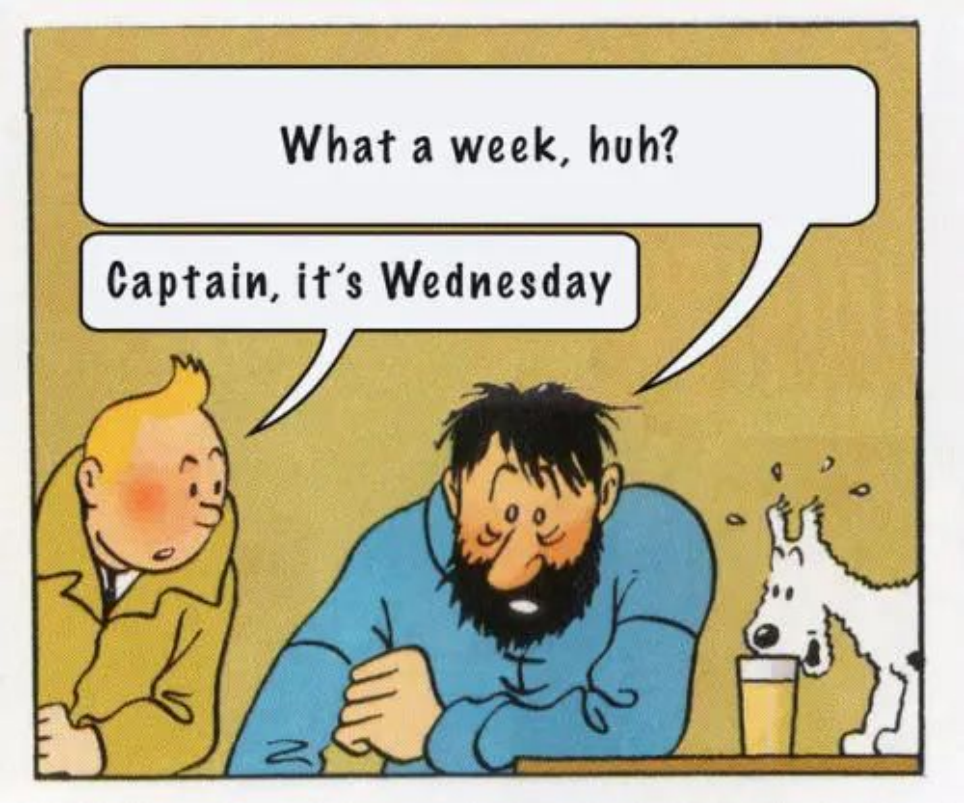
7/20: Midterm Review!



Divide and Conquer

Sorting &
Randomization

Data Structures

Graph Search

Dynamic Programming

Greed & Flow

Special Topics

Format

- 85 minutes
- Work briskly – there's not a lot to write, but there are many small separate problems to think about
  - Don't get so hung up on one that you don't attempt others!
  - General Stanford CS advice: don't leave anything blank
- The exam has a lot of opportunities to show depth of understanding. It's challenging but not intended to be gratuitously so, and I'll take the challenge into account grade-wise.

Lecture 1 Key Points

- The point of big-O notation (and big-Ω, and big-Θ) is to suppress multiplicative constants and lower-order terms.

- Know how to show something is – or isn't – big-O of something else. (like HW1 Question 1)

- Example: Show that n^n is not $\Theta(n!)$

Show: n^n is not $\Theta(n!)$

- $f(n)$ is $\Theta(g(n))$ if and only if it is both $O(g(n))$ and $\Omega(g(n))$. So here it suffices to show that one of these is false.

- Compare terms to get an idea:
  - 9^9 = `9 * 9 * 9 * 9 * 9 * 9 * 9 * 9 * 9`
  - $9!$ = `9 * 8 * 7 * 6 * 5 * 4 * 3 * 2 * 1`

- It sure looks like n^n dominates $n!$ term-by-term, and is not $O(n!)$, but we need to show this formally...

Show: n^n is not $O(n!)$

Suppose that n^n were $O(n!)$. Then there would exist c, n_0 such that for all $n \geq n_0$, $n^n \leq c \cdot n!$.

Notice that this means $c \geq \frac{n^n}{n!} = \frac{n \cdot n \cdot \ldots \cdot n}{n \cdot (n-1) \cdot \ldots \cdot 1} = \frac{n}{n} \cdot \frac{n}{n-1} \cdot \ldots \cdot \frac{n}{1}$.

Notice that the first term is 1, every other term is strictly greater than 1, and the last term is n.

But this means that if we just choose $n = c + 1$, the inequality is clearly false!

