Do not distribute course material

CS 6033: Design&Analysis of Algorithms I

Prerequisites

1) CS5403: Data Structures and Algorithms
2) CS6003: Foundations of Computer Science
3) A programming course beyond “Introduction to Programming”.

Prerequisite(s): Graduate status, CS 5403 and CS 6003.

Here is the catalogue description of CS5403

This course introduces data structures. Topics include program specifications and design; abstract data types; stacks, queues; dynamic storage allocation; sequential and linked implementation of stacks and queues; searching methods, sequential and binary; binary trees and general trees; hashing; computational complexity; sorting algorithms: selection sort, heap sort, mergesort and quicksort; comparison of sorting techniques and analysis.

Here is the catalogue description of CS6003

This course covers logic, sets, functions, relations, asymptotic notation, proof techniques, induction, combinatorics, discrete probability, recurrences, graphs, trees, mathematical models of computation and undecidability.

Description:

Review of basic data structures and mathematical tools.

Data structures: priority queues, binary search trees, balanced search trees, B-trees.

Algorithm design and analysis techniques illustrated in searching and sorting: heapsort, quicksort, sorting in linear time, medians and order statistics.

Design and analysis techniques: dynamic programming, greedy algorithms.

Graph algorithms: elementary graph algorithms (breadth-first search, depth-first search, topological sort, connected components, strongly connected components), minimum spanning trees, shortest paths. String algorithms.

Geometric algorithm.

Brief introduction to NP-completeness.

Course Work and Grading

Homework	10%
Class participation	10%
Exams (40% midterm, 40% final)	80%

Do the homework - check the answer key!

Your participation matters

Attendance at exams is mandatory. Make-up exams will only be given in the case of a emergency, such as illness, which must be documented, e.g. with a doctor's note. In such cases, you **must** notify me as early as possible, preferably **before** the exam is given. If you miss an exam without a valid excuse, you will receive a grade of zero for that exam. The exams will be closed book.

Cheating will not be tolerated. Absolutely no communication with other students is permitted on exams. Cheating may not only result in a zero grade for the exam, but potentially a zero for the entire course, and possible additional actions at my discretion including involving the CS department and the administration.

Please see the university policy https://engineering.nyu.edu/academics/code-of-conduct/academic-dishonesty

You may discuss general approaches on how to do the homework assignments with other students. You may work with one other student to work out the details of the questions, and to write up the solution. If you work with another student, you must put both names and netID's on top of the assignment. Additionally, if you work with a partner, only one of you will submit the assignment on NYU Classes (but both of you are responsible to make sure it was submitted). If you do not have your name on top of the assignment, you will not receive any points for that assignment. If you work together, you must fully understand the work you submit. Your submission must be your (and your partner's) work. If there is any evidence that the work is not yours (and your partner's) work (such as copying from others, from the Internet, paying a third party to carry out the work, etc) it will be considered academic dishonesty. You will receive a 0 for the assignment, you will be reported to the department and the Dean of Student Affairs, and potentially receive a F for this course.(See http://cis.poly.edu/policies/)

Course Work and Grading

Homework	
Class participation	
Exams (40% midter	

The end of the class is *not* the time to try to increase your grade. Instead, the moment you detect that you are performing poorly you should seek help from a TA during office hours, the course text, outside sources, etc so you can do a midcourse correction. Rather than panicking at the end of the course when *no remaining opportunities* exist to raise your average

Attendance at exams is mand
illness, which must be docume ted, e.g. with a doctor's note. In such cases, you **must** notify me as early as possible, preferably **before** the exam is given. If you miss an exam without a valid excuse, you will receive a grade of zero for that exam. The exams will be closed book.

Cheating will not be tolerated. Absolutely no communication with other students is permitted on exams. Cheating may not only result in a zero grade for the exam, but potentially a zero for the entire course, and possible additional actions at my discretion including involving the CS department and the administration.

Please see the university policy https://engineering.nyu.edu/academics/code-of-conduct/academic-dishonesty

You may discuss general approaches on how to do the homework assignments with other students. You may work with one other student to work out the details of the questions, and to write up the solution. If you work with another student, you must put both names and netID's on top of the assignment. Additionally, if you work with a partner, only one of you will submit the assignment on NYU Classes (but both of you are responsible to make sure it was submitted). If you do not have your name on top of the assignment, you will not receive any points for that assignment. If you work together, you must fully understand the work you submit. Your submission must be your (and your partner's) work. If there is any evidence that the work is not yours (and your partner's) work (such as copying from others, from the Internet, paying a third party to carry out the work, etc) it will be considered academic dishonesty. You will receive a 0 for the assignment, you will be reported to the department and the Dean of Student Affairs, and potentially receive a F for this course.(See http://cis.poly.edu/policies/)

APPROXIMATE SCHEDULE

- Chapters 1, 2 & 3. Introduction: What's an algorithm? Why do we want to study algorithms? Termination. Correctness. Performance. How to measure performance of an algorithm? Models of computation, abstract machines. RAM. Best-, worst-, and average-case performance. Review of asymptotic notation: big-O, big-Ω, and big-Θ; little-o, and little-ω.

- Chapters 10 and 6. Review of basic data structures. Abstract data types (ADTs). Common ADTs: arrays, stacks, queues, linked lists, priority queues, heaps, heapsort.

- Chapters 11, 12, 13, 14, and 18. Dictionaries ADT. Hashing. Balanced search trees (2-3 trees, 2-3-4 trees, and more generally (a, b)-trees, red-black trees). B-trees. Augmenting a balanced search tree.

- Chapter 21. Implementation: using Union Find problem and Data Structures

- Chapters 4, 7, 9 and 33.4. Divide-and-conquer algorithms. Review of recurrences and how to solve them. Master's theorem. Binary search. Mergesort. Quicksort. Median and order statistics. Deterministic linear-time selection. Fast integer multiplication (Karatsuba's algorithm). Fast matrix multiplication (Strassen's algorithm). Closest-pair problem.

- Chapters 22 and 23. Graph algorithms: elementary graph algorithms (breadth-first search, depth-first search, topological sort, connected components, strongly connected components), minimum spanning trees, shortest paths. Some graph algorithms will be presented later in the course as illustrations for different algorithm design paradigms.

- Chapter 15 and 24. Dynamic programming: Rod cutting. Matrix chain product. Longest common subsequence. Optimal binary search trees. Shortest path problems in graphs. Transitive closure.

- Chapter 16. Greedy algorithms: Activity selection. Huffman coding. Minimum spanning trees.

- Chapter 34. Undecidability and fundamentals of NP-completeness (both very briefly; one lecture).

Thurs Feb 18th is on a Monday schedule, should we have (remote?) class on Friday Feb 19 and then have a day off later in the semester, right after the midterm?

Lecture 1

- analyzing algorithm (introduce framework)
- discuss two sorting algorithms (insertion sort and merge sort)
- discuss pseudocode used in this class
- discuss loop invariant
- use asymptotic notation for running time analysis
- discuss “divide-and-conquer”

“It has often been said that a person does not really understand something until after teaching it to someone else. Actually a person does not really understand something until after teaching it to a computer, i.e., expressing it as an algorithm” Donald Knuth

Design and Analysis of Algorithms?

- “Design - is going from a problem specification to an algorithm.”
- “Analysis - is going from a description of an algorithm to some properties that we might care about: proof of its termination, proof of its correctness, determining its best/worst-case running time, space use etc.”

Boris Aronov

Why?

Algorithms are needed to:

Sending/Receiving Information over the Internet

Scheduling

Allocation of scarce resources

Reducing the size of data

Finding the shortest path

Human Genome Project

Electronic commerce

…

Computer program performance

We focus on the questions:

- How do we know an algorithm is correct
- How do we calculate the running time of an algorithm
- How do we compare two algorithms
- Can we find a “better” algorithm?

Sorting

Why Sort?

