ADA lecture notes

Analysis and design of algorithms

These are the notes for analysis and design of algorithms course. The professor says it'll be an interesting course, let's see about that. I am using Obsidian and this is an amazing markdown editor! It has a lot of community plugins. Anyways, study now... xD

Here is a somewhat detailed overview.

1. ADA/Lecture 1 : Introduction to the course and grading.
2. ADA/Lecture 2 : Mastering Master Theorem
3. ADA/Lecture 3 : DAC
4. ADA/Lecture 4 : DAC continued
5. ADA/Lecture 5 : Last DAC
6. ADA/Lecture 6 : Dynamic programming 1
7. ADA/Lecture 7 : Knapsack problem
8. ADA/Lecture 8 : Sequence alignment
9. ADA/Lecture 9 : Matrix chains
10. ADA/Lecture 10 : Watashiwa Greed
11. ADA/Lecture 11 : Graph coloring problem (Lecture sched)
12. ADA/Lecture 12 : Greedy 3
13. ADA/Lecture 13 : Graph Algorithms 1 - Undirected DFS
14. ADA/Lecture 14 : Checking 2-Edge connectivity
15. ADA/Lecture 15 : Union and find datastructure
16. ADA/Lecture 16 : Path compression
17. ADA/Lecture 17 : Bellman Ford (DP)
18. ADA/Lecture 18 : Bellman Ford Continued, Network Flows

Lecture->1

It's like DSA version 2 (in terms of management). Here's the link for previous year: ADA2020, ADA2022. Solutions and questions in this course are made by the instructor and hence making it public is not a good idea. So, these notes with stick around the lectures and maybe sometimes touching things but WILL NOT quote.

Evaluation

- Quizzes : 15% (n-1)
- Homework Assignments (Theory) : 15% (group of two)
- Programming Assignments : 10% (Foobar, No lab hours, Individual)
- Midsem : 30%
- Endsem : 30% Both theory

Multiplying large integers

Input : Two n-digit numbers A and B Output: Product $A \times B$ Primitive Ops: Add/Multiply two single digit integers (recall digital circuits adder)

- Classical pen-paper approach:
  - At max $2n$ operations per partial product, since n, we have $2n^2$
  - Summation of them, $2n^2$
  - Net $4n^2$
- Doing it differently: (Main idea: $\frac{n}{2}$ digits for each a, b, c, d)
  - $\overbrace{56}^{a}\overbrace{78}^{b} \times \overbrace{12}^{c}\overbrace{34}^{d}$
    1. Compute $a.c = 672$
    2. Compute $b.d = 2652$
    3. Compute $(a + b)(c + d) = 6164$
    4. Compute $\boxed{3. - 2. - 1.} = 2840$
    5. Put it all together $6720000 + 2652 + 284000$ (Notice the padding)
    6. Do it all recursively
  - Here's the recursive implementation, where $A = 10^{\frac{n}{2}} \cdot a + b, B = 10^{\frac{n}{2}} \cdot c + d$ and $A \times B = 10^n ac + 10^{\frac{n}{2}}(ad + bc) + bd$
    1. Recursively compute $a.c$
    2. Recursively compute $b.d$
    3. Recursively compute $(a+b)\cdot(c+d)$ (Karatsuba method, otherwise 4 recursive calls)
    4. Compute $\boxed{3. - 2. - 1.}$ for each call
    5. Pad and add!

Lecture->2

Analysis: The recurrence method

$T(n)$ = Runtime of Algorithm 1 for multiplying two n-digit numbers
Base Case : $n = 1, T(n) = c$: Multiplying two single digit numbers

Recurrence (for $n > 1$) : Express $T(n)$ as the runtime of recursive calls + additional work which maybe done in that call.

Recursively compute each ac, bd, bc, ad, work in this step

$$\begin{aligned} T(n) &= \overbrace{4T\left(\tfrac{n}{2}\right)}^{\text{computing ac,bd,bc,bd}} + \overbrace{c_1 \cdot n}^{\text{Adding 4 n/2 (+padded) digit no.s}} \\ T(1) &= \underbrace{c}_{\text{Base case}} \end{aligned}$$

Karatsuba's Algorithm (more like optimization) ($a + b$ may have $\frac{n}{2} + 1$ digits)

$$\begin{aligned} T(n) &= \overbrace{3T\left(\tfrac{n}{2}\right)}^{\text{n/2+1 but ignore}} + \overbrace{(c_2 + c_3) \cdot n}^{\text{Adding a+b, multiplying each other}} \\ T(1) &= \underbrace{c}_{\text{Base case}} \end{aligned}$$

Master Method / Master Theorem

A 'Black-box' method to solve many common recurrences in Algorithm design (especially DAC)

