Hi everyone !!

CS 374A Review MT2

Electric Boogaloo

ACM @ UIUC

April 12, 2025

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 Sariel'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 EdStem 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

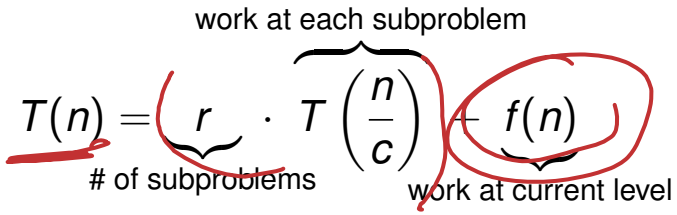
Template

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

- Similar to **induction**!

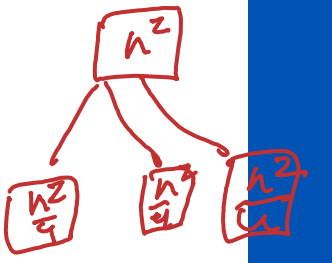
Recursion: Runtime Analysis

- **General Form:**

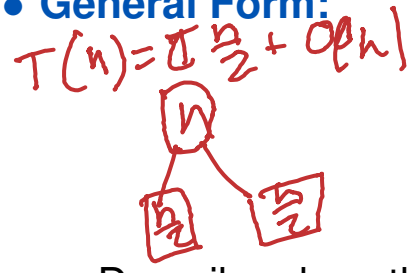


  - Describes how the amount of work changes between each level of recursion.
  - We can solve for a **time complexity** that describes the scaling behavior of the algorithm at hand.

Recursion: Runtime Analysis



- **General Form:**

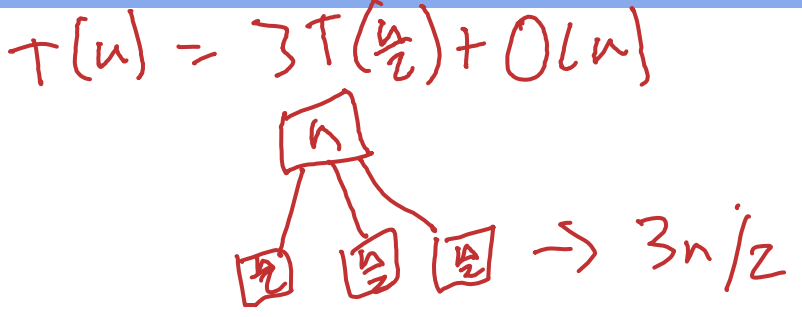


$$T(n) = \underbrace{r}_{\text{\# of subproblems}} \cdot \overbrace{T\left(\frac{n}{c}\right)}^{\text{work at each subproblem}} + \underbrace{f(n)}_{\text{work at current level}}$$

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

Master's Theorem

$$\begin{aligned}
\text{Decreasing: } r \cdot f(n/c) = \kappa \cdot f(n) \text{ where } \kappa < 1 &\implies T(n) = O(f(n)) \\
\text{Equal: } r \cdot f(n/c) = \kappa \cdot f(n) \text{ where } \kappa = 1 &\implies T(n) = O(f(n) \cdot \log_c n) \\
\text{Increasing: } r \cdot f(n/c) = \kappa \cdot f(n) \text{ where } \kappa > 1 &\implies T(n) = O(n^{\log_c r})
\end{aligned}$$



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 behavior of the algorithm at hand.
- **Master's Theorem**

Master's Theorem

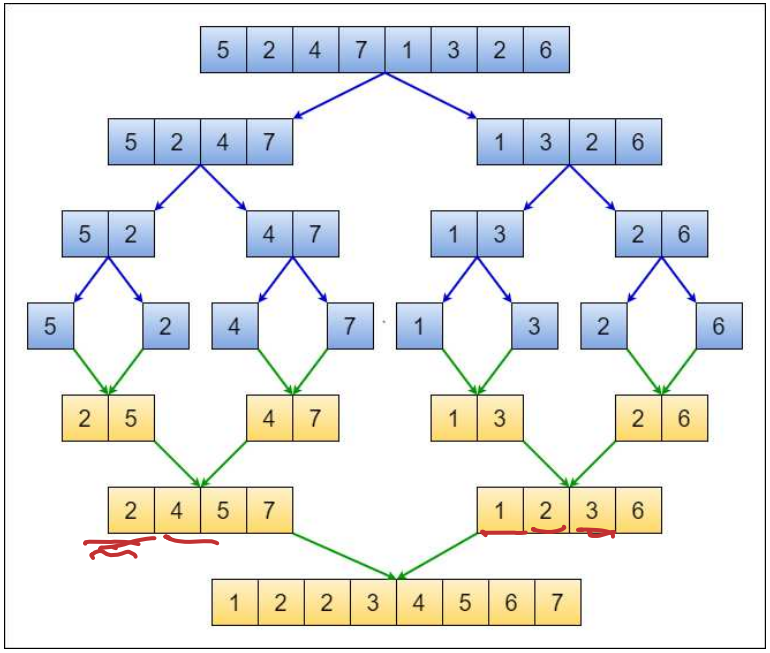
$$\begin{aligned}
\text{Decreasing: } r \cdot f(n/c) = \kappa \cdot f(n) \text{ where } \kappa < 1 &\implies T(n) = O(f(n)) \\
\text{Equal: } r \cdot f(n/c) = \kappa \cdot f(n) \text{ where } \kappa = 1 &\implies T(n) = O(f(n) \cdot log_c n) \\
\text{Increasing: } r \cdot f(n/c) = \kappa \cdot f(n) \text{ where } \kappa > 1 &\implies T(n) = O(n^{\log_c r})
\end{aligned}$$

  - **Intuition:** If each level contains 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.