Sorted data is easier to work with. Historically 25% of computer time was spent on sorting!

grouping

Testing if all the items are unique is easy if sorted!

Finding which item occurs most frequently is much faster if the items are sorted! O(n)

preprocessing:

Searching through 100,000,000 items (under the assumption of each item is equally likely to be searched for) takes ~50,000,000 comparisons if the items are unsorted vs ~ 27 comparisons using binary search. O(n) vs O(log(n))!

Finding the smallest, largest, K'th largest is O(1) if sorted.

subroutine in other algorithms

Optimal compression of text. Finding an optimal Huffman encoding starts by knowing the character frequency and sorting the characters by their frequencies.

Kruskal's algorithm for finding an optimal spanning tree starts by sorting the edge weights.

Finding which pair of items are closest to each other is easy if the items are sorted! O(n) time

Sorting Problem:

Input: $n \in \mathbb{Z}^+$ *keys*: $A = \langle a_1, a_2, \ldots, a_n \rangle$ where two keys can be compared using '<'

Output: $A' = \langle a'_1, a'_2, \ldots, a'_n \rangle$ is a permutation of $A = \langle a_1, a_2, \ldots, a_n \rangle$ where $a'_1 \leq a'_2 \leq \cdots \leq a'_n$

Some keys may contain satellite data.

Example:

Input Class list < (701, John Doe, 37 Baker Street), (231, Jane Doe, 19 Main Street), …, (200, John Smith, 10 Central Ave) > where the *key* is the student ID number.

Output < (200, John Smith, 10 Central Ave),(231, Jane Doe,19 Main Street), …, (701, John Doe, 37 Baker Street)>

Sorting Problem:

Input: $n \in \mathbb{Z}^+$ *keys*: $A = \langle a_1, a_2, \ldots, a_n \rangle$ where two keys can be compared using '<'

Output: $A' = \langle a'_1, a'_2, \ldots, a'_n \rangle$ is a permutation of $A = \langle a_1, a_2, \ldots, a_n \rangle$ where $a'_1 \leq a'_2 \leq$

Note that the variables changed their values. To avoid confusion, occasionally I will use a "prime" (') to signify that the value of the variable has changed.
If we need to discus the value at an intermediate point occasionally we will use "double primes" ("), superscripts or subscripts

Some keys may contain satellite data.

Example:

Input Class list < (701, John Doe, 37 Baker Street), (231, Jane Doe, 19 Main Street), …, (200, John Smith, 10 Central Ave) > where the *key* is the student ID number.

Output < (200, John Smith, 10 Central Ave),(231, Jane Doe,19 Main Street), …, (701, John Doe, 37 Baker Street)>

Design

Input:

20	17	43	25	4	72	15

Output:

4	15	17	20	25	43	72

Does it work? How long does it take to finish?

Insertion Sort

Lets code up our sorting algorithm

We won't code it exactly as we imagined it. Instead we will use our picture as inspiration for our algorithm

Algorithm

Algorithm is how you g
from the input
to the output.

Constructive!
instructions for how
to find the
answer

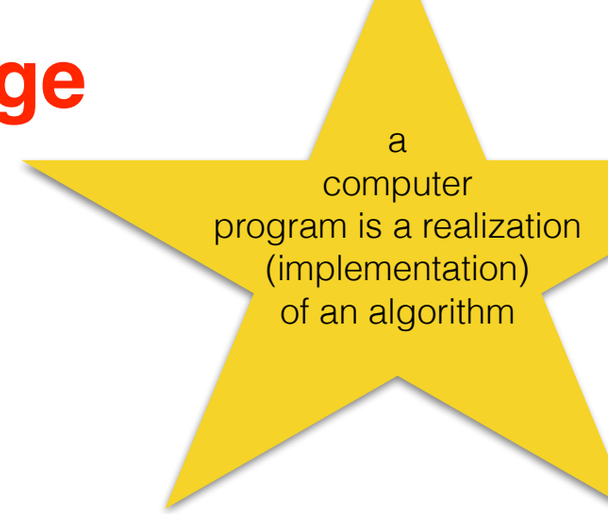
Notice: this means it must be precise!

… an **algorithm** is any well-defined computational procedure that takes some value, or set of values, as **input** and produces some value, or set of values, as **output**.

… tool for solving a well-specified **computational problem**.

"An algorithm is a finite, definite, effective procedure, with some input and some output." Knuth

Pseudocode - a mix of natural language and programming language concepts



- indent to show block structures
- **//** indicates comments
- **loops**: **for** (**to, down, by**) **while, repeat-until**. *The loop counter retains its value that caused the loop to terminate* (useful when proving the correctness of an algorithm)
- The notation "**..**" is used to indicate a (inclusive) **subarray**, e.g. **A[i..j]** indicates the elements <**A[i],A[i+1], ..., A[j]**>
- Object **attributes** and **methods** are accessed using the standard "**.**" notation, e.g. **A.length**
- **return** statements can return multiple values
- See the textbook for more information (pages 20-22)

A[1..j-1] sorted rest of array

How did the algorithm work? What invariant did we maintain?

How can we turn this idea into code?

function Insertion-Sort(A)

for *j* = 2 to *A.length*

key = A[*j*] // Insert A[*j*] into the sorted sequence A[1.. *j*-1]

i = *j* - 1

while *i* > 0 and A[*i*] > key

A[*i* + 1] = A[*i*]

i = *i* - 1

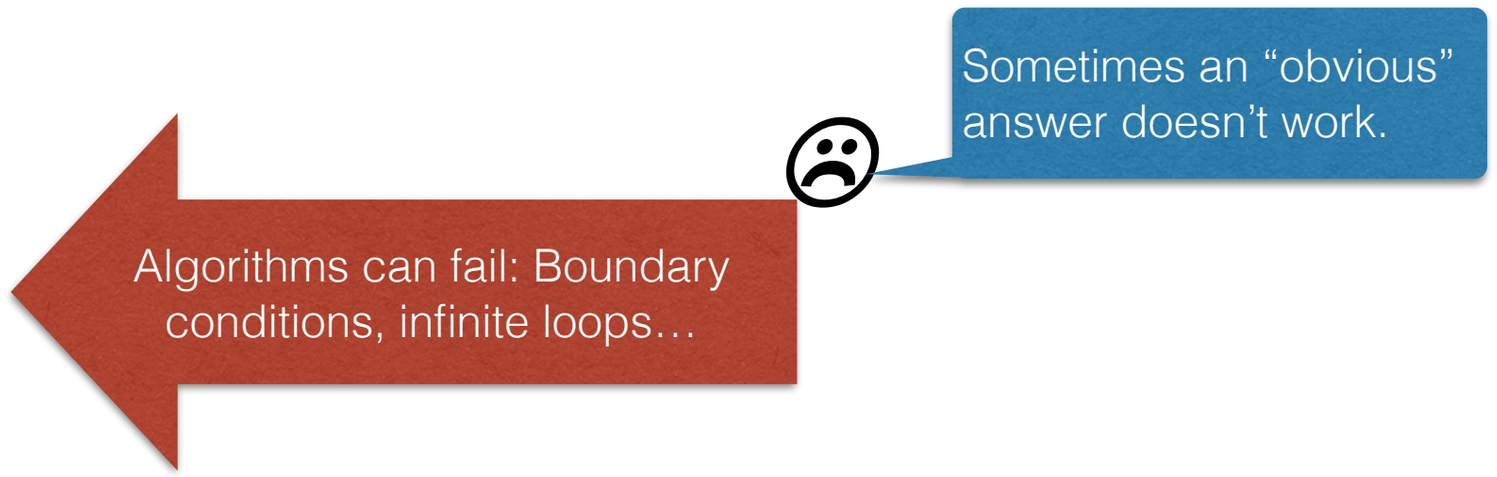
A[*i* + 1] = *key*

Note how we use key = A[j] to hold the item in A[j]

Does it work? How long does it take to finish?