Assumption: All the recursive calls are made on subproblems of equal size. (If not, use the proof we will do now)

Assumption (for proof): Both constants c are equal.

$$\begin{aligned} T(n) &= aT\left(\tfrac{n}{b}\right) + c \cdot n^d \\ T(1) &\leq c \end{aligned}$$

$$a \geq 1, b \geq 1, c, d \geq 0$$

a = Number of recursive calls
b = Shrinkage factor of subproblem size
d = Affects the runtime of the additional work (outside recursion)

Master Theorem (Simpler Version): Prof. says no need to *remember*

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

Example: Mergesort, $T(n) = 2T(n/2) + c \cdot n$ so $a = 2, b = 2, d = 1$, so case 1. $T(n) = \mathcal{O}(n \log n)$

Example: Binary search, $T(n) = T(n/2) + c \cdot n^0$, so $a = 1, b = 2, d = 0$, so case 1. $T(n) = \mathcal{O}(\log n)$

Example: Multiplication algorithm1, $T(n) = 4T(n/2)+c_1 n$, so $a = 4, b = 2, d = 1$, so case 3. $\mathcal{O}(n^{\log_4 2})$

Example: Multiplication algorithm (Karatsuba), $T(n) = 3T(n/2) + c_1 n$, so $a = 3, b = 2, d = 1$, so case 3. $\mathcal{O}(n^{\log_3 2})$

- The calculator uses Strassen Schonenhage: $\mathcal{O}(n \log n \log \log n)$
- New proposed solution: $\mathcal{O}(n \log n)$

Proof of master theorem (Simpler version)

Assumtions:
1. n is a power of b (the shrinkage factor)
2. Base case: $T(1) = c$ (same as n^d)

Main technique:

- Recursion Trees (eww)
  - Levels: $\boxed{\log_b n + 1}$
  - Subproblems at level j: $\boxed{a^j}$
  - Subproblem size at level j: $\boxed{\frac{n}{b^j}}$
  - *Total* work done outside recursive calls at level j: $\boxed{a^j \cdot \left(\frac{n}{b^j}\right)^d \cdot c} = n^d \left(\frac{a}{b^d}\right)^j \cdot c$
    Should be intuitive from the above equation, Good = a, bad = b^d, in the end we just sum it all.
  - So, work is sum of total work across all levels

$$\sum_{j=0}^{\log_b n+1} n^d \left(\frac{a}{b^d}\right)^j \cdot c$$

Lecture->3

Divide and Conquer Algorithms

1. Divide (break into several parts)
2. Conquer (Solve the smallest solvable)
3. Combine (subproblems)

Counting Inversions in an Array

Input: $1, 3, 5, 2, 4, 6$
Output: 3
Inversion pairs: $(3, 2), (5, 2), (5, 4)$ {kind of like bubble sort}
Golden Benchmark to get inversions: Sorted array (ascending)
Trivial algorithm = $\mathcal{O}(n^2)$
Today: $\mathcal{O}(n \log n)$
Q. Can we output all inversions in same time above? No, total $O(n^2)$ possible invs.

- Key Ideas
  - Suppose A is divided in to X and Y (possibly in the middle)
  - An inversion pair (i, j) is :
    1. Left inversion : Both (i, j) in X
    2. Right inversion : Both (i, j) in Y
    3. Split inversion : i in X and j in Y
  - Using recursion get $(1), (2)$ and after you get the results, count split inversions.
  - Here's the pseudo code

```
CountInv(array& A, length n):
    if n==1 return 0
    X = A[1,2,...n/2], Y=A[n/2+1,...,n]
```

```
    x = CountInv(X,n/2)
    y = CountInv(Y,n/2)
    x = CountSplitInv(X,n/2)
    return x+y+x
Copy
```

- Now, since split inversions wont be affected if we sort each X and Y (along the recursive calls) it'll get easier to count inversions. We will use this to count split inversions while merging. `count += (n/2 - i + 1)` where i is the iterator of X and we are using the combine/merge function. This takes advantage of sorting.

Lecture->4

Closest pair of points in 2D

Input: A set of points with 2 coordinates
Distance (d) between two points is *Euclidean distance* in $2D$
Output: $(a, b) : d(a, b)$ is minimum
Assumption: All points have x and y coordinates (Non distinct left as exercise)

The 1-D case:
Sort the given points, and linearly traverse. So complexity $\mathcal{O}(n \log n)$

Back to 2D:
P_x be the set sorted by x-coordinate
P_y be the set sorted by y-coordinate (independent of other coordinate)
Now we choose the median using P_x. Then we have two sets, Q,R on left and right of the median (assume median on Q).

