Project 3 Showcase

We'll be running a Project 3 Showcase day Sunday, May 11, from 12:00-15:00 in the Woz.

Show off your project 3 to other students! Or come by and see what creative ideas were developed!

Registration link: https://forms.gle/UmAH8YkNBqm8YBa8A

Lecture 37 (Sorting 6)

Radix vs. Comparison Sorting

CS61B, Spring 2025 @ UC Berkeley

Slides credit: Josh Hug

Radix Sorting Strings

Lecture 37, CS61B, Spring 2025

Merge Sort Runtime

Merge Sort requires Θ(N log N) compares.

What is Merge Sort's runtime on strings of length W? (Are comparisons necessarily constant time anymore?)

Merge Sort Runtime

Merge Sort requires Θ(N log N) compares.

What is Merge Sort's runtime on strings of length W?

- It depends!
  - Θ(N log N) if each comparison takes constant time.
    - Example: Strings are all different in top character.
  - Θ(WN log N) if each comparison takes Θ(W) time.
    - Example: Strings are all equal.

LSD vs. Merge Sort

The facts.

- Treating alphabet size as constant, LSD Sort has runtime Θ(WN).
- Merge Sort has runtime between Θ(N log N) and Θ(WN log N).

Which is better? It depends.

- When might LSD sort be faster?
- When might Merge Sort be faster?

LSD vs. Merge Sort

The facts:

- Treating alphabet size as constant, LSD Sort has runtime $\Theta(WN)$.
- Merge Sort is between $\Theta(N \log N)$ and $\Theta(WN \log N)$.

Which is better? It depends.

- When might LSD sort be faster?
  - Sufficiently large N.
  - If strings are very similar to each other.
    - Each Merge Sort comparison costs $\Theta(W)$ time.
- When might Merge Sort be faster?
  - If strings are highly dissimilar from each other.
    - Each Merge Sort comparison is very fast.

AAAAAAAAAAAAA.......AB
AAAAAAAAAAAAA.......AA
AAAAAAAAAAAAA.......AQ
...

IUYQWLKJASHLEIUHAD...
LIUHLIUHRGLIUEHWEF...
OZIUHIOHLHLZIEIUHF...
...

Radix Sorting Integers

Lecture 37, CS61B, Spring 2025

Linear Time Sorting

Issue: We don’t have a charAt method for integers.

- How would you LSD radix sort an array of integers?

Linear Time Sorting (My Answer)

Issue: We don't have a charAt method for integers.

- How would you LSD radix sort an array of integers?
  - Could convert into a String and treat as a base 10 number. Since maximum Java int is 2,000,000,000, W is also 10.
  - Could modify LSD radix sort to work natively on integers.
    - Instead of using charAt, maybe write a helper method like getDthDigit(int N, int d). Example: getDthDigit(15009, 2) = 5.

LSD Radix Sort on Integers

Note: There's no reason to stick with base 10!

- Could instead treat as a base 16, base 256, base 65536 number.

Example: 512,312 in base 16 is a 5 digit number:

- $512312_{10} = (7 \times 16^{4}) + (13 \times 16^{3}) + (1 \times 16^{2}) + (3 \times 16^{1}) + (8 \times 16^{0})$

Note this digit is greater than 9! That's OK, because we're in base 16.

Example: 512,312 in base 256 is a 3 digit number:

- $512312_{10} = (7 \times 256^{2}) + (209 \times 256^{1}) + (56 \times 256^{0})$

Note these digit are greater than 9! That's OK, because we're in base 256.

Relationship Between Base and Max # Digits

For Java integers:

- R=10, treat as a base 10 number. Up to 10 digits.
- R=16, treat as a base 16 number. Up to 8 digits.
- R=256, treat as a base 256 number. Up to 4 digits.
- R=65336, treat as a base 65536 number. Up to 2 digits.
- R=2147483647, treat as a base 2147483647 number (this is equivalent to counting sort). Has exactly 1 digit.

