DISCLAIMER: These notes are not necessarily an accurate representation of what I said during the class. They are mostly what I intend to say, and have not been carefully edited.

Goal of this lecture is to discuss how algorithmic efficiency is closely connected with recursion. Recall binary trees, we first show that once we can look at the whole sequence, balancing was never necessary.

Formally, we want to support the dynamic rank problem:

1. Insert some $x$

2. Find how many things are less than $x$.

The standard approach for this is to put the elements into a balanced binary tree. Then strange sequences like getting the $x$s in order causes the height of the tree to become large, after which much fun/headache ensue.

The idea of this week’s lecture is: what if we know all the $x$s ahead of time. Then we do not need to rebalance the tree, ever: we can sort all the $x$ values, and build a static binary tree on all the values.

Then in each node, we store a counter of how many nodes below it have been inserted: this values starts at 0, and anytime we insert $x$, we find $x$, and increment this counter for all its ancestors.

The other way to view this is we build a trie on the bit representation of the numbers: we access a node by reading its bit representation from front to back: left child corresponds to 0, right child corresponds to seeing a 1. In this representation, it’s also easier to not have keys in the middle of the tree: all important information sit on the leaves of such a tree.

This ability to avoid balancing is an important phenomenon. Balanced binary trees, as they are typically taught, are examples of online data structures: there is no knowledge of what future operations look like. If instead, we are given all future operations, it’s significantly easier for us to organize the search tree, or data structures in general.

In fact, in many cases, it’s even possible for us to completely omit data structures, by doing recursion instead. Here one thing that’s often easier to do is a 1-layer recursion that gets $O(\sqrt{n})$, instead of $O(\log n)$, per operation. For example, if we want to completely replace the BBST in the ranking problem, we can look at the indices of the next $\sqrt{n}$ accesses and queries, and:

1. Statically compute, based on the state of things so far, the answers to all the queries in this sublist ignoring all updates over the next $\sqrt{n}$ operations.
2. For each ‘new’ update, compute its effect on all the rank queries in this smaller list. The first operation takes $O(n)$, but is ran only $\sqrt{n}$ times, while the second operation takes $O(\sqrt{n})$ per query. So the cost still sums to $O(n^{1.5})$ total.

Observe that step (2) above is actually the same problem. This is because the rank problem is decomposable. That is, if we ‘zoom in’ on part of the problem (by statically solving the part before it), it becomes the same problem. This type of behavior tends to automatically give $O(n \log n)$ type algorithms.

For another example of a decomposable problem, consider axis-aligned box queries.

**Problem 0.1.** Given some points in 2D, support:

1. Insert a point.

2. Query how many points are in some rectangular box $[x_l, x_h] \times [y_l, y_h]$.

First, consider just the second operation. By doing inclusion/exclusion on four boxes, we can assume $x_l, y_l = 0$.

Then consider the median $x_h$ value, which we call $x_{mid}$. For all $x_h \geq mid$, we need to include the box

$$[0, x_{mid}] \times [0, y_h]$$

which is a 1-D query, involving all the points with $x \leq x_{mid}$. This can be answered in $O(n)$ time, after which we are left with two disjoint problems, one on all the points with $x < x_{mid}$, one on all points with $x > x_{mid}$. So we get the runtime recurrence

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$$

that solves to $O(n \log n)$.

Note that this generalizes to higher dimensions: in $d$ dimensions, the offline problem essentially needs $d - 1$ levels of divide-and-conquer, giving a runtime of $O(n \log^{d-1} n)$. Now we need to incorporate the updates.

**Claim 0.2.** Time is basically another dimension.

This is because if a point is inserted at time $t_1$ counts for a query is at time $t_2$ if and only if $t_1 < t_2$. So for the above 2-D problem, we just add an additional dimension, and turn things into a 3-D static query problem.

Note so far the updates and queries are static: we can decompose arbitrarily. A more sophisticated way of using data structures is the subsequent updates depend on results of previous queries. Here divide-and-conquer can still be used, as long as one divide on time first.

**Problem 0.3.** Given points $(x[1], y[1]) \ldots (x[n], y[n]) \in \mathbb{R}^2$, with associated values $v[i]$, solve the dynamic program:

$$DP[i] = v[i] + \min_{j < i} \{DP[j] + |x[i] - x[j]| + |y[i] - y[j]|\}$$

in $O(n \log^2 n)$ time.
The idea is to divide on time first. Let \( \text{mid} = n/2 \), we first solve \( x[1] \ldots x[\text{mid}] \) first, then handle all transitions from \( 1 \ldots \text{mid} \) to \( (\text{mid} + 1) \ldots n \). and then recurse on the second half.

To handle the recursion, note that if \( x[j] < x[i] \), the value is just \( x[i] - x[j] \), so we can group the \(-x[j]\) term together with \( DP[j] \), to \( DP[j] - v[j] \). Then (by some symmetry stuff) we’re effectively solving the transition

\[
\]

For this, we do another divide-and-conquer on the \( x \) values: do all the transitions from \( x[j] < \text{mid} \) to \( x[i] > \text{mid} \). Then the problem of solving

\[
\min_j DP[j] + |y[i] - y[j]|
\]

can be solved by sweeps in \( O(n) \) time each. As we did two layers of divide-and-conquer, the total runtime is \( O(n \log^2 n) \).

In my opinion, this type of approach simplifies graph data structures the most.

**Problem 0.4.** For an undirected weighted graph, and a sequence of reweighting updates, compute the weight of the max weight spanning tree after each update.

This is [https://codeforces.com/problemsets/acmsguru/problem/99999/529](https://codeforces.com/problemsets/acmsguru/problem/99999/529) There is a guide for this at [https://codeforces.com/blog/entry/16967](https://codeforces.com/blog/entry/16967)

First let’s try to get a \( n^{1.5} \) time algorithm for this. We take the edges involved in the next \( n^{0.5} \) modifies, remove them. Then the other edges are permanent during those steps, so we can just take their MST, \( T \). Furthermore, we then take the end points of the vertices involved in these \( n^{0.5} \) updates: if we remove some edge from \( T \), it must be the min weight edge on some path of \( T \). There are \( O(n^{0.5}) \) vertices, and we can use the following fact:

**Lemma 0.5.** Given a tree on \( n \) vertices, \( k \) of which are special, repeatedly performing:

1. remove a degree 1 non-special vertex,
2. contract away a degree 2 non-special vertex,

leaves us with \( O(k) \) vertices at the end.

Using this fact here brings us to \( O(n^{0.5}) \) vertices, so a total time of \( O(n^{1.5}) \). A multi-layer version of this gives \( O(n \log n) \).