



## **CS/ECE 374B Review MT2**

---

Divide and Conquer, Dynamic Programming, and Graphs



# Disclaimers and Logistics

---



- **Disclaimer:** Some of us are CAs, but we have not seen the exam. We have no idea what the questions are. However, we've taken the course and reviewed Kani's previous exams, so we have **suspicions** as to what the questions will be like.
- This review session is being recorded. Recordings and slides will be distributed on ACM's + HKN's website after the end.
- **Agenda:** We'll quickly review all topics likely to be covered, then go through a practice exam, then review individual topics by request.
  - Questions are designed to be written in the same style as Kani's previous exams but to be *slightly* harder, so don't worry if you don't get everything right away!
- Please let us know if we're going too fast/slow, not speaking loud enough/speaking too loud, etc.
- If you have a question anytime during the review session, please ask! Someone else almost surely has a similar question.
- We'll provide a feedback form at the end of the session.

# Recursion

- **Definition:** Reducing the problem to a smaller instance of itself, where eventually we can terminate in a base case.
  - Think: If we have a problem of size  $n$ , we want to continuously reduce to a problem smaller than  $n$ .
  - Example: Tower of Hanoi

## Template

```
1: procedure AmazingRecursiveAlgo( $n$ )
2:   if  $n ==$  [some base case] then
3:     |   return [value]
4:   else
5:     |   return AmazingRecursiveAlgo( $n - 1$ )
```

- Similar to **induction!**

# Recursion: Runtime Analysis



$$T(n) = 2T(n/2) + O(n)$$

- **General Form:**

$$T(n) = \underbrace{r}_{\text{\# of subproblems}} \cdot \overbrace{T\left(\frac{n}{c}\right)}^{\text{work at each subproblem}} + \underbrace{f(n)}_{\text{work at current level}}$$



- Describes how the amount of work changes between each level of recursion.
- We can solve for a **time complexity** that describes the scaling behaviour of the algorithm at hand.

# Recursion: Runtime Analysis



- **General Form:**

$$T(n) = \underbrace{r}_{\text{\# of subproblems}} \cdot \overbrace{T\left(\frac{n}{c}\right)}^{\text{work at each subproblem}} + \underbrace{f(n)}_{\text{work at current level}}$$

- Describes how the amount of work changes between each level of recursion.
  - We can solve for a **time complexity** that describes the scaling behaviour of the algorithm at hand.
- **Master's Theorem**

## Master's Theorem

Decreasing:  $r \cdot f(n/c) = \kappa \cdot f(n)$  where  $\kappa < 1 \implies T(n) = O(f(n))$

Equal:  $r \cdot f(n/c) = \kappa \cdot f(n)$  where  $\kappa = 1 \implies T(n) = O(f(n) \cdot \log_c n)$

Increasing:  $r \cdot f(n/c) = \kappa \cdot f(n)$  where  $\kappa > 1 \implies T(n) = O(n^{\log_c r})$

$$T(n) = 2T(n/2) + O(n)$$

# Recursion: Runtime Analysis



- **General Form:**

$$T(n) = \underbrace{r}_{\text{\# of subproblems}} \cdot \overbrace{T\left(\frac{n}{c}\right)}^{\text{work at each subproblem}} + \underbrace{f(n)}_{\text{work at current level}}$$

- Describes how the amount of work changes between each level of recursion.
  - We can solve for a **time complexity** that describes the scaling behaviour of the algorithm at hand.
- **Master's Theorem**

## Master's Theorem

Decreasing:  $r \cdot f(n/c) = \kappa \cdot f(n)$  where  $\kappa < 1 \implies T(n) = O(f(n))$

Equal:  $r \cdot f(n/c) = \kappa \cdot f(n)$  where  $\kappa = 1 \implies T(n) = O(f(n) \cdot \log_c n)$

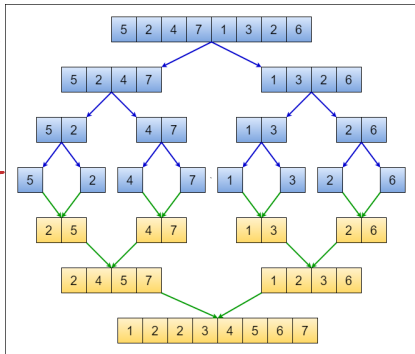
Increasing:  $r \cdot f(n/c) = \kappa \cdot f(n)$  where  $\kappa > 1 \implies T(n) = O(n^{\log_c r})$

- **Intuition:** If each level contains ~~the~~ more work than the level below it, then the root level will dominate. If each level contains the same amount of work, then we have  $\log_c n$  levels with  $f(n)$  work. If each level contains less work than the work below it, then the leaf nodes will dominate.

