Lecture 30: Final Review

CSE 373: Data Structures and Algorithms
HW 7 due tonight

Fill out surveys for 5 points of EC!
- Lecture survey
- Section survey
- Our survey

Nominate your Tas for Bob Bandes Award
After 373...

Get a job
- Prepare for interviews
- Attend CSE career fair in October

Get more coding practice
- Husky Code Student Project (email Kasey)
- Hackathons
- Research
- Open source projects

Become a TA
- Email Pim Lustig, pl@cs.washington.edu

Take more classes
- CSE 417 – more algorithms
- CSE 154 – web programming
- CSE 163 – python libraries for data science
- CSE 374 – C programming and unix
On the exam

Midterm Topics
- ADTs + data structures
- Asymptotic Analysis
  - Code Modeling (including recurrences)
  - Complexity Classes
  - Big O, Big Omega and Big Theta
- BST & AVL trees
- Hashing

Heaps
- Internal state of tree
- Array implementation

Graphs
- Graph definitions
- Graph implementations
- Graph algorithms
  - Traversals: BFS and DFS
  - Shortest-path: Dijkstra's algorithm
  - Topological sort
  - MST algorithms: Prim and Kruskal
  - Disjoint set data structure

Coding Projects
- Implementation of each data structure
- Best / Average / Worst case runtime of each data structure
- Testing strategies, debugging strategies

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

Memory and Locality
- How to leverage caching

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 unique or specialized about your chosen tool?
  - Given a scenario, how does your selection’s unique features contribute to a solution?
  - What is the runtime and memory usage of your selection?
- Given a scenario, what changes might you make to a design to better serve your goals?

NOT on the exam
- Finding close form of recurrences with tree method
- writing Java generics and Java interfaces
- writing JUnit
- writing Java syntax

CSE 373 19 SP - KASEY CHAMPION
ADTs and Data Structures you’re responsible for

For each of the listed algorithms make sure you understand:

In what situations it is useful
- What behavior does the ADT actually allow?
- What is unique about this ADT?

What are the different data structure implementation options for this ADT and how does that impact
- Ease of implementation
- Runtime
- Memory usage

ADTs
List
- Maintains order of elements
- Flexibility of item manipulation
- Useful in many situations
Stack
- First in last out ordering
Queue
- First in first out ordering
Set
- Unordered collection of unique elements
- Often used to test membership (contains)
Dictionary
- Collection of key value pairs
- Does not maintain order of elements
Tree
- Ordered hierarchy of elements
- Often used for traversals
Iterator
- Dictates specific order in which to move through elements in a collection
- Allows for a single pass through collection in one direction
Priority Queue
- Highest priority first out
Disjoint Set
- Set of sets
- Provides near constant time operations

Data Structures
Array
- Set capacity upon creation
- Each element has associated index for quick access
- Leverages caching
Linked Nodes
- Set of objects strung together by references
- Very flexible
- Requires traversals for access
Binary Search Tree
- Specific ordering of elements: left child < parent < right child
- Allows for quicker traversals
- Simple implementation
AVL Trees
- Same ordering as BST but self-balances to maintain optimal height for runtime
- Complex implementation
Hash table
- Optimized for close to constant time look up and insertion
- Optimality dictated by hash function and collision strategy
Heap
- Specific ordering: parent < children
- Fills from left to right to maintain optimal height
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?

The pros and cons of the algorithm in terms of
- Runtime – best, average and worst
- 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
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
}
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
}
In Place Heap Sort

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

Worst case runtime? O(n log n)
Best case runtime? O(n log n)
Average runtime? O(n log n)
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)
}
Quick Sort

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? $T(n) = \begin{cases} 
1 & \text{if } n \leq 1 \\
1 + T(n - 1) & \text{otherwise}
\end{cases}$ $= O(n^2)$

Best case runtime? $T(n) = \begin{cases} 
1 & \text{if } n \leq 1 \\
1 + 2T(n/2) & \text{otherwise}
\end{cases}$

Average runtime? $= O(n \log n)$

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
Arr[X >= 6] = largest
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?  $T(n) = \begin{cases} 
1 & \text{if } n \leq 1 \\
 n + 2T(n/2) & \text{otherwise} 
\end{cases} = O(n\log n)$

Best case runtime?  $T(n) = \begin{cases} 
1 & \text{if } n \leq 1 \\
 n + 2T(n/2) & \text{otherwise} 
\end{cases} = O(n\log n)$

Average runtime?

