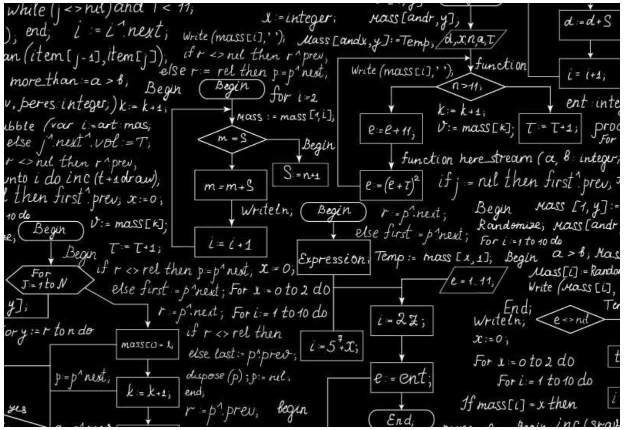
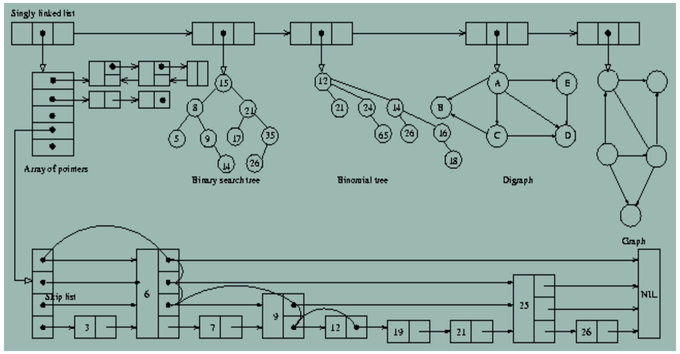
Why Data Structures?

Algorithms + Data Structures = Programs

-Niklaus Wirth



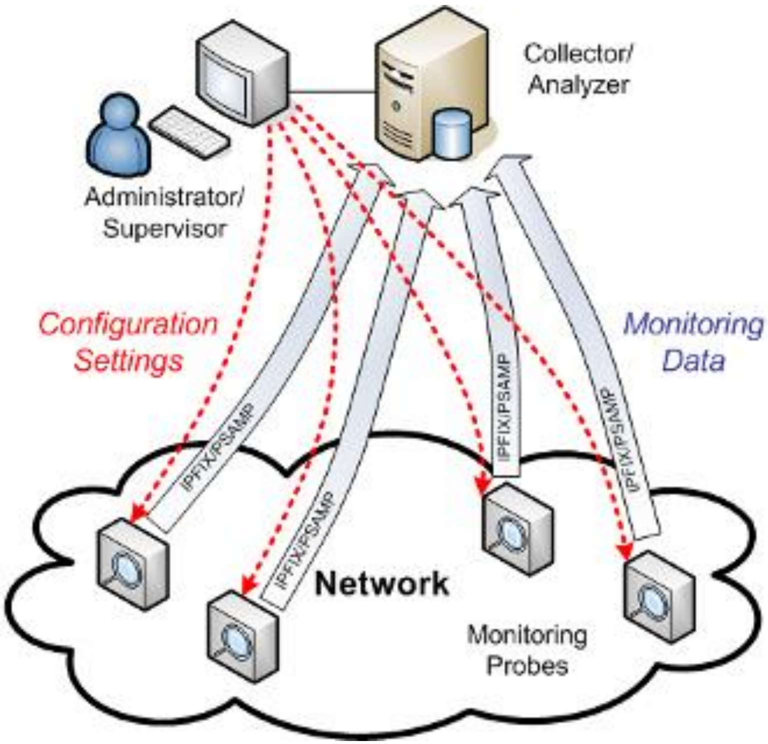


Motivating Applications

- Imagine you are in charge of maintaining a corporate network (or a major website such as Amazon)
  - High speed, high traffic volume, lots of users.
- Expected to perform with near perfect reliability, but is also under constant attack from malicious hackers
- Monitoring what is going through the network is complex:
  - Why is it slow?
  - Which machines have become compromised?
  - Which applications are eating up too much bandwidth?
  - Etc.

IP Network Monitoring

- Any monitoring software / engine must be extremely light weight, not add to the network load
  - These algorithms need smart data structures to track important statistics in real time



IP Network Monitoring

Consider this simple (toy) example:

- Is some IP address sending a lot of data to my network?
  - Which IP address sent the most data in last 1 minute?
  - How many different IP addresses in last 5 minutes?
  - Have I seen this IP address in the last 5 minutes?