# Divide and Conquer Algos: Merge Sort



- **Purpose:** Sort an arbitrary array.
- **Time Complexity:**  $O(n \log n)$
- **Intuition:** Three phases: (a) split the array in half, (b) sort each side, (c) merge the sorted halves by repeatedly comparing smallest elements on each side not yet inserted.

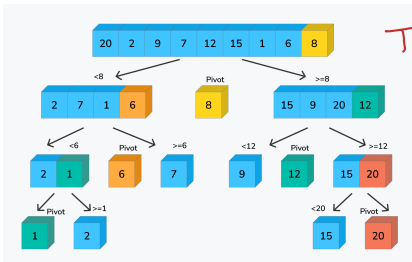


$$T(n) = 2T(n/2) + O(n)$$
$$O(n \log n)$$

# Divide and Conquer Algos: Quicksort



- **Purpose:** Sort an arbitrary array.
- **Time Complexity:** Avg:  $O(n \log n)$  | Worst:  $O(n^2)$  ( $O(n \log n)$  deterministic with quickselect partitioning)
- **Intuition:** Pick a pivot and rearrange the array such that all the elements that are less than the pivot value are to the left of the pivot value and all the elements that are greater than the pivot value are to the right of the pivot value. Then sort each side.
  - **Why the poor worst case performance?**
  - Because we can get unlucky and pick the worst possible pivot at every step.



$$T(n) = T(n-1) + O(n) \\ \sim O(n^2)$$



# Divide and Conquer Algos: Quickselect



- **Purpose:** Get the  $n^{\text{th}}$  smallest element in an arbitrary array.
- **Time Complexity:** Avg:  $O(n)$  | Worst;  $O(n^2)$ , ( $O(n)$  with MoM)
- **Intuition:** Pick a pivot  $P$  with a value  $P_V$  and rearrange the array such that all the elements that are less than  $P_V$  are to the left of  $P$  and all the elements that are greater than  $P_V$  are to the right of  $P$ , just like quick select. If the length of the array of elements that are less than  $P_V$  is greater than  $n$ , then we know that the  $n^{\text{th}}$  smallest element is to the left of  $P$  and we recurse on the left subarray. Otherwise, we know that the  $n^{\text{th}}$  smallest element is to the right of  $P$  and we recurse on the right subarray.
  - **Why the poor worst case performance?**
  - Again, because we can get unlucky and pick the worst possible pivot at every step.
  - We can guarantee linear performance with a better pivot-picking algorithm such as MedianOfMedians
    - Finds element that larger than  $\frac{3}{10}$  and smaller than  $\frac{7}{10}$  of the array's elements.
    - Runs in  $O(n)$  time

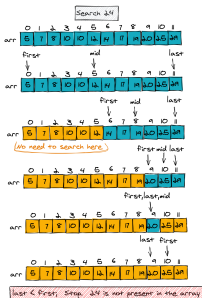
1 2 3 4 5  
1 2 3 4 5



# Divide and Conquer Algos: Binary Search



- **Purpose:** Find the existence of an element in a sorted array
- **Time Complexity:**  $O(\log n)$
- **Intuition:** Say we are trying to find the value  $n$ . Pick the middle element  $M$  in the array. If  $n > M$ , the element must be to the right of  $n$  and we recurse on the right. Otherwise, we recurse on the left.





# Backtracking

---



- Technique to methodically explore the solutions to a problem via the reduction to said problem to a smaller variant of itself, a.k.a **recursion**.
- Intuitively, think of the problem space as a maze that we are trying to find the exit of. For each path, you would traverse until you reach a dead end, at which point you **back track** to try a different path.
- To find recurrence, think "What information about a subset of my current problem space would be really nice to know?"

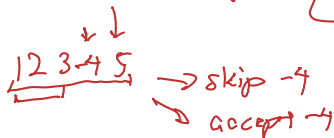
# Backtracking



- Technique to methodically explore the solutions to a problem via the reduction to said problem to a smaller variant of itself, a.k.a **recursion**.
- Intuitively, think of the problem space as a maze that we are trying to find the exit of. For each path, you would traverse until you reach a dead end, at which point you **back track** to try a different path.
- To find recurrence, think "What information about a subset of my current problem space would be really nice to know?" **Example:** Longest Increasing Subsequence
- "What is the length of a longest increasing subsequence in an arbitrary array?"