Interesting fact: Runtime depends on the alphabet size.

- As we saw with city sorting last time, R = 2147483647 will result in a very slow radix sort (since it’s just counting sort).

Another Computational Experiment

Results of a computational experiment:

- Treating as a base 256 number (4 digits), LSD radix sorting integers easily defeats Quicksort.

Sort	Base	# of Digits	Runtime
Java QuickSort	N/A	N/A	10.9 seconds
LSD Radix Sort	2^4 = 16	8	3.6 seconds
LSD Radix Sort	2^8 = 256	4	2.28 seconds
LSD Radix Sort	2^16 = 65536	2	3.66 seconds
LSD Radix Sort	2^30 = 1073741824	2	20 seconds

Sorting 100,000,000 integers

Sorting Summary

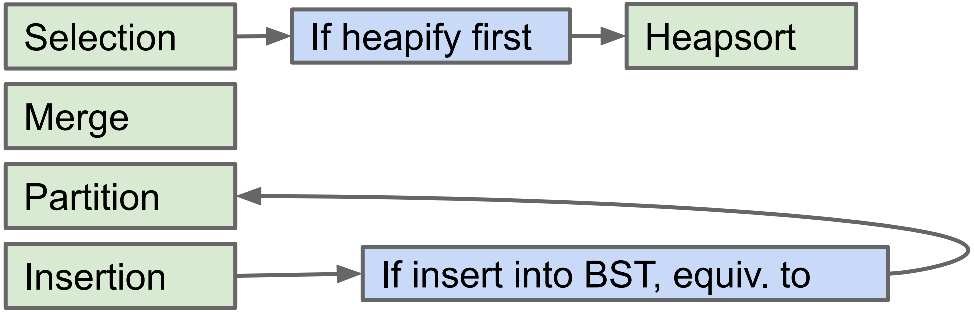
Lecture 37, CS61B, Spring 2025

Sorting Landscape

Below, we see the landscape of the sorting algorithms we've studied.

- Three basic flavors: Comparison, Alphabet, and Radix based.
- Each can be useful in different circumstances, but the important part was the analysis and the deep thought!
  - Hoping to teach you how to approach problems in general.

Comparison Based Sorting Algorithms:

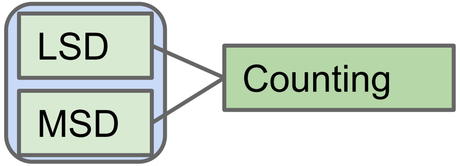


Small-Alphabet **(e.g. Integer) Sorting Algorithms**:

Counting

Radix Sorting Algorithms:

(require a sorting subroutine)



Sorting vs. Searching

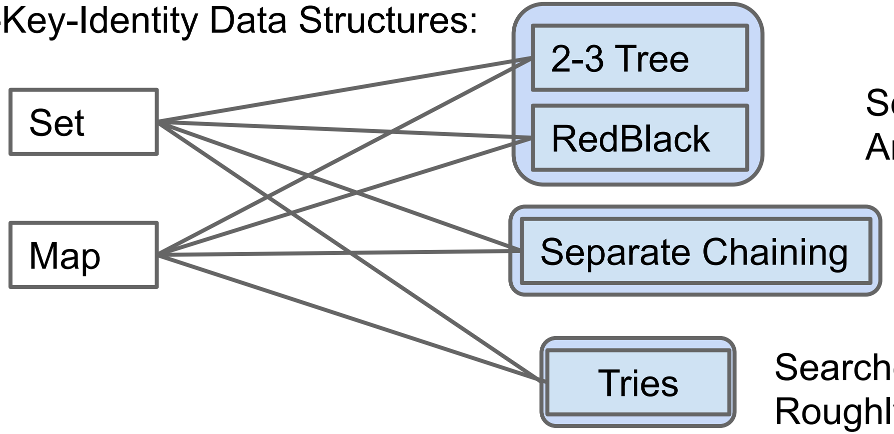
We've now concluded our study of the "sort problem."

