0.1. General Approaches

- There will be two sections:
  - The multiple choice section will evaluate your comprehension of material from the course. To study for this section, you should make sure you understand all of the slides and lecture notes. The approach I would take is not memorizing details (e.g. how to perform a RBT recoloring) because this is always information that you can lookup. Instead, I would focus on high-level takeaways from algorithms (e.g. How does randomized quicksort use randomness to thwart adversaries? What are the tradeoffs between pivot selection approaches?) To answer these types of questions, obviously you need a good understanding of how the algorithms work, but you certainly don’t need to memorize pseudocode.
  - The algorithm analysis and design will evaluate your ability to design algorithms of each of the five varieties. To study for this section, you should make sure you can answer the homework questions independently. If any of our solutions seems like we grabbed them from “thin-air”, ask us how we arrived at that solution!

0.2. Algorithmic Analysis

- **Asymptotics.** Questions involving asymptotics will be simple and/or well-motivated. I won’t ask for an asymptotic bound between two weird functions, but I will ask for an asymptotic bound of algorithms that you write.
- **Proofs of correctness.** Understand how a proof by induction works. We’ve seen several examples this quarter to review, including the proof of correctness for `insertion_sort`, `merge_sort`, `quicksort`, `selection_sort` (HW #1, Q3).
- **Best case/worst-case runtime vs. Big-$\Omega$/Big-$O$.** We can describe the best case runtime with a Big-$\Omega$ or Big-$O$ asymptotic bound.

```plaintext
algorithm insertion_sort(list A):
    for i = 0 to length(A)-1:
        let cur_value = A[i]
        let j = i - 1
        while j >= 0 and A[j] > cur_value:
            j = j - 1
        A[j+1] = cur_value

The best-case runtime is $\Theta(n)$. The worst-case runtime is $\Theta(n^2)$.
```

0.3. Divide-and-conquer

- **Compute runtime for a divide-and-conquer algorithm.** This involves writing and solving a recurrence relation. Approach: Try Master Method. If it doesn’t work because of unequal problem sizes, try substitution method; else, try recursion tree or iteration methods (whichever you prefer).
– **Master Method.** While I won’t ask you to rederive the Master Method, I think understanding its derivation helps with solving recurrences with recursion tree. (Lecture 2)

– **Inductive proof for substitution method.** Using fat base-cases, inductive hypothesis, and finishing the proof (Section 7/17). Recall that you cannot disprove a runtime with substitution method because you might have chosen the wrong form of the runtime as your inductive hypothesis i.e. $kn$ instead of $k_1n + k_2$ or $k_1n + k_2n^{1/2} + k_3$ (HW #1, Q6c).

- **Linear-time selection with select_k.** How does this it work? What are tradeoffs between different pivot selection approaches? Why are we looking for the median-of-medians? How do we prove its runtime? Where does the $n/5$ and $7n/10$ come from in the recurrence relation?

- **Tips for designing a divide-and-conquer algorithm.**
  - Think about how to decompose a problem into similar, smaller problems; this will be your recursive case.
  - Think about trivial problems you know how to solve; these will be your base case.

0.4. Comparison Lower Bounds/Linear Time Sorting

- **Lower bounding things.** Our $\Omega(n \log(n))$ lower-bound on comparison-based sorting algorithms is a good example of us lower-bounding things. Why are there $O(n!)$ leaves in the decision tree? Why must it be the case that we consider all of them? We’ve had several other examples of lower-bounding things (HW #2, Q3d; HW #2, Q6c; HW #4, Q4b).

- **Linear-time sorting.** Why does the comparison-based sorting lower bound not apply here? When can we use these algorithms? When are they useful?

- **Bucket sort.** How does it work? What happens if there aren’t enough buckets? What does it mean for bucket sort to be a stable sort?

- **Radix sort.** How does it work? Do we bucket sort from least to most significant or most to least significant? When would you prefer not to use it?

0.5. Trees

- **Fundamentals.** Be able to code up pre-order (root, left, right), post-order (left, right, root), in-order (left, root, right) traversals. What is a binary tree? What is a binary search tree? How can you find predecessors or successors? What are bounds for the height of a binary tree with $n$ vertices?

- **Red-Black Trees.** Why are they useful? How do they compare to other data structures for storing items? How do its properties guarantee that it remain balanced? Why do we need to recolor?

0.6. Randomized algorithms

- **Slow sorting with bogosort.** Why would we never use this sorting algorithm in practice? What about it makes its worst-case runtime $O(\infty)$?
• **Quick sorting with quicksort.**
  - If you understand this proof of correctness, then I think you’re solid with proving correctness by induction since both the main theorem and the lemma (proof of correctness of the partition function) involve induction.
  - How did we arrive at $O(n \log(n))$-time expected runtime? And what does this even mean?
  - What is an advantage of using randomness to randomly select the pivot, as opposed to choosing the first element of the sublist? Can we generalize this principle to randomized algorithms as a whole? What do we stand to gain by using a randomized algorithm?
  - Expected runtime over the inputs vs. on a specific worst-case input.

• **Quick selecting with quickselect.**
  - Does there exist a single worst-case input? How do we get the worst-case runtime?
  - How did we solve for the expected runtime? How did using phases help us?

