CS 170 - Class Notes

Michael Lin

October 12, 2020

1 Divide and Conquer

1.1 Master Theorem

For a recurrence relation that takes the form:

$$T(n) = aT(\frac{n}{b}) + O(n^d)$$

then

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } d > \log_b a \quad \text{The algorithm is root heavy} \\ \Theta(n^d \log n) & \text{if } d = \log_b a \\ \Theta(n^{\log n}) & \text{if } d < \log_b a \quad \text{The algorithm is leaf heavy} \end{cases}$$

1.2 Examples

1.2.1 Median Finding

1.2.2 Matrix Multiplication

2 Fast Fourier Transform

First, we know that evaluating a polynomial $A(x)$ takes only linear time because we as soon as we compute x^n we can multiply by x and get x^{n+1}

$$A(x) = a_0 + x(a_1 + x(a_2 + \dots x(a_{n-1}))) \qquad \text{(Horner's Rule)}$$

Moreover, addition also take linear time because for $C(x) = A(x) + B(x)$, we have $c_k = a_k + b_k$. However for matrix multiplication, we would have to match each term in one polynomial to every term in the other, resulting in an $O(n^2)$ runtime.

$$c_k = \sum_{j=0}^{k} a_j b_{k-j}$$

FFT can be used to achieve multiplication in just $O(n \log n)$ time.

2.1 Representation of Polynomial

- **Coefficient Vector:** $a_0 + a_1 x + a_2 x^2 \dots$
- **Roots Representation:** $c(x - r_0)(x - r_1) \dots (x - r_{n-1})$
- **Samples Representation:** (x_k, y_k) for $k = 0, 1 \dots n-1$ where x_k's distinct

We can convince ourselves that the runtime of the three operations for the representation can be summerized as followed:

Operations	Coeffs	Roots	Samples
Evaluation	$O(n)$	$O(n)$	$O(n^2)$
Addition	$O(n)$	∞	$O(n)$
Multiplication	$O(n^2)$	$O(n)$	$O(n)$

Ultimately, FFT is a algorithm that can convert a polynomial between its coefficient and samples representations in $n \log n$ time

2.2 The Algorithm - Divide and Conquer

Goal: $A(x)$ for $x \in X$ – From coefficients view to samples view

Divide: Divide into even and odd coefficient

$$A_{\text{even}}(x) = \sum_{k=0}^{n/2} a_{2k} x^k$$

$$A_{\text{odd}}(x) = \sum_{k=0}^{n/2} a_{2k+1} x^k$$

Combine: Obtain $A(x)$ given $A_{\text{even}}(x)$ and $A_{\text{odd}}(x)$

$$A(x) = A_{\text{even}}(x^2) + x A_{\text{odd}}(x^2)$$

Conquer: Therefore, we can recursively compute

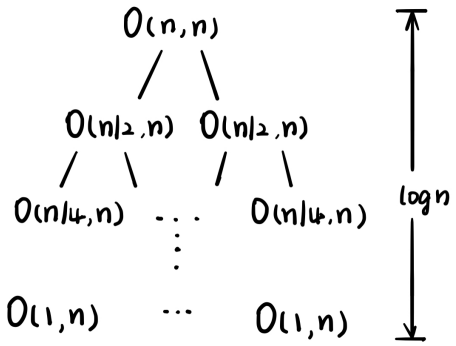
$$A_{\text{even}}(y) \text{ and } A_{\text{odd}}(y) \text{ for } y \in X^2 = \{x^2 | x \in X\}$$

Until we reach $n = 1$, that is when we have polynomial of degree one.

The big picture here is that in each recursive step, we compute a different polynomial of half the degree, but with a different set of the same size (X^2 has the same size as X). Here we obtain the recurrence

$$T(n, |X|) = 2T(n/2, X) + O(n + |X|)$$

This is not solvable by the Master's Theorem as it involves two variables. However, by drawing the recursion tree we soon discover that this is a bad recurrence – $|X|$ starts at N and never goes down.