1 2 3 -4 5

$$LIS(i, j) = \begin{cases} 0 & \text{if } i=0 \\ LIS(i-1, j) & \text{if } A[i] \geq A[j] \\ \max \begin{cases} LIS(i-1, j) \\ 1 + LIS(i-1, i) \end{cases} & \text{otherwise} \end{cases}$$



# Backtracking



- Technique to methodically explore the solutions to a problem via the reduction to said problem to a smaller variant of itself, a.k.a **recursion**.
- Intuitively, think of the problem space as a maze that we are trying to find the exit of. For each path, you would traverse until you reach a dead end, at which point you **back track** to try a different path.
- To find recurrence, think "What information about a subset of my current problem space would be really nice to know?" **Example:** Longest Increasing Subsequence
- "What is the length of a longest increasing subsequence in an arbitrary array?"

$$\text{LIS}(i, j) = \begin{cases} 0 & \text{if } i = 0 \\ \text{LIS}(i - 1, j) & \text{if } A[i] \geq A[j] \\ \max \begin{cases} \text{LIS}(i - 1, j) \\ 1 + \text{LIS}(i - 1, i) \end{cases} & \text{else} \end{cases}$$

# Backtracking



- Technique to methodically explore the solutions to a problem via the reduction to said problem to a smaller variant of itself, a.k.a **recursion**.
- Intuitively, think of the problem space as a maze that we are trying to find the exit of. For each path, you would traverse until you reach a dead end, at which point you **back track** to try a different path.
- To find recurrence, think "What information about a subset of my current problem space would be really nice to know?" **Example:** Longest Increasing Subsequence
- "What is the length of a longest increasing subsequence in an arbitrary array?"

$$\text{LIS}(i, j) = \begin{cases} 0 & \text{if } i = 0 \\ \text{LIS}(i - 1, j) & \text{if } A[i] \geq A[j] \\ \max \begin{cases} \text{LIS}(i - 1, j) \\ 1 + \text{LIS}(i - 1, i) \end{cases} & \text{else} \end{cases}$$

This kind of sucks; we're redoing computation that we've already done! What if instead, we computed all the subproblems beforehand, wrote down the solutions, then did the recursion?

# Dynamic Programming

---



- It's backtracking, but we compute all of the subproblems iteratively.
  - This idea of "writing things down" as to not repeat computation is called **memoization**



# Dynamic Programming

---



- It's backtracking, but we compute all of the subproblems iteratively.
  - This idea of "writing things down" as to not repeat computation is called **memoization**
- Alternatively, you can think about this recursively, except we check our memoization structure to see if we've computed anything before. If we have, we just use the computed result. Otherwise, we compute the subproblem.

# Dynamic Programming

---



- It's backtracking, but we compute all of the subproblems iteratively.
  - This idea of "writing things down" as to not repeat computation is called **memoization**
- Alternatively, you can think about this recursively, except we check our memoization structure to see if we've computed anything before. If we have, we just use the computed result. Otherwise, we compute the subproblem.
- For a DP solution, we need:
  1. English Description
  2. Recurrence
  3. Memoization Structure
  4. Solution Location
  5. Evaluation Order
  6. Runtime

# Dynamic Programming

---



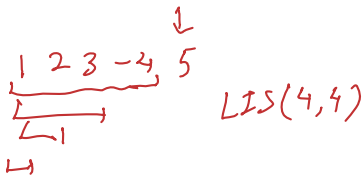
- It's backtracking, but we compute all of the subproblems iteratively.
  - This idea of "writing things down" as to not repeat computation is called **memoization**
- Alternatively, you can think about this recursively, except we check our memoization structure to see if we've computed anything before. If we have, we just use the computed result. Otherwise, we compute the subproblem.
- For a DP solution, we need:
  1. English Description
  2. Recurrence
  3. Memoization Structure
  4. Solution Location
  5. Evaluation Order
  6. Runtime
- **How to solve a DP:**
  - Identify how we can take advantage of a recursive call on a smaller subset of the input space.
  - Identity base cases
  - Identity recurrences (they should cover all possible cases at each step)

# Dynamic Programming



Let's look at the LIS example from before: "What is the length of a longest increasing subsequence in an arbitrary array?"

$$LIS[i,j] = \begin{cases} 1 & \text{if } i=j \\ LIS(i-1, j) & \text{if } A[i] \geq A[j] \\ \max \begin{cases} LIS(i-1, j) \\ 1 + LIS(i-1, i) \end{cases} & \text{otherwise} \end{cases}$$



LIS(4,4)

1 → 4

# Dynamic Programming



Let's look at the LIS example from before: "What is the length of a longest increasing subsequence in an arbitrary array?"