• **Global min-cut with karger.** How does it work? How does repeating Karger’s algorithm again and again help? Is this a general approach that can be used for Monte Carlo algorithms? How did we arrive at the $1/nC^2$ bound? How did we

• **Tips for designing a randomized algorithm.**
  - List out options about what could be randomly chosen, and think about whether it would help to do so.
  - For each thing that could be randomly chosen, think about how to randomly choose it.
  - If you’re having trouble designing it from scratch, design a non-randomized algorithm first and then try to convert that non-randomized algorithm into a randomized algorithm. This will allow you see all of the variables in play and decide which ones would be good to choose randomly (Lecture 4, majority element is a great example of this approach).

• **Hashing basics.** How do we construct a hash table? Where does the “randomness” come from? How does the randomness that we introduce inoculate us from adversarial inputs? How does hashing relate more generally to all randomized algorithms? Does hashing save us from worst-case inputs?

• **Universal Hash Families.** What is a universal hash family? Why is the definition $\Pr\{h(x) = h(y)\} \leq 1/n$ i.e. upper bound for probability of collision instead of $\Pr h(x) = i \leq 1/n$ i.e. upper bound for probability of getting hashed to a specific bucket? Functional knowledge: be able to identify when a hash family is universal. What are some examples of universal hash families? How do they compare to one another? Why might you prefer to use the $\mod p \mod n$ has family over the exhaustive set?

0.7. Graph algorithms

• **BFS/DFS.** What can you do with both? What can you only do with BFS? What can you only do with DFS? What are some applications in which BFS and DFS are useful? You should be able to write pseudocode for these algorithms from scratch.
• **Dijkstra’s algorithm for single-source shortest path.** How does it work? What invariant is being maintained? What greedy decision is being made? No need to memorize the proof of correctness, but I think being able to reason about it is helpful to understanding why it works. You should be able to write pseudocode for this algorithm from scratch.

• **Kosaraju’s algorithm for finding SCCs.** How does it work? What is the SCC metagraph?

• **Bellman-Ford.** What are advantages of this over running Dijkstra’s? What conditions must be true for this to work? How does it detect cycles?

• **Floyd-Warshall.** What are advantages of this over running Bellman-Ford for all vertices as source? What about doing the same with Dijkstra’s? What is the invariant? How is it different from Bellman-Ford’s invariant? How does it relate to 0/1 Knapsack’s invariant? What’s the table look like?

0.8. Greedy algorithms

• **Proving correctness.** You can use one of two techniques: greedy stays ahead or greedy exchange. I’m mostly been pushing for greedy exchange since I think it’s more easily applied to new instances, but you can use whichever one that pleases you. The general idea with greedy exchange argument is as follows:
  – Assume your greedy algorithm has made the same choices $S$ as an optimal solution $S$ up to some step $i$ (i.e. $i$ items have been added to $S$).
  – Exchange the next choice made by $S$ with the next choice made by your greedy algorithm.
  – Prove that you still have made the same choices as some other optimal solution. To do this, you explicitly construct $S$ to be the choices made by your greedy algorithm up to step $i + 1$ and then use the choices made by $S$ for the rest, and prove that this new solution $S$ is optimal/legal.

We’ve seen a couple examples at this point (Frog Hopping in Lecture, HW #5, Q3).

• **Tips for designing a greedy algorithm.**
  – If you were to implement a “quick-and-dirty” solution without regard for whether it was correct or not, what would it be? It’s probably the greedy solution.
  – Think about maximizing the local utility derived from making this decision (as opposed to delaying gratification by making a suboptimal choice now in order to make a super optimal choice later).

0.9. Dynamic Programming

• **Knapsack.** What are the differences between unbounded and 0/1 knapsack? Which one is harder and why?

• **Tips for designing a dynamic programming algorithm.**
  – Follow the four steps.
– The hardest steps are finding the optimal substructure and defining the recursive formulation. To find the optimal substructure, I like to think about what information would be nice to know at a given step. For example, in LCS, I’d choose some subproblem, say, the longest common subsequence of the first $i$ characters of the first word and the first $j$ characters of the second word. At this step, I know there can be two cases on the character that I’m looking at: either they’re the same or different. The same case is easy. The LCS of these strings must be the LCS of the first $i - 1$ characters of the first word and the first $j - 1$ characters of the second word. If they’re not the same, then I realize that either it might be the first string’s fault (if the first string was like `aaaaabaa` and the second word was `aaaaaaa`, and $i = j = 4$). It also might be the second string’s fault.

After defining the optimal substructure, I go to defining the recursive formulation. This step is just a mathematical translation of what you already know. It should involve some subset of max, min, sum, over sum set of values.

– Practice! (Google Dynamic Programming questions, and try at least three different ones. If you’re having trouble designing the optimal substructure or recursive formulation, I’ll try to do it and write out my thought process as I’m going.)

0.10. Additional topics

• You should probably take 30 min to understand what each of the complexity classes P, NP, NP-complete, and NP-hard mean and have a general understanding of the problems we used to study approximating NP-hard problems. Again, these topics will comprise $\frac{1}{5}$ of the exam.

• I wouldn’t spend much time learning to argue approximation schemes/fixed-parameter tractability/FPTAS for the final.

• Know the definition of max-flow and min-cut, that they are equivalent, and how Ford-Fulkerson works from a high level.

0.11. Thank you!