Quiz 1 Review

High Level

- Need to solve large problems n with constant-sized code, **correctly** and **efficiently**
- Analyzing running time: **How to count?**
  - **Asymptotics**
  - **Recurrences** (substitution, tree method, Master Theorem)
  - **Model of computation**: Word-RAM, Comparison
- How to solve an algorithms problem
  - Reduce to a problem you know how to solve
    - Use an algorithm you know (e.g. **sort**)
    - Use a data structure you know (e.g. **search**)
  - Design a new recursive algorithm (harder, mostly in 6.046)
    - Brute Force
    - Decrease & Conquer
    - Divide & Conquer (like merge sort)
    - Dynamic Programming (later in 6.006!)
    - Greedy/Incremental

Algorithm: Sorting

Reduce your problem to a problem you already know how to solve using known algorithms. You should know **how** each of these sorting algorithms are implemented, as well as be able to **choose** the right algorithm for a given task.

Algorithm	Time $O(\cdot)$	In-place?	Stable?	Comments
Insertion Sort	n^2	Y	Y	$O(nk)$ for k-proximate
Selection Sort	n^2	Y	N	$O(n)$ swaps
Merge Sort	$n \log n$	N	Y	stable, optimal comparison
AVL Sort	$n \log n$	N	Y	good if also need dynamic
Heap Sort	$n \log n$	Y	N	low space, optimal comparison
Counting Sort	$n + u$	N	Y	$O(n)$ when $u = O(n)$
Radix Sort	$n + n \log_n u$	N	Y	$O(n)$ when $u = O(n^c)$

Data Structures

Reduce your problem to using a data structure storing a set of items, supporting certain search and dynamic operations efficiently. You should know **how** each of these data structures implement the operations they support, as well as be able to **choose** the right data structure for a given task.

Sequence data structures support **extrinsic** operations that maintain, query, and modify an externally imposed order on items.

Sequence Data Structure	Operations $O(\cdot)$				
	Container	Static	Dynamic		
	`build(X)`	`get_at(i)` `set_at(i,x)`	`insert_first(x)` `delete_first()`	`insert_last(x)` `delete_last()`	`insert_at(i,x)` `delete_at(i)`
Array	n	1	n	n	n
Linked List	n	n	1	n	n
Dynamic Array	n	1	n	$1_{(a)}$	n
Sequence AVL	n	$\log n$	$\log n$	$\log n$	$\log n$

Set data structures support **intrinsic** operations that maintain, query, and modify a set of items based on what the items are, i.e., based on the **unique key** associated with each item.

Set Data Structure	Operations $O(\cdot)$				
	Container	Static	Dynamic	Order	
	`build(X)`	`find(k)`	`insert(x)` `delete(k)`	`find_min()` `find_max()`	`find_prev(k)` `find_next(k)`
Array	n	n	n	n	n
Sorted Array	$n \log n$	$\log n$	n	1	$\log n$
Direct Access	u	1	1	u	u
Hash Table	$n_{(e)}$	$1_{(e)}$	$1_{(a)(e)}$	n	n
Set AVL	$n \log n$	$\log n$	$\log n$	$\log n$	$\log n$

Priority Queues support a limited number of Set operations.

Priority Queue Data Structure	Operations $O(\cdot)$			
	`build(X)`	`insert(x)`	`delete_max()`	`find_max()`
Dynamic Array	n	$1_{(a)}$	n	n
Sorted Dyn. Array	$n \log n$	n	$1_{(a)}$	1
Set AVL	$n \log n$	$\log n$	$\log n$	$\log n$
Binary Heap	n	$\log n_{(a)}$	$\log n_{(a)}$	1

Problem Solving

Testing Strategies

- Read every problem first, rank them in the order of your confidence
- For most problems, you can receive $\geq 50\%$ of points in two sentences or less
- Probably better to do half the problems well than all the problems poorly

Types of problems

Type	Internals	Externals	Tests understanding of:
Mechanical	Y	N	how core material works
Reduction	N	Y	how to apply core material
Modification	Y	Y	how to adapt core material (augmentation, divide & conquer, amortization, etc.)

Questions to ask:

- Is this a Mechanical, Reduction, or Modification type problem?
- Is this problem about data structures? sorting? both?
- If data structures, do you need to support **Sequence** ops? **Set** ops? both?
- If stuck, is there an easy way to get a correct but inefficient algorithm?

Question yourself if you are:

- Trying to compute decimals, rationals, or real numbers
- Using Radix sort for every answer
- Augmenting a binary tree with something other than a subtree property

Data Structures Problems

- First solve using Sorting or Set/Sequence interfaces, choose algorithm/data structure after
- Describe all data structure(s) used (including **what data they store**) and their invariants
- Implement **every operation** we ask for in terms of your data structures
- Separate and label parts of your solution!

