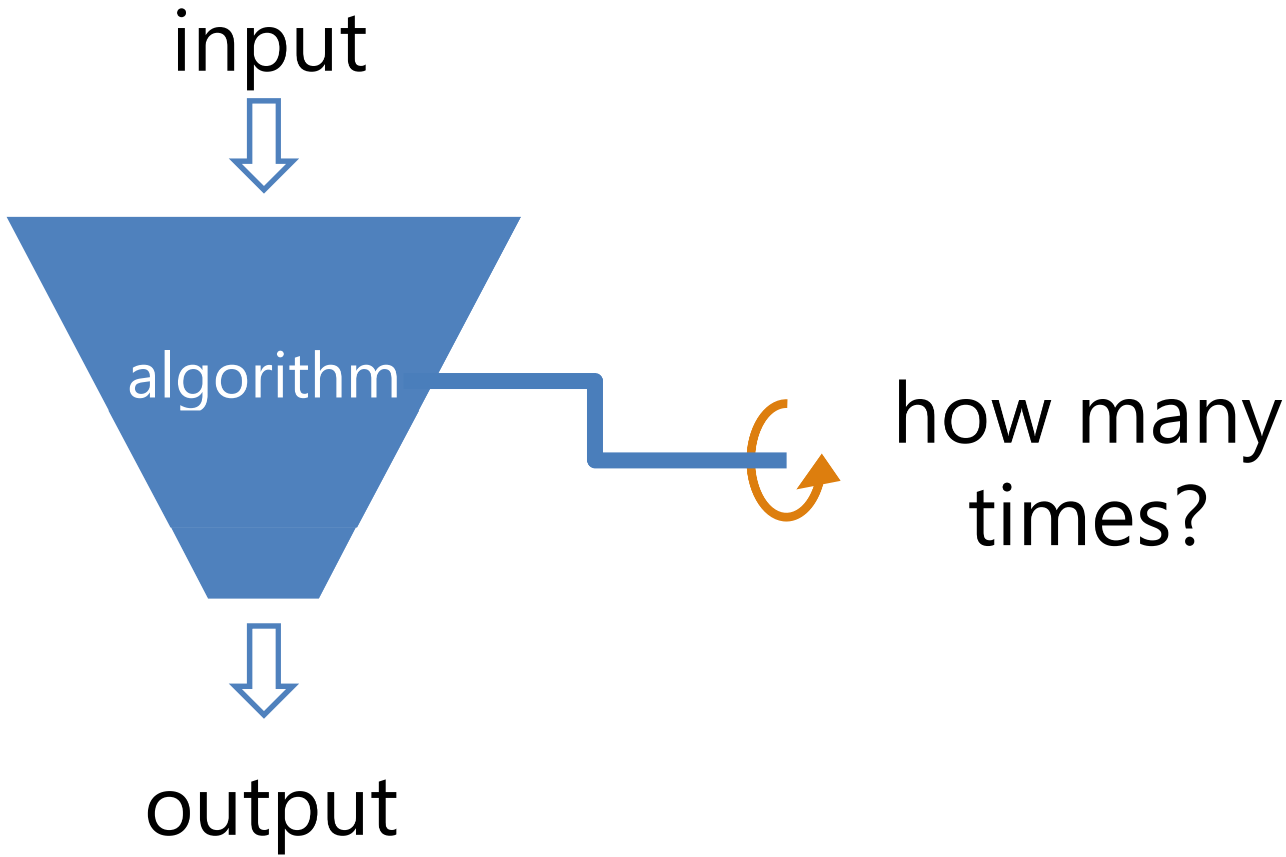
calculating running time

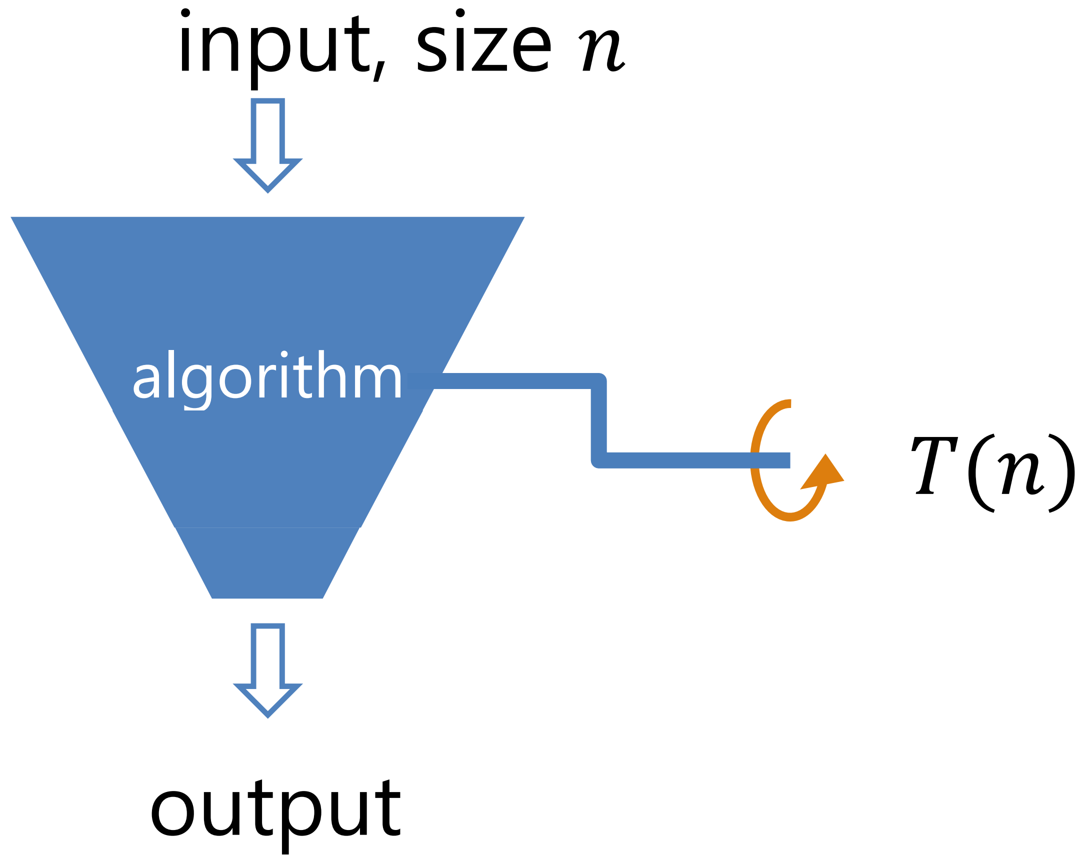
determine input size, n

assign cost to each line of code/algorithm

sum up all costs to produce function, $T(n)$

$T(n)$ is the time to execute for input size n


input
algorithm
how many
times?
output



input, size n
algorithm
$T(n)$
output

efficiency examples

```
statement1;
statement2;
statement3;
```

3

```
for (int i = 1; i <= n; i++) {
    statement4;
}
```

n

```
for (int i = 1; i <= n; i++) {
    statement5;
    statement6;
    statement7;
}
```

$3n$

$$T(n) = 4n + 3$$

efficiency examples 2

```
for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= n; j++) {
        statement1;
    }
}

for (int i = 1; i <= n; i++) {
    statement2;
    statement3;
    statement4;
    statement5;
}
```

n

$n \cdot n = n^2$

$4n$

$T(n) = n^2 + 4n$

how many statements will execute if $n = 10$? if $n = 1000$?

order of growth

example: $T(n) = \frac{2}{3}n^2 + 25n + 10$

order of growth is the change in runtime as n changes

consider runtime when n is extremely large

which term has the greatest influence on the value of $T(n)$ for extremely large n?

