CSE 101 Final Review

- **Toom–Cook algorithm:**

- Given two large integers, a and b, Toom–Cook splits up a and b into k smaller parts each of length l, and performs operations on the parts.
- The Karatsuba algorithm is a special case of Toom–Cook, where the number is split into two smaller ones.
- It reduces 4 multiplications to 3 and so operates at Θ(nlog(3)/log(2)), which is about **$\Theta(n^{1.585})$.**
- Ordinary long multiplication is equivalent to Toom-1, with complexity **$\Theta(n^2)$.**

- **Dijkstra's Algorithm:**
- **O((V + E)logV)**, using PQ
- $O(V^2)$, using an array

```
Dijkstra (x):
    PQ.add(0, x)
    for all other nodes y:                              //O(V)
            PQ. add(infinity, y)
    while ( PQ.size() > 0 ):
            curr = node with min_dist
            // find min, takes O(E): can be improved in PQ → O(1)
            for (y in curr.neighbors):          //O(degree(curr))
                    if (y.dist > curr.dist + edge_between_curr_and_y):
                            //you can choose remove the original (y.dist', y) or not,
                    //remove would save memory
                            PQ .remove(y.dist, y)              //O(logV)
                            y.dist = curr.dist + edge_between_curr_and_y
                            PQ .add(y.dist, y)                 //O(logV)
```

- Binary heap runtime:
-

Operation	Binary[11]
find-min	$\Theta(1)$
delete-min	$\Theta(\log n)$
insert	$O(\log n)$
decrease-key	$\Theta(\log n)$
merge	$\Theta(n)$

- **Binary heap problem on practice final:**

- ○ Initialize the heap by adding the 1st element in each list (k elements total) // O(klogk)
- ○ Repeat for each element: // O(nk)
    - ○ Remove the root of the heap (element **i** from the **ki**th list), and add in the next element from that list (**ki**th list) // O(logk)

Runtime in total: **O(nklogk)**

<u>Topics For Quiz 1</u>

- Max bandwidth path
    - Path
        - Simple path
            - No two edges are the same
        - **A single vertex is a trivial path to itself**
    - Objective
        - Over all possible paths p between v and u, find max BW(p)
- Graph reachability
    - Given a graph **G** and starting vertex **s**, give all reachable vertices v from s
        - **X:** a set of explored vertices
        - **F:** a set of reached but not explored vertices
        - **U:** a set of unreached vertices
    - procedure GraphSearch (G: directed graph, s: vertex)
        -
        - `Initialize X = empty, F = {s}, U = V - F.`
        - `While F is not empty:`
            - `Pick v in F.`
            - `For each neighbor u of v:`
                - `If u is not in X or F:`
                    - `move u from U to F.`
            - `Move v from F to X.`
        - `Return X.`
    - Time analysis: **O(|V| + |E|)**

- **chapter 3.1,3.2**

- How big is ur graph
    - |E|
        - As small as |V|, if smaller, then the graph degenerates - **sparse**
        - As large as $|V|^2$, all possible connections - **dense**
    - Adjacency matrix vs adjacency list
        - Matrix always takes $O(|V|^2)$ space, so if the graph is **sparse**, that would be wasteful.

        - List always takes O(|E|)
- DFS in **undirected graphs**
    - <u>**procedure** explore(G, v):</u>
        - visited(v);
        - previsited(v):
        - for each edge (v, u) ∈ E:
            - If not visited(u): explore(G, u)
        - postvisit(v);

    - <u>**procedure** dfs(G):</u>
        - for all v ∈ V:
            - visited(v) = false;
        - for all v ∈ V:
            - If not visited(v): explore(v)
    - Two steps when considering the runtime
        - Mark each spot as visited - **O(|V|)**
        - A loop in which scanned all edges - **O(|E|)**
- **Connectivity**
    - Connected components:
        - Subgraph internally connected but has edges to remaining graph
        - Each time DFS called explore, one connected component is picked out
- Previsit and povisit ordering
    - Define a counter **clock, initialized as 1**
        - <u>**procedure** previsit(v):</u>
            - Pre[v] = clock;
            - Clock++;
        - <u>**procedure** postvisit(v):</u>
            - Post[v] = clock;
            - Clock++;
    - **Property:**
        - For any nodes u and v, the two intervals [pre[u], post[u]] and [pre[v], post[v]] are either disjoint or contained within the other.

