Announcements

Asks of you
- Course evals by Sunday
- Course survey for 3 Socrative ec points
- TA nominations

Grades
- HW4 & HW6 scores go out later today
- HW

Logistics
- Final is on TUESDAY March 19th 8:30-10:20am here in Smith
- Erik Review: TODAY 3:30-5:30

What comes after 373?
- 417
- Projects
- TA
- internships
On the exam

Graphs

Graph definitions
- Directed vs undirected
- Weighted vs unweighted
- Walks vs paths vs cycles
- Self-loops and parallel edges
- Simple vs non-simple graphs (e.g. multigraphs)
- Trees, DAGs
- Strongly connected components

Graph representations
- Adjacency list
- Adjacency matrix
- Pros and cons of each

Graph algorithms
- Graph traversals: BFS and DFS
- Single-source shortest-path algorithms: Dijkstra's algorithm
- Topological sorts
- MST algorithms: Prim and Kruskal
- Disjoint sets

Sorting
- Quadratic sorts: insertion sort, selection sort
- Faster sorts: heap sort, merge sort, quick sort
- Understand the runtimes of all of the above (in the best and worst case)

Memory and Locality
- Basics of memory architecture
- Spatial and temporal locality

P vs NP
- Definitions of P, NP and NP Complete
- Understand what a reduction is

Design Decisions
- Given a scenario, what ADT, data structure implementation and/or algorithm is best optimized for your goals?
  - What is the purpose of the ADTs we’ve learned?
  - Given a scenario, determine whether one data structure would be a better fit then another (and explain why)
  - What is the optimal implementation of an ADT for a given situation?
- What is the runtime of a data structure's implementations of each ADT behavior?
- How can you leverage an algorithm to answer a given question?

NOT on the exam
- Java generics and Java interfaces.
- JUnit.
- Java syntax.
Algorithms you’re responsible for

For each of the listed algorithms make sure you understand:

In what situations it is useful
- What will this tell us about the data
- What state should the data be in to use it?

What the pros and cons of applying that algorithm are
- Runtime
- Memory usage

Heaps
- percolateUp
- percolateDown
- Floyd’s Build Heap

Sorting
- Insertion
- Selection
- Merge
- Quick
- Heap

Graphs
- Breadth First Search (BFS)
- Depth First Search (DFS)
- Dijkstra’s
- Topological Sort
- Prim’s MST
- Kruskal’s MST

Disjoint Sets
- Union by rank
- Path compression
Insertion Sort

public void insertionSort(collection) {
    for (entire list)
        if (currentItem is bigger than nextItem)
            int newIndex = findSpot(currentItem);
            shift(newIndex, currentItem);
}

public int findSpot(currentItem) {
    for (sorted list)
        if (spot found) return
}

public void shift(newIndex, currentItem) {
    for (i = currentItem > newIndex)
        item[i+1] = item[i]
        item[newIndex] = currentItem
}
**Selection Sort**

```java
public void selectionSort(collection) {
    for (entire list)
        int newIndex = findNextMin(currentItem);
        swap(newIndex, currentItem);
}
public int findNextMin(currentItem) {
    min = currentItem
    for (unsorted list)
        if (item < min)
            min = currentItem
    return min
}
public int swap(newIndex, currentItem) {
    temp = currentItem
    currentItem = newIndex
    newIndex = currentItem
}
```

- **Worst case runtime?** \(O(n^2)\)
- **Best case runtime?** \(O(n^2)\)
- **Average runtime?** \(O(n^2)\)
- **Stable?** Yes
- **In-place?** Yes

---

**Sorted Items** | **Current Item** | **Unsorted Items**
---|---|---
2 | 3 | 6 | 7 | 18 | 10 | 14 | 9 | 11 | 15
In Place Heap Sort

```java
public void inPlaceHeapSort(collection) {
    E[] heap = buildHeap(collection)
    for (n)
        output[n - i - 1] = removeMin(heap)
}
```

Complication: final array is reversed!
- Run reverse afterwards (O(n))
- Use a max heap
- Reverse compare function to emulate max heap