Sedgwick tilde (~) notation (from *Algorithms* textbook):

$g(n) \sim f(n)$ indicates that $\frac{f(n)}{g(n)}$ approaches 1 as n grows

$\sim f(n)$ represents any function $g(n)$, s.t. $g(n) \sim f(n)$

example: $T(n) = \frac{2}{3}n^2 + 6n + 10$

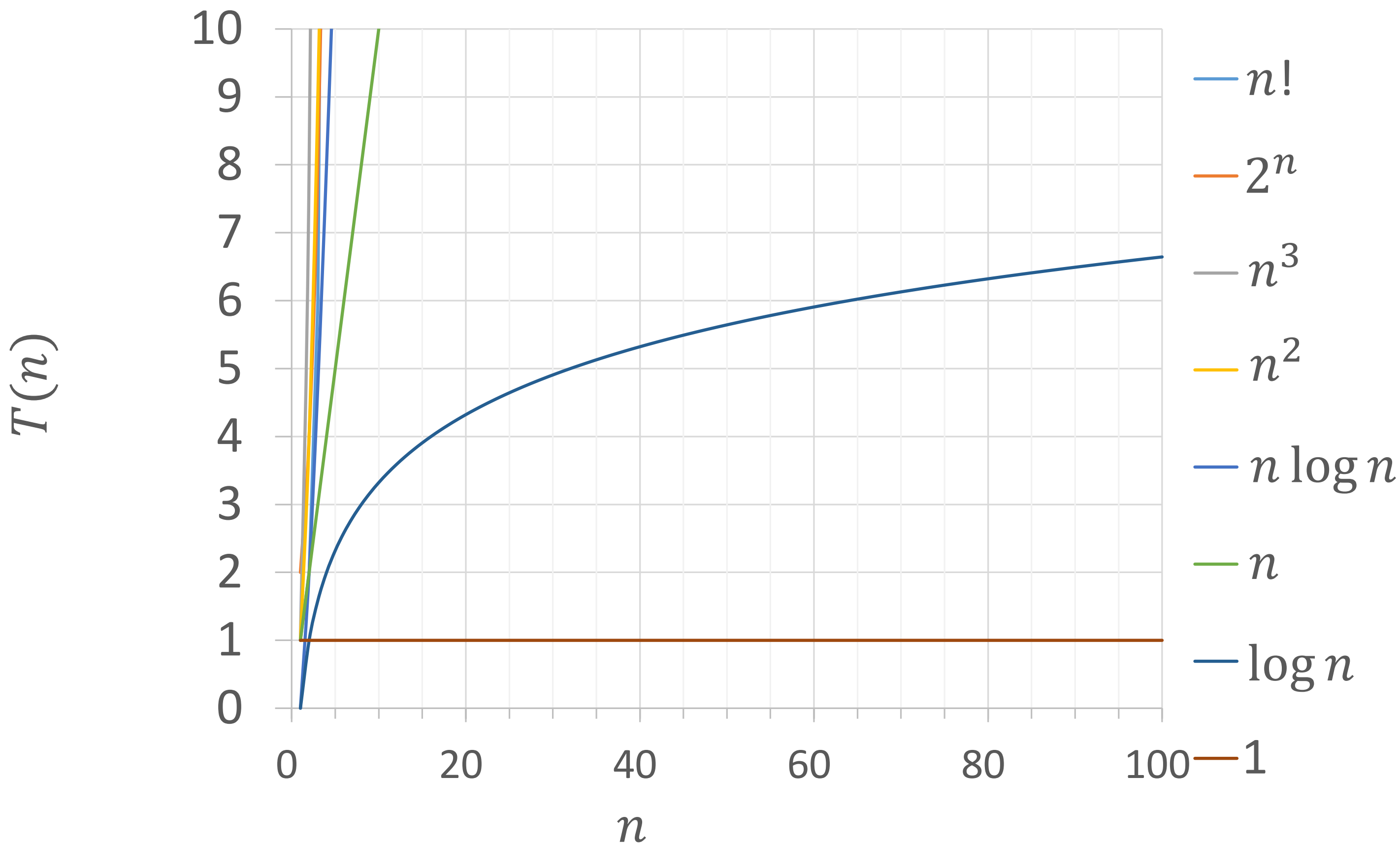
$$\therefore T(n) \sim \tfrac{2}{3}n^2$$