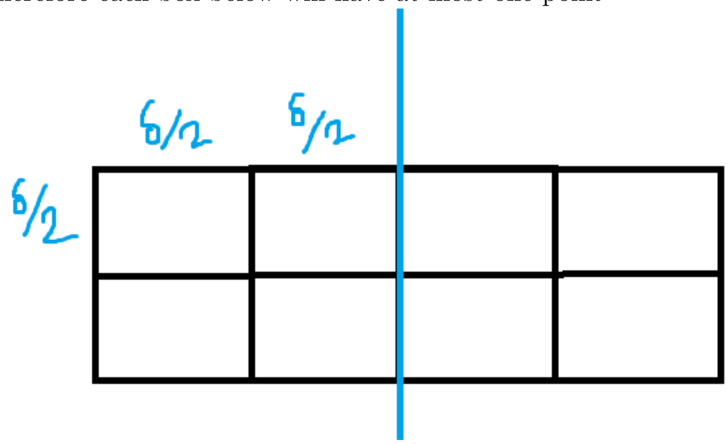
Recursion:

1. Compute Q_x, Q_y, R_x, R_y
2. $(p_1, q_1) = ClosestPair(Q_x, Q_y)$
3. $(p_2, q_2) = ClosestPair(R_x, R_y)$
4. Generate the sets Q_x, Q_y, R_x, R_y in $\mathcal{O}(n)$ time [Exercise]
5. $(p_3, q_3) = ClosestSplitPair(Q, R)$
6. get minimum of all 3

ClosestSplitPair:

1. Our search space will be restricted to $\pm\delta$ where $\delta = \min(ClosestPair(Q), ClosestPair(R))$
2. Note that this may not return a correct answer if the closest split pair does not lie in our restricted search space.
3. Calculations, we'll make it $\mathcal{O}(n)$; now assume the set S is the set of points contained in the predefined region.
4. Compute S_y (sorted by y coordinate) in linear time.

5. Traverse the list and apply the 1D algorithm but lookahead **seven** points instead of **one**. So complexity $\mathcal{O}(7n) \subseteq \mathcal{O}(n)$.
    - Proof of correctness of **seven**
    - We use the fact that our search space is restricted by $\pm\delta$ and what is δ
    - Therefore each box below will have at most one point



    - And also the box height is restricted by delta.
    - bdmtish, drumroll, so, we only need to compare with points which may be in these boxes. Therefore a linear algo.

Lecture->5

The selection problem

Input: An array (arbitrary), an integer i
Goal: Get i^{th} smallest number or i^{th} order statistic.

We'll see a linear time selection algo.

```
Select(A, length n, i)
    base: do something idk
    else:
        1. Choose any element as pivot = p
        2. Partition A around p (just like quicksort)
        3. Let j: position of p now
        4. if j==i then return P
        5. if j>i: recurse: return Select(A[1..j-1],j-1,i)
        6. if j<i: recurse: return Select(A[j+1..n], n-j, i-j)
Copy
```

Runtime:
We will analyze the worst case (i.e. we recurse on the larger subarray)

- $T(n) = c.n + T[\max(n - j, j - 1)]$

- Worst case: $c \cdot n + c \cdot (n-1) + c \cdot (n-2) \cdots \in \mathcal{O}(cn^2)$

So... `epic algorithm fail`.

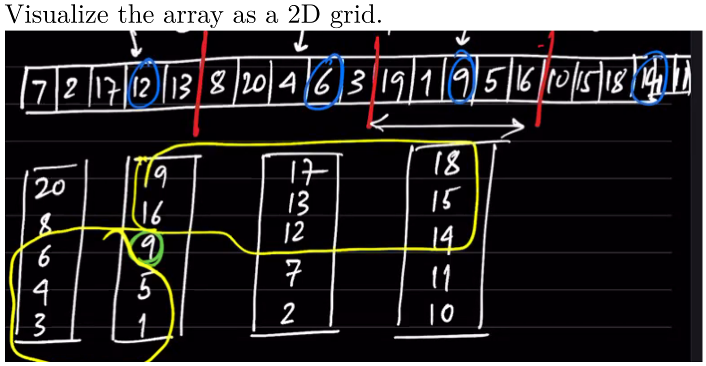
How to choose a good pivot fast?

Here are two choices

- Median (Best!) : But ...aren't you solving the median problem xD
- [30%-70%] (Good enough)
  - Randomization: Choose pivot as any element *uniformly at random*
  - Theorem: RSelect works in *expected* $\mathcal{O}(n)$ time
  - Deterministic pivot selection:
    * Break array A into buckets of 5 elements (i.e. $K = n/5$ groups)
    * Find median of each group $= e$ where $|e| = K$
    * pivot $p = Median(e)$
    * Let's re-write the algo:

```
DSelect(A,length n, i)
    base: idk
    else:
        1. Group A into chunks of 5
        2. e = set of medians from each
        3. p = DSelect( e, n/5 = K, K/2 )
        4. ...same as before
Copy
```

    * Runtime analysis: We have $T(n) = c \cdot n + T(n/5) + T(?)$
      · Key lemma: Median of e is bigger than (and also smaller than) atleast 30% of the set
      · Proof:
        Visualize the array as a 2D grid.