- During the data structures part of the class, we studied what we called the "search problem": Retrieve data of interest.
- There are some interesting connections between the two.

Name	Storage Operation(s)	Primary Retrieval Operation	Retrieve By:
List	`add(key)` `insert(key, index)`	`get(index)`	index
Map	`put(key, value)`	`get(key)`	key identity
Set	`add(key)`	`containsKey(key)`	key identity
PQ	`add(key)`	`getSmallest()`	key order (a.k.a. key size)
Disjoint Sets	`connect(int1, int2)`	`isConnected(int1, int2)`	two int values

Partial list of search problem data structures.

Search-By-Key-Identity Data Structures:

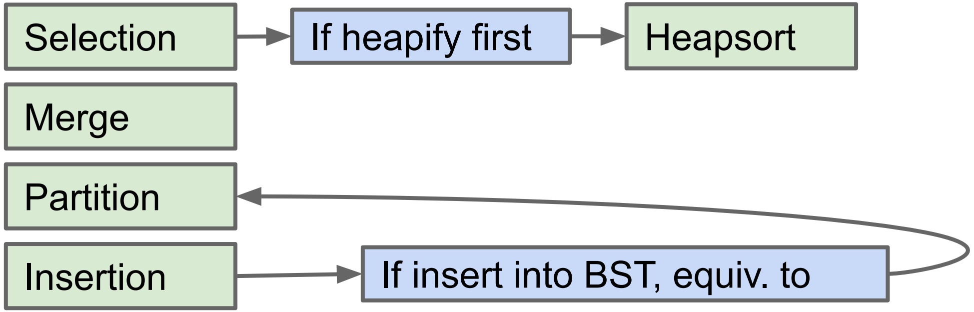


Searches using compareTo()
Analogous to **Comparison-Based**

Searches using hashCode() and equals()
Roughly Analogous to **Integer Sorting**

Searches digit-by-digit
Roughly Analogous to **Radix Sorting**

Comparison Based Sorting Algorithms:

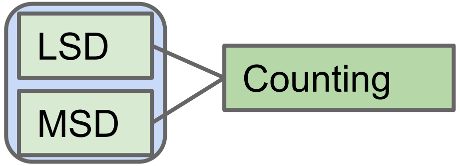


Small-Alphabet **(e.g. Integer) Sorting Algorithms**:

Counting

Radix Sorting Algorithms:

(require a sorting subroutine)



Going Even Further

There's plenty more to explore!

Many of these ideas can be mixed and matched with others. Examples:

- What if we use quicksort as a subroutine for MSD radix sort instead of counting sort?
- Implementing the `comparable` interface means an object can be stored in our compareTo-based data structures (e.g. TreeSet), or sorted with our comparison based sorts. Is there a single equivalent interface that would allow storage in a trie AND radix sorting? What would that interface look like?
- If an object has both digits AND is comparable, could we somehow use an LLRB to improve radix sort in some way?

Sounds of Sorting Algorithms

Starts with selection sort: https://www.youtube.com/watch?v=kPRA0W1kECg

Insertion sort: https://www.youtube.com/watch?v=kPRA0W1kECg&t=0m9s

Quicksort: https://www.youtube.com/watch?v=kPRA0W1kECg&t=0m38s

Mergesort: https://www.youtube.com/watch?v=kPRA0W1kECg&t=1m05s

Heapsort: https://www.youtube.com/watch?v=kPRA0W1kECg&t=1m28s

LSD sort: https://www.youtube.com/watch?v=kPRA0W1kECg&t=1m54s

MSD sort: https://www.youtube.com/watch?v=kPRA0W1kECg&t=2m10s

Shell's sort: https://www.youtube.com/watch?v=kPRA0W1kECg&t=3m37s

More sorts: https://www.youtube.com/watch?v=8MsTNqK3o_w

Questions to ponder (later… after class):

- How many items are sorted in the video for selection sort?
- Why does insertion sort take longer / more compares than selection sort?
- At what time stamp does the first partition complete for Quicksort?
- Could the size of the input used by mergesort in the video be a power of 2?
- What do the colors mean for heapsort?
- How many characters are in the alphabet used for the LSD sort problem?
- How many digits are in the keys used for the LSD sort problem?