example: $T(n) = \log(n) + 1$

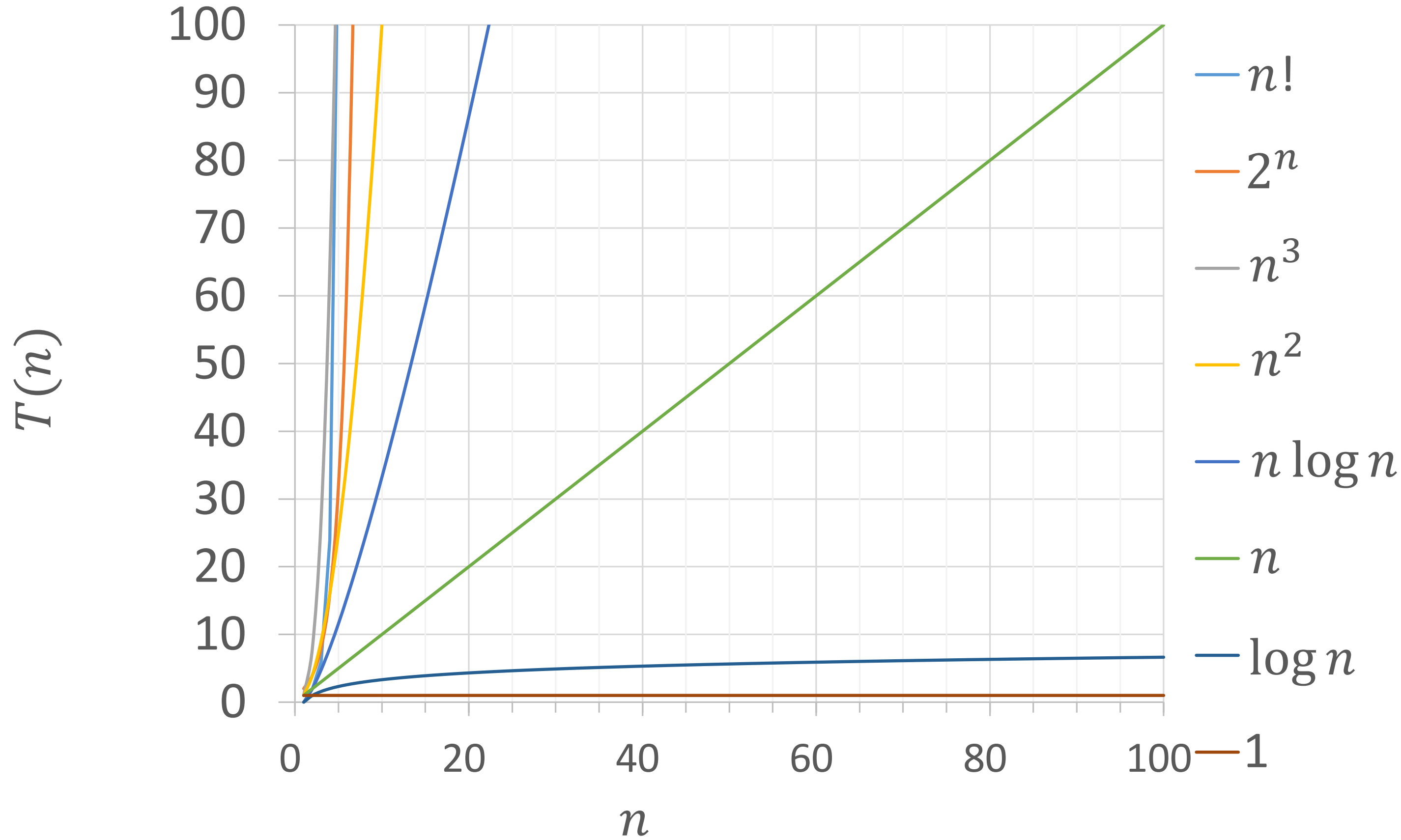
$$\therefore T(n) \sim \log(n)$$

common orders of growth

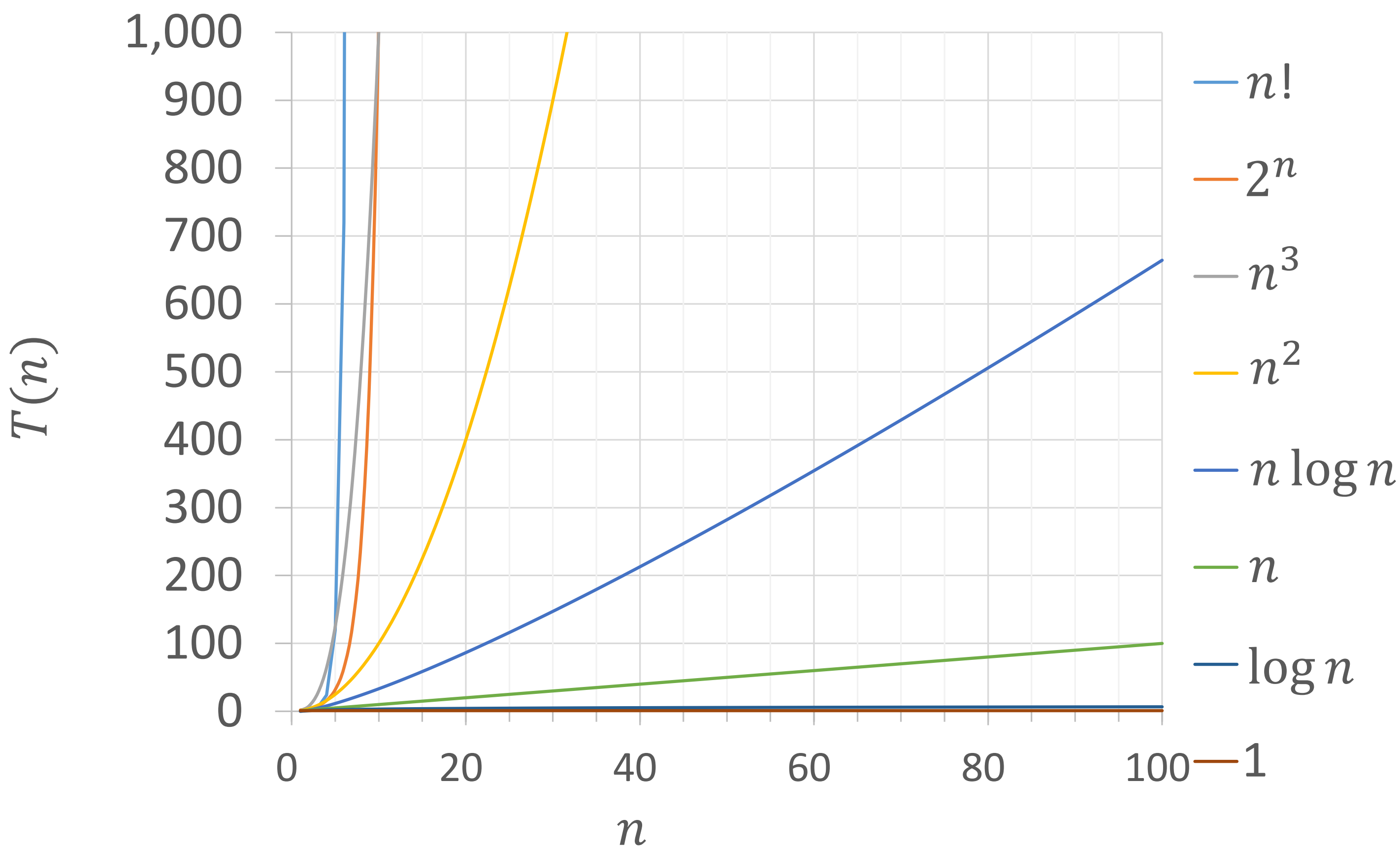
description	tilde approximation	order of growth
constant	$\sim K$	1
logarithmic	$\sim K \cdot \log(n)$	$\log(n)$
linear	$\sim K \cdot n$	n
linearithmic	$\sim K \cdot n \cdot \log(n)$	$n \cdot \log(n)$
quadratic	$\sim K \cdot n^2$	n^2
cubic	$\sim K \cdot n^3$	n^3
exponential	$\sim K \cdot 2^n$	2^n
factorial	$\sim K \cdot n!$	$n!$



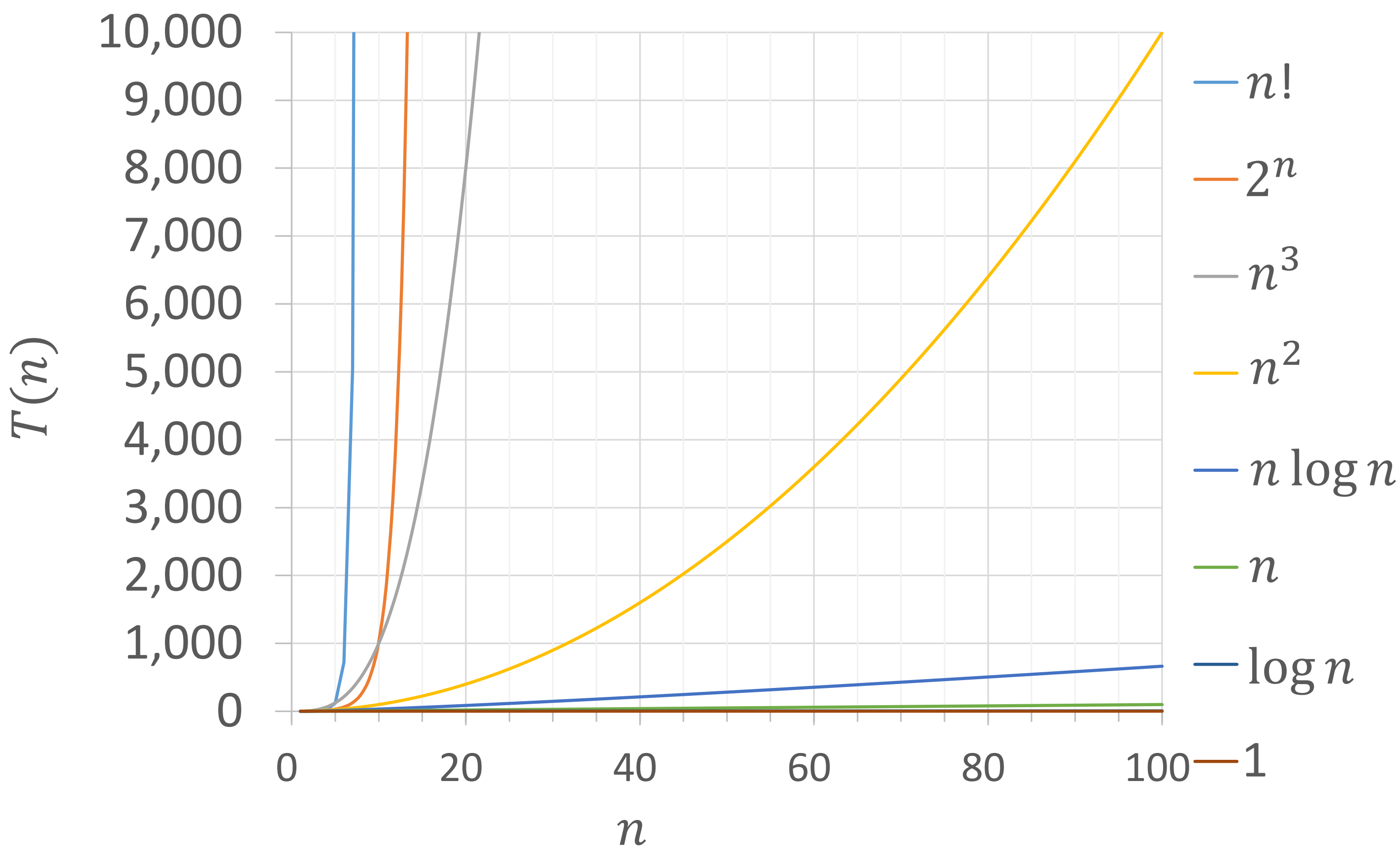
10
9
8
7
6
5
4
3
2
1
0
0
20
40
60
80
100
$T(n)$
n
$n!$
2^n
n^3
n^2
$n \log n$
n
$\log n$
1



