Lecture 1: Introduction

Objectives of this lecture

- Cover the course schedule and syllabus.
- Get started on dynamic programming and optimizations.
- Some high level points on how (not to) design algorithms.

1 Course Information

Website: `https://www.cs.cmu.edu/~yangp/15-451/`

2 Philosophy

Consider the following problem:

Problem: Optimal Leaf-Binary-Search-Tree (leaf-BST, meme name: Rock Combine)

Start with a sequence/list of n numbers. Every step, you can merge two neighboring numbers and replace by their sum, at cost of the new number created (aka. the sum). Find the minimum total cost to reduce everything to one number.

This problem is the same as finding an optimal binary search tree with all the entries at leaves in the order given. Here 'optimal' is measured in the sum over depths and weights: the weights can be viewed as the frequency by which the keys are accessed. This equivalence can be sketched as:

- each merger creates a new node with the previous two as children,
- the cost accounts for everything there getting deeper by 1,
- the restriction of merging neighbors only means the the ordering of leafs is preserved.

Note that the last item is where things differ from Huffman tree, which doesn't require the leafs are ordered. That is called the optimal binary tree problem. An input where the answers to these two problems differ is $< 1, 10, 1 >$: if we can ignore key orderings, we'd merge the two 1s together first.

The merge neighbors form, is easier to write a dynamic program. The key observation is that any number created in some intermediate point of the algorithm comes from an interval of the original list. This leads to a DP state where $DP[l][r]$ is the minimum cost to turn a range of numbers $[l, r]$ into a single number.

$$OPT_{i,j} = \begin{cases} 0 & \text{if } i = j \\ \sum_{i \le k \le j} a_k + \min_{i \le k < j} OPT_{i,k} + OPT_{k+1,j} & \text{otherwise} \end{cases}$$

This gives an $O(n^3)$ algorithm. This is usually step 1 towards designing a good algorithm: find an appropriate structure that one can mathemtically reason about. It corresponds to the first portion of this class, on dynamic programming.

Things below are for enrichmentn only. It (optimal leaf BST faster than n^2) will not be in any assessment.

The next step is to make these faster. We will systematically discuss these approaches in the data structures / amortized analysis portion of the course. This usually involves proving more things based on the structure identified earlier.

That is, we can treat the optimal k for each value of i and j as an array itself:

$$k^*_{ij} \triangleq \arg\min_{i \le k < j} OPT_{i,k} + OPT_{k+1,j}.$$

A somewhat intuitive, but not easy to prove fact is

Lemma 1: Monotonicity of Decision Point

or any $i < j < \hat{j}$, it holds that

$$k^*_{i,j} \le k^*_{i,\hat{j}}$$

A quick summary of using this fact for speedups:

1. $O(n^2 \log n)$ solution
   (a) Use the fact that for any $i < j < \hat{j}$, $k^*_{i,j} \le k^*_{i,\hat{j}}$
   (b) For each set of intervals of length L, compute the k^* for each one using binary search
   (c) Since computing each with binary search takes $O(n \log n)$ time and there are n total lengths for the intervals so the total complexity is $O(n^2 \log n)$
2. $O(n^2)$ solution
   (a) Use the fact that $k^*_{i,j-1} \le k^*_{i,j} \le k^8_{i+1,j}$

(b) Since $i, j-1$ and $i+1, j$ are shorter intervals, we can find $k^*_{i,j}$ by checking values from $k^*_{i,j-1}, k^*_{i+1,j}$.

(c) Getting all intervals of length L therefore requires $O(n)$ iterations:

$$\sum_i \left(k^*_{i+1,i+L} - k^*_{i,i+t-1} + 1\right) \leq O(n)$$

(d) Therefore computing all intervals takes $O(n^2)$

Both of these methods require looking at all intervals of length t together at once: note that because the transition formula only goes to smaller lengthed intervals, this plus going through ls in increasing order removes all issues with needing values of not-yet-computed states.