- IP address format: 192.168.0.0 (10001011001…0010)
- IPv4 has 32 bits, IPv6 has 128 bits
- Cannot afford to maintain a table of all possible IP addresses to see how much traffic each is sending.
- These are data structure problems, where obvious / naïve solutions are no good, and require creative / clever ideas.

Microprocessor Profiling

- Modern microprocessors run at GHz or higher speeds
- Yet they do an incredible amount of optimization for instruction scheduling, branch prediction, etc.

- Profiling or monitoring code tracks performance bottlenecks, looks for anomalies.
  – Compute memory access statistics
  – Correlations across resources, etc.
- Simple examples:
  – Which memory locations used the most in the last 1 sec?
  – Usage map over sliding time window
- Need for highly efficient dynamic data structures

A Puzzle

- An abstraction: Most Frequent Item
- You are shown a sequence of *N* positive integers
- Identify the one that occurs most frequently

Example:

4, 1, 3, 3, 2, 6, 3, 9, 3, 4, 1, 12, 19, 3, 1, 9

- However, your algorithm has access to only O(1) memory
  - "Streaming data"
  - Not stored, just seen once in the order it arrives
  - The order of arrival is arbitrary, with no pattern
  - What data structure will solve this problem?

A Puzzle: Most Frequent Item

- Items can be source IP addresses at a router
- The most frequent IP address can be useful to monitor suspicious traffic source
- More generally, find the top *K* frequent items
  - Targeted advertising
  - Amazon, Google, eBay, Alibaba may track items bought most frequently by various demographics

Another Puzzle

- An abstraction: The Majority Item
- You are shown a sequence of *N* positive integers
- Identify the one that occurs at least *N* / 2 times

- A: 4, 1, 3, 3, 2, 6, 3, 9, 3, 4, 1, 12, 19, 3, 1, 9, 1
- B: 4, 1, 3, 3, 2, 3, 3, 9, 3, 4, 1, 3, 19, 3, 3, 9, 3

- Sequence **A** has no majority, but **B** has one (item 3)
- Can a sequence have more than one majority?

- Again, your algorithm has access to only O(1) memory
  - What data structure will solve this problem?

Solving the Majority Puzzle

- Use two variables **M** (majority) and **C** (count).
- When next item, say, **X** arrives

  - `if` **C** = 0, **M** $\leftarrow$ **X** and **C** $\leftarrow$ **1**
  - `else if` **M** = **X**, **C** $\leftarrow$ **C** + 1
  - `else` **C** $\leftarrow$ **C** - 1

- Claim: At the end of sequence, **M** is the only possible candidate for majority.
  - Note that sequence may not have any majority.
  - But if there is a majority, **M** must be it.

Examples

Try the algorithm on following data streams:

- 1, 2, 1, 1, 2, 3, 2, 2, 2, 2, 3, 2, 1
- 1, 2, 1, 2, 1, 2, 1, 2, 3, 3, 1

Proof of Correctness

- Suppose item **Z** is the majority item.
- **Z** must become majority candidate **M** at some point (why?)
- While **M** = **Z**, only non-**Z** items cause counter to decrement
- "Charge" this decrement to that non-**Z** item
- Each non-**Z** item can only cancel one occurrence of **Z**
- But in total we have fewer than N / 2 non-**Z** items; they cannot cancel all occurrences of **Z**.
- In the end, **Z** must be stored as **M**, with a non-zero count **C**.

Solving the Majority Puzzle

- False Positives in Majority Puzzle.
  - What happens if the sequence does not have a majority?
  - **M** may contain a random item, with non-zero **C**.
  - Strictly, a second pass through the sequence is necessary to “confirm” that **M** is in fact the majority.

- But in our application, it suffices to just “tag” a malicious IP address, and monitor it for next few minutes.

Back to The Most Frequent Item Puzzle

- You are shown a sequence of *N* positive integers
- Identify most frequently occurring item

Example:

4, 1, 3, 3, 2, 6, 3, 9, 3, 4, 1, 12, 19, 3, 1, 9

- Streaming model (constant amount of memory)
- What clever idea will solve this problem?

An Impossible Result

- It cannot be done with just O(1) memory!
- Computing the Most Frequent Item(MFI) requires *N* space.

- An adversary-based argument:
  - The first half of the sequence has all distinct items
  - At all but one item, say **X,** is not remembered by algorithm.
  - In the second half, all items will be distinct, except **X** which occurs twice, which should be the MFI.
  - But if it occurs too early in the sequence of the second half, it won't be counted.