Is there a better algorithm?

Correctness?



1. Prove the algorithm produces the correct result if the input was valid
2. Prove that the algorithm always terminates

Sometimes easy to do, sometimes very complex to do.

Commonly, the principle of mathematical induction is used.

Often the loop invariant proves how the loop makes progress toward the goal of the loop

An invariant is a way to formalize what your variables mean. Bugs are hard to track. Use a loop invariant to check that your code does what you think it is doing

Proving Correctness using a Loop Invariant

Loop Invariant: *A property that is true before the start of each loop, and true just after the loop ends*

Loop Invariant similar to mathematical induction

Example loop invariant: At the start of each iteration of the for loop indexed by j, the subarray *A[1..j-1]* consists of the elements originally in *A[1..j-1]*, but in sorted order

Loop invariant has to be strong enough to prove the desired property works

When writing code think of the loop invariant first then code
What is the **milestone** of each iteration?

Often the loop invari
proves how the
makes progress tow
the goal of the loop

"… the loop invariant can be viewed as a more abstract specification of the loop, which characteriz
the deeper purpose of the loop beyond the details of th
implementation."

The loop invariant is NOT the loop conditional

Bug catcher!

Proving C Invariant

Loop Invariant: *A property that is true before the start of each loop, and true just after the loop ends*

Loop Invariant similar to mathematical induction

Example loop invariant: At the start of each iteration of the for loop indexed by j, the subarray *A[1..j-1]* consists of the elements originally in *A[1..j-1]*, but in sorted order

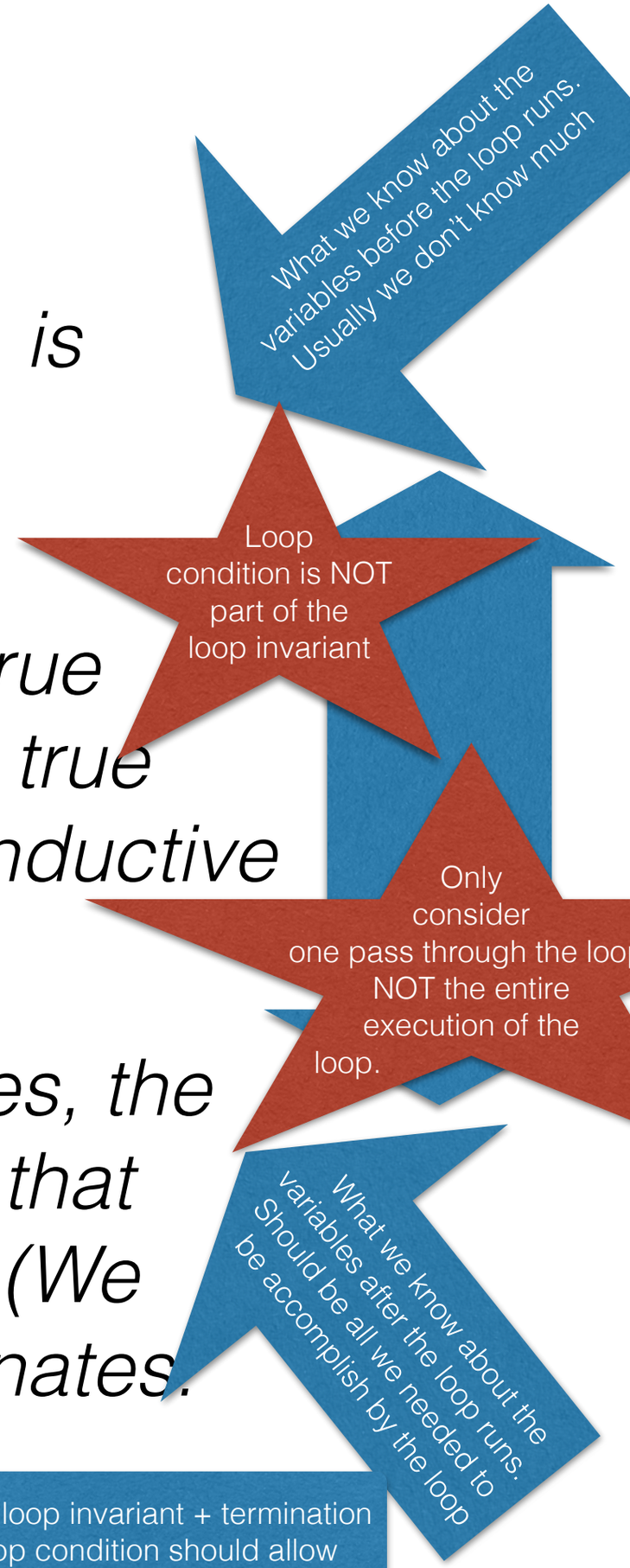
Loop invariant has to be strong enough to prove the desired property works

When writing code
think of the loop invariant
first then code
What is the **milestone** of each iteration?

1) **Initialization:** *Prove the loop invariant is true prior to the first iteration of the loop! (Similar to Base Case)*

2) **Maintenance:** *If the loop invariant is true before an iteration of the loop, it remains true before the next iteration. (Similar to the Inductive Step)*

3) **Termination:** *When the loop terminates, the loop invariant gives us a useful property that helps show that the algorithm is correct. (We "stop" the induction when the loop terminates. This is different from induction.)*



Two nice articles: https://yourbasic.org/algorithms/loop-invariants-explained/

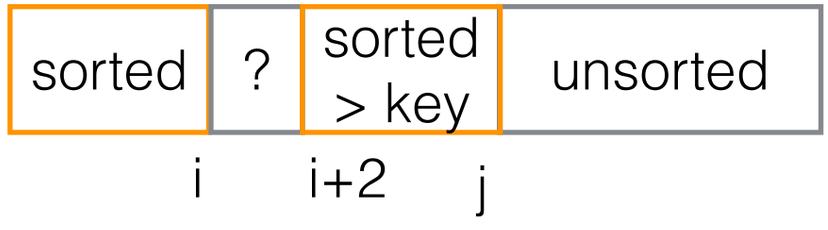
http://www.cs.cornell.edu/courses/cs2110/2017sp/online/loops/02loploopyq.pdf

There are two loops in the insertion sort algorithm so we need to prove two loop invariants.

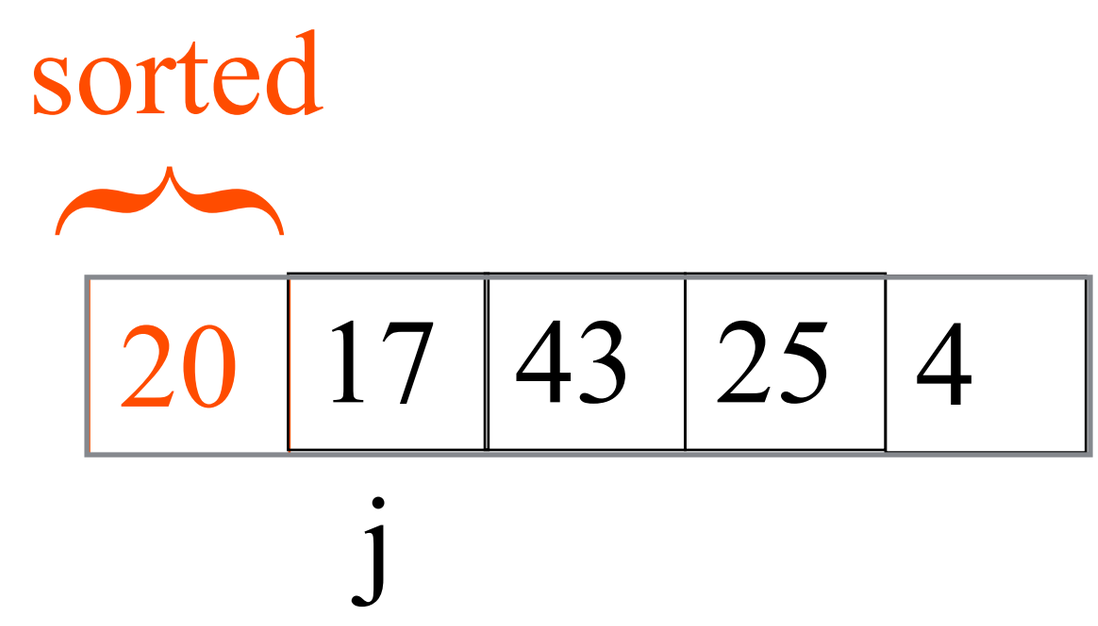
```
function Insertion-Sort(A)
  for j = 2 to A.length
      key = A[j]
      i = j - 1
  while i > 0 and A[i] > key
      A[i + 1] = A[i]
      i = i - 1
  A[i + 1] = key
```

