02-613 Week 6

Algorithms and Advanced Data Structures

Aidan Jan

September 29, 2025

Divide and Conquer

Divide and Conquer is a class of algorithms that involve the principle of cutting a problem into smaller parts, solving the smaller instance of the problem, and pieceing it back together.

Merge Sort

One of the most-studied algorithms that fall into the divide and conquer category is merge sort, a sorting algorithm that essentially goes as follows:

- Cut the array in half
- Recursively sort the left half and the right half
- Put the array back together.

```
MergeSort(L):
    if |L| = 1: return L
    elif |L| = 2: return [min(L), max(L)]
    else:
        L1 = MergeSort(L[0: |L| / 2])
        L2 = MergeSort(L[|L| / 2: |L|])
        return combine(L1, L2)
```

The runtime of this algorithm is $\mathrm{O}(n \log n)$.

Runtime Proof

We can prove the runtime of merge sort (like most other divide and conquer algorithms) by mathematical induction.

1. Show $T(k) \leq f(k)$ for "small" k
2. Assume $T(k) \leq f(k)$ for $k < n$
3. Show $T(n) \leq f(n)$

For step 1, if $k = 2$, then $c \cdot 2 \log 2 = 2c \leq T(n)$, where c is a constant. Now, assuming $k = n$,

$$\begin{aligned} T(n) &\leq 2T\left(\frac{n}{2}\right) + cn \\ &\leq 2\left(c\frac{n}{2}\log\left(\frac{n}{2}\right)\right) + cn \\ &\leq cn\log\left(\frac{n}{2}\right) + cn \\ &\leq cn\log n - cn\log 2 + 2cn \\ &\leq cn\log n \end{aligned}$$

It turns out this math would also work out for any running time larger than $n \log n$, such as n^2. However, we only care about the tightest runtime we can assign. It will not work for $O(n)$.

- Recall that $O(n \log n)$ is actually also $O(n^2)$.

The Master Theorem

For divide and conquer cases, we can typically write the runtime of the worst case as

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

where a is the number of subproblems the main problem gets split into per iteration, and b is the size of the subproblem relative to the original. For example for merge sort, $a = 2$ and $b = 2$, since every iteration splits into 2 subproblems, each half the size of the original. Finally, $f(n)$ is the runtime of the combine operation. For merge sort, $f(n) = n$.

The master theorem states that:

- If $n^{\log_b a} > f(n)$, then the final runtime would be $n^{\log_b a}$.
- If $f(n) > n^{\log_b a}$, then the final runtime would be $f(n)$.
- If $f(n) = n^{\log_b a}$, then the final runtime would be $f(n) \log(n^{\log_b a})$.

Closest Two Points on a Plane

Now a more practical application of divide and conquer: given a set of points on a 2D plane, find the pair of points that is closest together.

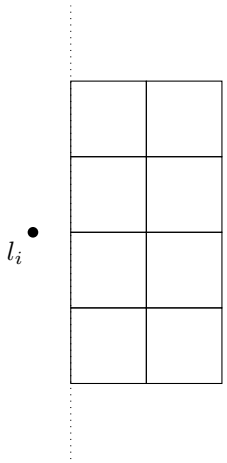
- The naive way is to calculate the distance between every pair of points. This obviously runs in $O(n^2)$. That's slow.
- It's possible to solve this problem in $O(n \log n)$ using a divide and conquer method.

The optimal algorithm goes as follows:

1. Sort the points by x-coordinate
2. Draw a line down the middle of the points, so that half the points are on the left of the split, and the other half are on the right.
3. Solve the smallest distance between two points in each half. (Let this distance be d.)
4. To combine, take the smaller distance from within each half, and check points over the boundary to see if there are two points closer together.

How do we combine?

- Since the smallest distance on either half is d, we only have to check points within d of our boundary of each side. From here, consider every point within d from the boundary on the left half, l_i.
- Record the y-coordinate of l_i, and create a 2×4 grid of squares on the right side, centered at the y-coordinate. Let each square have a side length of exactly $d/2$.



- Importantly, since each square has side length $d/2$ (and therefore a square diagonal is $\sqrt{d/2}$), there must be at maximum one point in each square. If there is more, we would have found that pair and marked it as the smallest, which would contradict our choice of d.
- Now, for each point near the divide, we need to check a constant number (8) possible points maximum on the other side of the line.
- Because of this constant checking, we can now assume that this combine function runs worst case on n points, and O(1) per point.
- This gives a divide and conquer runtime of $O(n \log n)$ in total!

Multiplication

Most intuitively, multiplication is repeated addition. (For computers, this would occur in the binary space.) Doing this with a n-bit number would require $O(n^2)$ time. However, there is a faster way.

Say x is an n-bit binary number. Let $m = n/2$. x_0 are the lower m bits, and x_1 is the upper m bits.

