CS 170 Midterm 1 Jeffrey Shen

1 Prologue

For any algorithm:

1. Is it correct?
2. How much time does it take, as function of n? Big-O Notation
3. Can we do better?

2 Divide-and-Conquer Algorithms

2.1

Divide-and-Conquer Method:

1. Breaking into subproblems that are smaller instances of same type of problem
2. Recursively solve subproblems
3. Appropriately combining solutions

Note: We want to reduce # of subproblems to make efficient

Ex) Can split a digit into

$x = \boxed{x_L}\,\boxed{x_R} = 2^{n/2} x_L + x_R$ (Karatsuba's Algorithm)

and take advantage of properties in a divide and conquer algorithm for mult

2.2 Master Theorem: problem size n, solve a subproblems of size $\frac{n}{b}$ and combine takes $O(n^d)$

If $T(n) = aT(\lceil n/b \rceil) + O(n^d)$ constants $a>0, b>1, d\geq 0$

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

2.3

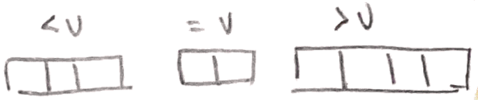
Ex) Mergesort – $O(n \log n)$ sort

- Split list into twohalves, sort the half, merge the two sorted sublists

```
func merge(x[1...k], y[1..l]):
    if k=0    return y[1...l]
    if l=0    return x[1...k]
    if x[1] ≤ y[1]
        return x[1] + merge(x[2...k], y[1...l])
    else
        return y[1] + merge(x[1...k], y[2...l])
```

```
Iterative
Q = [ ]
for i=1 to n:
    inject(Q, [a_i])
while |Q| > 1:
    inject(Q, merge(eject(Q), eject(Q)))
return eject(Q)
```

2.4



Find kth element in ↗

Find median by divide-and-conquer

1. Select a number v from list S randomly
2. Split list into $< v$, $= v$, $> v$
3. Search can be narrowed down to one of the lists by kth element

randomzing v $\quad T(n) \leq T\left(\frac{3n}{4}\right) + O(n) \Rightarrow O(n)$

2.5

Can break down matrix into blocks to simplify matrix multiplication, then use optimization from Volker Strassen

$$\begin{bmatrix} A & B \\ C & D \end{bmatrix}$$

2.6 Fast Fourier Transform

Multiplying two polynomials quicker in $O(n\log n)$ time

Degree d polynomial can be determined by $d+1$ distinct points

$$\begin{tikzcd} \boxed{\begin{matrix}\text{Coefficient Representation}\\ a_0, a_1, \ldots, a_d\end{matrix}} \arrow[r, shift left, "\text{Evaluation}"] & \boxed{\begin{matrix}\text{Value Representation}\\ A(x_0), A(x_1) \ldots A(x_{d-1})\end{matrix}} \arrow[l, shift left, "\text{Interpolation}"] \end{tikzcd}$$

Polynomial Multiplication

1) Selection : Pick points $x_0, \ldots x_{n-1}$ $\quad n \geq 2d+1$
2) Evaluation : Computes $A(x_0) \ldots A(x_{n-1})$ $\quad B(x_0) \ldots B(x_{n-1})$
3) Multiplication : $C(x_k) = A(x_k)B(x_k)$ for all $k = 0, \ldots, n-1$
4) Recover : Recover $C(x) = C_0 + C_1 x + \ldots + C_{2d} x^{2d}$

* FFT converts coefficient rep polynomial to value representation

* FFT^{-1} interpolates equation from value representation to coefficient rep

Fast Fourier Transform

1) Given $A(x)$, split into even and odd terms $A_e(x^2)$, $A_o(x^2)$

Then $\quad A(x) = A_e(x^2) + x_i A_o(x_i^2)$
$\qquad\; A(-x) = A_e(x^2) - x_i A_o(x^2)$

2) Choose complex nth roots of unity

$w^n = 1 \qquad n=2 : -1, 1 \qquad n=4 : 1, i, -1, -i \qquad n=2 : \begin{bmatrix} 1 & 1 \\ 1 & w \end{bmatrix}\begin{bmatrix} a \\ b \end{bmatrix}$

3) Solve Subproblems

$$\begin{array}{c|c|c} & \overset{2k}{0\ 2\ 4\ \ldots} \xrightarrow{k} & \overset{2k+1}{1\ 3\ 5\ \ldots} \\ \hline i\downarrow\ \begin{matrix}0\\1\\2\end{matrix} & FT_{n/2} & w^j FT_{n/2} \\ \hline \begin{matrix}0\\1\\2\\ \vdots\end{matrix} & FT_{n/2} & -w^j FT_{n/2} \end{array} \Rightarrow \begin{bmatrix} \boxed{W_{n/2}} \overset{A_e}{\begin{bmatrix} a_0 \\ a_2 \\ a_{n-2} \end{bmatrix}} + w^j \boxed{W_{n/2}} \overset{A_o}{\begin{bmatrix} a_1 \\ a_3 \\ \vdots \\ a_{n-1} \end{bmatrix}} \\ \boxed{W_{n/2}} \begin{bmatrix} a_0 \\ a_2 \\ \vdots \end{bmatrix} - w^j \boxed{W_{n/2}} \begin{bmatrix} a_1 \\ a_3 \\ \vdots \\ a_n \end{bmatrix} \end{bmatrix} = \begin{bmatrix} x + w^j y \\ x_1 + w^j y_2 \\ \vdots \\ x + w^j y \end{bmatrix} \begin{matrix} w_1 = 1 \\ w_2 = i \\ \vdots \\ w = -i \end{matrix}$$