$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$



122

Divide and Conquer Algos: Quickselect

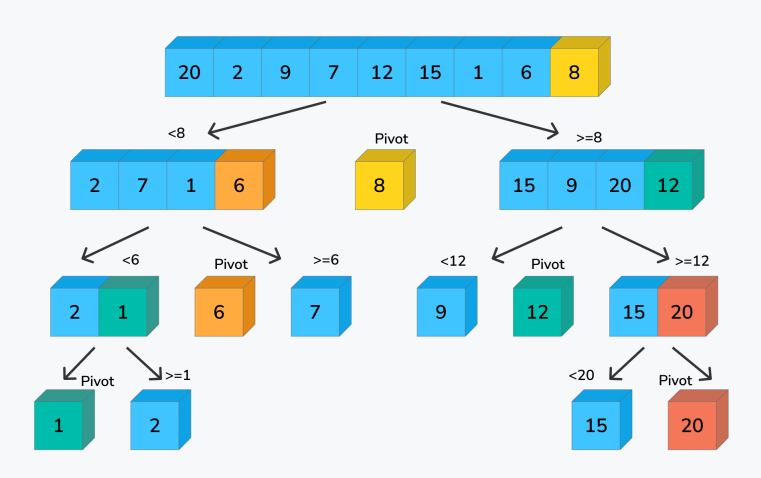
$T(n) = T(n/2) + O(n)$

$O(n)$

- **Purpose:** Get the n^{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^{th} smallest element is to the left of P and we recurse on the left subarray. Otherwise, we know that the n^{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

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.

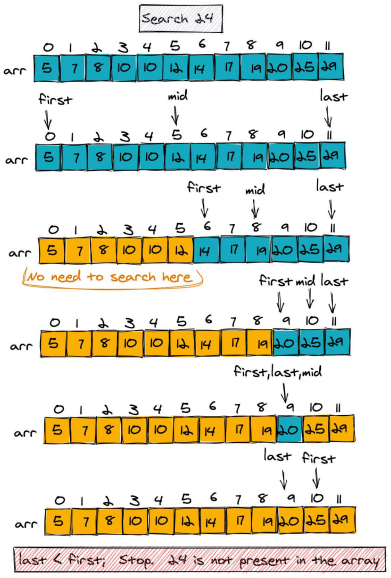


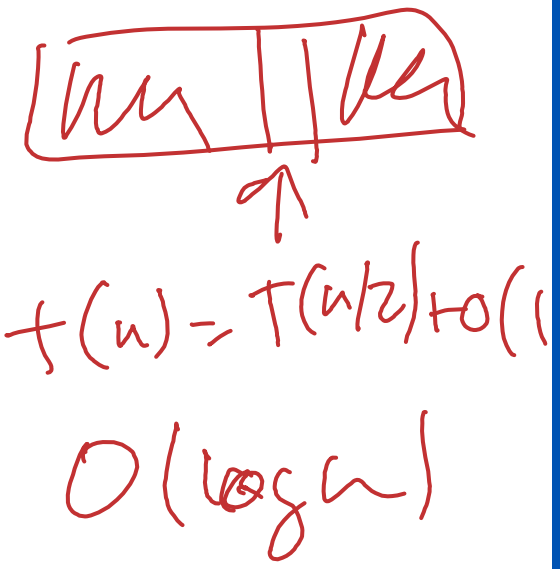
$T(n) = 2T(n/2) + O(n)$

$\rightarrow O(n \log n)$

Divide and Conquer Algos: Binary Search

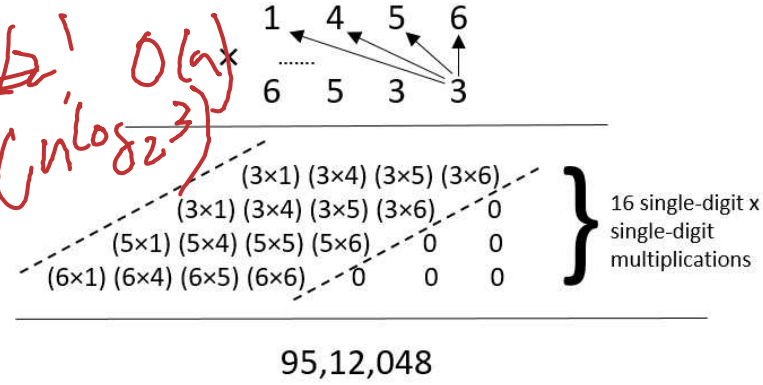
- **Purpose:** Find the existence of an element in a sorted array
- **Time Complexity**: $O(\log n)$
- **Intuition**: Say we are trying to find the value n. Pick the middle element M in the array. If $n > M$, the element must be to the right of n and we recurse on the right. Otherwise, we recurse on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





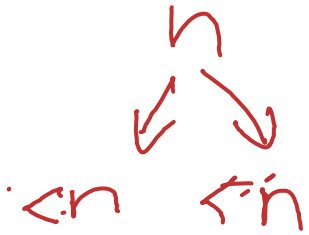
Divide and Conquer Algos: Karatsuba's

- **Purpose:** Multiplication
- **Time Complexity**: $O(n^{\log_2 3}) \approx O(n^{1.585})$
- 3 Phases:
  - **Divide**: Represent x as $x_0 p^n + x_1$, y as $y_0 p^n + y_1$
  - **Recurse**: Calculate $x_0 y_0$, $x_1 y_1$, $(x_0 + x_1)(y_0 + y_1)$
  - **Combine**: Use the three results to calculate our final answer.