At the bottom of the tree there are $2^{\log n}$ leaves, which would still give us n^2 running time.

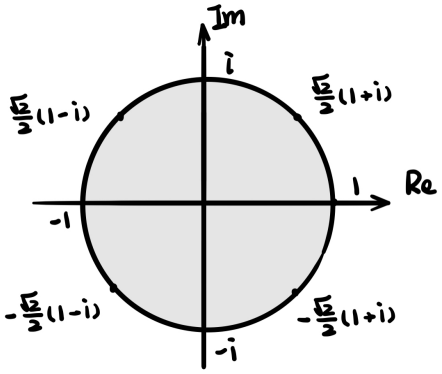
Nevertheless, this gives us an idea how to do better. If in each recursive step, we manage to also reduce the size that we are sampling the polynomial at. Or in another word, choose a set of X such that $|X^2| = |X|/2$. We can achieve $n \log n$ running time.

2.3 Nth Roots of Unity

Let's start with $X = \{1\}$ when $|X| = 1$. After that, what if we want two values in X? Remember that we want the feature where we square the values, we only have one value. Naturally, we will think of $X = \{-1, 1\}$. Moreover, we realize that there are two square roots for every number. We can keep taking square roots for all of them, and when we square them, it collapses into half of the size.

$$
\begin{aligned}
|X| = 1 &\Rightarrow X = \{1\} \\
|X| = 2 &\Rightarrow X = \{-1, 1\} \\
|X| = 4 &\Rightarrow X = \{i, -i, -1, 1\} \\
|X| = 8 &\Rightarrow X = \{\pm\frac{\sqrt{2}}{2}(1+i), \pm\frac{\sqrt{2}}{2}(1-i), i, -i, -1, 1\} \\
&\dots
\end{aligned}
$$

If we were to draw the numbers in the same set on a plane, you will see that they are evenly distributed in a unit circle.



It is because of such geometric property that we are able to reduce the size of the set in half when we square all of its elements. When you square a number, it's essentially doubling its angle, so half of its elements will overlap with each other.

We call the elements that satisfies $x^n = 1$ the **nth roots of unity**, and an element is said to be **primitive** if n is the smallest number that satisfies the condition, denoted by ω_n.

We can also express the nth roots of unity using Euler's formula. Since the points are evenly spaced, we have

$$\omega_n = e^{i\tau/n} \text{ where } \tau = 2\pi$$

So if we go back to 2.2, and instead of chosing an arbitrary set of X, we sample the polynomial at $\{1, \omega_n, \omega_n^2, \dots \omega_n^{n-1}\}$, we have a divide and conquer algorithm that runs in $n \log n$ time. That algorithm is called **Fast Fourier Transform (FFT)**

Furthermore, we can re-express the evaluation of the polynomial $a_0 + a_1x + a_2x^2 + \cdots + a_{n-1}x^{n-1}$ at $x = \{1, \omega, \omega^2, \dots \omega^{n-1}\}$ using the following matrix multiplication

$$
\begin{bmatrix}
1 & 1 & 1 & 1 & \cdots & 1 \\
1 & \omega & \omega^2 & \omega^3 & \cdots & \omega^{n-1} \\
1 & \omega^2 & \omega^4 & \omega^6 & \cdots & \omega^{2(n-1)} \\
1 & \omega^3 & \omega^6 & \omega^9 & \cdots & \omega^{3(n-1)} \\
\vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\
1 & \omega^{n-1} & \omega^{2(n-1)} & \omega^{3(n-1)} & \dots & \omega^{(n-1)(n-1)}
\end{bmatrix}
\begin{bmatrix}
a_0 \\ a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_{n-1}
\end{bmatrix}
$$

The corresponding transformation (denoted as F_n) is also known as the **Discrete Fourier Transform**. We notice that $F_n = F_n^T$ and $(F_n)_{jk} = \omega^{jk} = e^{i(jk)\tau/n}$. All the entries of F_n are nth roots of unity, which means raising any of them to the nth power gives 1.