**LIS-Iterative**( $A[1..n]$ ):

$A[n + 1] = \infty$

**for**  $j \leftarrow 0$  to  $n$

~~$LIS[0][j] = 1$~~   $LIS[0][j] = 1$

**for**  $i \leftarrow 1$  to  $n - 1$  **do**

**for**  $j \leftarrow i$  to  $n - 1$  **do** 

**if**  $A[i] \geq A[j]$

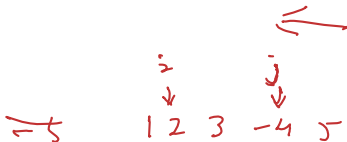
$LIS[i, j] = LIS[i - 1, j]$

**else**

$LIS[i, j] = \max \{ LIS[i - 1, j], 1 + LIS[i - 1, i] \}$

**return**  $LIS[n, n + 1]$

$A[0..2]$



# Graphs

- **Definition:** A set of vertices  $V$  connected by a set of edges  $E$ . Individual edges are notated as  $(u, v)$ , where  $u, v \in V$ .
  - They are usually represented as **adjacency lists** or **adjacency matrices**

$$G = (V, E)$$



$$u \rightarrow v$$

$$u \leftrightarrow v$$

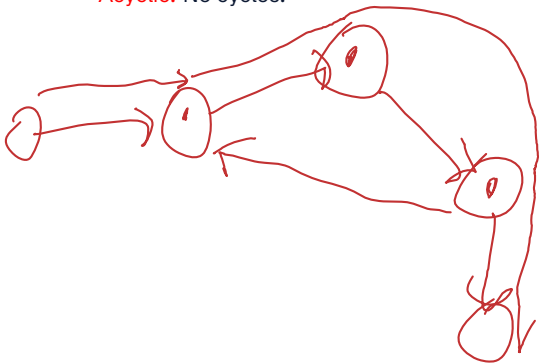


- **Definition:** A set of vertices  $V$  connected by a set of edges  $E$ . Individual edges are notated as  $(u, v)$ , where  $u, v \in V$ .
  - They are usually represented as **adjacency lists** or **adjacency matrices**
  - **Directed:** Each edge  $(u, v) \in E$  now has a direction  $u \rightarrow v$



# Graphs

- **Definition:** A set of vertices  $V$  connected by a set of edges  $E$ . Individual edges are notated as  $(u, v)$ , where  $u, v \in V$ .
  - They are usually represented as **adjacency lists** or **adjacency matrices**
  - **Directed:** Each edge  $(u, v) \in E$  now has a direction  $u \rightarrow v$
  - **Acyclic:** No cycles.





- **Definition:** A set of vertices  $V$  connected by a set of edges  $E$ . Individual edges are notated as  $(u, v)$ , where  $u, v \in V$ .
  - They are usually represented as **adjacency lists** or **adjacency matrices**
  - **Directed:** Each edge  $(u, v) \in E$  now has a direction  $u \rightarrow v$
  - **Acyclic:** No cycles.
- **Path:** A sequence of distinct vertices where each pair of consecutive vertices have an edge

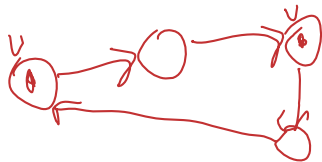
- **Definition:** A set of vertices  $V$  connected by a set of edges  $E$ . Individual edges are notated as  $(u, v)$ , where  $u, v \in V$ .
  - They are usually represented as **adjacency lists** or **adjacency matrices**
  - **Directed:** Each edge  $(u, v) \in E$  now has a direction  $u \rightarrow v$
  - **Acyclic:** No cycles.
- **Path:** A sequence of distinct vertices where each pair of consecutive vertices have an edge
- **Cycle:** A sequence of distinct vertices where each pair of consecutive vertices have an edge **and** the first and last vertices are connected.

- **Definition:** A set of vertices  $V$  connected by a set of edges  $E$ . Individual edges are notated as  $(u, v)$ , where  $u, v \in V$ .
  - They are usually represented as **adjacency lists** or **adjacency matrices**
  - **Directed:** Each edge  $(u, v) \in E$  now has a direction  $u \rightarrow v$
  - **Acyclic:** No cycles.
- **Path:** A sequence of distinct vertices where each pair of consecutive vertices have an edge
- **Cycle:** A sequence of distinct vertices where each pair of consecutive vertices have an edge **and** the first and last vertices are connected.
- **Connected:**  $u, v \in V$  are connected  $\iff$  there exists a path between  $u$  and  $v$ .



- **Definition:** A set of vertices  $V$  connected by a set of edges  $E$ . Individual edges are notated as  $(u, v)$ , where  $u, v \in V$ .
  - They are usually represented as **adjacency lists** or **adjacency matrices**
  - **Directed:** Each edge  $(u, v) \in E$  now has a direction  $u \rightarrow v$
  - **Acyclic:** No cycles.
- **Path:** A sequence of distinct vertices where each pair of consecutive vertices have an edge
- **Cycle:** A sequence of distinct vertices where each pair of consecutive vertices have an edge **and** the first and last vertices are connected.
- **Connected:**  $u, v \in V$  are connected  $\iff$  there exists a path between  $u$  and  $v$ .
- **Strongly Connected:**  $u, v \in V$  are strongly connected  $\iff$  there exists a path between  $u$  and  $v$  and from  $v$  to  $u$ .

