Analysis of Algorithms

There is often more than one way of doing something.

- implementing a collection of things
  - e.g. use an array or a linked list to hold the elements?
  - e.g. how to arrange the elements within the array or linked list
- algorithms
  - e.g. sorting – insertion sort, selection sort, …

Which way is better?

Multiple criteria:

- time
- space
- simplicity

analysis of algorithms focuses on evaluating these

Key Ideas – Time

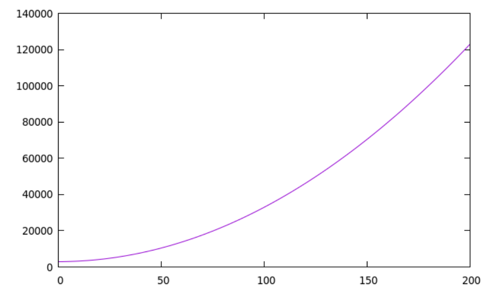
We are interested in:

- how the running time depends on the input size
  - described by a function T(n)
- the growth rate of T(n) rather than its actual value
  - the growth rate specifies how quickly T(n) increases with n
- asymptotic analysis
  - what happens in the long run

(The same ideas can be applied to analyzing space requirements.)

Running Time

T(n)



n – number of elements

- the input size *n* is the number of elements in the collection, being sorted, etc
- T(n) is the time is takes for the operation to run for an input size of *n*

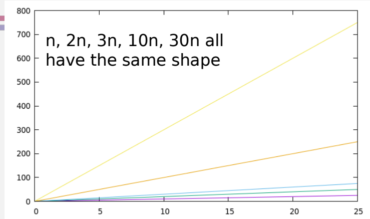
Growth Rates

Growth rate means we are looking at the shape of T(n) rather than its value.

Key observation #1 –

- multiplying T(n) by a constant doesn't change the shape

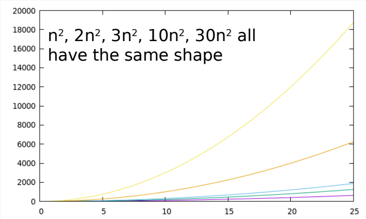
T(n)



n, 2n, 3n, 10n, 30n all have the same shape

n – number of elements

T(n)

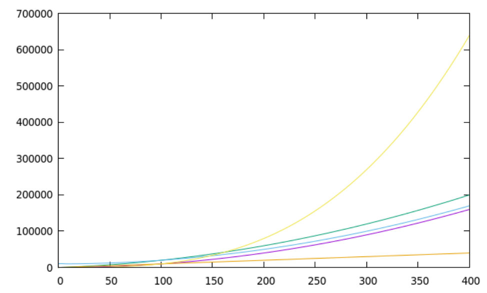


n^2, $2n^2$, $3n^2$, $10n^2$, $30n^2$ all have the same shape

n – number of elements

Growth Rate

T(n)



n – number of elements

$n^3/100$ has a different shape than n^2 – it grows much faster

n^2, $n^2+10000$, n^2+100n all have the same shape as n^2

100n has a different shape than n^2 – it grows much slower

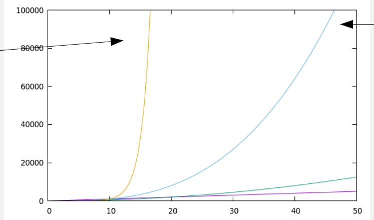
Key observation #2 –

- adding slower-growing terms to T(n) doesn't change the shape

Key Ideas – Time (and Space)

Why growth rate rather than actual elapsed time?

- growth rate provides an indication of whether the algorithm is practical for large problems



impossible beyond the very smallest problems

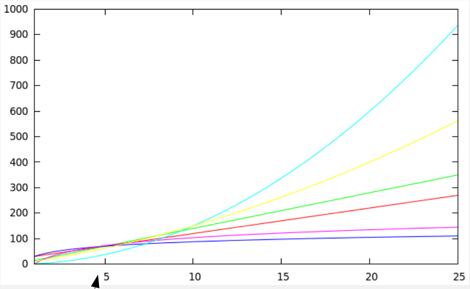
not ideal, but not out of the question

practical for large problem sizes

- growth rate is easier to analyze
  - allows analysis based on a high-level description of the algorithm rather than requiring implementation or careful counting with detailed pseudocode
  - avoids factors affecting elapsed time that are unrelated to the quality of the algorithm
    - e.g. machine speed and load, differences in programmer skill, compiler optimizations, specific input tested, …

Asymptotic Analysis

T(n)



n – number of elements

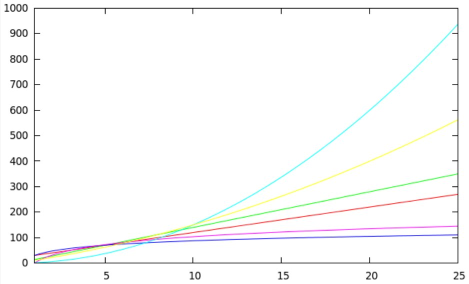
Asymptotic analysis means we are looking at what happens on this side of the chart...