<table>
<thead>
<tr>
<th>Worst case runtime?</th>
<th>O(nlogn)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Best case runtime?</td>
<td>O(nlogn)</td>
</tr>
<tr>
<td>Average runtime?</td>
<td>O(nlogn)</td>
</tr>
<tr>
<td>Stable?</td>
<td>No</td>
</tr>
<tr>
<td>In-place?</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Worst case runtime? O(nlogn)
Best case runtime? O(nlogn)
Average runtime? O(nlogn)
Stable? No
In-place? Yes
mergeSort(input) {
  if (input.length == 1)
    return
  else
    smallerHalf = mergeSort(new [0, ..., mid])
    largerHalf = mergeSort(new [mid + 1, ...])
  return merge(smallerHalf, largerHalf)
}

Worst case runtime?
Best case runtime?  T(n) = \[\begin{cases} 1 & \text{if } n \leq 1 \\ 2T(n/2) + n & \text{otherwise} \end{cases}\]
Average runtime? = \(O(n \log n)\)
Stable?  Yes
In-place?  No
Quick Sort

```java
quickSort(input) {
    if (input.length == 1)
        return
    else
        pivot = getPivot(input)
        smallerHalf = quickSort(getSmaller(pivot, input))
        largerHalf = quickSort(getBigger(pivot, input))
        return smallerHalf + pivot + largerHalf
}
```

Worst case runtime?

\[ T(n) = \begin{cases} 
1 & \text{if } n \leq 1 \\
 n + T(n - 1) & \text{otherwise} 
\end{cases} \]

Best case runtime?

\[ T(n) = \begin{cases} 
1 & \text{if } n \leq 1 \\
 n + 2T(n/2) & \text{otherwise} 
\end{cases} \]

Average runtime?

\[ T(n) = \begin{cases} 
1 & \text{if } n \leq 1 \\
 n + 2T(n/2) & \text{otherwise} 
\end{cases} \]

Stable?

No

In-place?

No
Better Quick Sort

Compare three elements: leftmost, rightmost and center
Swap elements if necessary so that
Arr[0] = smallest
Arr[center] = median of three

Low
X < 6

High
X >= 6
Better Quick Sort

```java
quickSort(input) {
    if (input.length == 1)
        return
    else
        pivot = getPivot(input)
        smallerHalf = quickSort(getSmaller(pivot, input))
        largerHalf = quickSort(getBigger(pivot, input))
    return smallerHalf + pivot + largerHalf
}
```

**Worst case runtime?**

**Best case runtime?**

\[ T(n) = \begin{cases} 
1 & \text{if } n \leq 1 \\
 n + 2T(n/2) & \text{otherwise}
\end{cases} \]

**Average runtime?**

**Stable?** No

**In-place?** Yes
A graph is defined by a pair of sets $G = (V, E)$ where...

- $V$ is a set of vertices
  - A vertex or “node” is a data entity

  \[ V = \{ A, B, C, D, E, F, G, H \} \]

- $E$ is a set of edges
  - An edge is a connection between two vertices

  \[ E = \{ (A, B), (A, C), (A, D), (A, H), (C, B), (B, D), (D, E), (D, F), (F, G), (G, H) \} \]
Graph Vocabulary

Graph Direction
- **Undirected graph** – edges have no direction and are two-way
  - \( V = \{ A, B, C \} \)
  - \( E = \{ (A, B), (B, C) \} \) inferred \((B, A)\) and \((C, B)\)
- **Directed graphs** – edges have direction and are thus one-way
  - \( V = \{ A, B, C \} \)
  - \( E = \{ (A, B), (B, C), (C, B) \} \)

Degree of a Vertex
- **Degree** – the number of edges containing that vertex
  - \( A : 1, B : 1, C : 1 \)
- **In-degree** – the number of directed edges that point to a vertex
  - \( A : 0, B : 2, C : 1 \)
- **Out-degree** – the number of directed edges that start at a vertex
  - \( A : 1, B : 1, C : 1 \)
Graph Vocabulary