- **Definition:** A set of vertices  $V$  connected by a set of edges  $E$ . Individual edges are notated as  $(u, v)$ , where  $u, v \in V$ .
  - They are usually represented as **adjacency lists** or **adjacency matrices**
  - **Directed:** Each edge  $(u, v) \in E$  now has a direction  $u \rightarrow v$
  - **Acyclic:** No cycles.
- **Path:** A sequence of distinct vertices where each pair of consecutive vertices have an edge
- **Cycle:** A sequence of distinct vertices where each pair of consecutive vertices have an edge **and** the first and last vertices are connected.
- **Connected:**  $u, v \in V$  are connected  $\iff$  there exists a path between  $u$  and  $v$ .
- **Strongly Connected:**  $u, v \in V$  are strongly connected  $\iff$  there exists a path between  $u$  and  $v$  and from  $v$  to  $u$ .
- **Connected Component (of  $u$ ):** The set of all vertices connected to  $u$ .



- **Definition:** A set of vertices  $V$  connected by a set of edges  $E$ . Individual edges are notated as  $(u, v)$ , where  $u, v \in V$ .
  - They are usually represented as **adjacency lists** or **adjacency matrices**
  - **Directed:** Each edge  $(u, v) \in E$  now has a direction  $u \rightarrow v$
  - **Acyclic:** No cycles.
- **Path:** A sequence of distinct vertices where each pair of consecutive vertices have an edge
- **Cycle:** A sequence of distinct vertices where each pair of consecutive vertices have an edge **and** the first and last vertices are connected.
- **Connected:**  $u, v \in V$  are connected  $\iff$  there exists a path between  $u$  and  $v$ .
- **Strongly Connected:**  $u, v \in V$  are strongly connected  $\iff$  there exists a path between  $u$  and  $v$  and from  $v$  to  $u$ .
- **Connected Component (of  $u$ ):** The set of all vertices connected to  $u$ .
- **Strongly Connected Component:** A set of vertices a strongly connected component if each pair of vertices are strongly connected.

- **Definition:** A set of vertices  $V$  connected by a set of edges  $E$ . Individual edges are notated as  $(u, v)$ , where  $u, v \in V$ .
  - They are usually represented as **adjacency lists** or **adjacency matrices**
  - **Directed:** Each edge  $(u, v) \in E$  now has a direction  $u \rightarrow v$
  - **Acyclic:** No cycles.
- **Path:** A sequence of distinct vertices where each pair of consecutive vertices have an edge
- **Cycle:** A sequence of distinct vertices where each pair of consecutive vertices have an edge **and** the first and last vertices are connected.
- **Connected:**  $u, v \in V$  are connected  $\iff$  there exists a path between  $u$  and  $v$ .
- **Strongly Connected:**  $u, v \in V$  are strongly connected  $\iff$  there exists a path between  $u$  and  $v$  and from  $v$  to  $u$ .
- **Connected Component (of  $u$ ):** The set of all vertices connected to  $u$ .
- **Strongly Connected Component:** A set of vertices a strongly connected component if each pair of vertices are strongly connected.

# Graph Algorithms: Traversal

---





# Graph Algorithms: Traversal

---



- **BFS:**
  - **Purpose:** Reachability, Shortest Path (unweighted graph)
  - **Implementation details:** Add your neighbours to a **queue**, pop from the queue to get next node
  - **Runtime:**  $O(V + E)$

# Graph Algorithms: Traversal

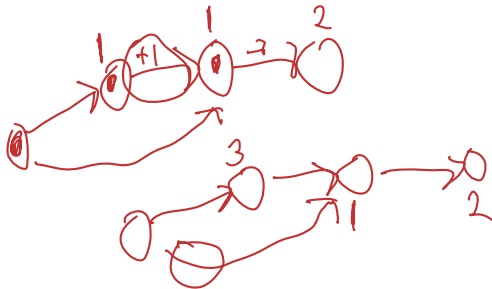


- **BFS:**

- **Purpose:** Reachability, Shortest Path (unweighted graph)
- **Implementation details:** Add your neighbours to a **queue**, pop from the queue to get next node
- **Runtime:**  $O(V + E)$

- **DFS:**

- **Purpose:** Reachability, toposort
- **Implementation details:** Add your neighbours to a **stack**, pop from the stack to get next node
- **Runtime:**  $O(V + E)$