Lessons for Data Structure Design

- Puzzles such as *Majority* and *Most Frequent Items* teach us important lessons:

  – Elegant interplay of data structure and algorithm

  – To solve a problem, we should understand its structure

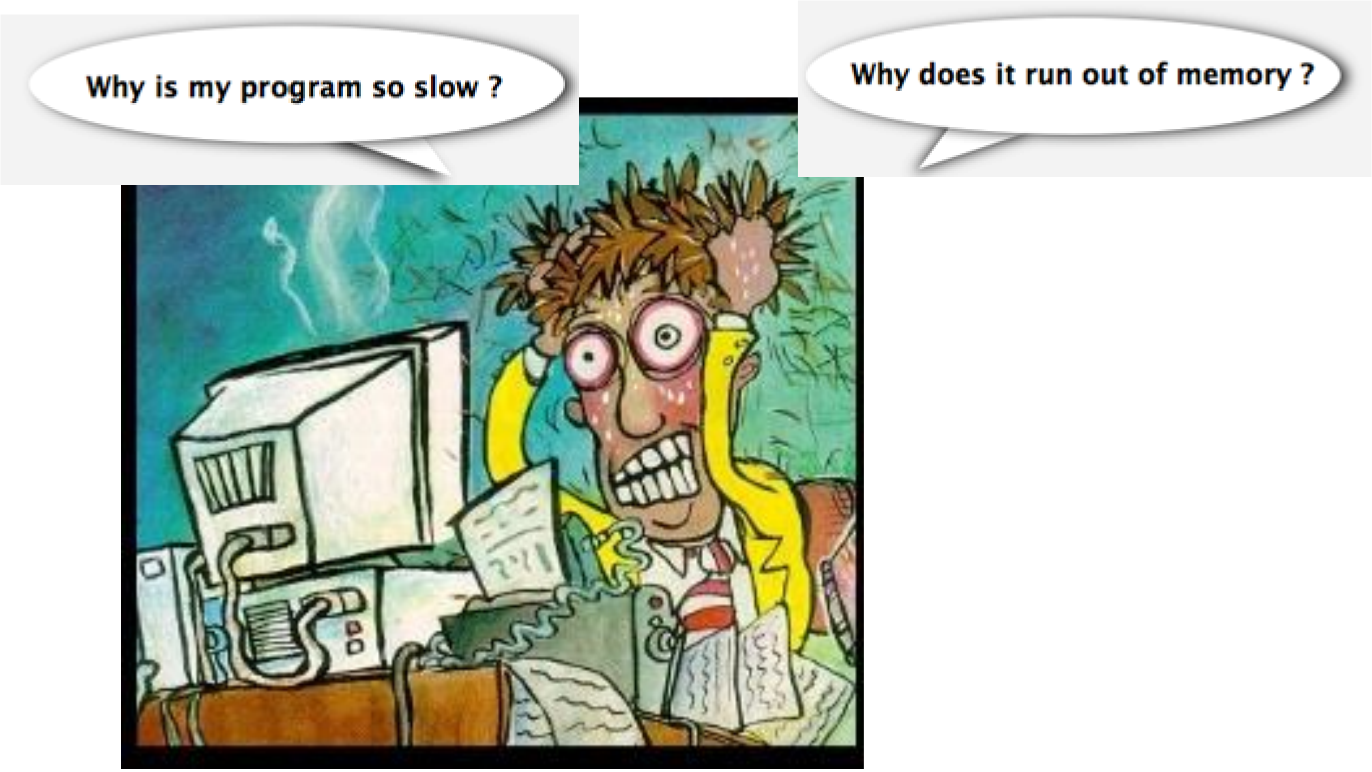
  – Correctness is intertwined with design / efficiency

  – Problems with superficial resemblance can have very different complexity

  – Do not blindly apply a data structure or algorithm without understanding the nature of the problem

Performance Bottleneck: Algorithm or Data Structure?

Will my program be able to solve a practical problem with large input?



Design and Analysis

Foundations of Algorithm and Data Structure Analysis:

- Data Structures (CS 321):
  - How to efficiently store, access, manage data?
  - How data structures effect algorithm's performance?
- Algorithm Design and Analysis (CS 421):
  - How to predict an algorithm's performance?
  - How well an algorithm scales up?
  - How to compare different algorithms for a problem?

CS 321 Course Objectives

At the end of the course, students will be:

- able to apply the most efficient known algorithms to solve searching and sorting problems.
- familiar with variety of different data structures, mainly tree structures, and their appropriate usage.
- able to choose appropriate data structures to implement certain algorithms.
- able to apply basic graph search algorithms, such as Breadth-First-Search and Depth-First-Search, to appropriate applications.