**Self loop** – an edge that starts and ends at the same vertex

**Parallel edges** – two edges with the same start and end vertices

**Simple graph** – a graph with no self-loops and no parallel edges
Assign each vertex a number from 0 to V – 1
Create a V x V array of Booleans
If (x,y) ∈ E then arr[x][y] = true

Runtime (in terms of V and E)
- get out - edges for a vertex O(v)
- get in – edges for a vertex O(v)
- decide if an edge exists O(1)
- insert an edge O(1)
- delete an edge O(1)
- delete a vertex
- add a vertex

How much space is used?
V²
Graph Vocabulary

**Dense Graph** – a graph with a lot of edges
\[ E \in \Theta(V^2) \]

**Sparse Graph** – a graph with “few” edges
\[ E \in \Theta(V) \]

An Adjacency Matrix seems a waste for a sparse graph...
Create a Dictionary of size V from type V to Collection of E
If (x,y) ∈ E then add y to the set associated with the key x

Runtime (in terms of V and E)
- get out - edges for a vertex O(1)
- get in - edges for a vertex O(V + E)
- decide if an edge exists O(1)
- insert an edge O(1)
- delete an edge O(1)
- delete a vertex
- add a vertex

How much space is used?
V + E
Walks and Paths

Walk – continuous set of edges leading from vertex to vertex

A list of vertices where if $i$ is some int where $0 < 1 < V_n$ every pair $(V_i, V_{i+1})$ in $E$ is true

Path – a walk that never visits the same vertex twice

[Diagrams showing connections between Winterfell, Castle Black, King's Landing, and Casterly Rock]
**Connected Graphs**

**Connected graph** – a graph where every vertex is connected to every other vertex via some path. It is not required for every vertex to have an edge to every other vertex.

There exists some way to get from each vertex to every other vertex.

**Connected Component** – a *subgraph* in which any two vertices are connected via some path, but is connected to no additional vertices in the *supergraph*.

- There exists some way to get from each vertex within the connected component to every other vertex in the connected component.
- A vertex with no edges is itself a connected component.
Breadth First Search

search(graph)
  toVisit.enqueue(first vertex)
  while(toVisit is not empty)
    current = toVisit.dequeue()
    for (V : current.neighbors())
      if (V is not in queue)
        toVisit.enqueue(v)
    visited.add(current)

Current node: I
Queue: B D E C F G H I
Visited: A B D E C F G H I
Depth First Search

dfs(graph)
    toVisit.push(first vertex)
    while(toVisit is not empty)
        current = toVisit.pop()
        for (V : current.neighbors())
            if (V is not in stack)
                toVisit.push(v)
        visited.add(current)

Current node: D

Stack: D B E I H G

Visited: A B E H G F I C D

How many times do you visit each node? 1 time each
How many times do you traverse each edge? Max 2 times each

- Putting them into toVisit
- Checking if they’re in toVisit

Runtime? O(V + 2E) = O(V + E) “graph linear”
Dijkstra’s Algorithm

Dijkstra(Graph G, Vertex source)
initialize distances to ∞
mark source as distance 0
mark all vertices unprocessed
while(there are unprocessed vertices){
  let u be the closest unprocessed vertex
  foreach(edge (u,v) leaving u){
    if(u.dist+weight(u,v) < v.dist){
      v.dist = u.dist+weight(u,v)
      v.predecessor = u
    }
  }
  mark u as processed
}

<table>
<thead>
<tr>
<th>Vertex</th>
<th>Distance</th>
<th>Predecessor</th>
<th>Processed</th>
</tr>
</thead>
<tbody>
<tr>
<td>s</td>
<td>0</td>
<td>--</td>
<td>Yes</td>
</tr>
<tr>
<td>w</td>
<td>1</td>
<td>s</td>
<td>Yes</td>
</tr>
<tr>
<td>x</td>
<td>2</td>
<td>w</td>
<td>Yes</td>
</tr>
<tr>
<td>u</td>
<td>20</td>
<td>s x</td>
<td>Yes</td>
</tr>
<tr>
<td>v</td>
<td>4</td>
<td>u</td>
<td>Yes</td>
</tr>
<tr>
<td>t</td>
<td>5</td>
<td>v</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Dijkstra’s Runtime