2.4 Polynomial Multiplication

Now that we have our FFT, doing polynomial multiplication becomes quite simple. For polynomial A and B, first thing we need is to compute their FFT

$$A^* = \text{FFT}(A) \text{ and } B^* = \text{FFT}(B)$$

What it does is that it converts A and B from coefficient vector into sample vector. Essentially, we have the samples of A and B, at the nth roots of unity $x_k = e^{ik\tau/n}$. Therefore the transformed version of our desired polynomial C^* is just the result of element-wise multiplication of A^* and B^*

$$C_k^* = A_k^* B_k^*$$

Then to compute C, we need the inverse fast fourier transform of C^*

$$C = F_n^{-1}(C^*)$$

Luckily, this is not too hard. Because F_n is almost unitary (the complex equivalent of orthonormal) except that each column has a length of $\frac{n}{2}$ instead of 1, and a very nice property of a unitary matrix is that the inverse is just its conjugate transpose. Here, we can divide each column in both F_n^H and F_n by $\frac{n}{2}$ to get a matrix whose columns are orthonormal, which gives us the following identity

$$\frac{1}{n} F_n^H F_n = I$$

Furthermore, we notice that F_n is symmetric, which means $F_n^H = \overline{F_n}$

$$F_n^{-1} = \frac{1}{n}\overline{F_n}$$

Therefore, for polynomial multiplication, we will first transform them into sample representation using FFT in $n \log n$ time, then do the multiplication in linear time, and use IFFT to transform back to coefficient representation in $n \log n$ time. The result is a polynomial multiplication algorithm that runs in $n \log n$ time.

3 Graph Search

Recall that a graph is defined as $G = (V, E)$, where V = Set of vertices, E = Set of edges. The elements of E may be **undirected pairs**, denoted by $e = \{v, w\}$, or **directed pairs**, denoted by $e = (v, w)$.

The applications of graph search includes: web crawling, social networking, network broadcast, garbage collection, solving puzzles and games.

3.1 Case Study: Pocket Cube

Suppose we have a 2x2x2 rubik's cube. We can model it using a configuration graph.

Configuration Graph:

- **Vertex for each possible state of the cube.** For a 2x2x2 cube, because we have 8 cubies, an each cubies has 3 possible twist #vertices $= 8! \cdot 3^8 = 264,539,520$.
- **Edge for each possible move.** Since every move is "undoable", the graph is undirected.

Amongst all the vertices, there's a certain **solved states**, and it is connected to all the possible moves from there. The graph will keep extending, until it exhausts all of the states. The maximum depth of this graph is also known as the **diameter** of the graph, which represents the maximum number of steps it takes to solve the rubik's cube. In the case of a 2x2x2, this number is 11. For 3x3x3, the number is 20, and it's roughly $\Theta(n^2/\lg n)$ for NxNxN.

3.2 Graph Representation

Adjacency List

For each vertex $u \in V$, we have adj[u] that stores all of the neighbors of u. adj[u] could be a linked list, or a hash set based on the hashed address of the vertices. More formally, adj[u] = $\{v \in V | (u, v) \in E\}$

In an object-oriented scenario, it is possible to defined **u.neighbors = adj[u]**. However, despite that this representation is cleaner, it means that the vertex v can only be involved in one graph structure, whereas if we are using adjacency lists, we can define multiple graphs using the same vertices.

Implicit Representation

In some cases such as the rubik's cube, we don't want to build the whole space of adj[u]. Instead we would have adj(u) being a function, or u.neighbors() being a method. **Compared with adjacency list this uses zero space because we are computing the neighbors on the go**, and if we are lucky we don't need to build the whole graph, we just need enough of it for us to find the answer.