- **chapter 3.3,3.4**

- **Types of edges**
    - Forward edges
        - Lead to a non-child descendant
        - If u is an ancestor of v, $[_u, [_v,]_v,]_u$
    - Back edges
        - Lead to ancestor
        - $[_v, [_u,]_u,]_v$
    - Cross edges
        - Lead to a node that has been explored.

- - **$[_v,]_v, [_u,]_u$**
- Directed acyclic graphs
    - **Property**
        - DFS reveals a back edge **iff** this G has a cycle
        - Every edges leads to a vertex with lower post number in a **DAG**
            - Sink: smallest post number
            - Source: highest post number
        - **Every DAG has at least one source and at least one sink**
            - Linearization:
                - Find a **source**, output it and delete from G
                - Repeat until the graph is empty

- **Strongly connected components (SCCs)**
    - Two nodes u and v are connected in a graph if there is a path from u to v and a path from v to u.
    - Property
        - Every directed graph is a DAG of its SCCs.
- Decomposition of SCCs
    - How to locate the **sink**
        - The SCC of highest post number must be a **source** SCC
        - We only need to reverse the whole graph, the **source SCC** of G' will be the **sink SCC** of the original graph.
    - How to continue once the first **sink** is discovered
        - Run DFS on G^R (step 1)
        - Run the directed connected components algorithm on G, and during DFS, process the vertices in decreasing order of their post numbers from step 1

- **chapter 4.1,4.2,4.3**

- **Distances**
    - The distance between two vertices is the length of the shortest path between them

- **BFS**
    - **procedure** bfs(G, s):
        - for all v ∈ V:
            - dist(v) = infinite
        - dist(s) = 0;
        - Q = [s] (queue containing only s)
        - while Q is not empty:
            - u = eject(Q)
            - for all edges (u, v) ∈ E:

- If dist(v) = infinite:
  - inject(Q, v)
  - dist(v) = dist(u) + 1

- Lengths on edges

- **chapter 4.4,4.5**

- **Dijkstra's algorithm**
  - All edges have positive length
  - Using priority queue
    - Insert
      - Add a new element to the set
    - Decrease key
      - Decrease the value of certain key
    - Delete min
      - Return the element with the smallest key,
      - remove it from the set
    - Make heap
      - Build a priority queue out of given elements

Quiz 2 Topics Lists

- MST, Trees, union/find data structure, Binary heap, Dijkstra, Prim, Kruskal
  - T/F, short answers, multiple choices
- Prove greedy algorithm - modify the solution

Greedy Algorithm

- **Format**
  - Instance: input
  - Solution format: output
  - Constraint: output's property to count as solution
  - Objective function: Quantity are we trying to max/min

Method 1: Modify-the-solution Recursion

- **Claim**
  - Let g_1 be the first greedy choice. Let OS be any other solution that meet all requirements and does not include g_1. Then there is a solution OS' that includes g_1, meets all constraints and is at least as good as OS.

- **Exchange argument**

- State what you know: g_1 meet the condition for the first choice. OS meets all of the constraints for the problem
- Define OS' in terms of g_1 and OS to include g_1. There might be multiple cases.

- **Prove exchange argument**
    - Show that OS' meet all constraints
    - Compare objective function of OS and OS' → OS' same or better

- **Prove the algorithm by induction**
    - For any instance I of size N, GS(I) is an optimal solution
    - By strong induction, Base case: N = 0 or 1, trivial case.
    - Assume for every instance I_2 of size $0 \leq n \leq N - 1$, $GS(I_2)$ is optimal solution. $GS(I) = g_1 + GS(I_2)$. By MTS lemma, there is an optimal solution OS' that also includes g_1. $OS' = g_1 + OS_2$, for some other solution OS_2 of instance I_2. Then by IH, GS(I) is at least as good as OS'. Since OS' is optimal, GS(I) is optimal.
    - $OS(I) \geq OS' = g_1 + S(I_2) \geq g_1 + GS(I_2) = GS(I)$

<u>**Method 2: Modify the solution - Iterative format**</u>

Basic idea:

- Prove for all $i \geq 1$
    - **Min -** value(GS_i) ≤ value(OS_i)
    - **Max -** cost(GS_i) ≤ cost(OS_i)

- **IterMTS:** Let $g_1, g_2, \ldots g_T$ be the decisions made in order by the greedy strategy. For each $0 \leq i \leq T$, there is an optimal solution OS_i that includes $g_1, g_2 \ldots, g_i$.