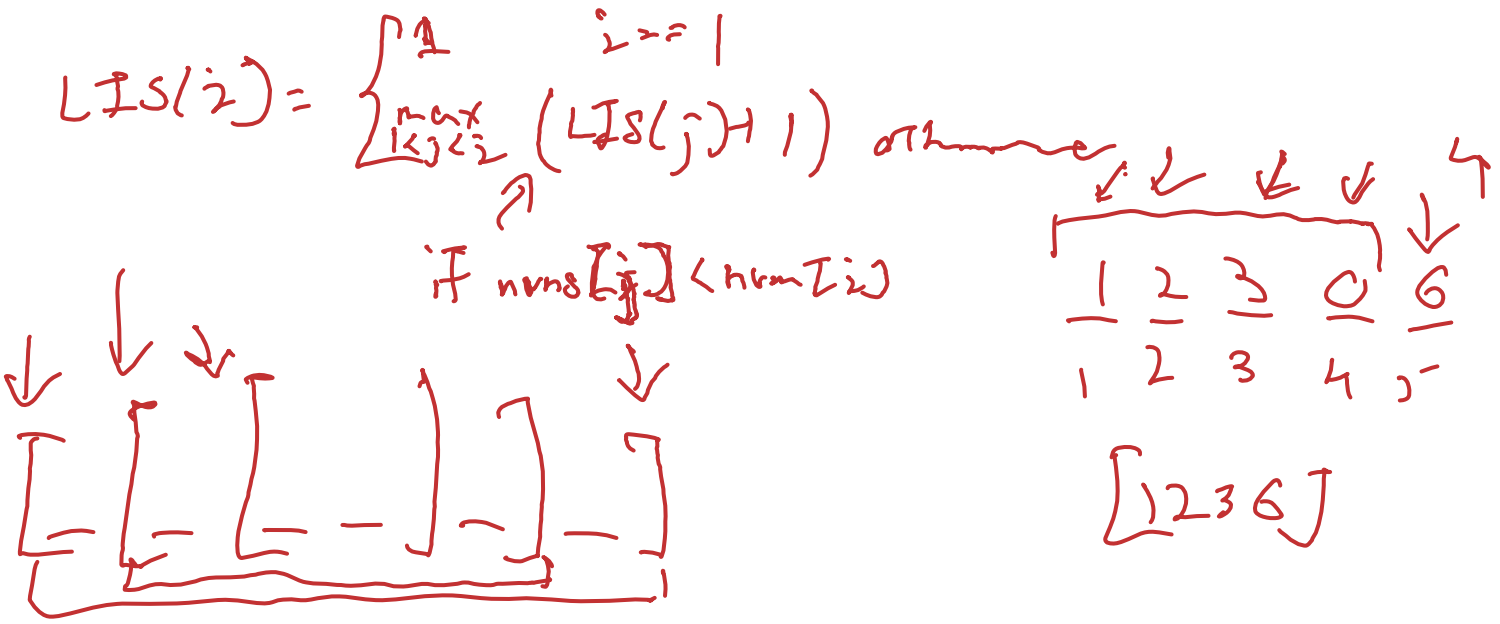
Backtracking

- 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?"



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

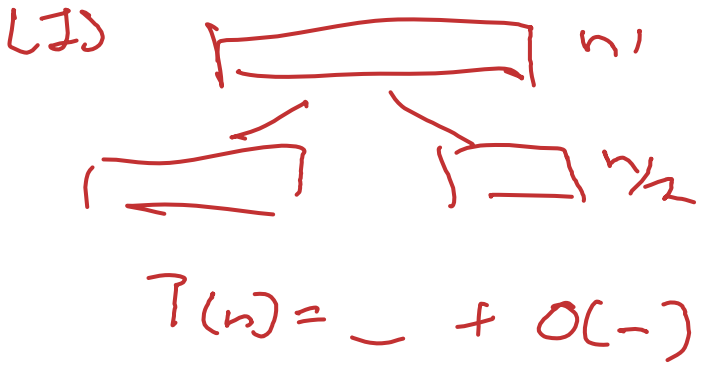
- It’s backtracking, but we compute all of the subproblems iteratively.
  - This idea of "writing things down" as to not repeat computation is called **memoization**

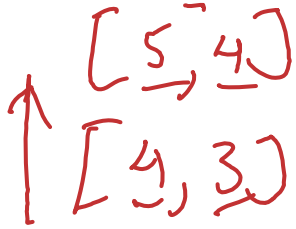
Dynamic Programming

- 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?"

$$LIS(i) = \begin{cases} 1 & i = 1 \\ \max_{1 \le j < i, A[j] < A[i]} (LIS(j) + 1) & \text{otherwise} \end{cases}$$

$LIS(n)$ $n = 5$

$LIS(n-1)$ $LIS(4)$ $i = 1 \to n$

Dynamic Programming

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

procedure LIS-ITERATIVE(A[1..n]):
- $A \leftarrow [1 \ldots n][1 \ldots n]$
- **for all** $i \leftarrow 1 \ldots n$ **do**
  - **for all** $j \leftarrow i \ldots n$ **do**
    - **if** $A[i] \leq A[j]$ **then**
      - $LIS[i][j] = 1$
    - **else**
      - $LIS[i][j] = 0$
- **for all** $i \leftarrow 1 \ldots n$ **do**
  - **for all** $j \leftarrow 2 \ldots n$ **do**
    - **if** $A[i] \geq A[j]$ **then**
      - $LIS[i][j] = LIS[i-1, j]$
    - **else**
      - $$LIS[i][j] = \max \begin{cases} LIS[i-1, j] \\ LIS[i-1, i] + 1 \end{cases}$$
- **return** LIS[n, n]

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**

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 $u \rightarrow v$

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 $u \to v$
  - **Acyclic:** No cycles.