Outer Loop Invariant: At the start of each iteration of the for loop indexed by j, the subarray *A[1..j-1]* consists of the elements originally in *A[1..j-1]*, but in sorted order

Inner Loop Invariant: At the start of each iteration of the while loop, the subarray *A[i+2..j]* consists of elements greater than the *key,* and contains the items originally in *A[i+1..j-1]* in the same sorted order. The items in position *A[1..i] are unchanged*



```
function Insertion-Sort(A)
  for j = 2 to A.length
      key = A[j]
      i = j - 1
      while i > 0 and A[i] > key
          A[i + 1] = A[i]
          i = i - 1
      A[i + 1] = key
```



Outer Loop Invariant: At the start of each iteration of the for loop indexed by j, the subarray *A[1..j-1]* consists of the elements originally in *A[1..j-1]*, but in sorted order

Initialization: Just before the first iteration, j = 2. The subarray A[1.. j-1]=[1.. 2-1] is the single element A[1] , which is the element originally in A[1], and it is trivially sorted.

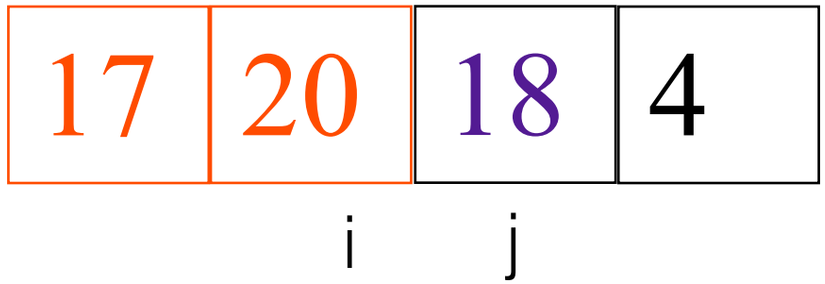
Maintenance: To be precise, we would need to state and prove a loop invariant for the "inner" **while** loop. Rather than getting bogged down in another loop invariant, we instead note that the body of the inner **while** loop works by moving A[j-1] , A[j-2] , A[j-3], and so on, by one position to the right until the proper position for *key* (which has the value that started out in A[j]) is found. At that point, the value of *key* is placed into this position.

Termination: The outer **for** loop ends when j > n, which occurs when j = n+1.

Therefore, j - 1 = n. Plugging n in for j in the loop invariant, the subarray A[1.. n] consists of the elements originally in A[1.. n] but in sorted order.

In other words, the entire array is sorted.

```
function Insertion-Sort(A)
  for j = 2 to A.length
      key = A[j]
      i = j - 1
      while i > 0 and A[i] > key
          A[i + 1] = A[i]
          i = i - 1
      A[i + 1] = key
```



Outer Loop Invariant: At the start of each iteration of the for loop, the subarray *A[1..j-1]* consists of the elements originally in *A[1..j-1]*, but in sorted order

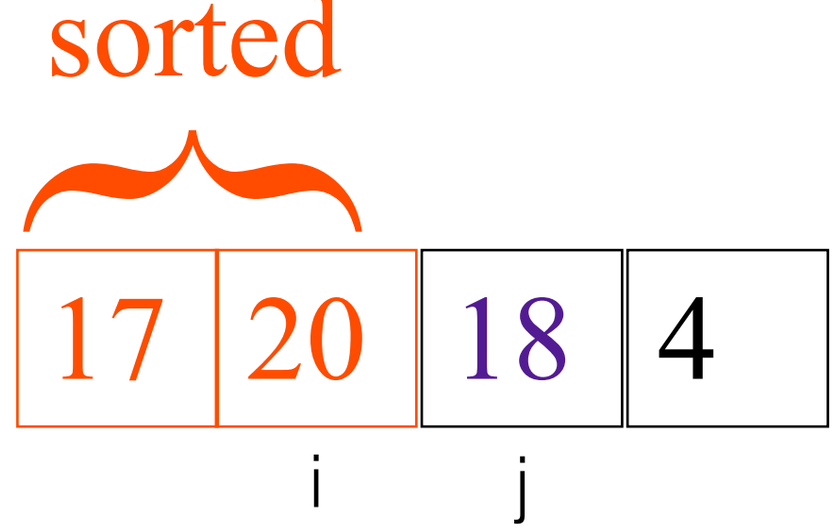
To verify that your loop is correct, please simulate your loop to verify your actual results against your proof.

Maintenance:

- We assume the **outer loop invariant** A[1..j-1] is the sorted version of the original A[1..j-1]
- Assign key = A[j], i = j-1
- Inner loop (Remember the termination condition is that A[1..i] & A[i+2..j] are sorted, contain all the items originally in A[1..j-1] together, and that A[i]<=key (or i≤0), A[i+2]>key
- Leaving room for us to set A[i+1] to key (The original A[j]).
- After incrementing j (*hidden in the for statement*), the loop invariant that A[1..j-1] must be sorted is true.

Next three slides contain additional information not included in the lecture

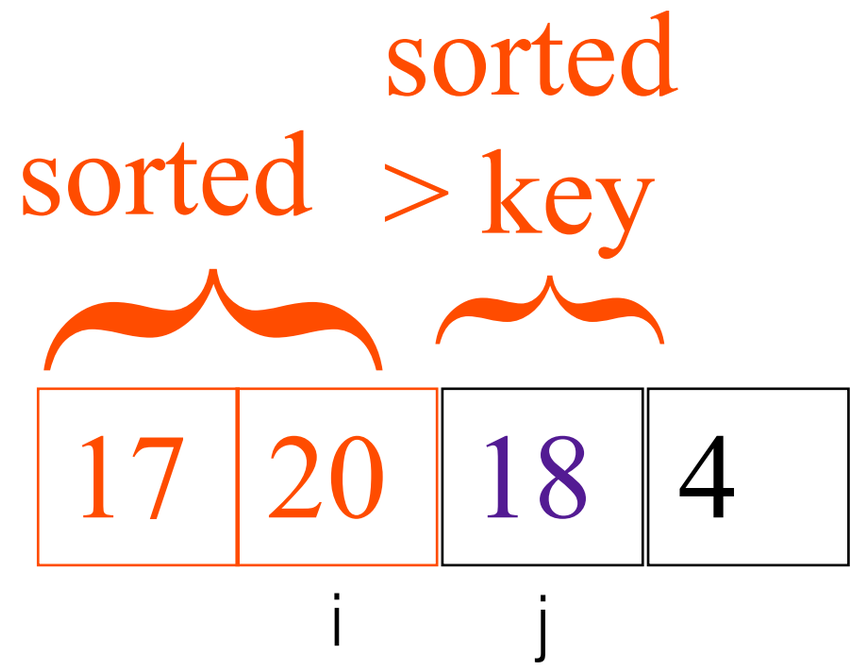
```
function Insertion-Sort(A)
  for j = 2 to A.length
      key = A[j]
      i = j - 1
      while i > 0 and A[i] > key
          A[i + 1] = A[i]
          i = i - 1
      A[i + 1] = key
```