(Subtlety: Because we might not have $c + 1 \geq n_0$, we instead choose $n = \max(c + 1, n_0)$.)

Lecture 2 Key Points

- Understand the cases of the Master Theorem (HW1): relationship between proliferation of the *number* of subproblems and shrinking of the *size* of individual subproblems. Does one of these factors win, making the tree top-heavy or bottom-heavy? Or do these factors balance out?

The Master Theorem

For recurrences of the form $T(n) = aT(\frac{n}{b}) + O(f(n))$,

$$T(n) = \begin{cases} O(n^d \log(n)) & \text{if } a = b^d \\ O(n^d) & \text{if } a < b^d \\ O(n^{\log_b(a)}) & \text{if } a > b^d \end{cases}$$

where a is the number of sub-problems, b is the factor by which the input size shrinks, and n^d is the work required to create all the sub-problems and combine their solutions. (Note that the big-O values here are tight; for instance, if $\Theta(n^2)$ work is required, use $d = 2$.)

- Use the Master Theorem to understand why Karatsuba's and Strassen's (from HW1) algorithms are asymptotically faster than naive integer / matrix multiplication.

Lecture 2 Key Points

- Understand how MergeSort works, and in particular the Merge step. Example: Does this implementation of the Merge step lead to MergeSort being **stable**? (NOTE: you would not have to read a specific language on an exam)

```
def merge(left_list, right_list):
    left_ptr = 0
    right_ptr = 0
    new_list = []
    while left_ptr < len(left_list) and right_ptr < len(right_list):
        if left_list[left_ptr] < right_list[right_ptr]:
            new_list.append(left_list[left_ptr])
            left_ptr += 1
        else:
            new_list.append(right_list[right_ptr])
            right_ptr += 1
    if left_ptr == len(left_list):
        new_list.extend(right_list[right_ptr:])
    else:
        new_list.extend(left_list[left_ptr:])
    return new_list
```

```
def merge(left_list, right_list):
    left_ptr = 0
    right_ptr = 0
    new_list = []
    while left_ptr < len(left_list) and right_ptr < len(right_list):
        if left_list[left_ptr] < right_list[right_ptr]:
            new_list.append(left_list[left_ptr])
            left_ptr += 1
        else:
            new_list.append(right_list[right_ptr])
            right_ptr += 1
    if left_ptr == len(left_list):
        new_list.extend(right_list[right_ptr:])
    else:
        new_list.extend(left_list[left_ptr:])
    return new_list
```

A problem – if these two elements are tied, it will actually take the one from right_list! A stable sort should keep tied elements in their original order.

We'd need to use <= to make this stable.

Lecture 3 Key Points

- Understand how k-Select and Partition work, and the parts of the recurrence in k-Select.

$$T(n) \leq T(n/5) + T(7n/10) + O(n)$$

What do these three parts come from?

Lecture 3 Key Points

- Understand how k-Select and Partition work, and the parts of the recurrence in k-Select.

$$T(n) \leq T(n/5) + T(7n/10) + O(n)$$

recursive call to k-Select to find the median of medians

recursive call to k-Select on the part of the list we know has the element we want

cost of finding the medians of all the little groups of 5, and of Partitioning around the estimated median of medians

Lecture 3 Key Points

- Understand how k-Select and Partition work, and the parts of the recurrence in k-Select.

$$T(n) \leq T(n/5) + T(7n/10) + O(n)$$

recursive call to k-Select to find the median of medians

recursive call to k-Select on the part of the list we know has the element we want

cost of finding the medians of all the little groups of 5, and of Partitioning around the estimated median of medians

Know how to show this is $O(n)$ using the substitution method.

Lecture 3 Key Points

- Understand RadixSort well enough to be able to walk through a simple example.

- Know that RadixSort isn't really "O(n)" when the number of digits d is itself O(log n).
  - e.g., for $b = 2$, $d = 3$, the only possible values are 000, 001, 010, 011, 100, 101, 110, 111. If we sort a list of distinct values, then $3 = d = \log_2 n = \log_2 2^3 = 3$.

- Understand the idea that a comparison-based sort method can't asymptotically beat n log n, because we have O(n!) leaves into a tree of decisions (one per sorting order), and the depth ends up having to be $\Omega(n \log n)$.

Lecture 4 Key Points

- When analyzing randomized algorithms, the expectation is over the randomness of the algorithm's choices, not the input. We assume the worst-case (bad guy) input. (See the $(1/2)^{100}$ vs. $(1/2)^{101}$ part of HW3 Problem 3)

- Understand how to use indicator variables, and that (for any variables, not just indicators), the expectation of a sum of variables (even if they're not independent!) is the sum of the expectations of the variables.