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 $u \to v$
  - **Acyclic:** No cycles.
- **Path:** A sequence of distinct vertices where each pair of consecutive vertices have an edge

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 $u \to v$
  - **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.

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 $u \to v$
  - **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.

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 $u \to v$
  - **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.

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 $u \to v$
  - **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.

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 $u \to v$
  - **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.

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 $u \to v$
  - **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

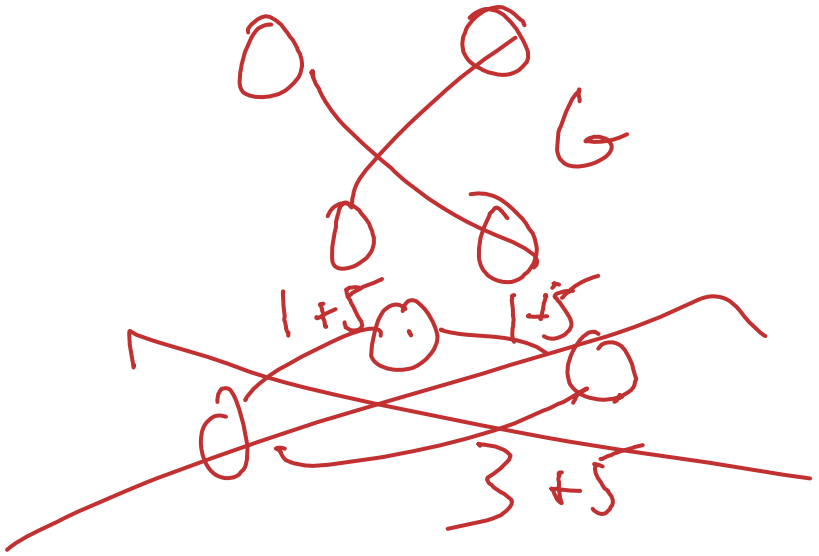
- **BFS:**
  - **Purpose:** Reachability, Shortest Path (unweighted graph)
  - **Implementation details:** Add your neighbours to a **queue**, pop from the queue to get next node
  - **Runtime:** $O(V + E)$

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

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



$G = (V, E)$ k neg

$G' = (V', E')$

$V' = V \times \{0 \ldots k\}$

$E' = (u, i) \to (v, i)$

$(u, i) \to (v, i+1)$

$wt(u \to v) + min\ wt$

Graph Algorithms: Shortest Path

- **Dijkstra's**
  - **Purpose:** SSSP, no negative edges
  - **Implementation:** Visit neighbours in **priority queue**
  - **Runtime:** $O(m \log n)$ (with **Quake Heaps**, $O(m + n \log n)$)
- **Bellman-Ford:**
  - **Purpose:** SSSP, yes negative weights. Will detect negative cycles.
  - **Implementation:** Dynamic Programming recurrence:
    - $d(v, k)$ is the shortest-walk distance from s to v using at most k edges
    - $d(v, k) = \min\left(d(v, k-1), \min_{u \to v} d(u, k-1) + \ell(u \to v)\right)$
  - **Runtime:** $O(mn)$

Graph Algorithms: Shortest Path

- **Dijkstra's**
  - **Purpose:** SSSP, no negative edges
  - **Implementation:** Visit neighbours in **priority queue**
  - **Runtime:** $O(m \log n)$ (with **Quake Heaps**, $O(m + n \log n)$)
- **Bellman-Ford:**
  - **Purpose:** SSSP, yes negative weights. Will detect negative cycles.
  - **Implementation:** Dynamic Programming recurrence:
    - $d(v, k)$ is the shortest-walk distance from s to v using at most k edges
    - $d(v, k) = \min\left(d(v, k-1), \min_{u \to v} d(u, k-1) + \ell(u \to v)\right)$
  - **Runtime:** $O(mn)$
- **Floyd-Warshall:**
  - **Purpose:** APSP, yes negative edge weights
  - **Implementation:** Dynamic Programming recurrence:
    - $d(u, v, i)$ is the shortest-path distance from u to v only going through vertices $1 \ldots i$.
    - $d(u, v, i) = \min\left(d(u, v, i), d(u, i, i-1) + d(i, v, i-1)\right)$
  - **Runtime:** $O(n^3)$

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

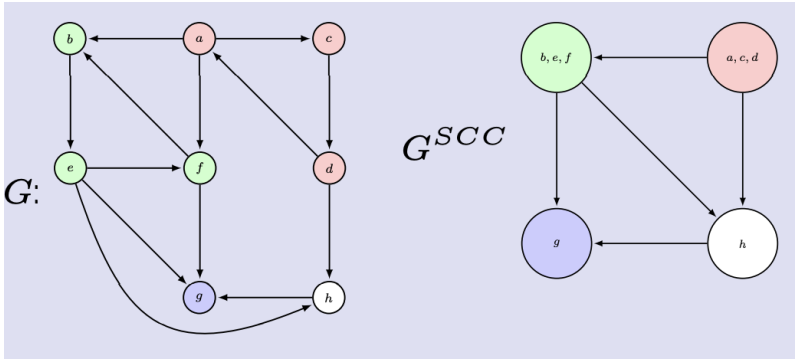
3 main algorithms:

- **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, Kosaraju'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 Algorithms: Longest Path

Longest path in a Directed Acyclic Graph (DAG)

- **Purpose:** To find the longest simple path (no repeating vertices) by weight in a graph which is guaranteed to be a DAG[1].
- **Runtime[2]:** $O(V + E)$
- **Returns:** The sum of the weights of the longest path in the DAG.

[1] Finding the longest path in other types of graphs is at least NP-hard.
[2] This is a relatively straight-forward DP on a DAG problem if you wish to derive it.