The first is to binary search: relabel all the intervals of length L to

$$1, 2, \ldots (n-L+1),$$

and abbreviate $k^*_{i,i+L-1}$ as k^*_i since we are only looking at things of the same lengths. Note that k^*_i is also (non-strictly) monotonically increasing in i. So once we figure out k^*_{mid} (for the middle one), we have:

$$k^*_i \leq k^*_{mid} \qquad \forall i \leq mid$$

so we can search only to the left of k^*_{mid} for indices $1 \ldots mid$, and only to the right of k^*_{mid} for indices $mid \ldots t$. The pseudocode for this is:

```
FindOptK($l, r, k^*_l, k^*_r$)
    1. If $r \leq l + O(1)$, find by brute force.
    2. Let $mid \leftarrow \lfloor (l+r)/2 \rfloor$.
    3. Compute $k^*_{mid}$ by looping over all of $[k^*_l, k^*_r]$.
    4. FindOptK($l, mid, k^*_l, k^*_{mid}$)
    5. FindOptK($mid, r, k^*_{mid}, k^*_r$)
```

This completely splits up the n possible candidates for k^*_i. Furthermore, this division process only goes $O(\log n)$ levels since the range of i we look for halves at each step. So the total cost is $O(n \log n)$. We will see such complex divide-and-conquer repeatedly in this course, although not in exactly format.

The second is to use the alternate inequality

$$k^*_{i,j-1} \leq k^*_{i,j} \leq k^*_{i+1,j}$$

to partition up the search space for i, j. Note that both $i, j-1$ and $i+1, j$ are shorter so the total number of things looked at is

$$\sum_i \left(k^*_{i+1,i+t} - k^*_{i,i+t-1} + 1\right) \leq n + n \leq O(n).$$

Can we do even faster? Dynamic programming seems doomed now because there are n^2 states.

This version, up to here, can be found at `https://atcoder.jp/contests/dp/tasks/dp_n`.

What about doing even faster?

Note that we are almost at the limit of the interval based structure identified earlier: there are just n^2 different intervals.

To go further, we need to start proving some states are not necessary. This often requires trying to formulate the problem using more advanced mathematical tools. The one that we will use the most is linear programming: they will be discussed at the start of the second portion of the class.

What about greedy?

A natural approach is to always merge together the two neighbors with smallest sum. This, in fact, works when the initial sequence can be permuted arbitrarily (it is known as Huffman code). However, the restriction of only merging neighbors makes this suboptimal though, consider for example

$$10, 9, 9, 10.$$

How to fix this? This natural next question leads to what I feel is one of the most important takeaways from this course:

AVOID PROPOSING GREEDY ALGORITHMS IF POSSIBLE

I suggest everyone try to converse with neighbors for 10 minutes to come up with something that the course staff cannot find an example to break...

I'm also quite sure LLMs can search up this algorithm by now.

Fix: Make things that have been merged 'transparent'. That is, once the two 9s above merged (into 18), the two 10s can 'see' each other through this new number, and merge.

I hope this illsutrates why it's better to go dynamic programming first: non-DP based approaches are so open-ended that one can easily spend a term trying to make something work. I suggest we all follow the 'do not attempt greedy until question explicitly tells you to' approach/agreement... this also means we will clearly indicate when a question involves greedy.

It turns out, after a lot of work, that the total cost of this equals to the optimal merging cost... Several things remain:

1. Recover a proper merging sequence from the output of this greedy process.
2. Implement this process in $O(n \log n)$ time (using some really fancy priority queues).
3. Prove that nothing can get better costs than this.

If you want a version of this that LLMs have a very hard time with, same problem, but on a cycle. Circle-square greedy generalizes naturally to it, but I have not been able to find a counter example, or attempted to prove its correctness.

If you want an open problem, remove the restriction that keys must sit on leaves. This is called the optimal search tree problem, and it's open whether there is faster than n^2 for it. Some recent progresses on it: `https://arxiv.org/abs/1505.00357`.

Finally, note that it's usually not the case that we know the frequency of accesses before calling the binary tree. This leads to the notion of online algorithms, where one is working with partial information. The wikipedia article on optimal binary trees has a good starting point for these: `https://en.wikipedia.org/wiki/Optimal_binary_search_tree#Dynamic_optimality`.