...not what is over here.

Key Ideas – Time (and Space)

Why asymptotic analysis?

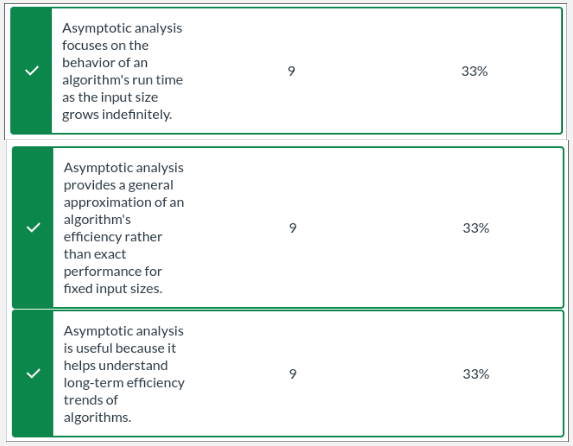
- differences in growth rate have a much larger effect on the actual running time when the input size is large



not much difference in value for small *n* – any algorithm is OK because nothing takes very long

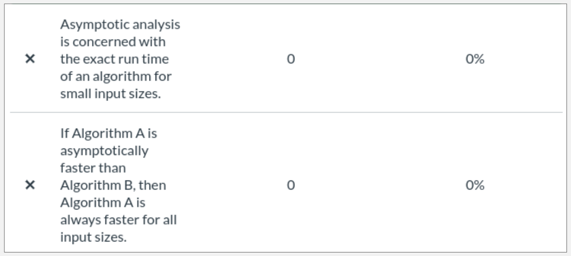
bigger difference in value for larger *n* – which algorithm is used matters more

Which of the following statements about asymptotic analysis are true? (Select all that apply.)



	Answer	Responses	Percent
✓	Asymptotic analysis focuses on the behavior of an algorithm's run time as the input size grows indefinitely.	9	33%
✓	Asymptotic analysis provides a general approximation of an algorithm's efficiency rather than exact performance for fixed input sizes.	9	33%
✓	Asymptotic analysis is useful because it helps understand long-term efficiency trends of algorithms.	9	33%

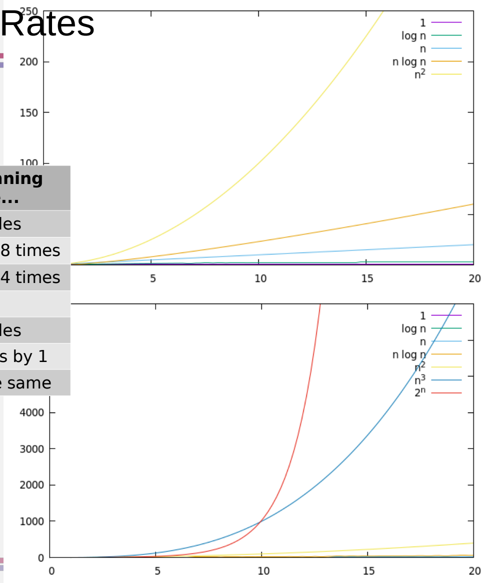
Which of the following statements about asymptotic analysis are true? (Select all that apply.)

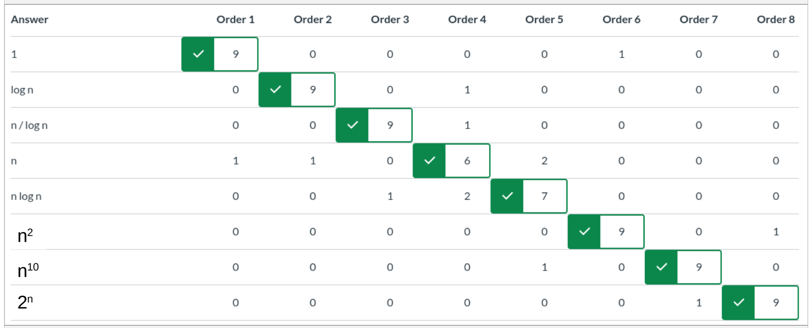


	Answer	Responses	Percent
×	Asymptotic analysis is concerned with the exact run time of an algorithm for small input sizes.	0	0%
×	If Algorithm A is asymptotically faster than Algorithm B, then Algorithm A is always faster for all input sizes.	0	0%