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 **(DO NOT MODIFY THE ALGORITHMS - MODIFY THE INPUTS!)**
4. Draw connection between how to result of the algorithm upon the new graph relates to the solution of the original question.

Recurrences and Asymptotics

Give a tight asymptotic bound for each recurrence:

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

$2n \lg \frac{n}{2} \Rightarrow 2n(\lg(n) - 1)$

$\lg \frac{n}{2^\ell} \Rightarrow (\lg n) - \ell$ at layer ℓ,

$4^\ell \frac{n}{2^\ell}(\log_2(n) - \ell)$

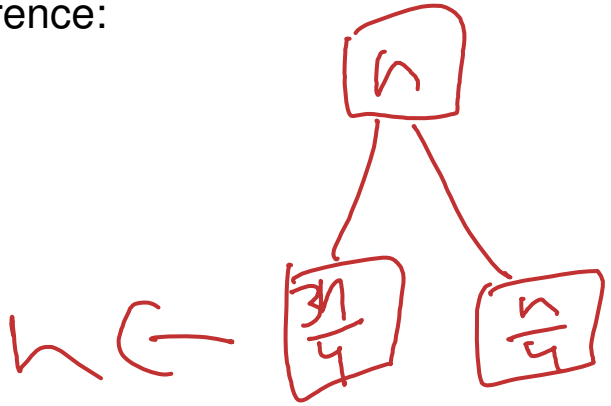
$\ell \sim \lg n \in O(n)$

$2^{\lg n - 1}\ n(\lg n - (\lg(n) - 1)) \Rightarrow O(n^2)$

Recurrences and Asymptotics

Give a tight asymptotic bound for each recurrence:

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



$\log_{4/3} n$ levels

$O(n \log n)$

Recurrences and Asymptotics

Give a tight asymptotic bound for each recurrence:

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

$$\log_a b = \frac{\log_c b}{\log_c a}$$

Recurrences and Asymptotics

Give a tight asymptotic bound for each recurrence:

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

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
- $n!$
- $\log(n + n^{374})$
- $\log n^n$
- $n^{1.000001}$
- 2^n
- $n \log n^5$
- $\frac{1}{\log_n 2}$

Recurrences and Asymptotics

Divide and Conquer

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

```
1: procedure StoogeSort(A[1 ... n])
2:     If A[1] > A[n], swap them.
3:     if n ≥ 3 then
4:         StoogeSort the initial 2/3 of A
5:         StoogeSort the final 2/3 of A
6:         StoogeSort the initial 2/3 of A (again)
```

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

Divide and Conquer

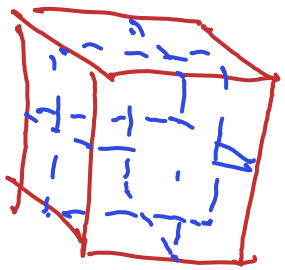
Divide and Conquer

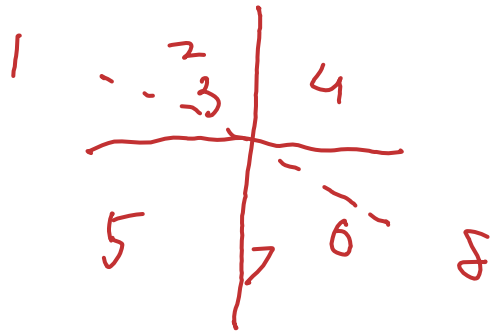
Suppose we are given a three-dimensional array $A[1 \ldots n, 1 \ldots n, 1 \ldots n]$ of distinct integers. An array element $A[i, j, k]$ is called a *local minimum* if it is smaller than its six immediate neighbors:

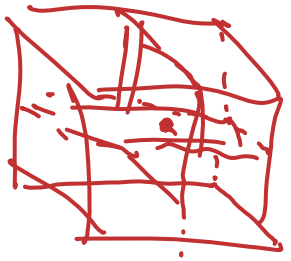
$$A[i,j,k] < \min \left\{ \begin{array}{ll} A[i-1,j,k], & A[i+1,j,k], \\ A[i,j-1,k], & A[i,j+1,k], \\ A[i,j,k-1], & A[i,j,k+1] \end{array} \right\}$$

To avoid edge cases, we assume all cells on the boundary of the array have value $+\infty$.