<= fill in in order of w

4) Do multiplication in value representation

5) Interpolate back using $M_n(w)^{-1} = \frac{1}{n} M_n(w^{-1})$

③ Decompositions of graphs

3.1

Graphs used for variety of problems, can be represented w/ adjacency matrix or adjacency list

3.2

Depth First Search is linear time algorithm that finds parts of graph that are reachable from vertex

```
procedure explore(G, v):
Input: G = (V, E) graph, v ∈ V
Output: visited(u) is true for nodes reachable
    visited(v) = true
    previsit(v)
    for each edge (v, u) ∈ E:
        if not visited(u): explore(G, u)
    postvisit(v)
```

```
procedure dfs(G)
    set all v ∈ V visited(v) = false
    for all v ∈ V:
        if not visited(v): explore(G, v)
```

Runtime: $O(|V|+|E|)$

We can set pre/post numbers in graph by

```
procedure previsit(v)
    pre[v] = clock
    clock += 1
```

```
procedure postvisit(v)
    post[v] = clock
    clock += 1
```

3.3

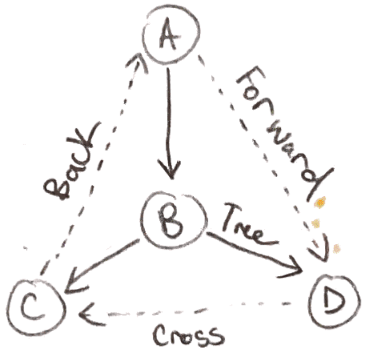
Edge Types:

Tree edge: part of DFS tree

Forward edge: from node to nonchild descendant in tree

Back edge: lead to an ancestor in tree

Cross edges: neither descendant or ancestor



pre/post ordering (u,v)

$\underset{u}{[}\quad \underset{v}{[}\quad \underset{v}{]}\quad \underset{u}{]}$ Tree / forward

$\underset{v}{[}\quad \underset{u}{[}\quad \underset{u}{]}\quad \underset{v}{]}$ Back

$[\quad]\quad [\quad]$ Cross

Properties

- Directed graph has cycle iff DFS reveals a backedge
- DAG can be linearized by performing in decreasing order of post numbers
- DAG has at least one source and one sink

highest post number is source, lowest is sink

(3.4)
In directed graphs we have strongly connected components where it is only connected if there is a path from u→v & v→u

- We can turn strongly connected components into a meta node and change any graph into a DAG

Properties

- If explore started at node v, terminates well all nodes reachable from v have been visited
- Node with highest post number must be in strongly connected component
- If C and C' are SCCs and edge from C to C', highest post in C > highest post in C'

To find SCC:

1. Run DFS on reversed G^R to find sink component
2. Run undirected connected component algo (in previsit set a cc[v] = count) on G and in DFS process nodes in decreasing order of post numbers in step 1

Runtime: $O(|V|+|E|)$

(4) Paths in Graphs

(4.1)
Difference between two nodes is the length of shortest path between them

(4.2)
Breadth First Search by using queue instead of a stack

Search by level away

```
procedure BFS (G, s):
    for all u ∈ V:
        dist (u) = ∞
    dist(s) = 0
    Q = [s]
    while Q not empty:
        u = eject (Q)
        for all edges (u,v) ∈ E:
            if dist (v) = ∞:
                inject (Q , v)
                dist(v) = dist(u) + 1
```

Runtime: $O(|V| + |E|)$

(4.4) Dijkstra's Algorithm works on graphs with non-negative weighted edges by using priority queue and exploring by next shortest path

---

```
procedure dijkstra(G, d, s):
    Input: Graph G = (V, E) directed or undirected, positive edge lengths {l_e : e ∈ E}, s ∈ V
    Output: For all vertices u reachable from s, dist(u) set to distance s to u
    for all u ∈ V:
        dist (u) = ∞
        prev (u) = nil
    dist( s) = 0
    H = makeQueue ( V)                dist value as keys
    while H is not empty:
        u = deletemin (H)
        for all edges ( u, v) ∈ E:
            if dist (v) > dist (u) + l(u,v):
                dist(v) = dist (u) + l(u,v)
                prev = u
                decreasekey (H, v)
```

Runtime : Binary Heap: $O((|V| + |E|) \log |V|)$

4.6

Bellman-Ford algorithm allows us to find shortest path on graph with negative weights by updating all edges |V|-1 times

---

```
procedure shortest-paths (G, l, s)
    Input: graph G edge length {l_e : e ∈ E} no negative cycles, vertex s ∈ V

    for all u ∈ V
        dist(u) = ∞, prev(u) = nil
    dist(s) = 0
    repeat |V|-1 times
        for all e ∈ E:
            update(e)
```

```
procedure update((u,v) ∈ E)
    dist(v) = min{dist(v), dist(u) + l(u,v)}
```

Runtime: $O(|V|+|E|)$

1.) Update gives correct distance if u is 2nd to last node in shortest path

2.) Never makes dist(v) too small, safe

If there is a negative cycle in the graph, there is no shortest path

To check, perform one last check to see if anything changes in all edges

We can also find shortest paths in dags in linear time by doing a topological sort

```
procedure dag-shortest-paths (G, l, s)
    Input: Dag G = (V, E)
           edge lengths {l_e : e ∈ E}; vertex s ∈ V
    Output: dist(u) is distance from s to u

    for all u ∈ V:
        dist(u) = ∞
        prev(u) = nil
    dist(s) = 0
    Linearize G
    for each u ∈ V, in linearized order
        for all edges (u,v) ∈ E:
            update(u,v)
```