        Here, pivot is 9. now, let's generalize.

    We have the recurrence $T(n) \leq c \cdot n + T(n/5) + T(0.7 \cdot n)$

Lecture->6

We shall begin DP now. Who doesn't like DP ;)

First example

Input: A set of balls arranged in a row; each ball has a weight
Output: Pick balls of maximum possible total weight; No two adjacent balls can be picked

E.g. [2,5,6,4,3,5], pick as much as you can but maximize the weight.
It's easy to see how a greedy one would fail as always.

Let's try something different, here are some observations:

1. Iff last ball is not part of the optimal solution, then Optimal solution = Optimal solution with the last ball removed from input (strictly smaller than input)
2. Iff Last ball is part of the optimal solution so the second last ball cannot be, then Optimal solution = Optimal solution with last 2 balls removed + weight of last ball

Formal recurrence:
Let $opt[i]$: The optimal solution with balls $\{b_1, b_2, ... b_i\} \forall i = 0, 1, 2 ...$ (Subproblem definition).
$opt[0] = 0, opt[1] = weight(1)$ (assuming balls are positive, in case of generalizing with negative, just change the base case)

$$opt[i] = \begin{cases} opt[i-1] & case1 \\ opt[i-2] + weight(i) & case2 \end{cases}$$

Trivial proof of case 1 and case 2.
Now, say balls are $b_1, b_2 ... b_n$, then the optimal solution can either have b_n or not have it at all. If we try both, we have the recursive solution as $T(n) = T(n-1) + T(n-2) + c$ which is the fibonacci/exponential complexity.

We also observe here that in the recursion, the distinct calls are only n (since `SelectBalls(n-i)`). If we memo(r)ise and look up in $\mathcal{O}(1)$, we're done.

Solution 1: Using some recursion with lookup

```
Tab: Array of size n (memoization)
def SelectBalls(b1,b2,b3...bn):
    if Tab[n] is valid return Tab[n] else:
    if n==0, Tab[n] = 0,
    elif n==1, Tab[n] = weight of b1,
    else:
        w1 = SelectBalls(b1,b2,...bn-1)
        w1 = SelectBalls(b1,b2,...bn-2)+weight(bn)
        Tab[n]=max(w1,w2)
Copy
```

Solution 2: Linear time (bottom up) iterative

```
Tab: Array of size n+1
def Selectballs(b1,b2...bn):
```

```
    Tab[0] = 0
    Tab[1] = b1
    for i = 2,3,...n:
        Tab[i] = max(Tab[i-1], Tab[i-2]+weight(bi))
    return Tab[n]
Copy
```

Finn.

Lecture->7

Knapsack problem

Input:

- A knapsack of size $W > 0$ (integer)
- n different indivisible iterms
- item i has weight $w_i > 0$ and value $v_i > 0$ (ints)

Goal: To fill the knapsack (without overloading) to maximise total value.

DP based attempt 1

$OPT[i]$: optimal solution considering items $\{1, 2, 3 \dots i\}$ and knapsack of size W.

Case 1: item i is not part of the solution $OPT[i]$, so $OPT[i] = OPT[i-1]$

Case 2: item i is part of the solution $OPT[i]$

- inclusion of i does not mean that we need to reject $i-1$.
- But, we also cannot reduce to $OPT[i-1]$, we need to pack as much value as possible in knapsack of size $W - w_i$ and we are not watching this param.

DP based attempt 2

$OPT[i, w]$: optimal solution considering items $\{1, 2, 3 \dots i\}$ and knapsack of size W.

Case 1: item i is not part of the solution $OPT[i, w]$, so $OPT[i] = OPT[i-1, w]$

Case 2: item i is part of the solution $OPT[i]$, then $OPT[i, w] = OPT[i-1, w - w_i] + v_i$ when $w \geq w_i$.

Base case: $OPT[0, w] = 0 \ \forall \ w \in [W]$

So, we can polish this.