Inner Loop Invariant: At the start of each iteration of the while loop, the subarray *A[i+2..j]* consists of elements greater than the *key*, and contains the items originally in *A[i+1..j-1]* in the same sorted order. The items in position *A[1..i]* are the original sorted items

Initialization: From the **outer loop invariant** we know *A[1..j-1] is* sorted before the while loop starts. We then create *i as j-1*. Thus the range *A[i+2..j] is empty (since i+2>j, thus it is trivially true and equivalent to the original A[i+1..j-1]).* A[1..i] has not been changed and contains the original sorted item.

```
function Insertion-Sort(A)
  for j = 2 to A.length
      key = A[j]
      i = j - 1
      while i > 0 and A[i] > key
          A[i + 1] = A[i]
          i = i - 1
      A[i + 1] = key
```

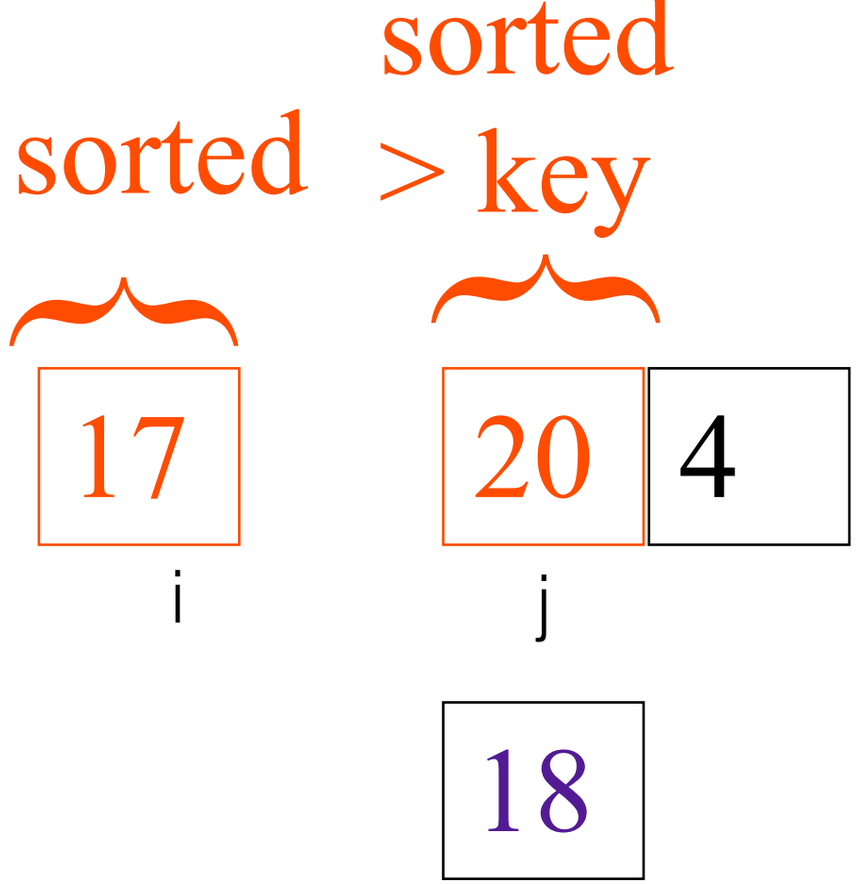


Inner Loop Invariant: At the start of each iteration of the while loop, the subarray *A[i+2..j]* consists of elements greater than the *key,* and contains the items originally in *A[i+1..j-1]* in the same sorted order. The items in position *A[1..i] are unchanged*

Maintenance:

- Prior Conditions: i>0, A[i]>key, A[i+2..j] are all greater than the key, and these are the items originally in A[i+1..j-1] in the same sorted order. The items originally in A[1..i] are unchanged.
- Set A[i] into A[i+1]
- Now, A[i+1]>key, A[i+1..j] are sorted, greater than key, and are the items originally in A[i..j-1] in the same sorted order
- Once we decrement i, now A[i+2]>key, and A[i+2..j] are sorted and greater than the key. A[1..i] are untouched.
- This fulfills the maintenance condition.

```
function Insertion-Sort(A)
  for j = 2 to A.length
      key = A[j]
      i = j - 1
      while i > 0 and A[i] > key
          A[i + 1] = A[i]
          i = i - 1
      A[i + 1] = key
```



Inner Loop Invariant: At the start of each iteration of the while loop, the subarray *A[i+2..j]* consists of elements greater than the *key,* and contains the items originally in *A[i+1..j-1]* in the same sorted order. The items in position *A[1..i]* are the original sorted item

Termination:

- while condition isn't satisfied: *i = 0 or A[i] <= key*
- loop invariant:
  - *A[1..i]* has not changed (and are sorted), and at most *key*,
  - *A[i+2..j]* are greater than the key and contains the original items in *A[i..j-1]* , still in sorted order.

Induction over the number of iterations!

Three things to prove:

Conjecture holds before the loop starts (initialization) (0 Iterations)

Conjecture holds holds at the end of each loop, so it is true at the beginning of the next loop (maintenance) (For arbitrary n completed loops)

The conjecture holds when the loop condition fails, allowing us to prove a useful property (termination)

In a **for** loop we test the loop invariant after the loop counter has been updated and before the loop condition is tested

How “efficient” is the algorithm?
(This will allow us to compare algorithms)

Want a way to ***analyze*** the algorithm…

i.e. *predict* the resources used (memory, communication band width, computer hardware, …)

We will focus on the computational time (i.e. how long it takes the algorithm to produce the output)

<u>Want a general theory that is:</u>

- *simple*
- *objective*
- *doesn’t change when the next generation of mac’s or pc’s arrive (except quantum computers …)*
- *predictive - gives a good estimate for large inputs*

Most algorithms take longer to run when the input size increases

How can we estimate the running time based on the size of the input?

Examples:

- *sorting n items*
- *finding the closest pair of n geographic locations*
- *finding the medium of n items*
- *multiplying two numbers*
- *finding the shortest path between two nodes in a graph*

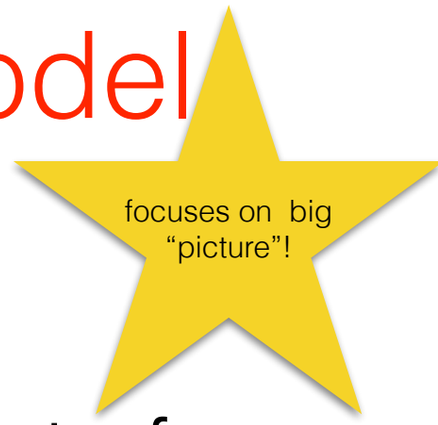
…what should the input size be?

Depends on the problem

There are many different models for how to estimate the running time of an algorithm.

logarithmic cost model has the number of steps proportional to the number of bits involved in the computation.

Random Access Machine (RAM) Model



Every machine instruction takes a constant amount of time (this is not true!)

Advantages:

- machine independent!
- doesn’t lose predictive power!
- language independent
- compiler independent

Focuses on rates of growth

RAM = Random Access Machine

A generic one-processor machine that contains *instructions commonly found* in real computers

In a RAM model, each of these takes a constant amount of time:

+, -, *, /, shift left, shift right, remainder, floor, ceiling

conditional branch, unconditional branch

subroutine call and return (the time for the subroutine to run is counted separately)

load, store, copy (we don't worry about if the item is in the cache or on the disk…)

We have as much memory as we need.

The RAM model assumes there is a limit on the word size. That integers are represented by c*lg(n) bits

We can estimate the number of RAM steps in an algorithm.