\[\text{Dijkstra(Graph } G, \text{ Vertex source)}\]
\[+V \text{ for (Vertex } v : G.\text{getVertices}()) \text{ { v.dist = INFINITY; }}\]

\[\text{G.\text{getVertex(source).dist = 0;}}\]

\[\text{initialize MPQ as a Min Priority Queue, add source}\]

\[\text{while(MPQ is not empty)}\{\]

\[\text{u = MPQ.removeMin(); } \text{+logV}\]

\[\text{for (Edge e : u.\text{getEdges(u)})}{\]

\[\text{oldDist = v.dist; newDist = u.dist+weight(u,v)}\]

\[\text{if(newDist < oldDist)}\{\]

\[\text{v.dist = newDist} \text{ +V}\]

\[\text{v.predecessor = u} \text{ +E}\]

\[\text{if(oldDist == INFINITY) } \text{MPQ.insert(v) } \text{+logV}\]

\[\text{else } \text{MPQ.updatePriority(v, newDist) } \text{+logV}\]

\[\text{}\}\]
\[\text{}\}\]

\[\text{Code Model} = C_1 + V + V(\log V + E(C_2 + 2\log V))\]
\[\text{= C_1 + V + VlogV + VEC_2 + VEC_3\log V}\]

\[\text{Tight O Bound} = O(VE\log V)\]

How often do we actually update the MPQ thanks to this if statement?
\[E \text{ times!}\]
\[\text{Tight O Bound} = O(V\log V + E\log V)\]
How Do We Find a Topological Ordering?

TopologicalSort(Graph G, Vertex source)

  count how many incoming edges each vertex has
  Collection toProcess = new Collection()
  foreach(Vertex v in G){
    if(v.edgesRemaining == 0)
      toProcess.insert(v)
  }
  topOrder = new List()
  while(toProcess is not empty){
    u = toProcess.remove()
    topOrder.insert(u)
    foreach(edge (u,v) leaving u){
      v.edgesRemaining--
      if(v.edgesRemaining == 0)
        toProcess.insert(v)
    }
  }

Math 126

CSE 142

CSE 143

CSE 373

CSE 374

CSE 417
How Do We Find a Topological Ordering?

TopologicalSort(Graph G, Vertex source)

1. count how many incoming edges each vertex has
2. Collection toProcess = new Collection()
3. foreach (Vertex v in G){
   if(v.edgesRemaining == 0)
      toProcess.insert(v)
}
4. topOrder = new List()
5. while (toProcess is not empty){
   u = toProcess.remove()
   topOrder.insert(u)
   foreach (edge (u,v) leaving u){
      v.edgesRemaining--
      if(v.edgesRemaining == 0)
         toProcess.insert(v)
   }
}

BFS
Graph linear
+ V + E

Pick something with O(1) insert / removal

+V

Runs as most once per edge
+E

O(V + E)
Practice

What is a possible ordering of the graph after a topological sort?

All possible orderings:
- e, d, b, c, f, a
- e, b, d, c, f, a
- e, b, c, d, f, a
- e, b, c, f, a

CSE 373 SP 18 - KASEY CHAMPION
Try it Out

PrimMST(Graph G)
initialize distances to $\infty$
mark source as distance 0
mark all vertices unprocessed
foreach(edge (source, v) ) {
    v.dist = weight(source,v)
    v.bestEdge = (source,v)
}
while(there are unprocessed vertices){
    let u be the closest unprocessed vertex
    add u.bestEdge to spanning tree
    foreach(edge (u,v) leaving u){
        if(weight(u,v) < v.dist && v unprocessed ){
            v.dist = weight(u,v)
            v.bestEdge = (u,v)
        }
    }
    mark u as processed
}
Try It Out

KruskalMST(Graph G)
initialize each vertex to be an independent component
sort the edges by weight
foreach(edge (u, v) in sorted order){
    if(u and v are in different components){
        add (u,v) to the MST
        Update u and v to be in the same component
    }
}