Recursive (with memoization)

```
M[i,w]: 2D array of size nxW, initialized to -1
def Knap(i,w):
```

```
    if M[i,w]==valid return M[i,w]
    if i==0: M[i,w]=0 #base case
    elif wi>w: M[i,w]=Knap(i-1,w)
    else M[i,w]=max(Knap(i-1,w), Knap(i-1,w-1)+vi)
    return M[i,w]
```
Copy

Dynamic (iterative) Algorithm

(Does not use additional stackspace so probably better)

```
M[i,w]: 2D array of size (n+1)x(W+1), init -1
def Knap(n,W):
    for w = 0 to W: M[0,w] = 0 #base case
    for i = 1 to n:
        for w = 0 to W:
            if wi>w: M[i,w] = M[i-1,w]
            else: max(M[i-1,w], M[i-1,w-1]+vi)
    return M[n,W]
```
Copy

Runtime $\mathcal{O}(nW)$ which is pseudopolynomial

Lecture -> 9

This lecture shall be skipped, this is same as assignment 2, Q3 (Robots in the auditorium problem)

Lecture -> 10

Minimizing Weighted Completion Time

A scheduling problem **Input**: Jobs $j_1, j_2 \dots j_n$, each job j_i has length l_i and weight w_i (kind of priority) **Output**: A schedule to minimize $\sum_{i=1}^{n} w_i \cdot C_i$ where C_i is the completion time of the i^{th} job. Idea: Typical mini max problem (refer SML, LDA/FDA) **Algorithm**: Sort the jobs in descending order of $\frac{w_i}{l_i}$ **Proof**: (Technique: Exchange argument/ WOP) idea:

- Start with the greedy schedule- Call it σ
- Suppose it is not optimal - Let $\sigma*$ be another schedule which is optimal
- Arrive at contradiction by producing even better solution

Lecture -> 12

Stable matching (/marriage)

Input: n wizards and n wands. Each wizard has a preference list, Each wand has a preference list. May generalize to m, n (however it is not guaranteed $\exists$ a stable matching).
Output: A stable matching
Stability: Unstable when for some matching M there are two candidates who might *cheat* on each other (both side should hold). Stable if no unstable lol.

The algorithm (Gale-Shapely):

From the wiki
Initialize: All wizards $w \in W$ and wands $v \in V$ are unmatched. **Iterate:** while $\exists\ w, w \ \to v$ and not tried every wand:

- assign wand if wand is single,
- check stable matching (i.e. check with wand if it's okay) and assign (w' is now unmatched),
- reject (i.e. w remains unmatched)

Observations:

1. Each wizard tries a wand in decreasing order of preference
2. Each wizard tries each wand at most once.
3. Once a wand has a holder, it never becomes free (it just gets a new wizard). However a wizard is dobee, they may become *free*.

Note: If the position of wands and wizards were flipped we may get a different stable matching. (move to bottom of the page)

Termination: Using observation 2, we see that n wizard can try at most n wands. Therefore, by $\mathcal{O}(n^2)$ we shall be done.

Proofs of correctness:

1. **Output is a Perfect Matching.**
   Proof: Suppose GS does not output a perfect matching, then $\exists$ some w who is free. Since we have n wizards as well as wands, $\exists$ some v which is also free. Also implies, v has not been tried, i.e. the algorithm has not terminated.
2. **Output is a Stable Matching** Proof: Suppose w, v is unstable (for the sake of contradiction)
   - Case 1, $w \overset{tried}{\not\to} v$. This means that $w \to v'$ which is at a lower priority, however this contradicts obeservation 1.
   - Case 2, $w \overset{tried}{\to} v$. However, then v would have already captured w' using observation 3. Even if w came by, w shall be rejected or w gets

the wand preliminarily but later forfeits their ownership.

We might consider the stable roommate problem i.e. pair n people given everyone has a preference list of $n-1$ remaining candidates. A stable matching may not exist.
In our setting, a stable matching shall always exist as shown above.

Now, size of input is $N = 2n^2 \in \Theta(n^2)$, then in terms of size of the input, our algorithm is just $\mathcal{O}(N)$.

To implement efficiently,
For each wand v, $Pref[v]$, we generate a converse preference list which essentially is $w \mapsto v$, so whenever a wizard is trying a wand, we can fetch their preference in constant time.

Which Stable-Matching does GS find?

Here are a few definitions lol.

- **Valid wand:** Wand v is a valid wand for wizard w if v can be assigned to w in *some* stable matching.
- **Wizard Optimal Matching:** A stable matching where *every* wizard gets the best possible valid wand.