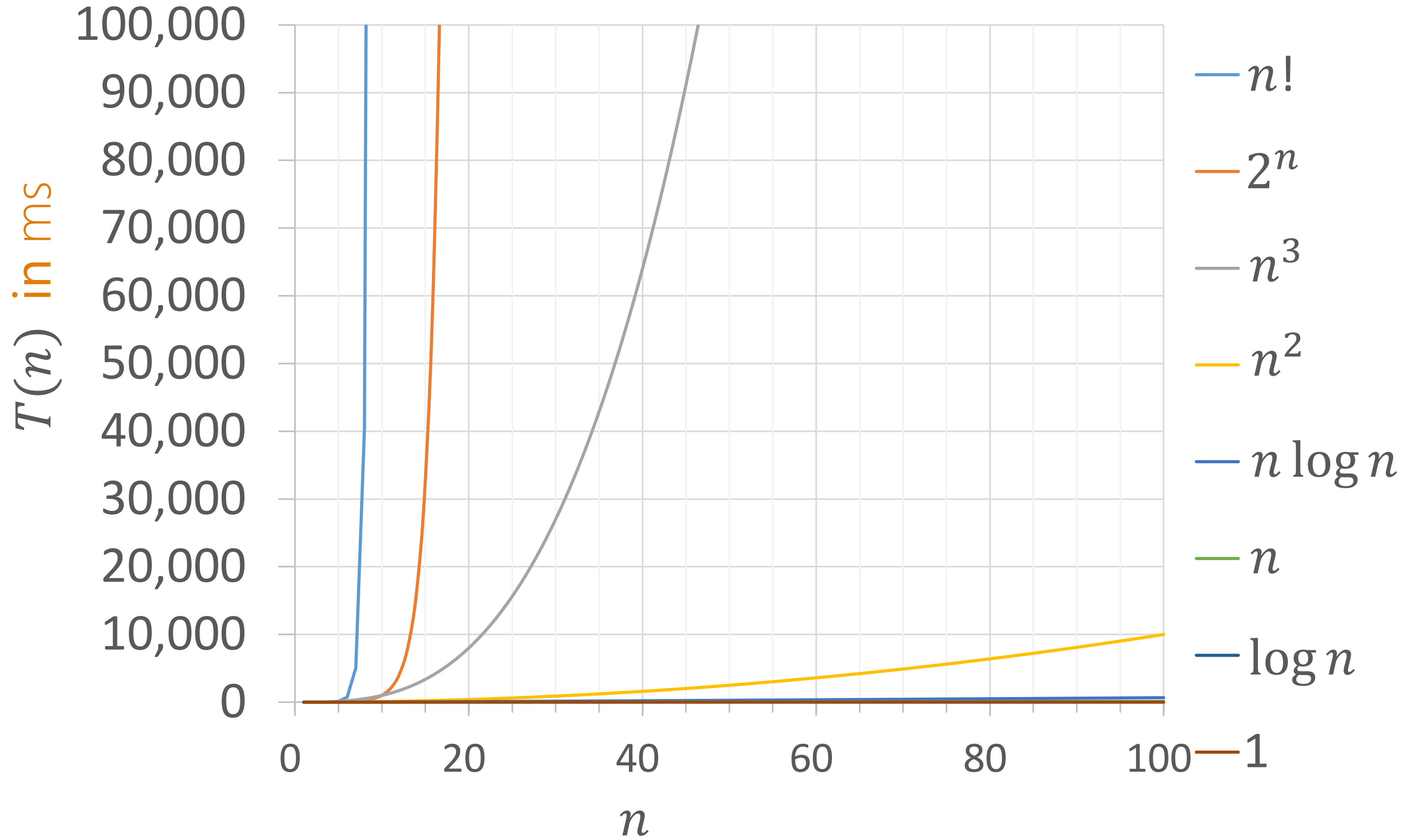
100
90
80
70
60
50
40
30
20
10
0
0
20
40
60
80
100
$T(n)$
n
$n!$
2^n
n^3
n^2
$n \log n$
n
$\log n$
1



1,000
900
800
700
600
500
400
300
200
100
0
0
20
40
60
80
100
$T(n)$
n
$n!$
2^n
n^3
n^2
$n \log n$
n
$\log n$
1