Abstracting Data Structures

Lecture 37, CS61B, Spring 2025

Data Structures we've covered

- Lists/Deques
  - Store a collection of items in order.
  - Adding/removing from end in constant (amortized) time
  - Getting/Setting in constant time
  - Sorting in N log N time (Quicksort, Mergesort, etc.)
- Disjoint Set
  - Store a graph
  - Can determine if two nodes are connected in $\alpha(N)$ time
- Set/Map
  - Store a collection of items with no order (or key-value pairs)
  - Adding/removing any item in constant (amortized) time
  - Getting/setting in constant (amortized) time
- Heap
  - Store a collection of items
  - Adding/removing the smallest in log(N) time
- Graph
  - Store a graph
  - Dijkstra's, A*, Prim's, Kruskal's, etc.

Problems we've solved

- Given a collection of N items with an order, we can
  - Sort them in N log N time
  - Add to the collection in constant time
  - Iterate over the items with constant overhead (constant time per item, N time total)
  - Repeatedly find the smallest in log N time (if we make our adds a bit slower)
- Given a collection of N distinct items with no order, we can
  - Add to/remove from the collection in constant time
  - Iterate over the items with constant overhead
  - Associate a value to each item (with no additional overhead)
  - Check if a particular item is in there in constant time
- Given a graph, we can
  - Find the shortest path from between two vertices in E log V time
  - Find the minimum spanning tree in E log V time
  - Find whether two nodes are connected in $\alpha(N)$ time

Practice Problems

Lecture 37, CS61B, Spring 2025

How to solve Algorithm Design Problems

Algorithm Design and Asymptotic Analysis are the two most "creative" parts of 61B

- Large portion of CS theory
- First step when making a new algorithm (don't start coding until you have a plan!)
- Often tested in interviews

There's no easy "trick" to solving these; the easiest way to get better is through practice.

Fortunately, there are many programming challenge websites which you can use to practice.

- Leetcode, Codeforces: Similar to interview questions, harder ones veer into competitive programming territory
- Advent of Code: Yearly code challenge that releases one problem per day in December
- Project Euler: More mathy than the others, less focused on runtime bounds

The rest of today will be algorithm design practice (goal is to describe an algorithm within a given runtime bound)

For each problem, I'll include the approximate difficulty of coming up with the algorithm from scratch (3/5 is around the difficulty of the hardest problems we'll ask on final exams). No specific information/algorithm in this section is considered in scope.

Duplicate

You are given a list of N integers (unsorted). Determine if the list contains at least one pair of duplicates

Example:

[1,2,3,10,5,8,10] -> True

[1,4,0,-5] -> False

Runtime Requirements:

O(N log N) : ★☆☆☆☆

O(N): ★★☆☆☆

Duplicate

You are given a list of N integers (unsorted). Determine if the list contains at least one pair of duplicates

Example:

[1,2,3,10,5,8,10] -> True

[1,4,0,-5] -> False

Runtime Requirements:

O(N log N) : ★☆☆☆☆

Sort the list, then run dup2 from Lecture 15

O(N): ★★☆☆☆

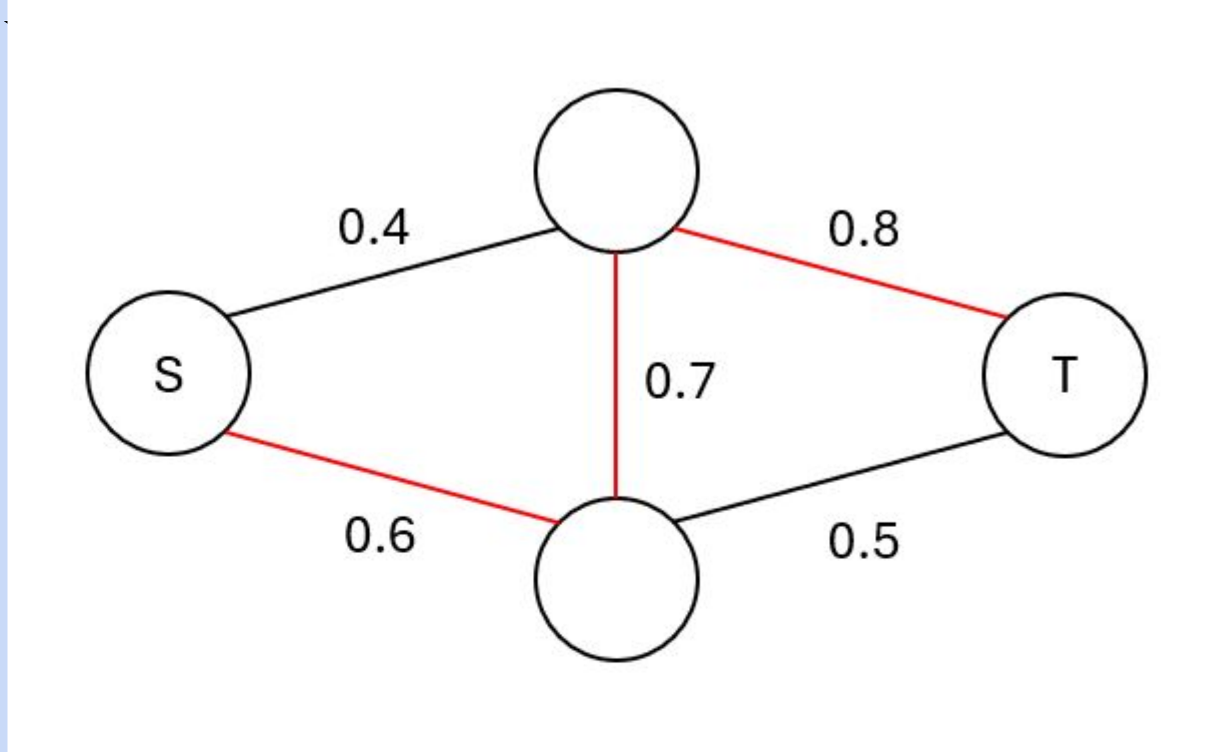
Insert all integers into a set, then check if that set contains exactly N items

Source: 61B Final Exam, Summer 2024

You are given a graph whose edges are values between 0 and 1 (not inclusive), a start vertex, and an end vertex. Find the path whose edges **multiply to the largest value.**

Example: 0.6 * 0.7 * 0.8 = 0.336, which is the largest possible value attainable in the below graph

Difficulty: ★★★☆



Source: 61B Final Exam, Summer 2024

You are given a graph whose edges are values between 0 and 1 (not inclusive), a start vertex, and an end vertex. Find the path whose edges **multiply to the largest value.**

Example: 0.6 * 0.7 * 0.8 = 0.336, which is the largest possible value attainable in the below graph

Difficulty: ★★★☆☆

Solution 1: Run Dijkstra's algorithm, but multiply instead of adding the weights, and use a max heap instead of a min heap. This works because the product of weights always decreases (you can't take an edge and increase the product).

Source: 61B Final Exam, Summer 2024

You are given a graph whose edges are values between 0 and 1 (not inclusive), a start vertex, and an end vertex. Find the path whose edges **multiply to the largest value.**

Example: 0.6 * 0.7 * 0.8 = 0.336, which is the largest possible value attainable in the below graph

Difficulty: ★★★☆☆

Solution 1: Run Dijkstra's algorithm, but multiply instead of adding the weights, and use a max heap instead of a min heap. This works because the product of weights always decreases (you can't take an edge and increase the product).

Solution 2: (★★★★☆) Create a new graph, but replace each edge of weight w with an edge of weight -log_2(w) (this is guaranteed to be positive, since the edge weights are between 0 and 1.