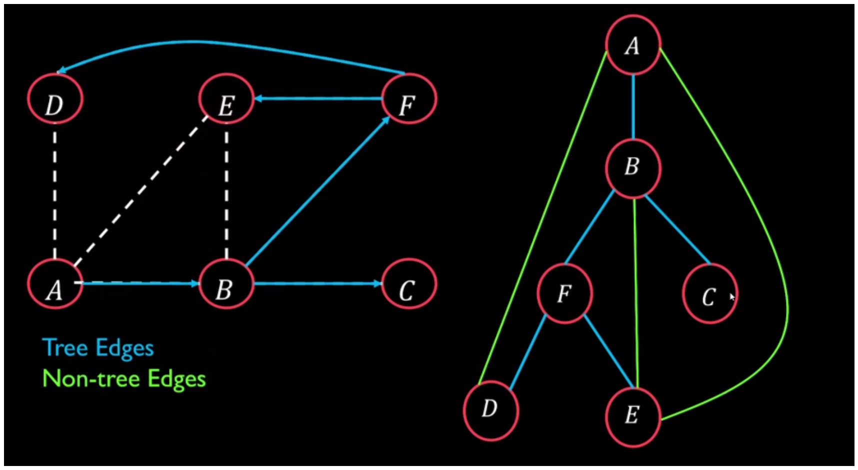
Theorem: GS produces a Wizard Optimal Matching. Therefore a unique stable matching.
Proof: Khud karlo
Corollary: GS produces a wand pessimal matching.

Lecture -> 13

- Revision of introductory GT (Tree, path, cycle, components, DS to store graphs)

DFS with timestamp (next lec.)

- Maintain a struct with Time stamp of arrival and departure for each vertex. Initialize to ∞ (disconnected components)
- Arrival time is the first arrival at that vertex
- Departure time is the last we will see that vertex (i.e. all possible subpaths have been explored)
- A DFS tree (formed by $n-1$ vertices, consider incidence of edge on vertex *only for a connected graph)
- Tree edge (used in DFS) and back edge (useless)
- An illustration



- **Property:** Ancestor relationship (only for *undirected DFS tree*). For a back-edge to exist. If (u, v) is a backedge then either of them must be an ancestor of the other.
  - For a directed graph, there shall be no need of ancestory. lol

```
visited: boolean[]
visited[v] = 0 forall v
arrival: number[]
departure: number[]
time: number = 0

function DFS(v){
    visited[v] = 1;
    arrival[v] = time++;
    for (w: adjlist of v)
        if (!visited[w]) DFS(w);
    departure[v] = time++
}
Copy
```

Time complexity arguments:

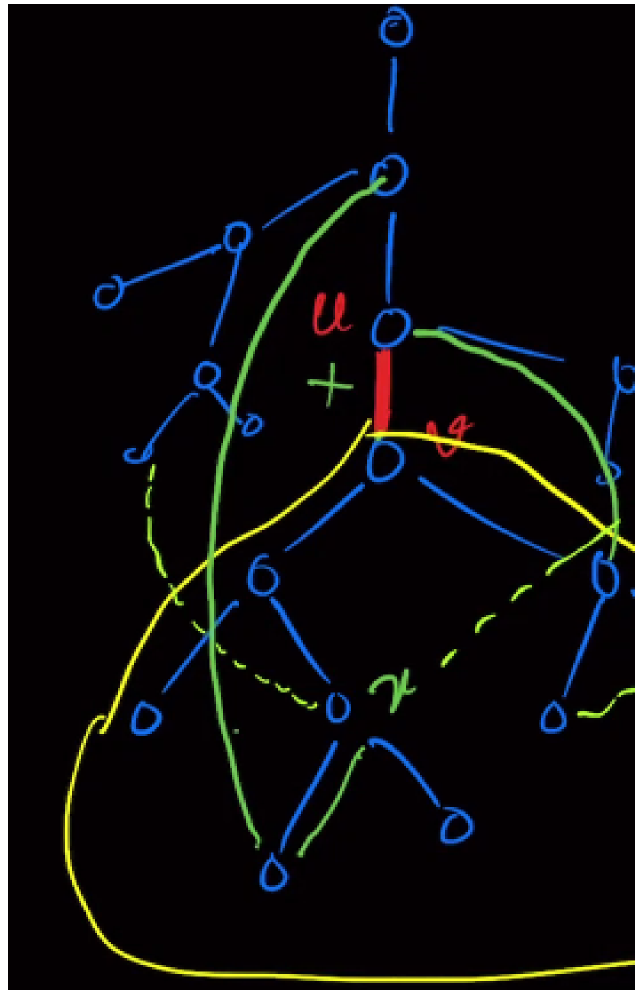
- We traverse the adjacency list of any vertex exactly once.
- So we see any edge e only once.
- And for initializers we have order $\mathcal{O}(V)$
- Final complexity is $\mathcal{O}(V + E)$, linear time.

Lecture -> 14

Application 1 of DFS

Checking 2-Edge Connectivity (Finding cut-edges or bridge-edges)

Naive: Repeat for every edge e, $G' = G - e$ is connected? Runtime is $\mathcal{O}(E(V + E))$



Smarter: Just one DFS! Recall DFS *tree* and back edges.