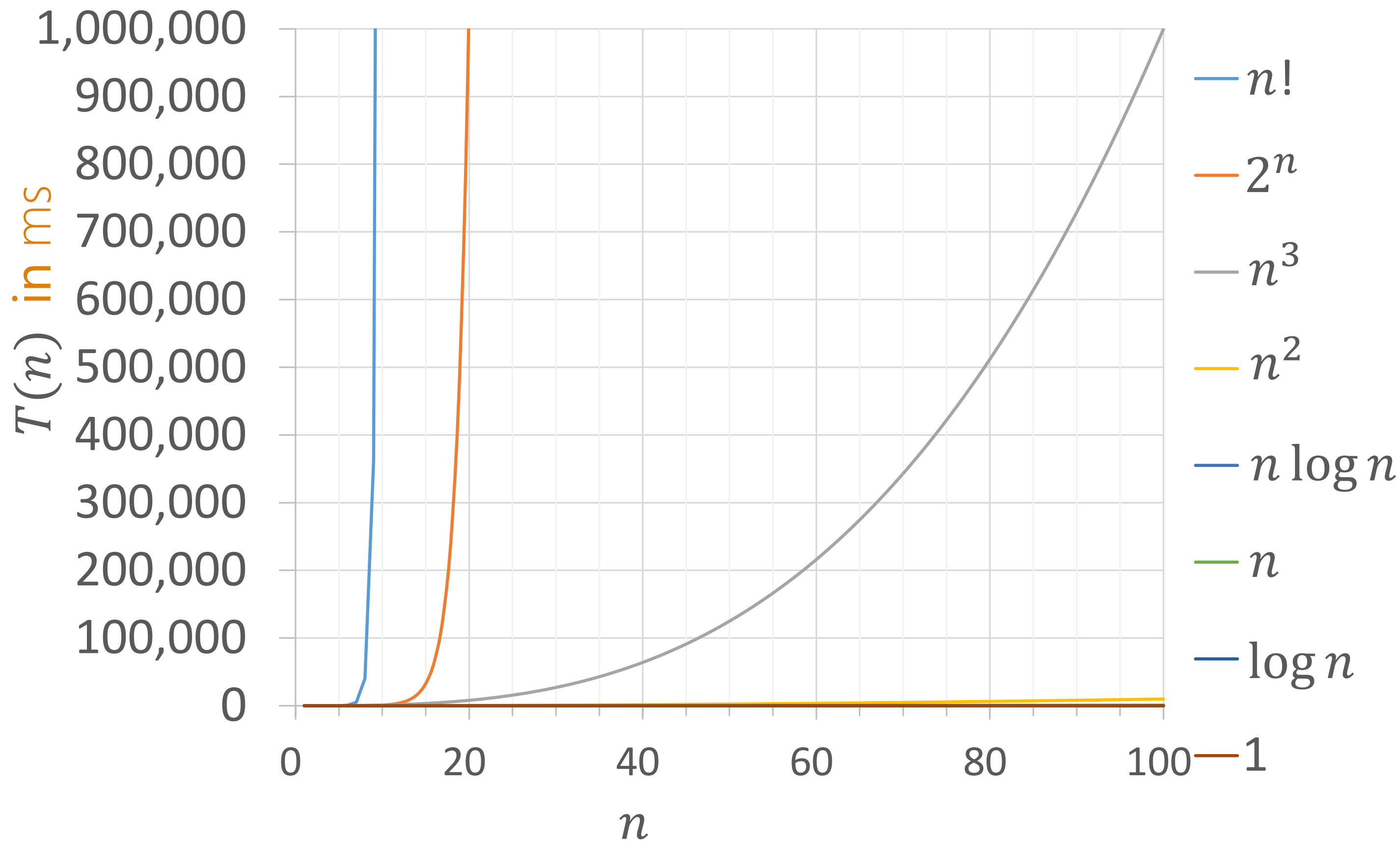

10,000
9,000
8,000
7,000
6,000
5,000
4,000
3,000
2,000
1,000
0
$T(n)$
0
20
40
60
80
100
n
$n!$
2^n
n^3
n^2
$n \log n$
n
$\log n$
1

~1.7 minutes

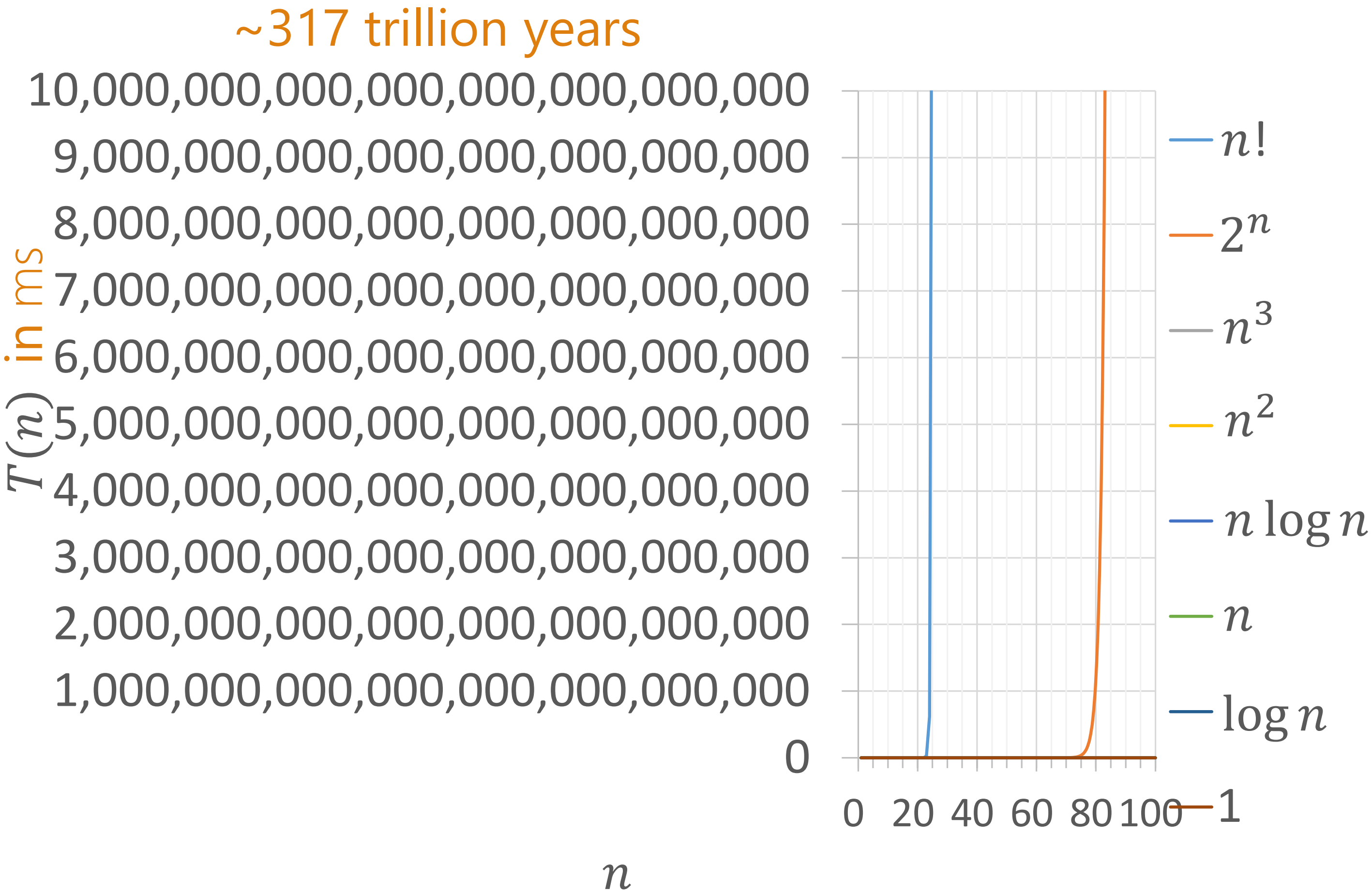


$T(n)$ in ms
100,000
90,000
80,000
70,000
60,000
50,000
40,000
30,000
20,000
10,000
0
0
20
40
60
80
100
n
$n!$
2^n
n^3
n^2
$n \log n$
n
$\log n$
1