- **Prove by induction:**
- *Base case*: For i = 0, we let OS_0 be any optimal solution. Since it doesn't have to agree with any greedy decisions.
- Assume that there is an optimal solution OS_{i-1} that includes g_1, g_2, g_{i-1}. If it also includes g_i, we set $OS_i = OS_{I-1}$. Otherwise,
    - Define OS_i in a way that leaves g_1 to g_{i-1} unchanged, but but changes i'th move of OS_{i-1} to g_i.
    - Prove that OS_i meets constraints
    - Compare obj(OS_i) and obj(OS_{i-1}).

<u>**Minimum Spanning Tree**</u>

- Given a graph G = (V, E), MST is a tree T = (V, E') that minimizes total weight of T. **Acyclic, connected.**

- Properties:
    - 1. Remove a cycle edge cannot disconnect graph.
    - 2. A tree on n nodes has **(n - 1)** edges.
    - 3. Any connected, undirected graph G = (V, E) with **|E| = |V| - 1 is a tree**.
    - 4. An undirected graph is a tree iff a unique path between any pairs of nodes.

- **Cut property**
    - Suppose edge X are part of a MST of G = (V, E). Pick any subset of nodes S for which X doesn't cross between S and V - S, and let e be the lightest edge across this partition. Then X U {e} is a part of some MST.

- **Union/Find data structure**
    - Make root of the shorter tree point to the root of larger tree
    - Properties:
        - For any x, rank(x) < rank (P(x))
        - A node of rank k has at least 2^k descendants. Rank k + 1: union 2k.
        - If there are n elements, there can be at most n / 2^k nodes of rank k.
    - Runtime: **find/union → O(log(n)).**
    - Path compression: reduce runtime to near O(1).

- **Kruskal's algorithm**

```
For all u belongs to V:
      Makeset (u);
X : {}
Sort the edges E by weight;
For all edges (u, v) belongs to E, in increasing order of weight:
      If find(u) ≠ find(v):
            Add edge (u, v) to X.
            Union (u, v)
Return X
```

- **Prim's algorithm**

```
X = {}
Repeat until |X| = |V| - 1
      Pick a subset of V S for which X has no edge between S and V - S.
      Let e be the min edge between S and V - S
      X = X U {e}
Return X.
```

Shortest Path in graph

- Dijkstra's algorithm

```
For all u belong to V:
      dist(u) = infinity
      prev(u) = null
dis(s) = 0

H = makequeue(V) (using dist-values as keys)
While H is not empty:
      U = deletemin(H)
      For all edges (u, v) belongs to E:
            If dist(v) > dist(u) + l(u, v)
            prev(v) = u
            decreasekey(H, v)
```

- **Runtime: depends on priority queue implementation**

Implementation	`deletemin`	`insert`/ `decreasekey`	$\lvert V\rvert \times$ `deletemin` $+$ $(\lvert V\rvert + \lvert E\rvert) \times$ `insert`
Array	$O(\lvert V\rvert)$	$O(1)$	$O(\lvert V\rvert^2)$
Binary heap	$O(\log \lvert V\rvert)$	$O(\log \lvert V\rvert)$	$O((\lvert V\rvert + \lvert E\rvert) \log \lvert V\rvert)$
d-ary heap	$O(\frac{d \log \lvert V\rvert}{\log d})$	$O(\frac{\log \lvert V\rvert}{\log d})$	$O((\lvert V\rvert \cdot d + \lvert E\rvert) \frac{\log \lvert V\rvert}{\log d})$
Fibonacci heap	$O(\log \lvert V\rvert)$	$O(1)$ (amortized)	$O(\lvert V\rvert \log \lvert V\rvert + \lvert E\rvert)$

- **Binary Heap:**
    - Each level is filled from left to right,
    - Key (parent) < children.

CHAPTER 2
Divide and Conquer

- Strategy
    - Break problem into *subproblems* that are themselves smaller instances of the same type of problem
    - Recursively solving these subproblems
    - Appropriately combining their answers

2. 1 Multiplication

- Observation: (KS mult)
  - (a + b x) (b + c y) can be done with **three** multiplication, since bc + ad = **(a + b) (c + d)** - **ac** - **bd**.
  - $$x = \boxed{x_L}\boxed{x_R} = 2^{n/2}x_L + x_R$$
    $$y = \boxed{y_L}\boxed{y_R} = 2^{n/2}y_L + y_R.$$
  - $$xy = (2^{n/2}x_L + x_R)(2^{n/2}y_L + y_R) = 2^n x_L y_L + 2^{n/2}(x_L y_R + x_R y_L) + x_R y_R.$$