- Make a DFS tree, plot the back edges
- Note, for any edge $e \leftarrow (uv)$, if $\exists$ a back edge from subtree of v to ancestor of v then uv is not a bridge edge.
- dbe: deepest back edge

```
Check-2EC(v)
    visited[v] = 1;
    arr[v] = time++;
    dbe = arr[v];
    for (all w in neighbourhood of v):
        if !visited[w]:
            dbe = min(dbe, Check-2EC[w]);
```

```
        else:
            dbe = min(dbe, arr[w])
    if dbe = arr[v] then "report and abort"
    return dbe
Copy
```

- [not correct above] Take care of always true else condition for `uv` case (neighbourhood)
- another error (didnt understand) Runtime, same as DFS lol.

Application 2 of DFS

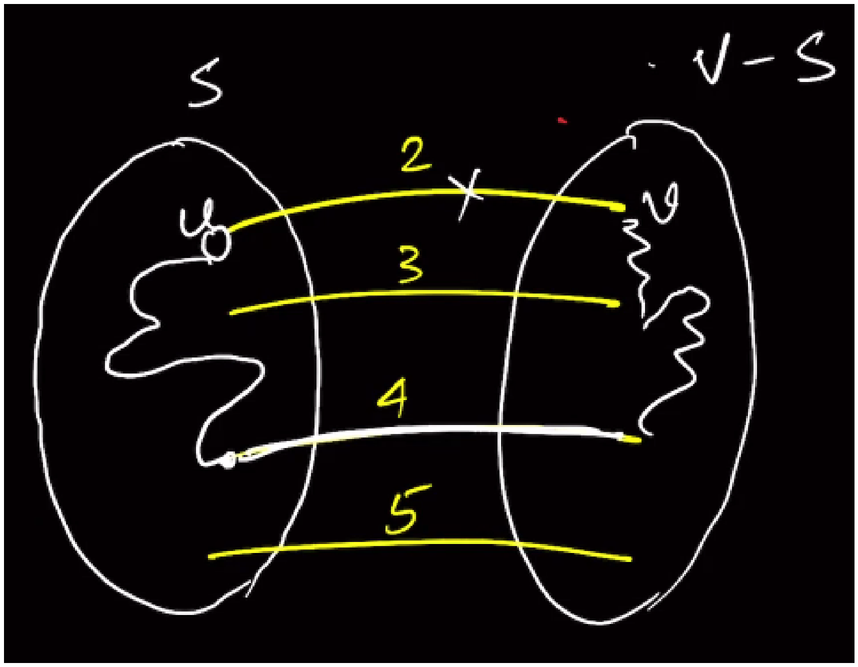
Checking Planarity of graphs (Ref. graph theory for def. of planar graph) Read about it Runtime is linear, POG

Minimum Spanning Tree

- Recall weighted graph and minimum sum of weight, priority Q

Lecture -> 15

- Cuts in a graph (combinations of vertices thrown in 2 buckets). Number of cuts given n vertices $2^{n-1} - 1$.
- **Cut property**: Consider any cut $(S, V - S)$. Then atleast the cheapest edge crossing S is part of the MST. Zhe proof: Say edges of form (uv) with weights $\{2, 3, 4, 5\}$ with vertex u in S and v in $V - S$. Say an arbitrary edge (except the one with lowest cost) from these is chosen to create a MST.



Now, the moment we choose the least weighted edge, we see a cycle. Finn.

NOTE: We do not claim that the two vertex sets as a result of a cut will be connected. Why?

Kruskal's Algorithm

naive and greedy (F)

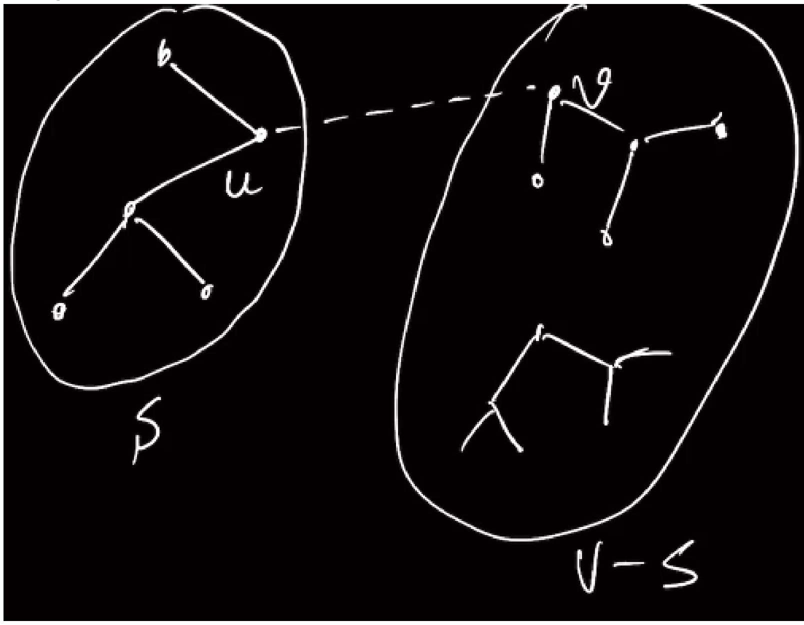
```
T = {}
edges.sort(increasing=True)
for i in [1..m]
    if T union {e_i} does not form a cycle
        T=T union {e_i}
Copy
```