Common Growth Rates

f(n)	when n...	the running time...
2^n	increases by 1	doubles
n^3	doubles	increases 8 times
n^2	doubles	increases 4 times
n log n		
n	doubles	doubles
log n	doubles	increases by 1
1	doubles	stays the same





Answer	Order 1	Order 2	Order 3	Order 4	Order 5	Order 6	Order 7	Order 8
1	✓ 9	0	0	0	0	1	0	0
log n	0	✓ 9	0	1	0	0	0	0
n / log n	0	0	✓ 9	1	0	0	0	0
n	1	1	0	✓ 6	2	0	0	0
n log n	0	0	1	2	✓ 7	0	0	0
n^2	0	0	0	0	0	✓ 9	0	1
n^{10}	0	0	0	0	1	0	✓ 9	0
2^n	0	0	0	0	0	0	1	✓ 9

Implications for Algorithm Design

Θ	fast computer	1000x faster
1	n is irrelevant	n is irrelevant
log n	any n is fine	any n is fine
n n log n	still practical for n = 1,000,000	still practical for n = 1,000,000,000
n^2	usable up to n = 10,000 hopeless for n > 1,000,000	usable up to n = 300,000 hopeless for n > 30,000,000
2^n	impractical for n > 40	impractical for n > 50
n!	useless for n ≥ 20	useless for n ≥ 22

Implications for Algorithm Design

Θ	running time on fast computer	characteristics of typical tasks with the specified running time
1	n is irrelevant	examine/do only a fixed number of things
log n	any n is fine	repeatedly eliminate a fraction of the search space e.g. binary search
n	still practical for n = 1,000,000	examine each object a fixed number of times e.g. sequential search
n log n		divide-and-conquer with linear time per step e.g. mergesort, quicksort
n^2	usable up to n = 10,000 hopeless for n > 1,000,000	examine all pairs (nested loops) e.g. insertion sort, selection sort
n^3		examine all triples
2^n	impractical for n > 40	enumerate all subsets
n!	useless for n ≥ 20	enumerate all permutations

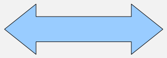
Analysis of Algorithms – "Sloppy" Counting

T(n) is the time it takes for the program to run on an input of size *n*.

e.g. the time it takes to insert an element into an array with *n* elements or to sort *n* numbers

"Sloppy" Counting

T(n) is the time it takes for the program to run on an input of size *n*.



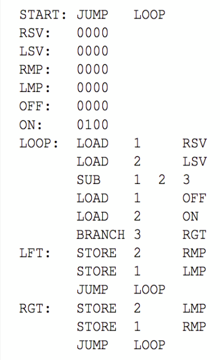
constant multiplicative factor (time per machine instruction)

```
START: JUMP    LOOP
RSV:   0000
LSV:   0000
RMP:   0000
LMP:   0000
OFF:   0000
ON:    0100
LOOP:  LOAD    1       RSV
       LOAD    2       LSV
       SUB     1   2   3
       LOAD    1       OFF
       LOAD    2       ON
       BRANCH  3       RGT
LFT:   STORE   2       RMP
       STORE   1       LMP
       JUMP    LOOP
RGT:   STORE   2       LMP
       STORE   1       RMP
       JUMP    LOOP
```

The computer is actually executing machine language instructions.

Since each machine language instruction takes the same amount of time to carry out, T(n) is proportional to the number of instructions executed – and has the same growth rate. So we can count the number of machine language instructions executed to understand T(n) instead timing the running program with a stopwatch.

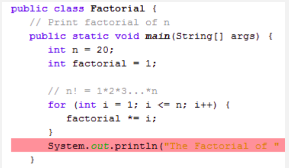
“Sloppy” Counting



```
START: JUMP   LOOP
RSV:   0000
LSV:   0000
RMP:   0000
LMP:   0000
OFF:   0000
ON:    0100
LOOP:  LOAD   1       RSV
       LOAD   2       LSV
       SUB    1   2   3
       LOAD   1       OFF
       LOAD   2       ON
       BRANCH 3       RGT
LFT:   STORE  2       RMP
       STORE  1       LMP
       JUMP   LOOP
RGT:   STORE  2       LMP
       STORE  1       RMP
       JUMP   LOOP
```

The computer is actually executing machine language instructions.

constant multiplicative factor (at most a small number of machine language instructions per Java statement)



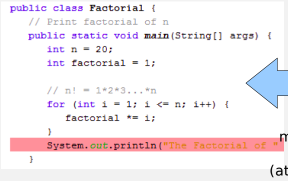
```
public class Factorial {
   // Print factorial of n
   public static void main(String[] args) {
      int n = 20;
      int factorial = 1;

      // n! = 1*2*3...*n
      for (int i = 1; i <= n; i++) {
         factorial *= i;
      }
      System.out.println("The Factorial of "
   }
```

The program is written in Java.