Describe and analyze an algorithm to find a local minimum in A as quickly as possible.







```
localMin2D(A [1..n][1..n][1..n]):
  if n < 20:
      brute force 3D min
  inputs = concat(A[1][1..n][1..n], A[n][1..n][1..n], A[1..n][1..n][1], A[1..n][1..n][n], A[1..n][1][1..n],
          A[1..n][n][1..n], A[⌊n/2⌋][1..n][1..n],A[1..n][⌊n/2⌋][1..n],A[1..n][1..n][⌊n/2⌋])
  minRow, minCol, minHeight = globalMin(inputs)
  if A[minRow][minCol][minHeight] < A[minRow-1][minCol][minHeight]
              and A[minRow][minCol][minHeight] < A[minRow+1][minCol][minHeight]
              and A[minRow][minCol][minHeight] < A[minRow][minCol+1][minHeight]
              and A[minRow][minCol][minHeight] < A[minRow][minCol-1][minHeight]
              and A[minRow][minCol][minHeight] < A[minRow][minCol][minHeight+1]
              and A[minRow][minCol][minHeight] < A[minRow][minCol][minHeight-1]:
      return True
  if minRow == ⌊n/2⌋ :
      if A[minRow][minCol][minHeight] > A[minRow-1][minCol][minHeight]:
          if minCol ≤ ⌊n/2⌋ and minHeight ≤ ⌊n/2⌋:
              return localMin(A[1..⌊n/2⌋][1..⌊n/2⌋],[1..⌊n/2⌋])
          else if minCol ≤ ⌊n/2⌋ and minHeight > ⌊n/2⌋:
              return localMin(A[1..⌊n/2⌋][1..⌊n/2⌋][⌊n/2⌋ ..n])
          else if minCol > ⌊n/2⌋ and minHeight ≤ ⌊n/2⌋:
              return localMin(A[1..⌊n/2⌋][⌊n/2⌋ ..n][1..⌊n/2⌋])
          else:
              return localMin(A[1..⌊n/2⌋][⌊n/2⌋ ..n][⌊n/2⌋ ..n])
      else if A[minRow][minCol][minHeight] > A[minRow+1][minCol][minHeight]:
          if minCol ≤ ⌊n/2⌋ and minHeight ≤ ⌊n/2⌋:
              return localMin(A[⌊n/2⌋ ..n][1..⌊n/2⌋][1..⌊n/2⌋])
          else if minCol ≤ ⌊n/2⌋ and minHeight > ⌊n/2⌋:
              return localMin(A[⌊n/2⌋ ..n][1..⌊n/2⌋][⌊n/2⌋ ..n])
          else if minCol > ⌊n/2⌋ and minHeight ≤ ⌊n/2⌋:
              return localMin(A[⌊n/2⌋ ..n][⌊n/2⌋ ..n][1..⌊n/2⌋])
          else:
              return localMin(A[⌊n/2⌋ ..n][⌊n/2⌋ ..n][⌊n/2⌋ ..n])
  if minCol == ⌊n/2⌋ :
      if A[minRow][minCol][minHeight] > A[minRow][minCol-1][minHeight]:
          if minRow ≤ ⌊n/2⌋ and minCol ≤ ⌊n/2⌋:
              return localMin(A[1..⌊n/2⌋][1..⌊n/2⌋],[1..⌊n/2⌋])
          else if minRow ≤ ⌊n/2⌋ and minCol > ⌊n/2⌋:
              return localMin(A[1..⌊n/2⌋][1..⌊n/2⌋],[⌊n/2⌋ ..n])
          else if minRow > ⌊n/2⌋ and minCol ≤ ⌊n/2⌋:
              return localMin(A[⌊n/2⌋ ..n][1..⌊n/2⌋][1..⌊n/2⌋])
          else:
              return localMin(A[⌊n/2⌋ ..n][1..⌊n/2⌋][⌊n/2⌋ ..n])
      else if A[minRow][minCol][minHeight] > A[minRow][minCol+1][minHeight]:
          if minRow ≤ ⌊n/2⌋ and minCol ≤ ⌊n/2⌋:
              return localMin(A[1..⌊n/2⌋][⌊n/2⌋ ..n][1..⌊n/2⌋])
          else if minRow ≤ ⌊n/2⌋ and minCol > ⌊n/2⌋:
              return localMin(A[1..⌊n/2⌋][⌊n/2⌋ ..n][⌊n/2⌋ ..n])
          else if minRow > ⌊n/2⌋ and minCol ≤ ⌊n/2⌋:
              return localMin(A[⌊n/2⌋ ..n][⌊n/2⌋ ..n][1..⌊n/2⌋])
          else:
              return localMin(A[⌊n/2⌋ ..n][⌊n/2⌋ ..n][⌊n/2⌋ ..n])
  ...
```

```
if minHeight == ⌊n/2⌋ :
  if A[minRow][minCol][minHeight] > A[minRow][minCol][minHeight-1]:
      if minRow ≤ ⌊n/2⌋ and minCol ≤ ⌊n/2⌋:
          return localMin(A[1..⌊n/2⌋][1..⌊n/2⌋][1..⌊n/2⌋])
      else if minRow ≤ ⌊n/2⌋ and minCol > ⌊n/2⌋:
          return localMin(A[1..⌊n/2⌋][⌊n/2⌋ ..n][1..⌊n/2⌋])
      else if minRow > ⌊n/2⌋ and minCol ≤ ⌊n/2⌋:
          return localMin(A[⌊n/2⌋ ..n][1..⌊n/2⌋][1..⌊n/2⌋])
      else:
          return localMin(A[⌊n/2⌋ ..n][⌊n/2⌋ ..n][1..⌊n/2⌋])
  else if A[minRow][minCol][minHeight] > A[minRow][minCol][minHeight+1]:
      if minRow ≤ ⌊n/2⌋ and minCol ≤ ⌊n/2⌋:
          return localMin(A[1..⌊n/2⌋][1..⌊n/2⌋][⌊n/2⌋ ..n])
      else if minRow ≤ ⌊n/2⌋ and minCol > ⌊n/2⌋:
          return localMin(A[1..⌊n/2⌋][⌊n/2⌋ ..n][⌊n/2⌋ ..n])
      else if minRow > ⌊n/2⌋ and minCol ≤ ⌊n/2⌋:
          return localMin(A[⌊n/2⌋ ..n][1..⌊n/2⌋][⌊n/2⌋ ..n])
      else:
          return localMin(A[⌊n/2⌋ ..n][⌊n/2⌋ ..n][⌊n/2⌋ ..n])
```

This algorithm gets the local minimum of a 3D array by first finding the global minimum in the list that is a concatenation of the boundary faces and the center faces along the x, y, and z axes. To guarantee that there is a local minimum in a 3D array, we always needs to guarantee that the boundary cells are all larger than the local minimum. Just like part a, we can use a divide and conquer scheme as well as recursion.