- Multiplication strategy:
  - General: T(n) = 4 * T(n/2) + O(n) → O($\mathbf{n^2}$)
  - Reduced: T(n) = 3 * T(n/2) + O(n) → O($\mathbf{n^{1.59}}$)
  - Proof idea:
    - changes in the branching factor of recursion tree
    - Geometric increase from O(n) (k = 0) to O ($n^{\log 2\ 3}$) (k = $\log_2 n$).
    - DPK p. 52 - 53

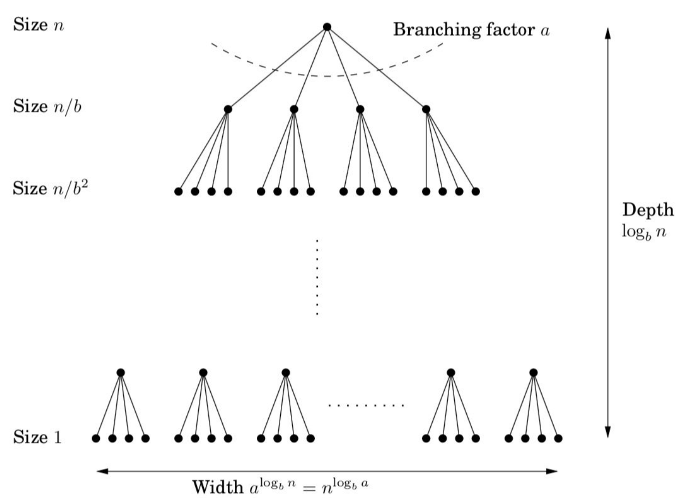
- K-terms (in general)
    - Split up number into k equally sized parts. Combine them with 2k - 1 multiplications instead of k^2.
    - $T(n) = (2k - 1)\ T(n/k) + O(n)$. → $T(n) = O(n^{\log(2k-1)/\log(k)})$

2. 2 Recurrence relationship

- Master Theorem:
  If $T(n) = aT([n/b]) + O(n^d)$ for some constants $a > 0$, $b > 1$ and $d \geq 0$. Then
    - $O(n^d)$ if $d > \log_b a$
    - $O(n^d \log n)$ if $d = \log_b a$
    - $O(n^{logb\backslash a})$ if $d < \log_b a$

- Proof idea:

Figure 2.3 Each problem of size n is divided into a subproblems of size n/b.



    -
    - K th level made up of a^k subproblems, each of size n / b^k. Total work for each level is

$$a^k \times O\left(\frac{n}{b^k}\right)^d \quad = \quad O(n^d) \times \left(\frac{a}{b^d}\right)^k.$$

    - A geometric series with **ratio $r = a / b^d$**
        - If $r < 1$, series decreasing, first term
        - If $r > 1$, sum is last term
        - If $r = 1$, all logn terms are equal to n^d

2.3 Merge Sort - All sorting algorithm that relies on comparisons takes n log(n).

- Algorithm: split into sub parts, recursively sort, and merge the list.

```
Function mergesort (a[1...n])
If n > 1:
      Return merge (mergesort(a[1...n/2]), mergesort(a[n/2]+1...n)
Else
      Return a

Function 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 x1 + merge(x[2...k], y[1...l])
Else
      Return y1 + merge(x[1...k], y[2...l])
```

- $T(n) = 2\,T(n / 2) + O(n) \rightarrow$ **O(nlog(n))**

- Mergesort has the lower bound runtime for sorting. Consider the binary sorting tree. Every leaf is a permutation. Then there are n! Leafs. → n log(n)

- Proof: strong induction.

2.3.1 Quick sort

- ```
  Procedure quicksort(a[1..n])
  If n ≤ 1
        Return a
  Set v to be a random element in a
  Partition a into SL, Sv, SR
  Return quicksort(SL) SV quicksort(SR)
  ```