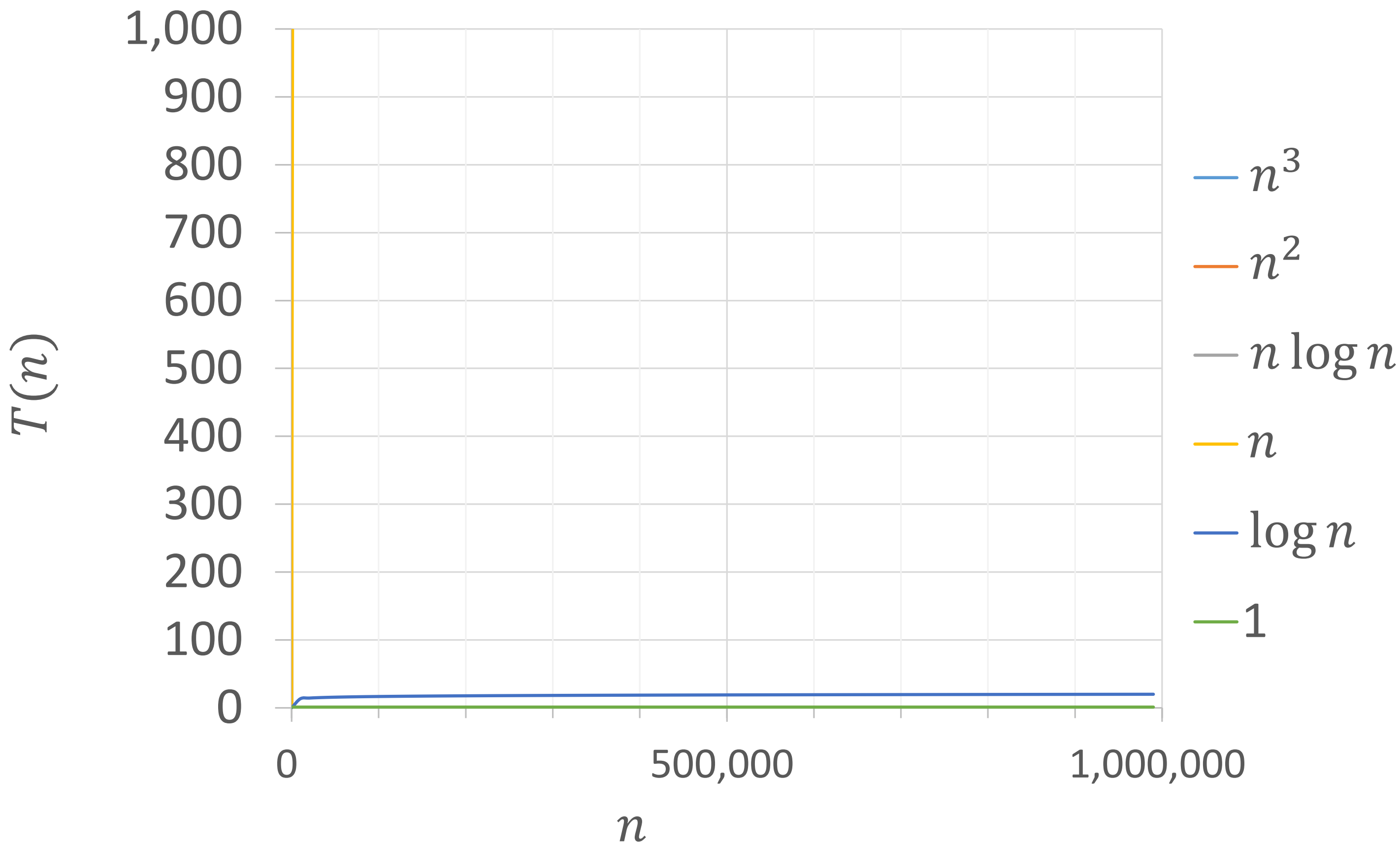
~17 minutes

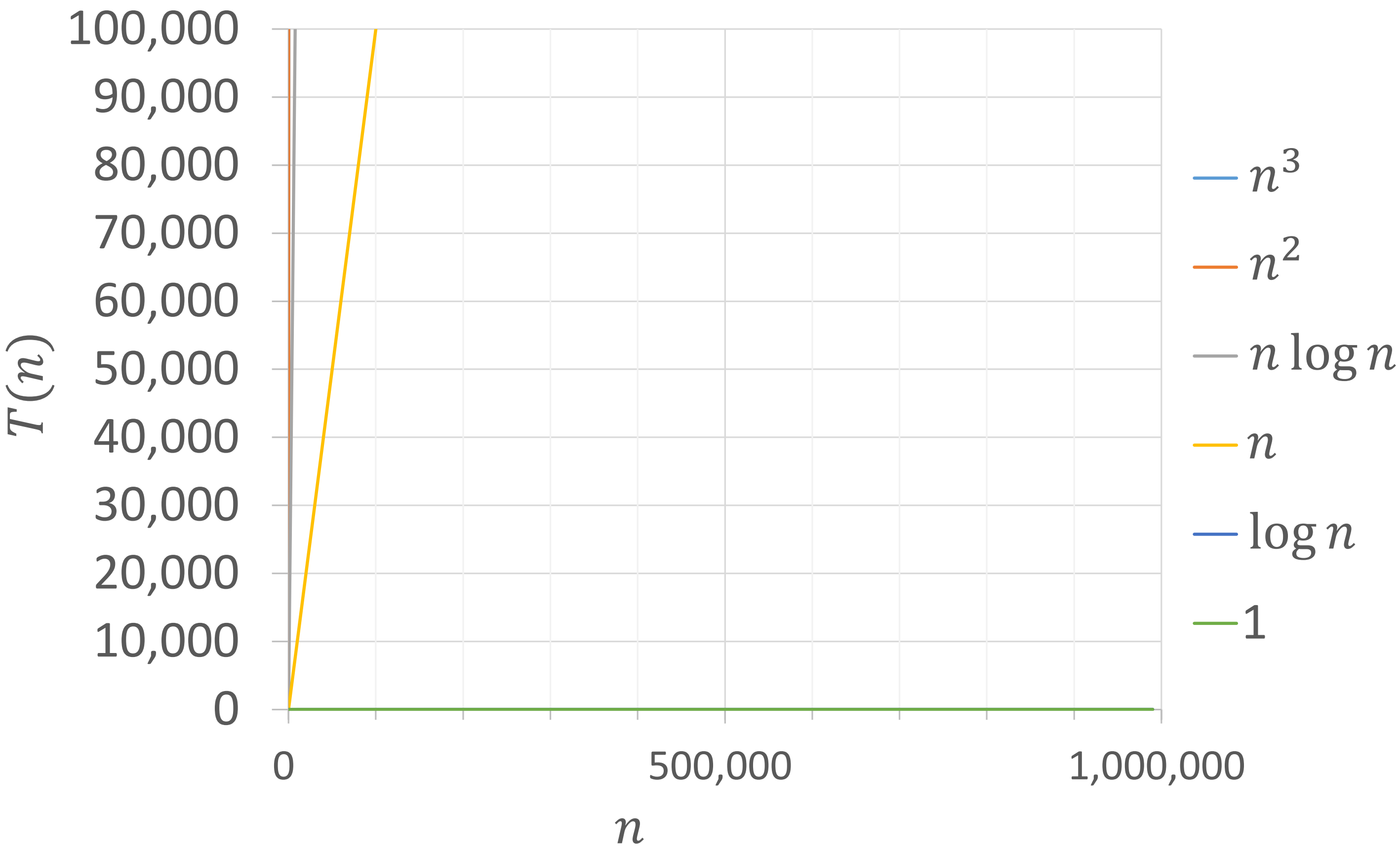


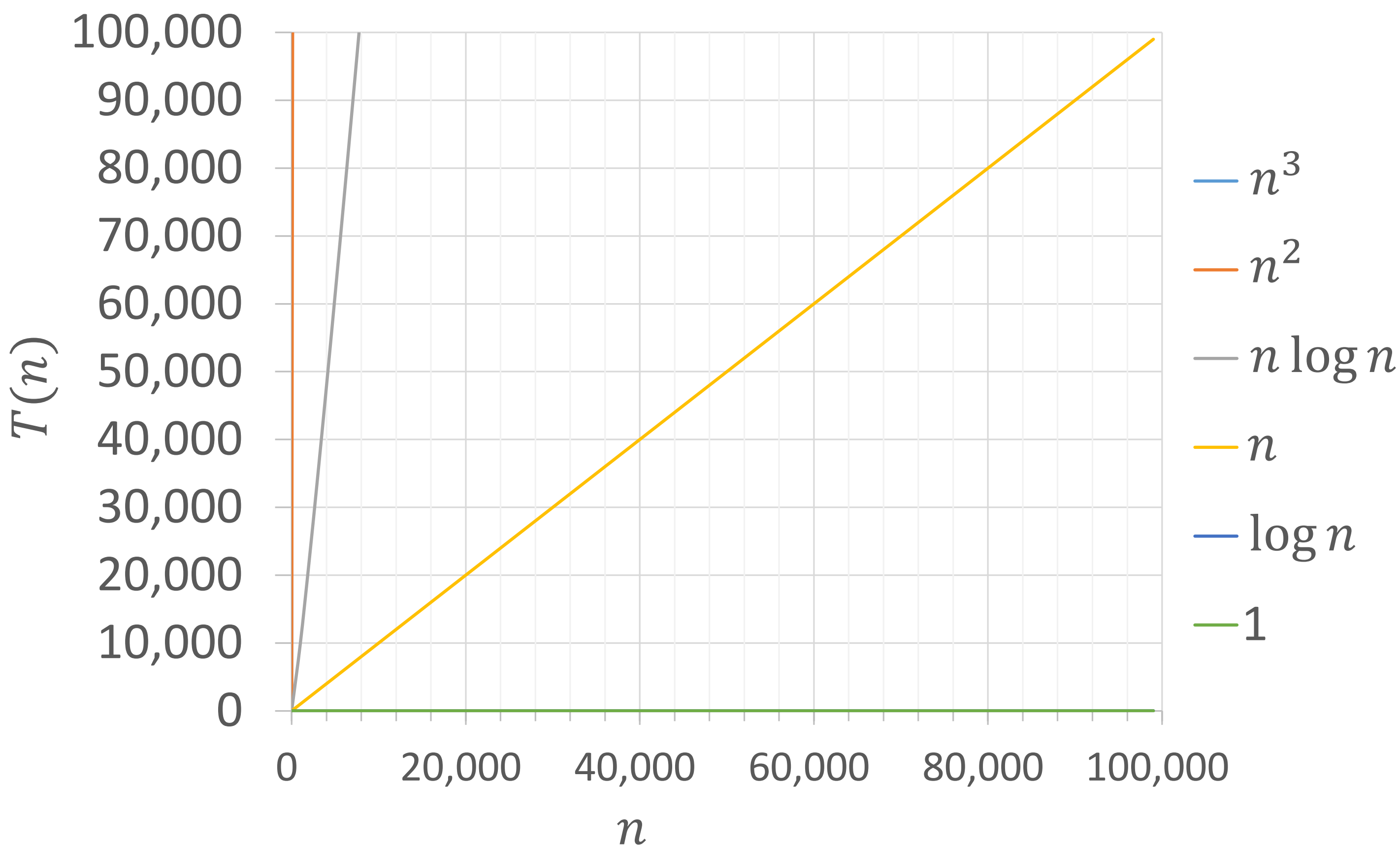
$T(n)$ in ms
1,000,000
900,000
800,000
700,000
600,000
500,000
400,000
300,000
200,000
100,000
0
0
20
40
60
80
100
n
$n!$
2^n
n^3
n^2
$n \log n$
n
$\log n$
1


~317 trillion years
10,000,000,000,000,000,000,000,000
9,000,000,000,000,000,000,000,000
8,000,000,000,000,000,000,000,000
7,000,000,000,000,000,000,000,000
6,000,000,000,000,000,000,000,000
5,000,000,000,000,000,000,000,000