Runtime: For the sorting part $\mathcal{O}(m \log n)$. Notice a simple graph. For the cycle detection part run a DFS with time complexity $\mathcal{O}(n)$. n because it is a tree so edges have the same order. Therefore the loop has total complexity of $\mathcal{O}(nm)$.

Correctness (Kruskal produces a MST) Proof: Say it does not produce a spanning tree.

1. No cycle? (But see `if` conditions)

2. A vertex is not connected. But it will always pick atleast an edge (therefore not creating a cycle), therefore spanning tree.



3. Minimality: Using the cut property
Also since sorted, we can add the least weighted edge. Finn

A faster method

Universe $P = \{p_1, p_2, p_2 ... p_n\}$, Set system $S_1, S_2 ..., S_k$ s.t. they are disjoint and $P = S_1 \bigcup S_2 \bigcup S_3 ... \bigcup S_k$ So, we can create a routine to find whether p_i belongs to which set. Then we apply a cycle detection if they belong to the same set. Finally, a union to continue the tree.

So, let's make a data structure. The idea, maintain an array, with elements as p_i such that each can host a pointer. In case of a union, we change pointer of the smaller set to the larger set. So we will always have a ***leader***. So, the find operation can easily follow the pointers and return the leader.

Claim: The max length of any chain with n nodes is $\leq \log_2 n$ Using induction. For $n = 2$ it is definitely 1. Suppose we have a set with n nodes formed by $n_1 \bigcup n_2$. WLOG, $[n_1] \leq [n_2]$. So, length of chain of $n_1 \leq \log_2 n_1$ and $n_2 \leq \log_2 n_2$. So length of longest chain = $\max(\log_2 n_2, \log_2 n_1 + 1)$. Argue that $\log_2 n_2 \leq \log_2(n_1 + n_2)$, therefore the maximum chain turns to be $\log_2 n$.

Finn.

We are going to $\log^*$ idk tf it is.

Lecture -> 16

Patch compression in Union DS

- Modify find such that the nodes directly point to the leader after reaching it.
- **Theorem:** m Find/Union operations can be doe in $\mathcal{O}(m \log^* n)$ time.
  - $\log^* n =$ no. of times you take log to make the input close to 1.

Shortest paths in directed graphs

- $\cdot 2^{n-2} + 1$ total paths. Consider starting and ending points, so the remaining $n-2$ vertices may or may not be included. And, another with direct path.
- For our algorithm $\mathbb{D}$ijkstra, we assume all *edge lengths are non-negative.*
- We maintain 2 sets, S containing the vertices for which the shortest path has been frozen, and $V - S$ for which shortest path might be modified.
- Initially S has the start vertex. We can freeze the neighbour with least edge weight (since non negative)
- Then, at each step shift the vertex w with minimum $d[w]$ to S. Where $d[v]$ is just the least or computed distance (which maybe just a tight upper bound) to vertex v from start. Note, once chosen, the vertex is now in the frozen set.
  - $\forall$ outgoing edges from W to neighbour v' in $V - S$ Update $d[v'] = \min(d[v'], d[w] + l_{wv'})$
- The pseudocode

$$d[v] = +\infty \ \forall \ v \in s; d[s] = 0$$
$$\text{H = Min-Heap(V, d[ ])}$$
$$\textit{while}(H \neq \textit{null})\{$$
$$w = \textit{deletemin}(H)$$
$$\forall\text{outgoing edges (w,v) to H}$$
$$d[v] = \min(d[v], d[w] + l_{wv})$$
$$\text{Update H (but not unnecessarily)}$$
$$\}$$

- Runtime analysis
  1. Takes $\mathcal{O}(V \log V)$: We call heapify v times.
  2. $\mathcal{O}(E \log V)$: Since inner loop runs at most E many times and inserting new edge(or updating in the minheap)
  3. Remember to augment the DS with auxilliary strucures to get $d[v]$ easily (Minheap by default only gives access to min element)
- Proof of correctness (why was dijksrta right):
  - wtf?