Not every statement in Java translates into the same number of machine language instructions, but there is an upper bound – say, one Java statement translates into at most 5 machine language instructions. Then T(n) is proportional to the number of Java statements executed – and has the same growth rate. So we can count the number of Java statements executed to understand T(n) instead of counting machine language instructions or timing the running program with a stopwatch.

“Sloppy” Counting

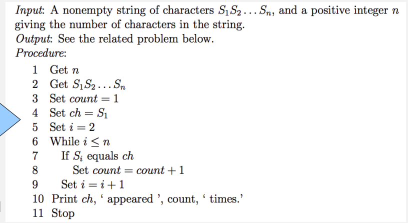


```
public class Factorial {
   // Print factorial of n
   public static void main(String[] args) {
      int n = 20;
      int factorial = 1;

      // n! = 1*2*3...*n
      for (int i = 1; i <= n; i++) {
         factorial *= i;
      }
      System.out.println("The Factorial of "
   }
```

The program is written in Java.

constant multiplicative factor (at most a small number of Java statements per pseudocode step)



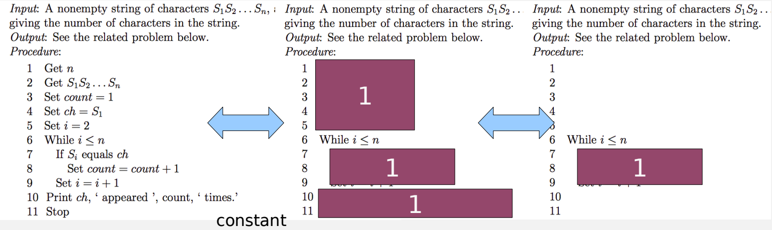
Input: A nonempty string of characters $S_1S_2\ldots S_n$, and a positive integer n giving the number of characters in the string.
Output: See the related problem below.
Procedure:

```
Get $n$
Get $S_1S_2\ldots S_n$
Set $count = 1$
Set $ch = S_1$
Set $i = 2$
While $i \le n$
   If $S_i$ equals $ch$
      Set $count = count + 1$
   Set $i = i + 1$
Print $ch$, ‘ appeared ’, count, ‘ times.’
Stop
```

An algorithm is expressed in pseudocode.

Not every pseudocode step translates into the same number of Java statements, but there is an upper bound – say, one pseudocode step translates into at most 5 Java statements. Then T(n) is proportional to the number of pseudocode steps executed – and has the same growth rate. So we can count the number of pseudocode steps executed to understand T(n) instead of counting Java statements or machine language statements or timing the running program with a stopwatch.

“Sloppy” Counting



Input: A nonempty string of characters $S_1S_2\ldots S_n$,
giving the number of characters in the string.
Output: See the related problem below.
Procedure:

```
Get $n$
Get $S_1S_2\ldots S_n$
Set $count = 1$
Set $ch = S_1$
Set $i = 2$
While $i \le n$
   If $S_i$ equals $ch$
      Set $count = count + 1$
   Set $i = i + 1$
Print $ch$, ‘ appeared ’, count, ‘ times.’
Stop
```

constant multiplicative factor (at most a small number of steps per block)

drop lower-order terms

Not every block contains the same number of pseudocode steps, but there is an upper bound – say, one block contains at most 10 steps. Then T(n) is thus proportional to the number of blocks executed – and has the same growth rate. So we can count the number of blocks executed to understand T(n) instead counting statements or timing the running program with a stopwatch.

The two blocks before and after the while loop don't affect the asymptotic behavior – as *n* grows, the difference in value between n and n+2 becomes insignificant compared to the value itself. So we can focus just on the most-repeated parts to understand T(n).

“Sloppy” Counting

Thus –

- focus on loops, and how the number of loop repetitions depends on the size of the input
  - identify what repeats the most

But –

- be aware of hidden loops – a method call is not one line of code, but rather all of the lines of code in its body ⚠