4,000,000,000,000,000,000,000,000
3,000,000,000,000,000,000,000,000
2,000,000,000,000,000,000,000,000
1,000,000,000,000,000,000,000,000
0
$T(n)$ in ms
0 20 40 60 80 100
n
$n!$
2^n
n^3
n^2
$n \log n$
n
$\log n$
1

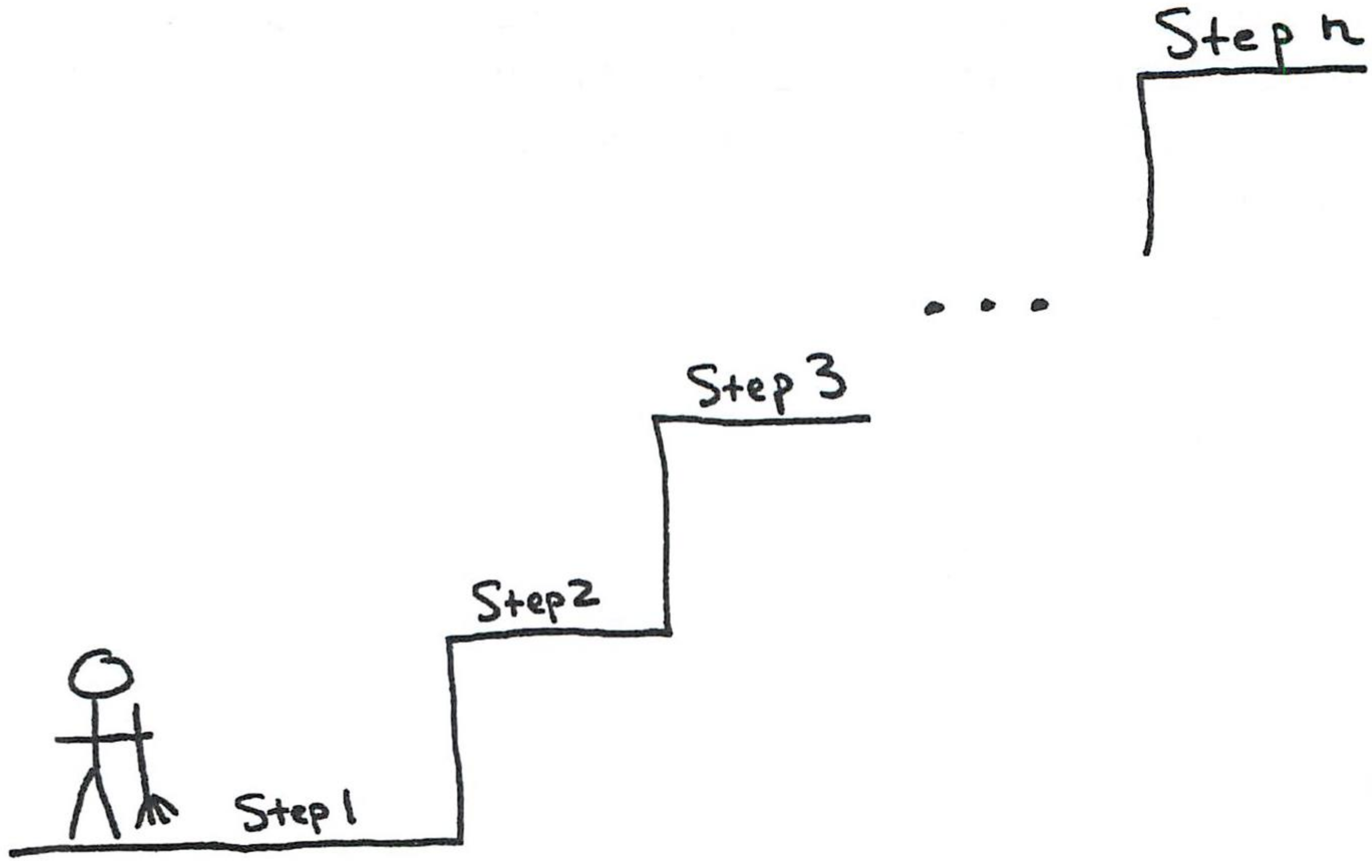
alternatively…


1,000
900
800
700
600
500
400
300
200
100
0
$T(n)$
0
500,000
1,000,000
n
n^3
n^2
$n \log n$
n
$\log n$
1


100,000
90,000
80,000
70,000
60,000
50,000
40,000
30,000
20,000
10,000
0
$T(n)$
0
500,000
1,000,000
n
n^3
n^2
$n \log n$
n
$\log n$
1