Stable? No

In-place? Yes
Graph: Formal Definition

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
  - for undirected graph: $A : 1$, $B : 2$, $C : 1$
- **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

![Diagram of a self loop with Petyr as the vertex]

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

![Diagram of parallel edges with Dany and Drogo as the vertices]

**Simple graph** – a graph with no self-loops and no parallel edges

![Diagram of a simple graph with Margaery, Renly, Loras, and Brienne as the vertices]
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 \((x, y) \in E\) 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 $0 < i < Vn$ every pair $(V_i, V_{i+1})$ in $E$ is true

Path – a walk that never visits the same vertex twice
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

\[ \text{Dijkstra}(\text{Graph } G, \text{ Vertex source}) \]

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

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

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

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

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

\[ +\log V \]

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

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

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

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

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

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

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

\[ \} \]

\[ \} \]

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

\[ = C_1 + V + V\log V + VEC_2 + VEC_3\log V \]

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

This actually doesn’t run all E times – for every iteration of the outer loop. It actually will run E times in total; if every vertex is only removed from the priority queue (processed) once, then we will examine each edge once. So each line inside this foreach gets multiplied by a single E instead of E * V.

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

BFS
Graph linear
+ V + E

Pick something with O(1) insert / removal

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
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
  }

<table>
<thead>
<tr>
<th>Vertex</th>
<th>Distance</th>
<th>Best Edge</th>
<th>Processed</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>-</td>
<td>X</td>
<td>✓</td>
</tr>
<tr>
<td>B</td>
<td>2</td>
<td>(A, B)</td>
<td>✓</td>
</tr>
<tr>
<td>C</td>
<td>4</td>
<td>(A, C)</td>
<td>✓</td>
</tr>
<tr>
<td>D</td>
<td>7-2</td>
<td>(A,-D)(C, D)</td>
<td>✓</td>
</tr>
<tr>
<td>E</td>
<td>6-5</td>
<td>(B,-E)(C, E)</td>
<td>✓</td>
</tr>
<tr>
<td>F</td>
<td>3</td>
<td>(B, F)</td>
<td>✓</td>
</tr>
<tr>
<td>G</td>
<td>50</td>
<td>(B, G)</td>
<td>✓</td>
</tr>
</tbody>
</table>
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); +V
    }
    sort edges in ascending order by weight
    foreach(edge (u, v)){
        if(findMST(v) is not in findMST(u)){
            union(u, v) +E
        }
    }

How many times will we call union?
V – 1
-> +Vunion + EfindMST
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.

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 $\text{makeSet}(x)$

$\text{makeSet}(0)$
$\text{makeSet}(1)$
$\text{makeSet}(2)$
$\text{makeSet}(3)$
$\text{makeSet}(4)$
$\text{makeSet}(5)$

Worst case runtime?
$O(1)$
Implement `findSet(x)`

- `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  rank = 2

rank = 0  rank = 1
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

rank = 3

Store (rank * -1) - 1

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

\textbf{makeSet(x)}

Without Optimizations \quad O(1)

With Optimizations \quad O(1)

\textbf{findSet(x)}

Without Optimizations \quad O(n)

With Optimizations \quad \text{Best case: O(1) Worst case: O(logn)}

\textbf{union(x, y)}

Without Optimizations \quad O(n)

With Optimizations \quad \text{Best case: O(1) Worst case: O(logn)}
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
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: $O(n1000000), O(10000000000000n^2)$

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
P

P (stands for “Polynomial”)
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)
# Homework 2 Data Structures

## DoubleLinkedList

<table>
<thead>
<tr>
<th>Function</th>
<th>Best Case Runtime</th>
<th>Average Runtime</th>
<th>Worst Case Runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>add(T item)</td>
<td>O(1)</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
<tr>
<td>T remove()</td>
<td>O(1)</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
<tr>
<td>T get(int index)</td>
<td>O(1)</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
<tr>
<td>set(int index, T item)</td>
<td>O(1)</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
<tr>
<td>insert(int index, T item)</td>
<td>O(1)</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
<tr>
<td>T delete(int index)</td>
<td>O(1)</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
<tr>
<td>int indexOf(T item)</td>
<td>O(1)</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
</tbody>
</table>