We first search through the boundary faces as well as the square slices that cut the cube in half against the x, y and z directions for the global min by scanning through each face. If the point we arrive act is a local min, we are done. If not, that means one of its neighbours is less than itself. We also know that the neighbour is not in any one of the faces we just searched. The slices that we just searched break up the cube in to octant. We recurse the algorithm on the octant that the neighbour that is less than the current global min of the above specified region is in. We can do this because a local min within his octant will be a local min within the either cube, and we know that a local min exists. (We know it exists as we can think about the each octant as a metric space in R^4. Since it is closed and bounded, by extreme value theorem, the minimum exists.) We continue this octant division an search until a local min is found.

Since we are searching 6 slices of the cube and then using that to search a region that is constructed with side lengths that is a half of the original region, we can write the recurrence relation as follows:

$T(n) = 6n^2 + T(n/2)$.

The work is dominated at the root and we do $6n^2$ per level, so the runtime is $O(n^2)$.

■

Divide and Conquer

Dynamic Programming

A shuffle of two strings X and Y is formed by interspersing the characters into a new string, keeping the characters of X and Y in the same order. For example, the string BANANAANANAS is a shuffle of the strings BANANA and ANANAS in several different ways.

BANANAANANAS BANANAANANAS BANANAANANAS

Similarly, the strings PRODGYRNAMAMMIINCG and DYPRONGARMAMMICING are both shuffles of the strings DYNAMIC and PROGRAMMING:

PRODGYRNAMAMMIINCG DYPRONGARMAMMICING

1. Given three strings $A[1..m]$, $B[1..n]$, and $C[1..m+n]$, describe and analyze an algorithm to determine whether C is a shuffle of A and B.
2. Given three strings $A[1..m]$, $B[1..n]$, and $C[1..m+n]$, describe and analyze an algorithm to determine the *number of different ways* that A and B can be shuffled to obtain C.

Dynamic Programming

There is exactly one optimal answer.

Dynamic Programming

The **StupidScript** language includes a binary operator @ that computes the *average* of its two arguments. For example, the StupidScript code `print(3 @ 6)` would print 4.5, because (3 + 6)/2 = 4.5.
Expressions like 3@7@4 that use the @ operator more than once yield different results when they are evaluated in different orders:

$$(3@7)@4 = 5@4 = 4.5 \quad \text{but} \quad 3@(7@4) = 3@5.5 = 4.25$$

Here is a larger example:

$$\begin{aligned}(((((8@6)@7)@5)@3)@(0@9)) &= 4.5\\ ((8@6)@(7@5))@((3@0)@9) &= 5.875\\ (8@(6@(7@(5@(3@0)))))@9 &= 7.890625\end{aligned}$$

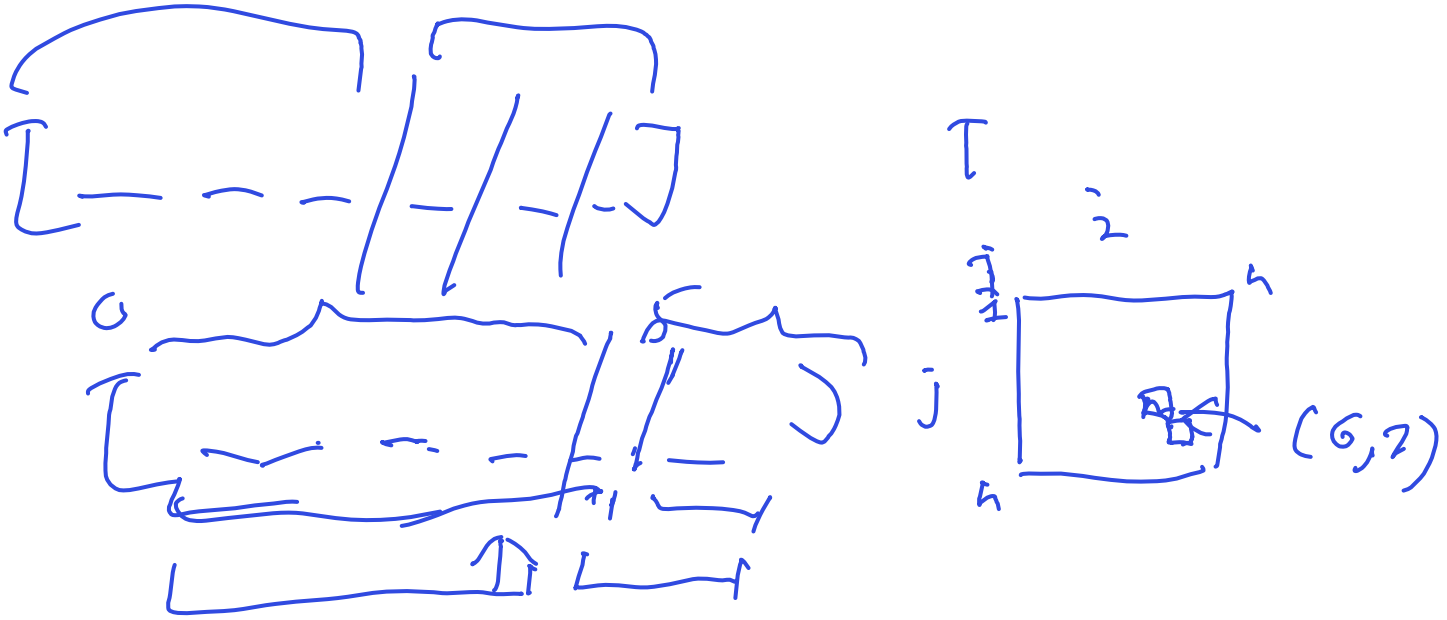
Your goal for this problem is to describe and analyze an algorithm to compute, given a sequence of integers separated by @ signs, the **largest possible** value the expression can take by adding parentheses. Your input is an array $A[1 \ldots n]$ listing the sequence of integers.