- Runtime: O(nlog(n)) expected runtime.
  ```

2.4 Median - all selection algorithm takes O(n)

- Sorting takes O(n log(n)) time, but we only care about the middle not the ordering.

- **Selection**
  - *Input*: list of numbers S, in integer k
  - *Output*: the kth smallest element of k

-
$$\text{selection}(S,k) = \begin{cases} \text{selection}(S_L,k) & \text{if } k \le |S_L| \\ v & \text{if } |S_L| < k \le |S_L| + |S_v| \\ \text{selection}(S_R, k - |S_L| - |S_v|) & \text{if } k > |S_L| + |S_v|. \end{cases}$$
- Shrink size of the sub-problem as max [S_L, S_R]
- If v is picked as the middle point, T(n) = T(n / 2) + O(n).

- **Efficiency Analysis**
  - Randomly choose v.
    - Best case: all mediums are picked. → O(n)
    - Worst case: pick in decreasing/increasing order → O(n^2)
  - Close to **best case**

- Prove by fair coin (p. 61-62) E = 1 + ½ E, E = 2. T(n) ≤ T(3n/4) + O(n).
- On average, expected in linear time **O(n)**.
- **Quicksort** takes O(n logn) on average, outperforms other sorting; use the same way to pick v as median to sort the array.

2.5 Matrix Multiplication

- Matrix multiplication computes n^2 cells, each take O(n). → $O(n^3)$.

- Break into subproblems, *blockwise*.

$$XY = \begin{bmatrix} A & B \\ C & D \end{bmatrix} \begin{bmatrix} E & F \\ G & H \end{bmatrix} = \begin{bmatrix} AE+BG & AF+BH \\ CE+DG & CF+DH \end{bmatrix}$$

-
- $T(n) = 8\,T(n / 2) + O(n^2)$ → $\mathbf{O(n^3)}$
- Improved by genius algebra:

$$XY = \begin{bmatrix} P_5+P_4-P_2+P_6 & P_1+P_2 \\ P_3+P_4 & P_1+P_5-P_3-P_7 \end{bmatrix}$$

where

$$\begin{aligned} P_1 &= A(F-H) & P_5 &= (A+D)(E+H) \\ P_2 &= (A+B)H & P_6 &= (B-D)(G+H) \\ P_3 &= (C+D)E & P_7 &= (A-C)(E+F) \\ P_4 &= D(G-E) \end{aligned}$$

- Reduced runtime to $T(n) = \mathbf{7}\,T(n / 2) + O(n^2)$. → $\mathbf{O(n^{2.81})}$

Lecture Notes

5- 17 Search, sort, select.

- Reduced and conquer
    - T(n) = aT(n - b)
    - If a > 1, takes exponential time.

- **Search**
  Input: Sorted list of integers; target integer.
  Output: index of the target.

    - Binary tree: log(n); Any search algorithm takes **O(log(n))**.
    - Degenerate into two parts, solve and combine.

- **Sort**

- List of sorting methods.
  Bubble sort, insertion sort, selection sort → **O(n^2)**
  Quicksort, mergesort → **O(nlog(n))**.

- Runtime:
  - N! Comparisons must be made
  - Traverse down the binary search tree with n! Leaves. → log(n!) < n log(n).
  - **O(nlog(n))**: best runtime for any sorting algorithm that relies on comparisons between elements.

5-22 DC examples

Power of two

- Given n, compute the digits of 2^n in decimal.

  - Cn digits.
  - ```
    Procedure PoT(n)
    If n = 0: return 1
    If n = 1: return 2
    P = PoT(n/2)                          // even: p = 2^(n/2);
                                          // odd: p = 2^((n-1)/2)
    P = KSMult(P, P)
    If n mod 2 = 1: P = add(P, P).
    Return P
    ```

  - Runtime: T(n) = T(n/2) + O($n^{1.58}$)

Making a binary heap

- Insert n elements, each take O(log(n)). In total takes O(nlog(n)).

- DC: put (o_1, k_1) aside, break remaining part into 2 halves. Make object 1 the root and tickle it down

- T(n) = 2 * T(n/2) + O(logn).

- Cheat MS: L(n) = 2L(n/2) + 1; U(n) = 2* U(n/2) + $n^{1/2}$; → T(n) = **O(n)**.

Greatest overlap
    ```

- Sort the list, and break into two part based on the median value.
- Get the greatest overlap on two sub problems.
- Get the greatest overlap between two subsets.

Minimum Distance

- Base:
    - If n = 2, return the distance
- Break into 2 halves of size n/2, (by x-value)
- Gives us the min distance on each side, d_L, d_R
- Compare $x - d_L \leq x_i \leq x + d_L$

2.6 Fast Fourier Transform

- Multiply two degree-d polynomials.
- Will not be on the exam.

- Polynomials:
    - $A(x) = a_0 + a_1x^1 + \ldots + a_{n-1} x^{n-1}$