Minimizing the sum of -log(x) - log(y) - log(z) -...

is the same as maximizing the sum of log(x) + log(y) + log(z) +...,

which is the same as maximizing log(xyz…),

which is the same as maximizing xyz…

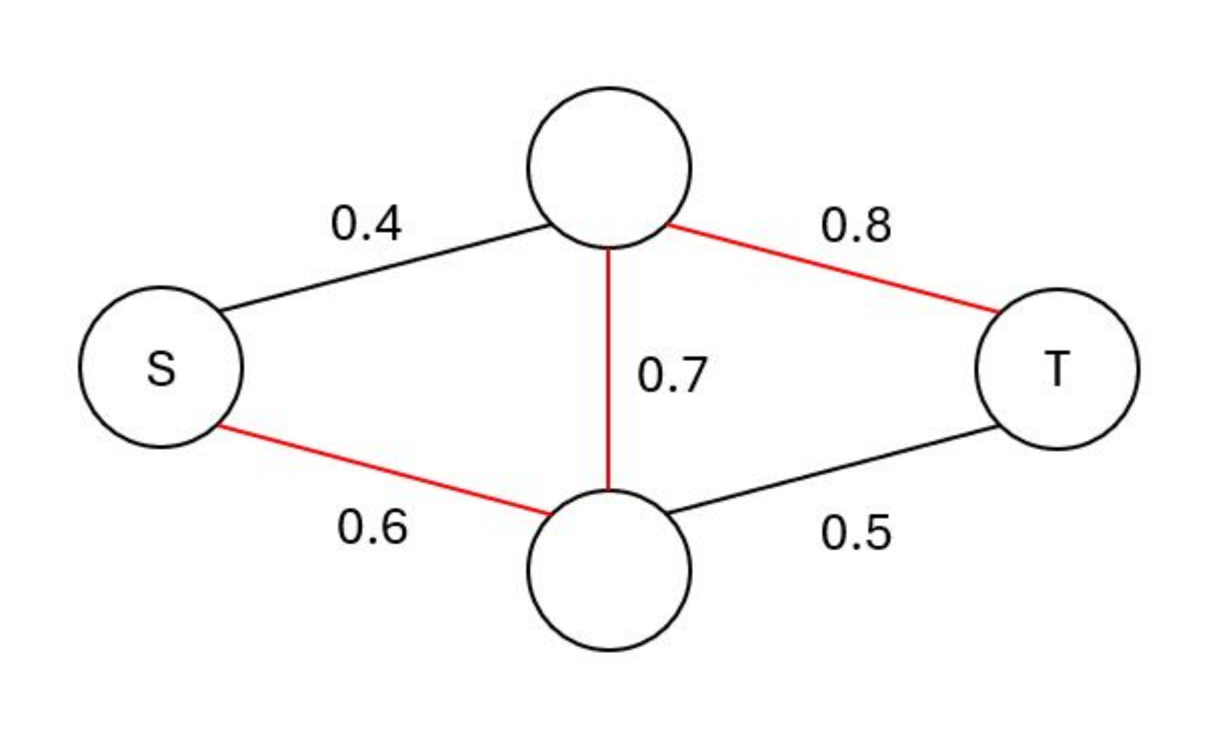
So if we run Dijkstra's on the new graph, we'll get the correct path.

A Slight Modification

You are given a graph whose edges are values between 0 and 1 (not inclusive), a start vertex, and an end vertex. Find the path whose edges **add to the largest value without visiting any node more than once.**

Example: 0.6 + 0.7 + 0.8 = 0.21, which is the largest possible value attainable in the below graph

Difficulty: ★★★★



A Slight Modification

You are given a graph whose edges are values between 0 and 1 (not inclusive), a start vertex, and an end vertex. Find the path whose edges **add to the largest value without visiting any node more than once.**

Example: 0.6 + 0.7 + 0.8 = 0.21, which is the largest possible value attainable in the below graph

Difficulty: ★★★★★★★★★★★★★★★★★★★★★★★★★★★

Currently no known solution in polynomial time (and if you do solve this in polynomial time, you earn $1000000)

Just like in asymptotic analysis, it's often hard to determine the difference between an easy and a hard problem.