- Review the analysis of QuickSort, although you are not responsible for reproducing it. Understand the ideas behind its parts.

A kinda bad True/False question that I cut

The probability that QuickSort attains its worst-case $O(n^2)$ running time is exactly $2^{n-1} / n!$

A kinda bad True/False question that I cut

The probability that QuickSort attains its worst-case $\Theta(n^2)$ running time is exactly $2^{n-1} / n!$

If we want the probability that we always pick the worst-case pivot, then we have a 2/n chance on the first pick (i.e., we pick either end), then a 2/(n-1) chance on the next pick, and so on. So that expression above looks right, and giving this justification would have earned a lot of partial credit…

A kinda bad True/False question that I cut

The probability that QuickSort attains its worst-case $\Theta(n^2)$ running time is exactly $2^{n-1} / n!$

If we want the probability that we always pick the worst-case pivot, then we have a 2/n chance on the first pick (i.e., we pick either end), then a 2/(n-1) chance on the next pick, and so on. So that expression above looks right, and giving this justification would have earned a lot of partial credit…

But the premise is wrong! The algorithm doesn't have to make the very *worst decisions to have an $\Theta(n^2)$ runtime. So the real probability is higher.*

A kinda bad True/False question that I cut

The probability that QuickSort attains its worst-case $\Theta(n^2)$ running time is exactly $2^{n-1} / n!$

If we want the probability that we always pick the worst-case pivot, then we have a 2/n chance on the first pick (i.e., we pick either end), then a 2/(n-1) chance on the next pick, and so on. So that expression above looks right, and giving this justification would have earned a lot of partial credit...

But the premise is wrong! The algorithm doesn't have to make the very *worst decisions to have an $\Theta(n^2)$ runtime. So the real probability is higher. This felt too sneaky, and also, it's a little vague to say when an individual runtime crosses over from the n* log *n to the n^2 mark, so to speak.*

Lecture 4 Key Points

- Understand what Karger's algorithm does (and the central idea of edge contractions), and also be familiar with the analysis (it's not as bad as the one for QuickSort, and it's on the exam reference page).

Lecture 5 Key Points

- Understand how to use hash tables as accessories that help solve problems like 2-SUM.
- Understand the chaining implementation of hash tables.
- Make sure you're totally clear on what the parts of the definition of universal hashing mean...

Universal hashing

Let $\mathcal{U}$ be the universe of all elements we could put in our hash functions, and let $\mathcal{H} = \{h_1, ..., h_m\}$ be a family of hash functions hashing these elements to n buckets. Then $\mathcal{H}$ is universal if and only if for every pair u_i, u_j of distinct elements in $\mathcal{U}$,

$$\mathbf{Pr}_{h \text{ in } \mathcal{H}}[h(u_i) = h(u_j)] \leq \frac{1}{n}$$

where the probability is over a (uniformly) random choice of a hash function h from the family $\mathcal{H}$.

Lecture 5 Key Points

Example: Suppose that there exists a pair of values *x*, *y* in the universe for which any two hash functions in a family *H* hash *x* and *y* to different buckets. Is *H* universal?

Universal hashing

Let $\mathcal{U}$ be the universe of all elements we could put in our hash functions, and let $\mathcal{H} = \{h_1, ..., h_m\}$ be a family of hash functions hashing these elements to n buckets. Then $\mathcal{H}$ is universal if and only if for every pair u_i, u_j of distinct elements in $\mathcal{U}$,

$$\mathbf{Pr}_{h \text{ in } \mathcal{H}}[h(u_i) = h(u_j)] \leq \frac{1}{n}$$

where the probability is over a (uniformly) random choice of a hash function h from the family $\mathcal{H}$.

Lecture 5 Key Points

Example: Suppose that there exists a pair of values *x*, *y* in the universe for which any hash function in a family *H* hashes *x* and *y* to different buckets. Is *H* universal? Not necessarily! We need to demonstrate that the collision probability is low for *any* such pair.

Universal hashing

Let $\mathcal{U}$ be the universe of all elements we could put in our hash functions, and let $\mathcal{H} = \{h_1, ..., h_m\}$ be a family of hash functions hashing these elements to n buckets. Then $\mathcal{H}$ is universal if and only if for every pair u_i, u_j of distinct elements in $\mathcal{U}$,

$$\mathbf{Pr}_{h \text{ in } \mathcal{H}}[h(u_i) = h(u_j)] \leq \frac{1}{n}$$

where the probability is over a (uniformly) random choice of a hash function h from the family $\mathcal{H}$.