# Graph Algorithms: Shortest Path

---



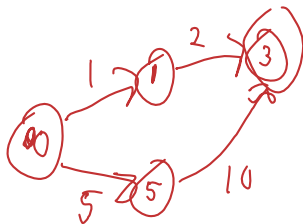
# Graph Algorithms: Shortest Path



- Dijkstra's
  - **Purpose:** SSSP, no negative edges
  - **Implementation:** Visit neighbours in **priority queue**
  - **Runtime:**  $O(m + n \log n)$

$$m = |E|$$

$$n = |V|$$



# Graph Algorithms: Shortest Path



- **Dijkstra's**
  - **Purpose:** SSSP, no negative edges
  - **Implementation:** Visit neighbours in **priority queue**
  - **Runtime:**  $O(m + n \log n)$
- **Bellman-Ford:**
  - **Purpose:** SSSP, yes negative weights. Will detect negative cycles.
  - **Implementation:** Dynamic Programming recurrence
  - **Runtime:**  $O(mn)$

BF( $v, n$ ) LSP  $u \rightarrow v$  using at most  
 $n$  edges

# Graph Algorithms: Shortest Path



- **Dijkstra's**
  - **Purpose:** SSSP, no negative edges
  - **Implementation:** Visit neighbours in **priority queue**
  - **Runtime:**  $O(m + n \log n)$
- **Bellman-Ford:**
  - **Purpose:** SSSP, yes negative weights. Will detect negative cycles.
  - **Implementation:** Dynamic Programming recurrence
  - **Runtime:**  $O(mn)$
- **Floyd-Warshall:**
  - **Purpose:** APSP, yes negative edge weights
  - **Implementation:** Dynamic Programming recurrence
  - **Runtime:**  $O(n^3)$

$FS(u, v, i) =$  SP  $u \rightarrow v$  only going through  
 $v$ 's  $1 \dots i$

# Graph Algorithms: MSTs

---



3 main algorithms:

- **Prim-Jarnik:** Keep a priority queue for edges to be added to the tree. Start the tree at some arbitrarily selected root vertex. When adding a vertex, add all of its neighbors to the queue. Runtime:  $O(|E| \log |V|)$ ,  $O(|V| \log |V| + |E|)$  using Quake heaps.

# Graph Algorithms: MSTs

---



3 main algorithms:

- **Prim-Jarnik:** Keep a priority queue for edges to be added to the tree. Start the tree at some arbitrarily selected root vertex. When adding a vertex, add all of its neighbors to the queue. Runtime:  $O(|E| \log |V|)$ ,  $O(|V| \log |V| + |E|)$  using Quake heaps.
- **Kruskal:** Keep a disjoint-sets data structure to keep track of connected components. Sort the edges, then in order, add each edge if it connects two components. Runtime:  $O(|E| \log |V|)$ .



# Graph Algorithms: MSTs

---



3 main algorithms:

- **Prim-Jarnik:** Keep a priority queue for edges to be added to the tree. Start the tree at some arbitrarily selected root vertex. When adding a vertex, add all of its neighbors to the queue. Runtime:  $O(|E| \log |V|)$ ,  $O(|V| \log |V| + |E|)$  using Quake heaps.
- **Kruskal:** Keep a disjoint-sets data structure to keep track of connected components. Sort the edges, then in order, add each edge if it connects two components. Runtime:  $O(|E| \log |V|)$ .
- **Borůvka:** No fancy data structures! Just find smallest edge going out of each vertex, then contract all edges that you selected! Runtime:  $O(|E| \log |V|)$

# Graph Algorithms: MSTs

---



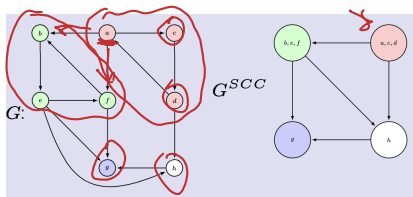
3 main algorithms:

- **Prim-Jarnik**: Keep a priority queue for edges to be added to the tree. Start the tree at some arbitrarily selected root vertex. When adding a vertex, add all of its neighbors to the queue. Runtime:  $O(|E| \log |V|)$ ,  $O(|V| \log |V| + |E|)$  using Quake heaps.
- **Kruskal**: Keep a disjoint-sets data structure to keep track of connected components. Sort the edges, then in order, add each edge if it connects two components. Runtime:  $O(|E| \log |V|)$ .
- **Borůvka**: No fancy data structures! Just find smallest edge going out of each vertex, then contract all edges that you selected! Runtime:  $O(|E| \log |V|)$
- Faster (but way more complicated algorithms) exist. **Yao** (1975):  $O(|E| \log \log |V|)$  with a modification of Borůvka's (using linear-time median selection). **Karger-Klein-Tarjan** (1995):  $O(|E|)$  in expectation, **Chazelle** (2000):  $O(|E| \alpha(|V|, |E|))$  deterministic

# Graph Algorithms: SCC

## SCC-Finding Algorithms (Tarjan's, Kosuraju's)

- **Purpose:** To identify (and collapse) SCCs in a (directed) graph
- **Runtime:**  $O(V + E)$
- **Returns:** A metagraph that has one node for each SCC.