Problem 1. Restaurant Lineup (S19 Q1)

Popular restaurant Criminal Seafood does not take reservations, but maintains a wait list where customers who have been on the wait list longer are seated earlier. Sometimes customers decide to eat somewhere else, so the restaurant must remove them from the wait list. Assume each customer has a different name, and no two customers are added to the wait list at the exact same time. Design a database to help Criminal Seafood maintain its wait list supporting the following operations, each in $O(1)$ time. State whether each operation running time is worst-case, amortized, and/or expected.

`build()`	initialize an empty database
`add_name(`x`)`	add name x to the back of the wait list
`remove_name(`x`)`	remove name x from the wait list
`seat()`	remove and return the name of the customer from the front of the wait list

Solution: Maintain a doubly-linked list containing customers on the wait list in order, maintaining a pointer to the front of the linked list corresponding to the front of the wait list, and a pointer to the back of the linked list corresponding to the back of the wait list. Also maintain a hash table mapping each customer name to the linked list node containing that customer. To implement `add_name(`x`)`, create a new linked list node containing name x and add it to the back of the linked list in worst-case $O(1)$ time. Then add name x to the hash table pointing to the newly created node in amortized expected $O(1)$ time. To implement `remove_name(`x`)`, lookup name x in the hash table in and remove the mapped node from the linked list in expected $O(1)$ time. Lastly, to implement `seat()`, remove the node from the front of the linked list containing name x, remove name x from the hash table, and then return x, in amortized expected $O(1)$ time.

Problem 2. Rainy Research (S19 Q1)

Mether Wan is a scientist who studies global rainfall. Mether often receives data measurements from a large set of deployed sensors. Each collected data measurement is a triple of integers (r, ℓ, t), where r is a positive amount of rainfall measured at latitude ℓ at time t. The **peak rainfall** at latitude ℓ **since** time t is the maximum rainfall of any measurement at latitude ℓ measured at a time greater than or equal to t (or zero if no such measurement exists). Describe a database that can store Mether's sensor data and support the following operations, each in worst-case $O(\log n)$ time where n is the number of measurements in the database at the time of the operation.

`build()`	initialize an empty database
`record_data(`r, ℓ, t`)`	add a rainfall measurement r at latitude ℓ at time t
`peak_rainfall(`ℓ, t`)`	return the peak rainfall at latitude ℓ **since** time t

Solution: Maintain a Set AVL tree L storing distinct measurement latitudes, where each latitude ℓ maps to a rainfall Set AVL tree $R(\ell)$ containing all measurement triples with latitude ℓ, keyed by time. We only store nodes associated with measurements, so the height of each Set AVL tree is bounded by $O(\log n)$. For each rainfall tree, augment each node p with the maximum rainfall $p.m$ of any measurement within p's subtree. This augmentation can be maintained in constant time at a node p by taking the maximum of the rainfall at p and the augmented maximums of p's left and right children (if they exist); thus this augmentation can be maintained without effecting the asymptotic running time of standard AVL tree operations.

To implement `record_data(`r, ℓ, t`)`, search L for latitude ℓ in worst-case $O(\log n)$ time. If ℓ does not exist in L, insert a new node corresponding to ℓ mapping to a new empty rainfall Set AVL tree, also in $O(\log n)$ time. In either case, insert the measurement triple to $R(\ell)$, for a total running time of worst-case $O(\log n)$.

To implement `peak_rainfall(`ℓ, t`)`, search L for latitude ℓ in worst-case $O(\log n)$ time. If ℓ does not exist, return zero. Otherwise, perform a one-sided range query on $R(\ell)$ to find the peak rainfall at latitude ℓ since time t. Specifically, let $\text{peak}(v, t)$ be the maximum rainfall of any measurement in node v's subtree measured at time $\geq t$ (or zero if v is not a node):

$$\text{peak}(v, t) = \begin{cases} \max\{v.item.r, v.right.m, \text{peak}(v.left, t)\} & \text{if } v.t \geq t \\ \text{peak}(v.right, t) & \text{if } v.t < t \end{cases}.$$

Then peak rainfall is simply $\text{peak}(v, t)$ with v being the root of the tree, which can be computed using at most $O(\log n)$ recursive calls. So this operation runs in worst-case $O(\log n)$ time.

Note, this problem can also be solved where each latitude AVL tree is keyed by rainfall, augmenting nodes with maximum time in subtree. We leave this as an exercise to the reader.