Why do you earn $1,000,000 for solving this problem?

Reducing Hamilton Path

The **Hamilton Path Problem** is the problem of, given a graph, determining if there is a path that goes through all vertices exactly once.

Problem X is the problem of, given a graph whose edges are values between 0 and 1 (not inclusive), a start vertex, and an end vertex, finding the path whose edges add to the largest value without visiting any node more than once.

Show that the Hamilton Path Problem **reduces** to Problem X; that is, if you find an efficient solution to Problem X, then you have an efficient solution to the Hamilton Path Problem (★★★☆☆)

Reducing Hamilton Path

The **Hamilton Path Problem** is the problem of, given a graph, determining if there is a path that goes through all vertices exactly once.

Problem X is the problem of, given a graph whose edges are values between 0 and 1 (not inclusive), a start vertex, and an end vertex, finding the path whose edges add to the largest value without visiting any node more than once.

Show that the Hamilton Path Problem **reduces** to Problem X; that is, if you find an efficient solution to Problem X, then you have an efficient solution to the Hamilton Path Problem (★★★☆☆)

Create a new graph with the same topology as the original graph, but with edge weights 0.5. Add two new "dummy" nodes S and T that are connected to all other nodes with weight 0.1 Run Problem X on this new graph from S to T, and return True if the path returned has length 0.2+0.5x(|V|-1)

Beans and Plates

A line of W plates are placed in a row, and numbered 1, 2, …, W.

You're given N pairs of numbers (a,b), 1≤a≤b≤W.

For each pair of numbers, you place one bean on each plate from plate a to plate b (inclusive).

Example:

If W = 5, and you're given the pairs (2,5), (1,4), (3,3), the plates look like:

[0,0,0,0,0]->[0,1,1,1,1]->[1,2,2,2,1]->[1,2,3,2,1]

Find the total number of beans on all plates in O(NW) time (★☆☆☆☆)

Find the total number of beans on all plates in O(N) time (★★★☆☆)

Find the number of plates with an odd number of beans on them in O(N log N) time (★★★★★)

Beans and Plates

A line of W plates are placed in a row, and numbered 1, 2, ..., W.

You're given N pairs of numbers (a,b), $1 \le a \le b \le W$.

For each pair of numbers, you place one bean on each plate from plate a to plate b (inclusive).

Example:

If W = 5, and you're given the pairs (2,5), (1,4), (3,3), the plates look like:

[0,0,0,0,0]->[0,1,1,1,1]->[1,2,2,2,1]->[1,2,3,2,1]

Find the total number of beans on all plates in O(NW) time (★☆☆☆☆)

Simulate the procedure on an array of length W

Find the total number of beans on all plates in O(N) time (★★★☆☆)

Each pair adds b-a+1 beans in total. Sum this over all pairs

Beans and Plates

A line of W plates are placed in a row, and numbered 1, 2, …, W.

You're given N pairs of numbers (a,b), 1≤a≤b≤W.

For each pair of numbers, you place one bean on each plate from plate a to plate b (inclusive).

Example:

If W = 5, and you're given the pairs (2,5), (1,4), (3,3), the plates look like:

[0,0,0,0,0]->[0,1,1,1,1]->[1,2,2,2,1]->[1,2,3,2,1]

Find the number of plates with an odd number of beans on them in O(N log N) time (★★★★★)

Create a list lst, and insert into the list a and b+1 for each pair (In the example, we insert 2,6,1,5,3,4).

Sort this list. This creates a list of every time the number of beans in the list changes from odd to even or vice versa. So the odd-bean plates are the ones between lst[0] and lst[1]-1, lst[2] and lst[3]-1, etc. Sum these values together.