# Graph Problems: General Stuff



## How to solve graph problems:

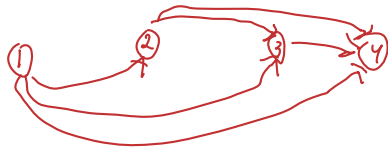
1. Identify type of problem (Reachability, Shortest Path, SCC)
2. Construct new graph
  - Add sources/sinks
  - Add vertices via  $V' = V \times \{\text{some set}\}$  (Useful for tracking states)
  - Add vertices via  $E' = E \times \{\text{some set}\}$  (Useful for allowing/prohibit certain behaviour)
3. Apply some stock algorithm
4. Draw connection between how to result of the algorithm upon the new graph relates to the solution of the original question.

$A(n)$

1, 2, 3, ...

$O(n) V$

$O(n^2) E$



# Recurrences and Asymptotics



Give a tight asymptotic bound for each recurrence:

- $T(n) = 4T(\frac{n}{2}) + n \log_2 n$

$$l, \quad 4^l \cdot \frac{n}{2^l} \log \frac{n}{2^l} = 4^l \cdot \left( \frac{n}{2^l} (\log n - l) \right)$$

$$\log n \quad \longrightarrow$$



$$4^{\log_2 n} \cdot \frac{n}{2^{\log_2 n}}$$

$$4^{\log_2 n} = (2^2)^{\log_2 n} = 2^{2 \log_2 n} = n^2 \quad \left( \text{So } (n^2) \right)$$

$$(a^b)^c = a^{bc}$$

# Recurrences and Asymptotics



Give a tight asymptotic bound for each recurrence:

- $T(n) = 4T(\frac{n}{2}) + n \log_2 n$
- $T(n) = T(\frac{3n}{4}) + T(\frac{n}{4}) + 5n$

$$5n$$
$$\frac{3 \cdot 5n}{4} + \frac{5n}{4} = 5n$$

$$\log_{4/3} n \sim \log n$$

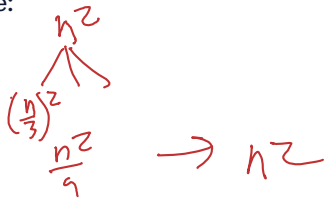
$$O(n \log n)$$

# Recurrences and Asymptotics



Give a tight asymptotic bound for each recurrence:

- $T(n) = 4T(\frac{n}{2}) + n \log_2 n$
- $T(n) = T(\frac{3n}{4}) + T(\frac{n}{4}) + 5n$
- $T(n) = 9T(\frac{n}{3}) + n^2$



$$n^2 \log_3 n \rightarrow O(n^2 \log n)$$

# Recurrences and Asymptotics



Give a tight asymptotic bound for each recurrence:

- $T(n) = 4T(\frac{n}{2}) + n \log_2 n$
- $T(n) = T(\frac{3n}{4}) + T(\frac{n}{4}) + 5n$
- $T(n) = 9T(\frac{n}{3}) + n^2$

Group the following functions s.t.  $f$  and  $g$  are in the same group if  $f(x) \sim \Theta(g(x))$ , and sort the groups by runtime:

- $n^n$
- $\log \log n$
- $374^n$
- $n!$

$$\log \log n \ll \log n \ll n \log n \ll n^{1.0001} \ll 2^n \ll 374^n \ll n! \ll n^n$$

- $\log(n + n^{374}) \ll \log(n^{375}) = 375 \log n \sim O(\log n)$
- $\log n^n \sim n \log n$
- $n^{1.000001}$

- $2^n$

- $n \log n^5$

- $\frac{1}{\log_n 2}$

$$\frac{1}{\log_n 2} = \frac{1}{\frac{\log_2 2}{\log_2 n}} = \log_2 n \sim O(\log n)$$



# Divide and Conquer



Consider the following (correct!) in-place sorting algorithm:

- 1: **procedure** StoogeSort( $A[1 \dots n]$ )
- 2:     If  $A[1] > A[n]$ , swap them. *O(1)*
- 3:     **if**  $n \geq 3$  **then**
- 4:         StoogeSort the initial 2/3 of  $A$   *$T(n/3)$*
- 5:         StoogeSort the final 2/3 of  $A$   *$T(2n/3)$*
- 6:         StoogeSort the initial 2/3 of  $A$  (again)  *$T(2n/3)$*