Lecture 5 Key Points

- You do not need to know modular arithmetic for the midterm, but there could still be a universal hashing question that isn't about that.

- Understand how Bloom filters work, e.g., in the basic scenario in HW3 Problem 2 (not the more complicated one with multiple filters). In what sense do they "fill up"?

Lecture 6 Key Points

- Be able to walk through inserting into (and deleting the minimum from) min-heaps implemented as arrays, as seen in class. Create and practice with some examples.

- Understand the basic properties of trees, what makes a binary search tree special, and what makes a balanced binary search tree special.

- Be able to use the rules for red-black trees (given on the exam) to talk about legal and illegal trees. You don't need to worry about the mechanics of inserting or deleting, since we didn't cover the rotation rules, but know the time bounds.

Lecture 6 Key Points

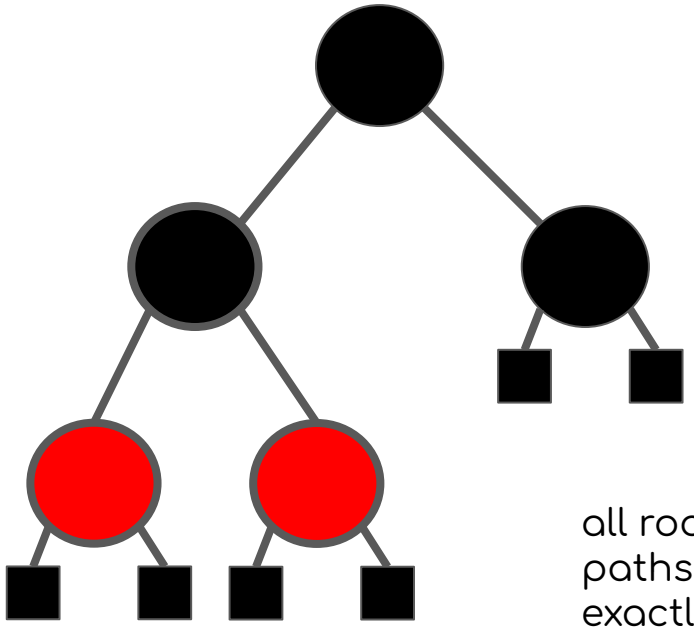
Example: In the implementation of min-heaps we saw, they are restricted to having certain tree shapes (all complete binary trees except perhaps for the last level). Is it always possible for these same shapes to have their nodes colored to be valid red-black trees?

Red-black tree rules

- Every node is colored red or black.
- The root node is a black node.
- Any missing children (a leaf has two of these, an internal node with only one child has one of these) are NILs that count as black nodes.
- Children of a red node are black nodes.
- For any node x, all paths from x to NILs have the same number of black nodes.

Lecture 6 Key Points

Yes – the idea of the implementation of min-heaps is to keep them balanced, which is also what red-black trees are trying to do. It turns out to be easy to do this coloring – make every node black, unless there is an incomplete bottom level, in which case we make all of those nodes red. Then all of the rules are satisfied.



all root-to-NIL paths have exactly 3 black nodes

Lecture 7 Key Points

- Know how you would implement BFS to solve problems like searching for a node, finding shortest path lengths, counting connected components, and even finding shortest total distances in graphs with small weights.

- Understand Dijkstra's Algorithm well enough to work through an example, and know where the $n \log n$ in its running time comes from. Know why we use a special heap and what it is good at (decrease-key).
  - You do not need to know the proof of correctness of Dijkstra's or the details of Fibonacci heaps.

- No need to know amortization for the midterm. Also, all graphs on the midterm are undirected.

Lecture 7 Key Points

Example: Why would it be inefficient to use Dijkstra's Algorithm if you knew your graph was a weighted tree?

Lecture 7 Key Points

Example: Why would it be inefficient to use Dijkstra's Algorithm if you knew your graph was a weighted tree?

The power of Dijkstra's is to find a shortest path out of many possible paths, but in a tree, there is exactly one path between any two nodes. We might as well use BFS to find this path and then calculate the total cost of its parts – we would not need to use the heap of Dijkstra's algorithm, so we would dodge the $O(n \log n)$ part of the cost.

We didn't talk in class about how we would modify BFS to find actual paths, but it's not hard (each node stores the node from which it was discovered)

Final Thoughts

- Remember, don't spend too much time writing out unnecessarily complex answers. Even a bulleted list is fine as long as it's clear that you have the key ideas.
- Work at a good pace and don't get hung up for too long on anything. Most questions are of similar point value, so focus on getting as many as you can.
- Try not to worry about what score corresponds to what overall grade, etc., and just do your best! You're gonna crush it!