By assuming our computer can execute a certain number of RAM steps per second, we can estimate the actual run time

Running time of an algorithm is

$$\sum (\text{cost of statment}) \cdot (\text{number of times statement is executed})$$

Time taken by an algorithm depends on the *size* of its input

Time taken by an algorithm depends on the *particular* input

Insertion Sort

For j = 2, 3, 4,…, n,
let t_j be the number of times
the while loop *test* is executed

function Insertion-Sort(A)

```
for j = 2 to A.length
    key = A[j]
    // Insert A[j] into the sorted sequence A[1..j-1]
    i = j - 1
    while i > 0 and A[i] > key
        A[i + 1] = A[i]
        i = i - 1
    A[i + 1] = key
```

# times executed	#instructions for this step
n	c_1
n-1	c_2
n-1	c_3
?	c_4
?	c_5
?	c_6
n-1	c_7

Total #steps?

$$T(n) = (n)c_1 + (n-1)c_2 + (n-1)c_3 + ?c_4 + ?c_5 + ?c_6 + (n-1)c_7$$

Insertion Sort

For j = 2, 3, 4,…, n,
let t_j be the number of times
the while loop *test* is executed

function Insertion-Sort(A)

```
for j = 2 to A.length
    key = A[j]
    // Insert A[j] into the sorted sequence A[1..j-1]
    i = j - 1
    while i > 0 and A[i] > key
        A[i + 1] = A[i]
        i = i - 1
    A[i + 1] = key
```

# times executed	#instructions for this step
n	c_1
n-1	c_2
n-1	c_3
$\sum_{j=2}^{n} t_j$	c_4
$\sum_{j=2}^{n} (t_j - 1)$	c_5
$\sum_{j=2}^{n} (t_j - 1)$	c_6
n-1	c_7

Total #steps

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=2}^{n} t_j + c_5 \sum_{j=2}^{n} (t_j - 1) + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7(n-1)$$

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=2}^{n} t_j + c_5 \sum_{j=2}^{n} (t_j - 1) + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7(n-1)$$

Number of Steps?

Best?

Best case (means fewest number of steps)

Average?

We will consider this case when we cover randomized algorithms

Average case (very hard to define the “typical” input)

Worst Case?

Worst case (means the largest number of possible steps for some input)

When people ask for the running time, they usually mean worst case

Insertion Sort

For j = 2, 3, 4,…, n,
let t_j be the number of times
the while loop test is executed

function Insertion-Sort(A)

```
for j = 2 to A.length
    key = A[j]
    // Insert A[j] into the sorted sequence A[1..j-1]
    i = j - 1
    while i > 0 and A[i] > key
        A[i + 1] = A[i]
        i = i - 1
    A[i + 1] = key
```

# times executed	#instructions for this step
n	c_1
n-1	c_2
n-1	c_3
$\sum_{j=2}^{n} t_j$	c_4
$\sum_{j=2}^{n} (t_j - 1)$	c_5
$\sum_{j=2}^{n} (t_j - 1)$	c_6
n-1	c_7

Total #steps

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=2}^{n} t_j + c_5 \sum_{j=2}^{n} (t_j - 1) + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7(n-1)$$

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=2}^{n} t_j + c_5 \sum_{j=2}^{n} (t_j - 1) + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7(n-1)$$

Number of Steps?

Best case (means fewest number of steps)

$T(n) = an + b$ linear!

Average case (very hard to define the “typical” input)

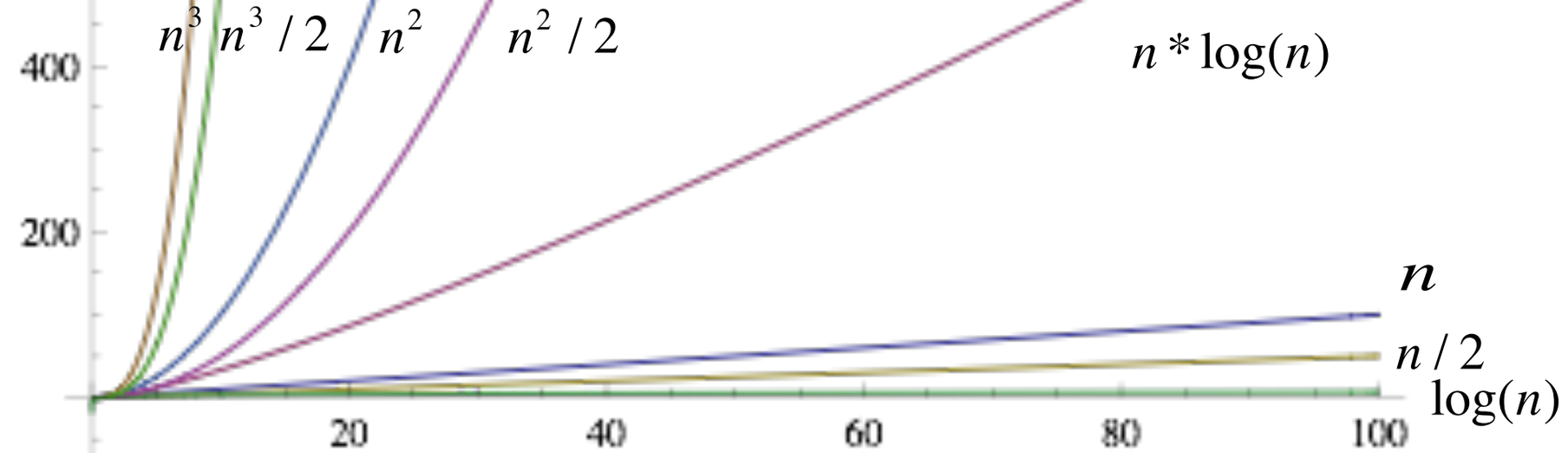
$T(n) = an^2 + bn + c$

Worst case (means the largest number of possible steps for some input)

$T(n) = an^2 + bn + c$ quadratic…

Do we need that much precision?

How do we (roughly) estimate the running time?




```
Plot[{n, n*Log[2, n], n/2, Log[2, n], n^2, n^2/2, n^3, n^3/2}, {n, 0, 100},
 PlotRange -> 500]
```

$\log(n)$	n	$n\log(n)$	$n^2/2$	n^2	$n^3/2$	n^3
6.6	100	664	5000	10000	50000	1000000
7.6	200	1529	20000	40000	4000000	8000000
8.2	300	2469	45000	90000	13500000	27000000
8.6	400	3458	80000	160000	32000000	64000000
…	…	…	…	…	…	…
9.96	1000	9966	500000	1000000	500000000	1000000000

$10^6/1{,}000{,}000{,}000 = 1/1{,}000$

$10^6 * \log(10^6)/1{,}000{,}000{,}000$ still less than a second

$10^{12}/1{,}000{,}000{,}000 = 10^3$ approx 16.6 minutes

$10^{18}/1{,}000{,}000{,}000 = 10^9$ approx 31.7 years…

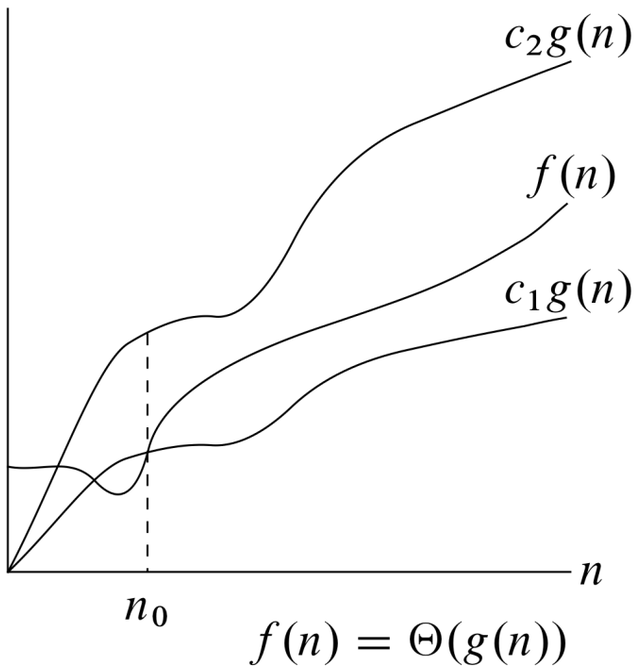
Asymptotic Notation

- used to describe the growth of (any) function whose domain is the set of natural numbers: 0, 1, 2, 3, …