Written Assessments

- Assignments (50 %):
  - 3 Homeworks
  - 4 Programming Assignments
  - 9 Exercises
- Exams:
  - Exam 1 (15 %)
  - Exam 2 (15 %)
  - Final Exam (20%)

Grading Policy

- Homeworks will not be accepted late.
- Programming assignments:
  - must be submitted electronically to the instructor by 11.00PM of the due date to avoid any penalty.
  - Within one week after the deadline, you can still submit your assignment. However, a 20% late submission penalty will be applied.
  - No submission will be accepted after one week past the due date.

Grading Policy (cont'd)

- All students should submit correct and complete files to the instructor.
- Any accidentally wrong or incomplete submission may need to submitted again and will incur the late submission penalty.
- Any submissions, late or otherwise, that cannot be compiled or that cause runtime errors will receive 0 points.

Academic Honesty

- Each student must work independently unless specified otherwise.
- Determination of academic dishonesty is at the discretion of the instructor of the course within the policy guidelines of the University.

Example: The Sorting Problem

INPUT: A sequence of n numbers $A = (a_1, a_2, \ldots, a_n)$

OUTPUT: A permutation $A' = (a'_1, a'_2, \ldots, a'_n)$ of the input sequence such that $a'_1 \leq a'_2 \leq \ldots \leq a'_n$

Example:

- Input sequence: A = [6, 5, 3, 1, 8, 7, 2, 4]

An instance of the problem

- Output sequence: A′ = [1, 2, 3, 4, 5, 6, 7, 8]

Insertion Sort: Example

6 5 3 1 8 7 2 4

Insertion Sort: Algorithm

```
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
```

Algorithms Analysis

- Input size: size of the input
  - For the sorting problem, it is the number of items to be sorted, n
  - For the integer multiplication problem, it is the total number of bits needed to represent the integers in input
  - For graph problems, it is the number of vertices $|V|$ and the number of edges $|E|$.

Algorithms Analysis

- Running time: number of primitive operations (or steps) executed by the algorithm
  - It is a function of the input size

Insertion Sort Running Time

The cost of each step is constant

```
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
```

Insertion Sort Running Time

The *`for`* loop is executed *(n – 1)* times

```
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
```

Insertion Sort Running Time

```
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
```

The `while` loop is executed t_j times

- t_j varies with `j` and the sequence of items to be sorted

Insertion Sort Running Time

- The input size is *n* (number of items)
- The running time depends on the type of input we have:
  - <u>Best case</u>: the sequence is already sorted
  - <u>Worst case</u>: the sequence is in reverse sorted order

Insertion Sort Running Time

- Best case: the sequence is already sorted
  - the `for` loop is executed n times
  - the `while` loop is executed $t_j = 1$ time, for all $j = 2, \ldots, n$
  - The running time in this case is a linear function of n - O(n)

Insertion Sort Running Time

- Worst case: the sequence is in reverse sorted order
  - the `for` loop is executed *n* times
  - for each `j` = 2, …, *n* compare `A[j]` with each element of the sorted sub-sequence `A[1… j -1]`, then the `while` loop is executed $t_i = j$ times
  - The running time of insertion sort is which is a quadratic function of *n* – $O(n^2)$

$$\sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1$$

Insertion Sort Running Time

- In general, we are interested in the worst-case running time
  - It gives us an upper bound on the running time for any input
- Average case: But average case is often as bad as the worst case
  - For insertion sort:
    - Do j / 2 comparisons in the `while` loop, so $t_i = j / 2$ for each $j = 2, \ldots, n$
    - So, the running time is again a quadratic function of n – $O(n^2)$

Generalizing the Majority Problem

- Identify k items, each appearing more than N / $(k+1)$ times.
- Note that simple majority is the case of $k = 1$.

Generalizing the Majority Problem

- Find k items, each appearing more than N / (k+1) times.

- Use k (majority, count) tuples $(\mathbf{M_1}, \mathbf{C_1}), \ldots, (\mathbf{M_k}, \mathbf{C_k})$.
- When next item, say, **X** arrives

  - if $\mathbf{X} = \mathbf{M_j}$ for some **j**, set $\mathbf{C_j} = \mathbf{C_j} + 1$
  - elseif some counter **i** zero, set $\mathbf{M_i} = \mathbf{X}$ and $\mathbf{C_i} = 1$
  - else decrement all counters $\mathbf{C_j} = \mathbf{C_j} - 1$;

- Verify for yourselves this algorithm is correct.