For our rubik's cube problem, computing all the possible states for a larger cube would not be possible. But representing a state is easy, and knowing what's reachable from that state is also relatively easy.

3.3 Breadth First Search

Goal

- Visit all nodes reachable from a given node $u \in V$
- Achieve $O(V + E)$
- Look at nodes reachable in { 0, 1, 2 ... } moves
- Avoid duplicates

Algorithm 1 Breadth First Search

```
1:  procedure BFS(s, adj)      ▷ Given the starting vertex s and the graph as an adjacency list
2:      level = { s: 0 }                                  ▷ The minimum steps to reach a vertex u
3:      parent = { s: None }
4:                ▷ Optional, the parent vertex of u. This later helps us find the shortest path
5:      i = 1
6:      frontier = [ s ]                                  ▷ All vertices at level i − 1
7:      while frontier do                                 ▷ frontier is not empty
8:          next = [ ]
9:          for u in frontier do
10:             for v in adj[u] do                        ▷ Look at all vertices at the next level
11:                 if v not in level then
12:                             ▷ Checking duplicates – if we have visited v, it should have a level
13:                     level[v] = i
14:                     parent[v] = u
15:                     next.append(v)
16:         frontier = next
17:         i = i + 1
```

Runtime Analysis

When we look at frontier we see that the edges only enter it once, because before one vertex enters frontier it would be marked in level. Therefore in a single BFS procedure with starting point s and adjacency list adj the running time of would be the sum of all the vertices in the adjacency list.

$$\sum_{u \in V} |\text{adj}[u]| = \begin{cases} 2|E| & \text{undirected graph} \\ |E| & \text{directed graph} \end{cases}$$

On top of that, we also spend order V time because we need to touch every vertex. So the total running time would be $\Theta(V + E)$

3.4 Depth First Search

Similar to BFS, DFS can be used to explore the vertices in a graph. However, unlike BFS which explore a graph layer by layer, DFS uses recursion which gives a more straightforward algorithm for the **connectivity** of a graph.

Algorithm 2 Depth First Search

```
procedure DFS-Visit(s, adj)                ▷ Recursivly visit all the nodes reachable from s
    for v in adj[s] do
        if v not in parents then
            parents[v] = s
            DEF-Visit(v, adj)

procedure DFS(V, adj)
    parents = { }
                                           ▷ A list keeping track of all visited nodes also works
    for s in V do
        if s not in parents then           ▷ This step is to find all possible starting points
            parents[s] = None
            DFS-Visit(s, adj)
            ▷ If a node has not yet been visited, we recursivly explore all connected nodes from s
```

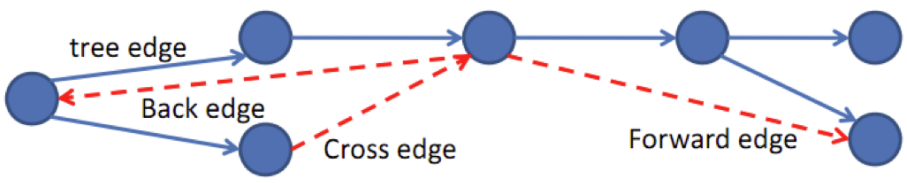
Runtime Analysis

In procedure DFS, we visit each vertex once so that's $O(V)$ to begin with, and inside each iteration of vertex v, we pay for what's inside the adjacency list adj[v]. So just as in the case for BFS, we need to add order E time, and the result would be $\Theta(V + E)$.

Edge Classification

The edges we traverse as we execute a depth-first search can be classified into four edge types.

1. If v is visited for the first time as we traverse the edge (u, v), then the edge is a **tree edge**.
2. Else, v has already been visited:
   (a) If v is an ancestor of u, then edge (u, v) is a **back edge**.
   (b) Else, if v is a descendant of u, then edge (u, v) is a **forward edge**.
   (c) Else, if v is neither an ancestor or descendant of u, then edge (u, v) is a **cross edge.**



This is useful for many problems, here we are going to explore two of them: **Cycle Detection** and **Topological Sort**.