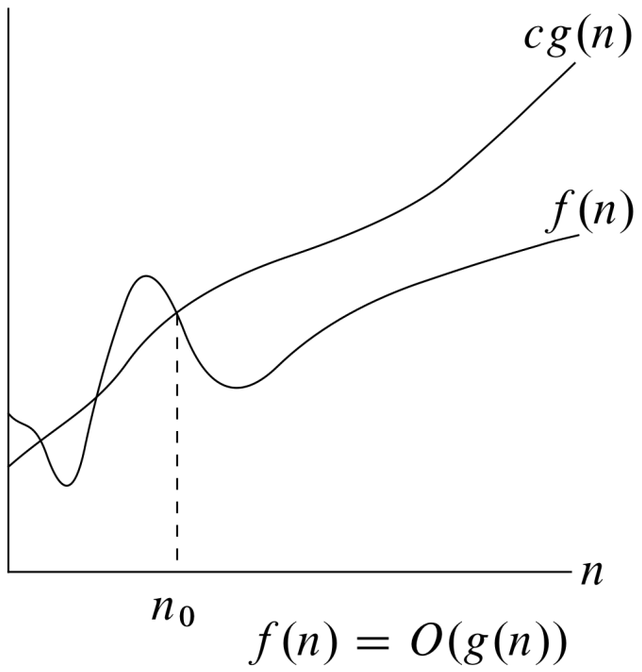
- Θ, O, Ω, o, ω

1. Drop lower-order terms

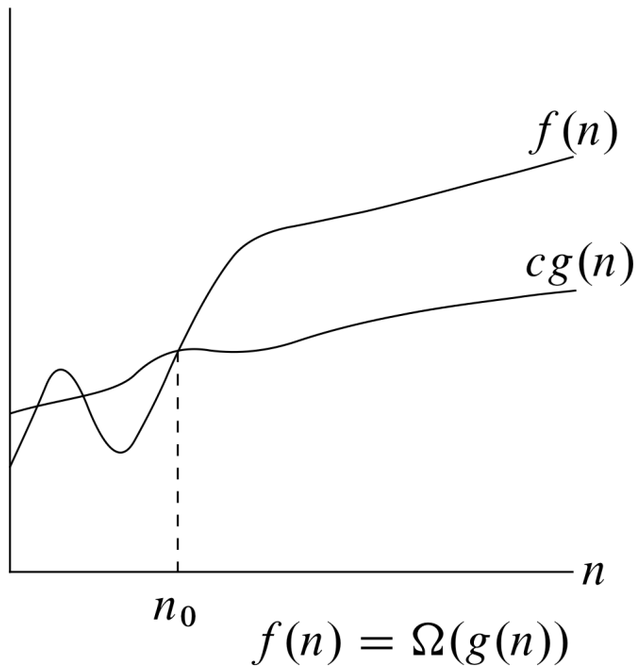
2. Drop constant coefficients



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



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



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

O, o, Ω, ω, Θ

Asymptotic Upper Bound Notation

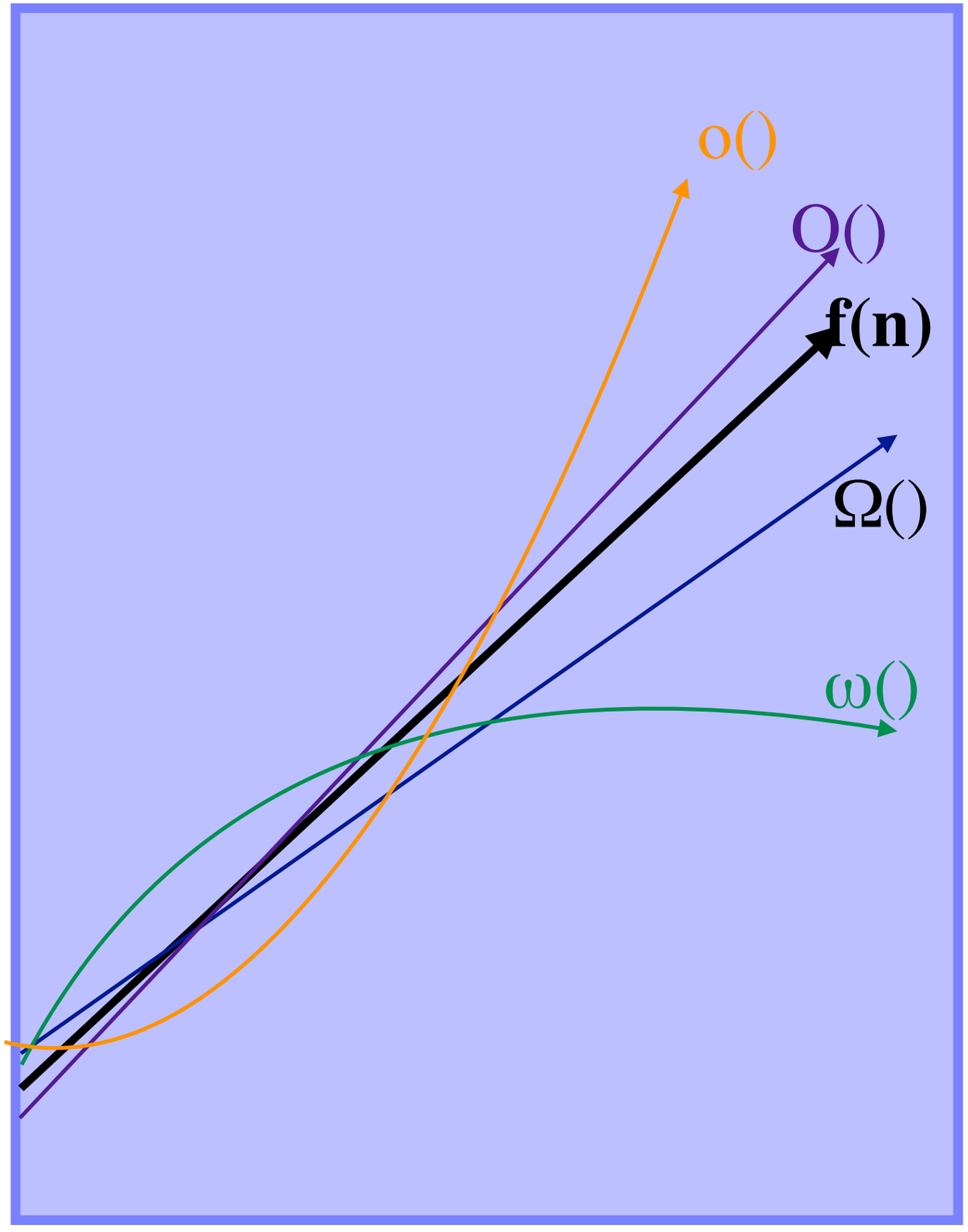
- Big-O, O(): f(n) is O(g(n)) if *there exists* a $c > 0$ and an n_0 such that $f(n) \leq cg(n)$ for all $n \geq n_0$
- Little-o, o(): f(n) is o(g(n)) if *for any* constant $c > 0$ there exists n_0 such that $f(n) < cg(n)$ for all $n \geq n_0$,

Asymptotic Lower Bound Notation

- Ω Asymptotic lower bound: f(n) is Ω(g(n)) if *there exists* a $c > 0$ and an n_0 such that $f(n) \geq cg(n)$ for all $n \geq n_0$
- Little-Omega, ω(): f(n) is ω(g(n)) if *for any* constant $c > 0$ there exists n_0 such that $f(n) > cg(n)$ for all $n \geq n_0$,