Give a *tight* asymptotic bound on the runtime of StoogeSort, in terms of  $n$ .

$$3^{\log_3 3/2 n} \rightarrow 3^{\frac{\log_3 n}{\log_3 3/2}}$$
$$\rightarrow n^{\log_3 3 - \log_3 2}$$
$$O(n^{1 - \log_3 2})$$

*w.p.h. #v*

$3^d \rightarrow 3^d$

$d = \log_{3/2} n$

$$3^{\log_{3/2} n}$$

*recursion tree diagram:*

```
graph TD
    A[ ] --- B[ ]
    A --- C[ ]
    A --- D[ ]
```

$$T(n) \leq 3 \cdot T(2n/3) + O(1)$$

Given a program represented as a directed graph  $G = (V, E)$  and function  $R$  where:

- Each vertex  $v \in V$  represents a function
- Each edge  $u \rightarrow v \in E$  represents that function  $u$  can call  $v$  internally
- $R(v)$  represents the "resource" that function  $v$  has *exclusive* access to during execution.  $R(v) \in \{1, 2, \dots, k\}$ .

You are given the "main function"  $s \in V$ . Considering that an execution path cannot have two functions using the same resource at the same time, write an algorithm to find the set  $D \subseteq V$  of functions that could not possibly be called as a result of running  $s$ .

# Dynamic Programming



You are given an array of integers  $A[1..2n]$ . On each turn, you choose two arbitrary integers with no other numbered integers between them. You then earn points equal to the division of your two chosen numbers, and both chosen integers are removed. Once a square is removed, it cannot be chosen in any future turns. Your goal is to remove all of the numbers and to earn as many points as possible. Describe an algorithm to find the maximum number of points you can earn in a game.

MaxScore(3, 8)

4	3	6	5	9	1	2	3	7	3	4	8
4	3	6	5	9	1	2	3			4	8
4	3	6	5			2	3			4	8
4	3	6					3			4	8
4	3									4	8
										4	8

+  $\frac{7}{3}$  points!

+  $\frac{9}{1}$  points!

+  $\frac{5}{2}$  points!

+  $\frac{6}{3}$  points!

+  $\frac{4}{3}$  points!

+  $\frac{4}{8}$  points!

All done!

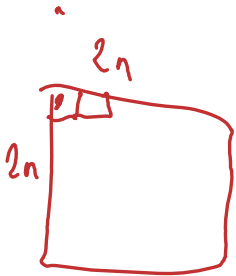
MaxScore(5, 6) + MaxScore(3, 4) + 145(7, 8)

$A[1..2n]$

$$\text{MaxScore}(i, j) = \begin{cases} i = j - 1 & 1/j \\ i \geq j & -\infty \\ i - j \% 2 = 0 & -\infty \end{cases} 0$$

otherwise

$\text{MaxScore}(1, 4)$



$(2n)^2$

for all  $i \leq a, b \leq j$   $a < b$

max  $\left\{ \begin{aligned} &a/b + \text{MaxScore}(i, a-1) + \\ &\text{MaxScore}(a+1, b-1) + \text{MaxScore}(b+1, j) \end{aligned} \right.$

for  $i$  in 2

$(2n)^2 \cdot (2n)^2$

$16n^4 \rightarrow O(n^4)$

# Graphs

[go.acm.illinois.edu/CS374b\\_mt2-feedback](http://go.acm.illinois.edu/CS374b_mt2-feedback)



It's late at night, and you're walking home. You have a graph  $G = (V, E)$  describing Champaign's road network, with each edge annotated as to whether or not the edge is lit by streetlights. Describe and analyze an efficient algorithm to calculate the shortest  $s \rightarrow t$  path for a given  $(s, t)$ , where at most  $k$  edges are unlit.

[go.acm.illinois.edu/CS374b\\_mt2-feedback](http://go.acm.illinois.edu/CS374b_mt2-feedback)

underscores

directed  
weighted

SP in  $G'$  from  $(u, j) \rightarrow (t, k)$  represents

the SP in  $G$  from  $u \rightarrow t$  using  $|u, j|$  unlit edge

$$V' = V \times \{1 \dots k\}$$

$$(u, i) \rightarrow (v, i+1) \in E' \text{ if } u \rightarrow v \text{ unlit, } i < k$$

$$(u, i) \rightarrow (v, i) \text{ if } i \text{ lit}$$

Run Dijkstra

$$\text{get } \min_{0 \leq i \leq k} (d((s, 0), (t, i)))$$

$$|V'| = k|V|$$

$$|E'| = k|E|$$

$$RT: E' + V' \log V' = O(k(E + V \log V))$$