For example, if your input sequence is $[3, 7, 4]$, your algorithm should return 4.5, and if your input sequence is $[8, 6, 7, 5, 3, 0, 9]$, your algorithm should return 7.890625. Assume all arithmetic operations (including @) can be performed exactly in $O(1)$ time.

Dynamic Programming

$$SS(i,j) = \begin{cases} A[i] & i = j \\ \max\limits_{i \le k < j} SS(i,k) \oplus SS(k+1,j) & \end{cases}$$

$O(n)$



Graphs

You are in Austin, TX, which is lit at night by "moon towers"- high, powerful lamps which illuminate multiple blocks. You are given a weighted, undirected graph $G = (V, E)$ as a map of Austin, as well as a list of moon towers represented as pairs (p, v) where $v \in V$ is the location of the tower and $p \in \mathbb{R}_+$ is the *power* of the moon tower.

(a) A vertex v is considered *lit* if there exists a moon tower m s.t. $d(m, v) < p(m)$. Describe and analyze and algorithm to find all *lit* vertices.

$G' = (V', E')$

$V' = V \cup \{start\}$

$E' = E \cup \{start \to m,$ wt: max pwr $- p(m)\}$

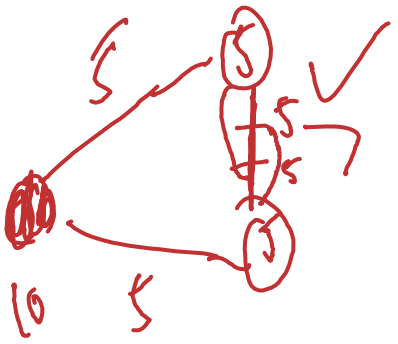
Dijkstra's, find all within max pwr of start

Graphs

You are in Austin, TX, which is lit at night by “moon towers”- high, powerful lamps which illuminate multiple blocks. You are given a weighted, undirected graph $G = (V, E)$ as a map of Austin, as well as a list of moon towers represented as pairs (p, v) where $v \in V$ is the location of the tower and $p \in \mathbb{R}_+$ is the *power* of the moon tower.

(a) A vertex v is considered *lit* if there exists a moon tower m s.t. $d(m, v) < p(m)$. Describe and analyze and algorithm to find all *lit* vertices.

(b) An edge $u \leftrightarrow v$ is *lit* if for all $0 \leq h \leq \ell(u \leftrightarrow v)$, there exists a moontower m s.t. either $d(m, u) + h < p(m)$ or $d(m, v) + \ell(u \leftrightarrow v) - h < p(m)$. Describe and analyze and algorithm to find all *lit* edges.



for $u \to v \in E$
if max - d(s,u)
+ max - d(s, v)
$\geq \ell(u,v)$;
else good / bad

Graphs

You are in Austin, TX, which is lit at night by "moon towers"- high, powerful lamps which illuminate multiple blocks. You are given a weighted, undirected graph $G = (V, E)$ as a map of Austin, as well as a list of moon towers represented as pairs (p, v) where $v \in V$ is the location of the tower and $p \in \mathbb{R}_+$ is the *power* of the moon tower.

(a) A vertex v is considered *lit* if there exists a moon tower m s.t. $d(m, v) < p(m)$. Describe and analyze and algorithm to find all *lit* vertices.

(b) An edge $u \leftrightarrow v$ is *lit* if for all $0 \leq h \leq \ell(u \leftrightarrow v)$, there exists a moontower m s.t. either $d(m, u) + h < p(m)$ or $d(m, v) + \ell(u \leftrightarrow v) - h < p(m)$. Describe and analyze and algorithm to find all *lit* edges.

(c) Using your answers to parts (a) and (b), describe and analyze and algorithm to calculate the shortest $s \to t$ path where at most k edges are unlit.

$G' = (V', E')$

$V' = V \times \{0 \ldots k\}$

$E' = (u,i) \to (v,i)$ if u-v lit

$(u,i) \to (v,i+1)$ else

Graphs

Graphs

New question!

(a) A train station is considered *critical* if it breaking down would result in one city in the network no longer being able to reach another city in the network by any path. Describe and analyze an algorithm to, given a train station t and a network G, determine if t is a *critical* train station,

Graphs

New question!

(a) A train station is considered *critical* if it breaking down would result in one city in the network no longer being able to reach another city in the network by any path. Describe and analyze an algorithm to, given a train station t and a network G, determine if t is a *critical* train station,

(b) Describe and analyze an algorithm to find all *critical* train stations.

Graphs

Graphs

The Cheery Hells neighborhood of Sham-Poobanana runs a popular and well-regulated Halloween celebration. To regulate the flood of costumed children, the Cheery Hells Neighborhood Association has designated a walking direction for each stretch of sidewalk.

At the entrance, each child receives a map of the neighborhood, in the form of a directed graph G, whose vertices represent houses. Each edge $v \to w$ indicates that one can walk directly from house v to house w following the designated sidewalk directions. A special vertex s designates the entrance to Cheery Hells. Children can visit houses as many times as they like, but biometric scanners at every house ensure that each child receives candy only at their *first* visit to each house.

The children of Cheery Hells have published a secret web site listing the amount of candy that each house in Cheery Hells will give to each visitor.

Describe and analyze an algorithm to compute the maximum amount of candy that a single child can obtain in a walk through Cheery Hells, starting at the entrance node s. The input to your algorithm is the directed graph G, along with a non-negative integer v.candy for each vertex v, describing the amount of candy the corresponding house gives to each first-time visitor.

Graphs

Feedback

- Further questions in ACM (Siebel 1104) after this
- Please fill out the feedback form:
  `go.acm.illinois.edu/cs374a_mt2_feedback`