In this case,

$$x = x_1 2^m + x_0$$
$$y = y_1 2^m + y_0$$

Now, when we multiply these two numbers, we have

$$\begin{aligned} xy &= (x_1 2^m + x_0)(y_1 2^m + y_0) \\ &= x_1 y_1 2^{2m} + (x_1 y_0 + x_0 y_1) 2^m + x_0 y_0 \end{aligned}$$

We can then expand the middle term even more:

$$x_1 y_0 + x_0 y_1 = (x_1 + x_0)(y_1 + y_0) - x_1 y_1 - x_0 y_0$$

Why does this help? We just turned 2 multiplications into three!

- However, notice that the parts of these multiplications have already been used. This actually turns two multiplication into one, as long as we cache from the steps before!

Proof

$$T(2^k) = 3T(2^{k-1}) + c2^k$$

where T is the number of elementary operations, and k is the number of bits of the number. To multiply k-bit numbers, since we do three multiplications of two numbers half as large and a multiplication of 2^k (bit shift), we get the above equation. Simplifying,

$$\begin{aligned}
\frac{T(2^k)}{3^k} &= \frac{3T(2^{k-1})}{3^k} + \frac{c2^k}{3^k} \\
\frac{T(2^k)}{3^k} &= \frac{T(2^{k-1})}{3^{k-1}} + \frac{c2^k}{3^k} \\
\frac{T(2^k)}{3^k} &= \left(\frac{T(2^{k-2})}{3^{k-2}} + \frac{c2^{k-1}}{3^{k-1}}\right) + \frac{c2^k}{3^k}
\end{aligned}$$

We can keep breaking down the time of the middle term using mathematical induction. (We know the base case is true, since $T(1) = 1$.) This means this equation can be simplified to:

$$\frac{T(1)}{3} + c\sum_{i=2}^{k} \frac{2^i}{3^i}$$

Now, since $\frac{2^i}{3^i}$ will eventually converge to some number (since its rate is less than 1), we can assign a boundary, β, that this sum must be under. We can then rewrite this in terms of β:

$$\begin{aligned}
\frac{T(2^k)}{3^k} &\leq \beta \\
T(2^k) &\leq \beta 3^k \\
&= \beta 2^{\log 3^k} \\
&= \beta 2^k \log 3 \\
&= \beta n^{\log 3}
\end{aligned}$$

Since beta is a constant, we can drop it, and say our time complexity is $\Theta(n^{\log 3}) = \Theta(n^{1.58})$.

- Note that log here is in base-2.

Matrix Multiplication

Note that we can do matrix multiplication in a similar way to addition.

- Break our $n \times n$ matrices into four $\frac{n}{2} \times \frac{n}{2}$ matrices. We now have 8 small matrices total.

$$A \times B = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

- We can calculate the result of these 8 multiplications in 7, if we do some smart grouping.
- Our runtime for a $n \times n$ matrix becomes $T(n) = 7T(n/2) + cn^2$.
- Final runtime after simplifying using the same math gives $\mathrm{O}(n^{\log 7})$

Binary Search

Normally if we want to search for an object in a data set, it takes $\mathrm{O}(n)$, since we have to check every element. Can we do better?

- We can do better. Set up a binary search tree. First, set a (arbitrary) node to be the root.

- From the root, when a number is inserted, if it is smaller in the root, append to the left subtree. If larger, then append to the right.
- **BST Property:** Given a node (k, v), all nodes in left subtree have keys $< k$, and all nodes in right subtree have keys $> k$.
- To find a node, start from the root. If the value you are looking for is lower, go left. If the value is larger, then look right.
- To insert, follow the same process as find except when you get to the end, append the node.
- To delete, find the node to delete, u, and mark its parent p.
  - if u is a leaf, just delete,
  - if u has 1 child c, delete u, and make c a child of p.
  - if u has 2 children, find the smallest node in the right subtree of u, delete it, and replace u with it.

The issue is, what if our root is very small, and all the nodes we append are bigger? Then, the tree grows only to one side, and eventually turns into a linked list.

- This makes the search time $O(n)$ instead of $O(\log n)$!
- Therefore, we occasionally have to rebalance the tree and pay the $O(n)$ time. (Read each value in order of the tree, which gives you sorted list, pick the median and set it as root, and reinsert all the other values.)

Averaging

We have two types of averages:

- **Randomized average:** Suppose we roll a fair 6-sided dice. The average value should be 3.5. However, the average could also be 6, albeit very unlikely.
- **Amortized average:** Suppose we have a deck of cards, numbers 1-10 with 4 repeats each. If we sample 40 cards without replacement, the guaranteed average is 5.5. There is no chance of anything else.
  - Amortized averages have strong restraints on what the numbers can be.
  - This is commonly used in video games because it oftentimes feels more fair than full randomness.

QuickSort

- Pick a random number as a "pivot", and sort each side into whether it is smaller than or larger than the pivot. The sides need not be balanced.
- Recursively split each half, and combine them as you go up.
- In this case, *on average*, the tree will be mostly balanced, since on average half the numbers will be bigger, and half will be smaller.
- This is a form of amortized average!