<table>
<thead>
<tr>
<th>Edge</th>
<th>Include?</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>(A,C)</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>(C,E)</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>(A,B)</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>(A,D)</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>(C,D)</td>
<td>No</td>
<td>Cycle A,C,D,A</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Edge (cont.)</th>
<th>Inc?</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>(B,F)</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>(D,E)</td>
<td>No</td>
<td>Cycle A,C,E,D,A</td>
</tr>
<tr>
<td>(D,F)</td>
<td>No</td>
<td>Cycle A,D,F,B,A</td>
</tr>
<tr>
<td>(E,F)</td>
<td>No</td>
<td>Cycle A,C,E,F,D,A</td>
</tr>
<tr>
<td>(C,F)</td>
<td>No</td>
<td>Cycle C,A,B,F,C</td>
</tr>
</tbody>
</table>
Kruskal’s Algorithm Implementation

KruskalMST(Graph G)
initialize each vertex to be an independent component
sort the edges by weight
foreach(edge (u, v) in sorted order){
  if(u and v are in different components){
    add (u,v) to the MST
    update u and v to be in the same component
  }
}

KruskalMST(Graph G)
foreach (V : vertices) {
  makeMST(v); +?
}
sort edges in ascending order by weight
foreach(edge (u, v)){
  if(findMST(v) is not in findMST(u)){
    union(u, v) +?
  }
}
Strongly Connected Components

**Strongly Connected Component**

A subgraph C such that every pair of vertices in C is connected via some path in both directions, and there is no other vertex which is connected to every vertex of C in both directions.

Note: the direction of the edges matters!
Why Find SCCs?

Graphs are useful because they encode relationships between arbitrary objects.

We’ve found the strongly connected components of G.

Let’s build a new graph out of them! Call it H

- Have a vertex for each of the strongly connected components
- Add an edge from component 1 to component 2 if there is an edge from a vertex inside 1 to one inside 2.
Implement makeSet(x)

makeSet(0)
makeSet(1)
makeSet(2)
makeSet(3)
makeSet(4)
makeSet(5)

Worst case runtime?

O(1)
Implement `findSet(x)`

```plaintext
findSet(0)  
findSet(3)  
findSet(5)  
```

Worst case runtime?

$O(n)$

Worst case runtime of union?

$O(n)$
Implement union(x, y)

union(3, 5)
union(2, 1)
union(2, 5)
Improving union

Problem: Trees can be unbalanced

Solution: Union-by-rank!
- let rank(x) be a number representing the upper bound of the height of x so rank(x) >= height(x)
- Keep track of rank of all trees
- When unioning make the tree with larger rank the root
- If it’s a tie, pick one randomly and increase rank by one

```
rank = 0
4
```
```
rank = 2
2
  1
  3
  5
```
```
rank = 0
0
```
```
rank = 1
4
```
Improving findSet()

Problem: Every time we call findSet() you must traverse all the levels of the tree to find representative

Solution: Path Compression
- Collapse tree into fewer levels by updating parent pointer of each node you visit
- Whenever you call findSet() update each node you touch’s parent pointer to point directly to overallRoot

```
findSet(5)
findSet(4)
```

Does this improve the worst case runtimes?

findSet is more likely to be \( O(1) \) than \( O(\log(n)) \)
Array Implementation

rank = 0

rank = 3

Store (rank * -1) - 1

Each “node” now only takes 4 bytes of memory instead of 32
## Optimized Disjoint Set Runtime

### `makeSet(x)`

<table>
<thead>
<tr>
<th></th>
<th>Without Optimizations</th>
<th>With Optimizations</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><strong>O(1)</strong></td>
<td><strong>O(1)</strong></td>
</tr>
</tbody>
</table>

### `findSet(x)`

<table>
<thead>
<tr>
<th></th>
<th>Without Optimizations</th>
<th>With Optimizations</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><strong>O(n)</strong></td>
<td><strong>Best case: O(1) Worst case: O(logn)</strong></td>
</tr>
</tbody>
</table>

### `union(x, y)`

<table>
<thead>
<tr>
<th></th>
<th>Without Optimizations</th>
<th>With Optimizations</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><strong>O(n)</strong></td>
<td><strong>Best case: O(1) Worst case: O(logn)</strong></td>
</tr>
</tbody>
</table>
Scenario #1

You are going to Disneyland for spring break! You’ve never been, so you want to make sure you hit ALL the rides.