Asymptotic Equality Notation

- Big-Theta, Θ(): f(n) is Θ(g(n)) if it is both O(g(n)) and Ω(g(n))



Asymptotic Notation Examples

- When we say a function f(n) is Θ(n) we can think of f(n) as an anonymous linear function

- We an use asymptotic notation in a formula:

$$4n^2 + 3n + 4 = 4n^2 + \Theta(n)$$

means that $4n^2 + 3n + 4 = 4n^2 + f(n)$ where f(n) is a function in Θ(n)

- We can use asymptotic notation in a recurrence relation:

$$T(n) = T(n-1) + T(n-2) + \Theta(n)$$

if we want the asymptotic behavior of T(n), and not an exact number

Asymptotic Notation Intuition

- Running time may be a complicated function of the input length

$$2n^3 + 3n^2 \log n + 18\, n + 21\, \log^5 n + 21$$

- We are concerned about the behavior of the function on large input (the run times are almost always fast on small input value). As n becomes large the high order $2n^3$ term dominates

$$2n^3 + 3n^2 \log n + 18\, n + 21\, \log^5 n + 21 \sim 2n^3$$

- Additionally, the coefficient 2 in $2n^3$ can depend not only on the algorithm but also on the machine and other external factors not related to the algorithm, thus we omit it as well

- With each of these simplifications, we get less precise but gain a level of abstraction that makes are analysis more useable and meaningful. (We are doing mathematical modeling)

Insesrtion Sort Running times:

Best case (means fewest number of steps)

$T(n) = an + b \text{ is } \Theta(n)$

Average case (very hard to define the “typical” input)

$T(n) = an^2 + bn + c \text{ is } \Theta(n^2)$

Worst case (means the largest number of possible steps for some input)

$T(n) = an^2 + bn + c \text{ is } \Theta(n^2)$

Read more at: https://www.khanacademy.org/computing/computer-science/algorithms/insertion-sort/a/analysis-of-insertion-sort

Choice

Algorithmic Design

Correct?

How much time does the algorithm take, as a function of n?

Is there a better way?

Modified version https://en.wikipedia.org/wiki/Book#/media/File:Bucheinband.15.Jh.r.Inkunabel.jpg

Modified from https://upload.wikimedia.org/wikipedia/commons/d/d4/Milkau_B%C3%BCcherschrank_mit_angekettetem_Buch_aus_der_Bibliothek_von_Cesena_109-2.jpg

"Traditionally, routines in which the algorithm contains at least two recursive calls are called divide-and-conquer.
...
Also, the subproblems usually must be disjoint..."

"In divide and conquer,
the recursion is the divide,
the overhead is the conquer."

Running time analysis is often done by a recurrence relation

Divide and Conquer

A way to design an algorithm

Divide the problem into a number of subproblems that are smaller instances of the *same* problem.

Conquer the subproblems (typically) by solving them recursively.

Base case: If the subproblems are small enough,
just solve them by brute force.

Combine the subproblem solutions to give a solution to the original problem.

Merge sort is used in data bases!
It is a key component in Hadoop, a distributed file system.

Merge Sort

Merge Sort

43	37	20	25	4	72	15	19

DIVIDE IN HALF

43	37	20	25

4	72	15	19

CONQUER: SORT EACH HALF

20	25	37	43

↑ left

4	15	19	72

↑ right

COMBINE: MERGE

4	15	19	20	25	37	43	72

Pseudocode

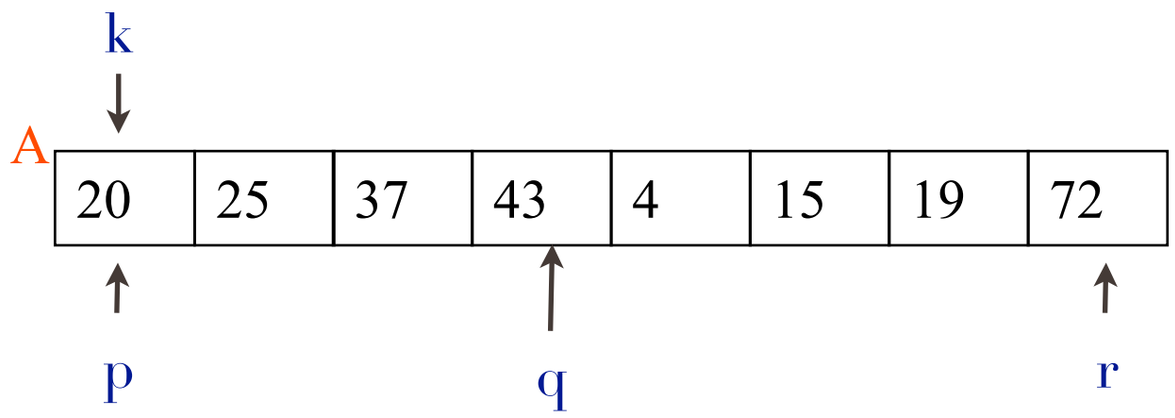
```
MERGE-SORT(A, p, r)
  if p < r                          ← check for base case
    q = ⌊(p + r)/2⌋                 ← divide
    MERGE-SORT(A, p, q)             ← conquer
    MERGE-SORT(A, q+1, r)           ← conquer
    MERGE(A, p, q, r)               ← combine
```

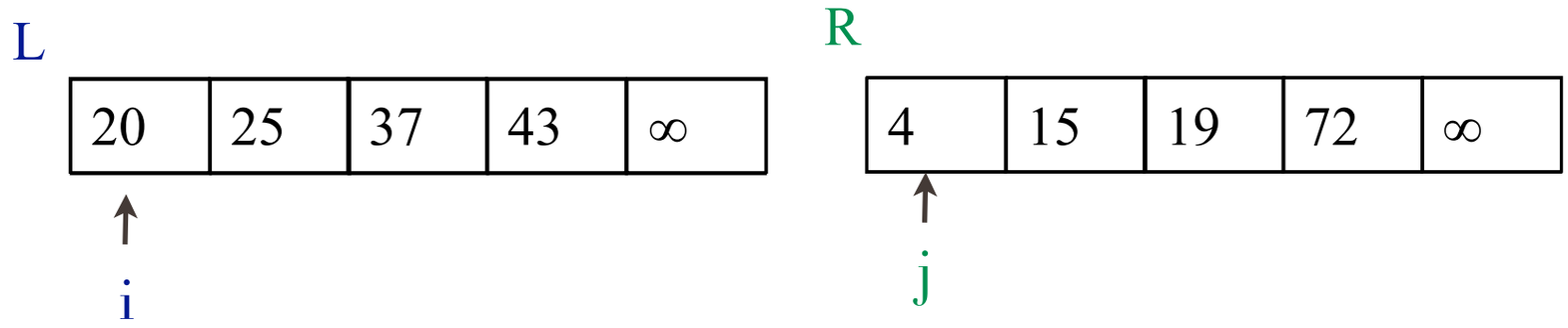
MERGE-SORT(A, 1, 1)

MERGE-SORT(A, 1, 2)

MERGE-SORT(A, 1, 4)

MERGE-SORT(A, 1, 8)





MERGE(A, p, q, r)

$n_1 = q - p + 1$
$n_2 = r - q$
let $L[1..n_1 + 1]$ and $R[1..n_2 + 1]$ be new arrays
for $i = 1$ **to** n_1
    $L[i] = A[p + i - 1]$
for $j = 1$ **to** n_2
    $R[j] = A[q + j]$
$L[n_1 + 1] = \infty$
$R[n_2 + 1] = \infty$
$i = 1$
$j = 1$
for $k = p$ **to** r
    if $L[i] \leq R[j]$
        $A[k] = L[i]$
        $i = i + 1$
    else $A[k] = R[j]$
        $j = j + 1$

See page 31 and 33 in the textbook for a proof that the merge function works correctly