## ArrayDictionary

<table>
<thead>
<tr>
<th>Function</th>
<th>Best Case Runtime</th>
<th>Average Runtime</th>
<th>Worst Case Runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>V get(K key)</td>
<td>O(1)</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
<tr>
<td>put(K key, V value)</td>
<td>O(1)</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
<tr>
<td>V remove(K key)</td>
<td>O(1)</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
<tr>
<td>boolean containsKey(K Key)</td>
<td>O(1)</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
</tbody>
</table>
### ChainedHashDictionary

<table>
<thead>
<tr>
<th>Function</th>
<th>Best Case Runtime</th>
<th>Average Runtime</th>
<th>Worst Case Runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>V get(K key)</td>
<td>O(1)</td>
<td>O((\lambda))</td>
<td>O(n)</td>
</tr>
<tr>
<td>put(K key, V value)</td>
<td>O(1)</td>
<td>O((\lambda))</td>
<td>O(n)</td>
</tr>
<tr>
<td>V remove(K key)</td>
<td>O(1)</td>
<td>O((\lambda))</td>
<td>O(n)</td>
</tr>
<tr>
<td>boolean containsKey(K Key)</td>
<td>O(1)</td>
<td>O((\lambda))</td>
<td>O(n)</td>
</tr>
</tbody>
</table>

### ChainedHashSet

<table>
<thead>
<tr>
<th>Function</th>
<th>Best Case Runtime</th>
<th>Average Runtime</th>
<th>Worst Case Runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>add(T item)</td>
<td>O(1)</td>
<td>O((\lambda))</td>
<td>O(n)</td>
</tr>
<tr>
<td>remove(T item)</td>
<td>O(1)</td>
<td>O((\lambda))</td>
<td>O(n)</td>
</tr>
<tr>
<td>boolean contains(T item)</td>
<td>O(1)</td>
<td>O((\lambda))</td>
<td>O(n)</td>
</tr>
</tbody>
</table>
### Homework 5 Data Structures

#### ArrayHeap

<table>
<thead>
<tr>
<th>Function</th>
<th>Best Case Runtime</th>
<th>Average Runtime</th>
<th>Worst Case Runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>T removeMin()</td>
<td>O(1)</td>
<td>O(log n)</td>
<td>O(log n)</td>
</tr>
<tr>
<td>T peekMin()</td>
<td>O(1)</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
<tr>
<td>add(T item)</td>
<td>O(1)</td>
<td>O(log n)</td>
<td>O(n)</td>
</tr>
<tr>
<td>boolean contains(T item)</td>
<td>O(1)</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
<tr>
<td>remove(T item)</td>
<td>O(1)</td>
<td>O(log n)</td>
<td>O(log n)</td>
</tr>
<tr>
<td>replace(T oldItem, T newItem)</td>
<td>O(1)</td>
<td>O(log n)</td>
<td>O(log n)</td>
</tr>
</tbody>
</table>
ArrayDisjointSet

<table>
<thead>
<tr>
<th>Function</th>
<th>Best Case Runtime</th>
<th>Average Runtime</th>
<th>Worst Case Runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>makeSet(T item)</td>
<td>O(1)</td>
<td>O(1)</td>
<td>O(n)</td>
</tr>
<tr>
<td>int findSet(T item)</td>
<td>O(1)</td>
<td>O(1)</td>
<td>O(logn)</td>
</tr>
<tr>
<td>union(T item1, T item2)</td>
<td>O(1)</td>
<td>O(1)</td>
<td>O(logn)</td>
</tr>
</tbody>
</table>

Graph

<table>
<thead>
<tr>
<th>Function</th>
<th>Best Case Runtime</th>
<th>Average Runtime</th>
<th>Worst Case Runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>int numVertices()</td>
<td>O(1)</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
<tr>
<td>int numEdges()</td>
<td>O(1)</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
<tr>
<td>ISet&lt;E&gt; findMinimumSpanningTree()</td>
<td>O(ElogE)</td>
<td>O(ElogE)</td>
<td>O(ElogE)</td>
</tr>
<tr>
<td>IList&lt;E&gt; findShortestPathBetween(V start, V end)</td>
<td>O(1)</td>
<td>you don’t need to know this</td>
<td>O(VlogV + ElogV)</td>
</tr>
</tbody>
</table>
What to test?

**Expected behavior**
- The main use case scenario
- Does your code do what it should given friendly conditions?

**Forbidden Input**
- What are all the ways the user can mess up?

**Empty/Null**
- Protect yourself!
- How do things get started?
- 0, -1, null, empty collections

**Boundary/Edge Cases**
- First items
- Last item
- Full collections

**Scale**
- Is there a difference between 10, 100, 1000, 10000 items?