Is there a graph algorithm that would help?

**BFS or DFS**

How would you draw the graph?
- What are the vertices?
  - Rides
- What are the edges?
  - Walkways

BFS = 0 1 2 3 5 6 7 8 9 4 10
DFS = 0 3 5 6 7 8 9 10 1 4 2
Scenario #1 continued

Now that you have your basic graph of Disneyland, what might the following graph items represent in this context?

**Weighted edges**
- Walkway distances
- Walking times
- Foot traffic

**Directed edges**
- Entrances and exits
- Crowd control for fireworks
- Parade routes

**Self Loops**
- Looping a ride

**Parallel Edges**
- Foot traffic at different times of day
- Walkways and train routes
Scenario #2

You are a Disneyland employee and you need to rope off as many miles of walkways as you can for the fireworks while leaving guests access to all the rides.

Is there a graph algorithm that would help?

MST

How would you draw the graph?
- What are the vertices?
  Rides
- What are the edges?
  Walkways with distances
Scenario #3

You arrive at Disneyland and you want to visit all the rides, but do the least amount of walking possible. If you start at the Flag Pole, plan the shortest walk to each of the attractions.

Is there a graph algorithm that would help?

Dijkstra’s

How would you draw the graph?
- What are the vertices?
  Rides
- What are the edges?
  Walkways with distances
Scenario #2b

Now that you know the shortest distance to each attraction, can you make a plan to visit all the attractions with the least amount of total walking?

Nope! This is the travelling salesman problem which is much more complicated than Dijkstra’s. (NP Hard, more on this later)
Scenario #3

You have great taste so you are riding Space Mountain. Your friend makes poor choices so they are riding Splash Mountain. You decide to meet at the castle, how long before you can meet up?

Is there a graph algorithm that would help? Dijkstra’s
What information do our edges need to store?
Walking times
How do we apply the algorithm?
- Run Dijkstra’s from Splash Mountain.
- Run Dijkstra’s from Space Mountain.
- Take the larger of the two times.
Types of Problems

**Decision Problem** – any arbitrary yes-or-no question on an infinite set of inputs. Resolution to problem can be represented by a Boolean value.
- IS-PRIME: is X a prime number? (where X is some input)
- IS-SORTED: is this list of numbers sorted?
- EQUAL: is X equal to Y? (for however X and Y define equality)

**Solvable** – a decision problem is solvable if there exists some algorithm that given any input or instance can correctly produce either a “yes” or “no” answer.
- Not all problems are solvable!
  - Example: Halting problem

**Efficient algorithm** – an algorithm is efficient if the worst case bound is a polynomial. The growth rate of this is such that you can actually run it on a computer in practice.
- Definitely efficient: $O(1)$, $O(n)$, $O(n \log n)$, $O(n^2)$
- Technically efficient: $O(n^{1000000})$, $O(100000000000000n^2)$

Everything we’ve talked about in class so far has been **solvable** and **efficient**...
Weighted Graphs: A Reduction

Transform Input

Unweighted Shortest Paths

Transform Output
The set of all decision problems that have an algorithm that runs in time $O(n^k)$ for some constant $k$.

The decision version of all problems we’ve solved in this class are in $P$.

$P$ is an example of a “complexity class”
A set of problems that can be solved under some limitations (e.g. with some amount of memory or in some amount of time).
I’ll know it when I see it.

More formally,

**NP (stands for “nondeterministic polynomial”)**

The set of all decision problems such that if the answer is YES, there is a proof of that which can be verified in polynomial time.

It’s a common misconception that NP stands for “not polynomial”
Please never ever ever ever say that.

Please.

Every time you do a theoretical computer scientist sheds a single tear.

(That theoretical computer scientist is me)