Cycle Detection

We claim that G has a cycle $\Leftrightarrow$ DFS(G) has a back edge.

Proof. We need to proof the proposition in both direction.

$\Leftarrow$: If there's a back edge, by definition, it makes a cycle.

$\Rightarrow$: Given a cycle $\{v_0, v_1, \ldots, v_k\}$, we can assume that v_0 is the first vertex in the cycle we visited during DFS. By induction, we can prove that we will visit v_k before finish v_0. This argument can be made for any v_i, we can think of it as "balanced parentheses" $\underset{u}{(} \ldots \underset{k}{(} \ldots \underset{k}{)} \ldots \underset{u}{)}$

Therefore, when we visited v_k, v_0 will still be on the stack. We will consider (v_k, v_0), and because v_0 is still being processed, we will mark it as a back edge. □

Topological Sort

Example: Given Directed Acylic Graph (DAG), where vertices represent tasks & edges represent dependencies, order tasks without violating dependencies.

The topological sort result is the **reverse of DFS finishing times** (time at which DFS-Visit(v) finishes).

Proof. To prove the correctness, we need to prove that for any edge $e = (u, v)$ u is ordered before v, that is, in DFS-Visit, v finishes before u finishes.

- if u visited before v:

  Before visit to u finishes, will visit v – via (u, v) or otherwise. By "balanced parentheses", v will finish before u.

- if v visited before u:

  Because graph is acyclic, u **cannot be reached from** v. That means visit to v finishes before visiting u.

□

3.5 Connected Components

For an undirected graph G, BFS(v) or DFS(v) will visit every other vertex in the same connected component as v. We can mark every vertex visited from a BFS/DFS from v as being "owned" by v. As we iterate through all the vertices, we execute a BFS/DFS starting from a vertex if it has no owner (i.e. it is part of an undiscovered connected component) and mark all the vertices visited in that BFS/DFS.

The runtime of this algorithm is $\Theta(V + E)$ since each vertex is visited twice (once by iterating through it in the outer loop, another by visiting it in BFS/DFS) and each edge is visited once (in BFS/DFS).

However, the algorithm does not work with directed graph. For undirected graphs, finding a path from u to v implies that there exists a path from v to u. This is not the case for directed graphs.

Strongly Connected Components

We define Strongly Connected Components ( ) as components in directed graphs where any two vertices has a path in between each other.

The intuition that will help us separate a directed graph into strongly connected components is realizing that **a SCC with its edges' directions reversed is still a SCC.** We will introduce G^T, which is the transpose of directed graph G. G^T and G are the same graph except the edge directions are reversed in G^T , i.e. if edge (u, v) is in G, then the edge (v, u) is in G^T.

4 Shortest Path in Weighted Graphs

Definition:

$$G(V, E, W)$$

5 Dynamic Prgramming

Dynamic Programming (DP) is a powerful, general algorithmic design technique. It can be think of as "careful brute force", or "subproblems + reuse". Similar to divide and conquer, you take a problem, split it into subproblems, solve the subproblems and reuse the solution to the subproblems.

5.1 Fibonacci Number

$$F_1 = 1 \ , \ F_2 = 1$$

$$F_n = F_{n-1} + F_{n-2}$$

Goal: Compute F_n

Here, the naive approach would be just recursively calling $F_n = F_{n-1} + F_{n-2}$ until we reach $n = 2$. However this give us the recurrence

$$T(n) = T(n-1) + T(n-2) + \Theta(1)$$

The correct answer for this recurrence is approximatly φ^n where φ is the golden ratio $1.618\ldots$. However, we can also simplify it like so

$$T(n) \leq 2(n-2) = \Theta(2^{n/2})$$

Memoized DP Algorithm

memo =