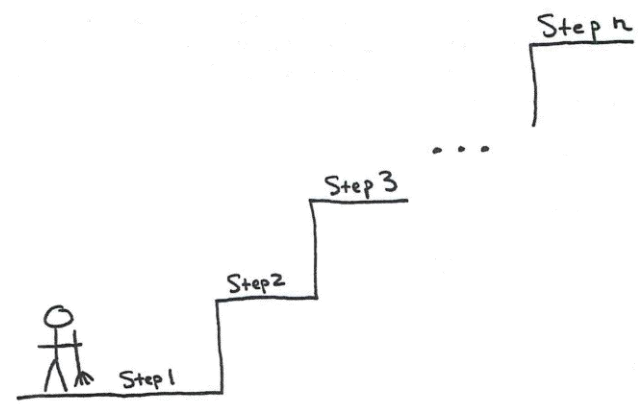

100,000
90,000
80,000
70,000
60,000
50,000
40,000
30,000
20,000
10,000
0
$T(n)$
0
20,000
40,000
60,000
80,000
100,000
n
n^3
n^2
$n \log n$
n
$\log n$
1

example: sweeping staircase


Step n
Step 3
Step 2
Step 1

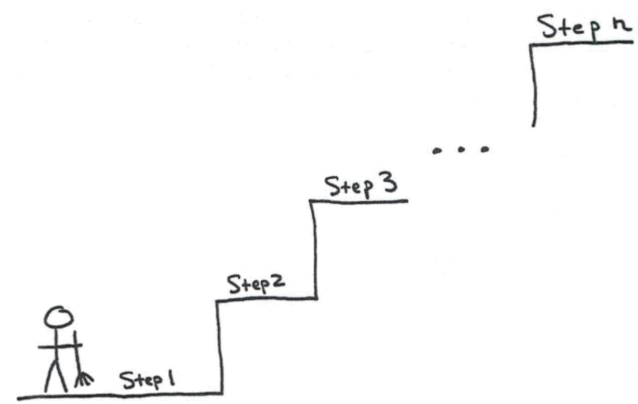
input: a person at the bottom of a staircase with n steps that must be swept clean

output: all n steps have been swept clean



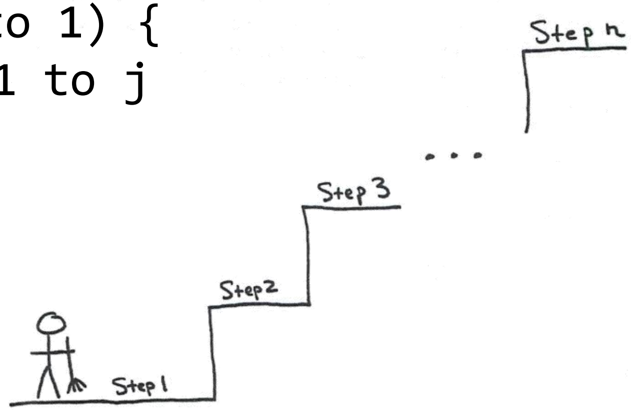
algorithm A:

```
// climb to top, sweep top to bottom
for (each stair i from 1 to n-1) {
   step up onto step i+1
}
for (each stair j from n-1 down to 1) {
   sweep debris of step j+1 to step j
   step down to step j
}
sweep debris to trash
```



algorithm B:

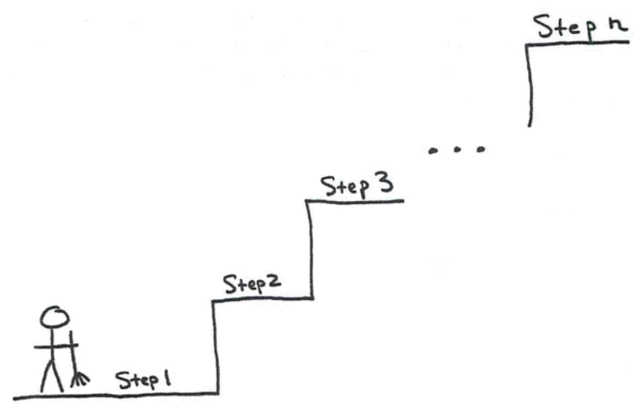
```
// clean bottom to top, making sure all
// steps below are clean
for (each stair i from 1 to n-1) {
   for (each stair j from 1 to i) {
         step up onto step j+1
   }
   for (each stair j from i down to 1) {
         sweep debris from step j+1 to j
         step down to step j
   }
   sweep debris to trash
}
```



based on intuition (and your own stair-sweeping experience), which algorithm will be “better”?

calculate $T(n)$ for algorithm A:

```
// climb to top, sweep top to bottom
for (each stair i from 1 to n-1) {
   step up onto step i+1
}
for (each stair j from n-1 down to 1) {
   sweep debris of step j+1 to step j
   step down to step j
}
sweep debris to trash
```

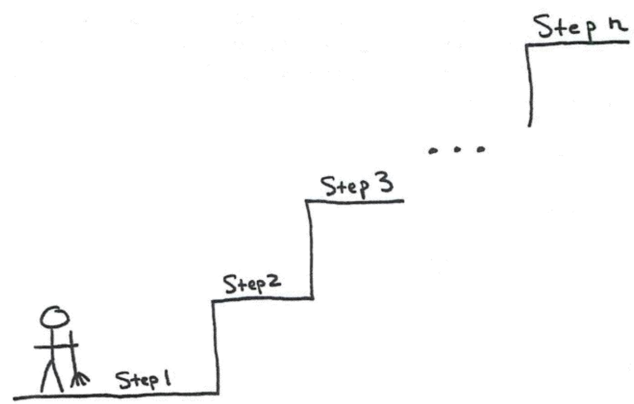


assume equal cost for:

step up / step down / sweep

calculate $T(n)$ for algorithm B:

```
for (each stair i from 1 to n-1) {
   for (each stair j from 1 to i) {
         step up onto step j+1
   }
   for (each stair j from i down to 1) {
         sweep debris from step j+1 to j
         step down to step j
   }
   sweep debris to trash
}
```



assume equal cost for:

step up / step down / sweep

what is the order of growth of algorithm A?

what is the order of growth of algorithm B?

clicker questions

let $T(n)$ be a function that returns the time it takes an algorithm to run given an input size of n

for Algorithm A, $T(n) = \log n$
and for Algorithm B, $T(n) = 10$

which algorithm scales better? ("scales better" means can run faster at larger input sizes, we also say, which algorithm has "better asymptotic performance")

A. A

B. B

C. A & B scale the same

which of these rates of growth is ordered from best to worst (a good rate of growth is one that scales well, i.e. has better asymptotic performance, which means “runs faster at larger input sizes” as input size goes to infinity)

A. factorial, logarithmic, linear, exponential, n-log-n, quadratic constant
B. exponential, logarithmic, factorial, linear, n-log-n, constant, quadratic
C. n-log-n, factorial, exponential, quadratic, linear, logarithmic constant
D. exponential, logarithmic ,factorial, n-log-n, linear, quadratic, constant
E. constant, logarithmic, linear, n-log-n, quadratic, exponential, factorial

Your co-worker, Bob, has an algorithm. Bob decides to test its performance. Bob feeds it an input with a size of 1000 and the algorithm finishes in 1.0 seconds. Bob feeds it an input with a size of 2000 and the algorithm finishes in 0.01 seconds. Bob does not understand why the algorithm ran faster with a larger input size. What are possible reasons the algorithm ran faster?

A. Bob needs to collect more samples at each input size because timings can be variable
B. maybe it was a worst case input at size 1000 and a best case input at size 2000
C. algorithms usually run faster the larger the input size
D. A & B
E. A & B & C

Your co-worker, Bob, has an algorithm. Bob decides to test its performance. Bob feeds it an input with a size of 1000 and the algorithm finishes in 0.9 seconds. Bob feeds it an input with a size of 2000 and the algorithm finishes in 4.1 seconds. Bob doubles the input size to 4000, and the algorithm finishes in 15.8 seconds. For an input size of 16000, of the times given below, which is the most likely running time of the algorithm?

A. 28 seconds
B. 32 seconds
C. 64 seconds
D. 128 seconds
E. 256 seconds

lecture summary

introduction to program complexity

- linear search
- best, worst, average case

analytical approach

- calculating running time
- order of growth

next:

binary search