CSE 101
Final Review Session Part 1

March 16, 2018
Final review session consists of two parts.

Part 1 covers:
- Dynamic Programming
- Greedy algorithms

Part 2 covers:
- Network flows
- NP-completeness
- Linear Programming
Outline of Exam

- 6 questions, 60 points total
- 30 points are “mechanical”: flow, LP, short answer/TTK, ...
- About 20 points are algorithm “design” questions: Greed, DP
- About 10 points on NP-Completeness reduction
- Google Doc with “advice” exists: linked from Piazza @872
True/False questions.

- The complexity of the DFS is $O(|V| + |E|)$. 

False
The complexity of the DFS is $O(|V| + |E|)$. True

Dijkstra's algorithm should be used on graphs with negative weights. False

Bellman-Ford can find negative cycles in graphs. True

There exists a searching algorithm based on comparisons with time complexity of $O(\log \log(n))$, which finds some number $x$ in an array of $n$ elements. False

Prim's algorithm can have $O(|E|\log|V|)$ time complexity if it is implemented with binary heaps. True

The lower-bound for all sorting algorithms based on comparisons is $O(n)$, where $n$ is the number of elements that must be sorted. False
True/False questions.

- The complexity of the DFS is $O(|V| + |E|)$. **True**
- Dijkstra’s algorithm should be used on graphs with negative weights. **False**
True/False questions.

- The complexity of the DFS is $O(|V| + |E|)$.  **True**
- Dijkstra’s algorithm should be used on graphs with negative weights.  **False**
True/False questions.

- The complexity of the DFS is $O(|V| + |E|)$. True
- Dijkstra’s algorithm should be used on graphs with negative weights. False
- Bellman-Ford can find negative cycles in graphs.
True/False questions.

- The complexity of the DFS is $O(|V| + |E|)$. **True**
- Dijkstra’s algorithm should be used on graphs with negative weights. **False**
- Bellman-Ford can find negative cycles in graphs. **True**
True/False questions.

- The complexity of the DFS is $\mathcal{O}(|V| + |E|)$. **True**
- Dijkstra’s algorithm should be used on graphs with negative weights. **False**
- Bellman-Ford can find negative cycles in graphs. **True**
- There exists a searching algorithm based on comparisons with time complexity of $\mathcal{O}(\log \log(n))$, which finds some number $x$ in an array of $n$ elements.
True/False questions.

- The complexity of the DFS is $O(|V| + |E|)$. **True**
- Dijkstra’s algorithm should be used on graphs with negative weights. **False**
- Bellman-Ford can find negative cycles in graphs. **True**
- There exists a searching algorithm based on comparisons with time complexity of $O(\log \log(n))$, which finds some number $x$ in an array of $n$ elements. **False**
True/False questions.

- The complexity of the DFS is $O(|V| + |E|)$. **True**
- Dijkstra’s algorithm should be used on graphs with negative weights. **False**
- Bellman-Ford can find negative cycles in graphs. **True**
- There exists a searching algorithm based on comparisons with time complexity of $O(\log \log(n))$, which finds some number $x$ in an array of $n$ elements. **False**
- Prim’s algorithm can have $O(|E|\log|V|)$ time complexity if it is implemented with binary heaps.
True/False questions.

- The complexity of the DFS is $O(|V| + |E|)$. **True**
- Dijkstra’s algorithm should be used on graphs with negative weights. **False**
- Bellman-Ford can find negative cycles in graphs. **True**
- There exists a searching algorithm based on comparisons with time complexity of $O(\log \log(n))$, which finds some number $x$ in an array of $n$ elements. **False**
- Prim’s algorithm can have $O(|E|\log|V|)$ time complexity if it is implemented with binary heaps. **True**
True/False questions.

- The complexity of the DFS is $O(|V| + |E|)$. **True**
- Dijkstra’s algorithm should be used on graphs with negative weights. **False**
- Bellman-Ford can find negative cycles in graphs. **True**
- There exists a searching algorithm based on comparisons with time complexity of $O(\log \log(n))$, which finds some number $x$ in an array of $n$ elements. **False**
- Prim’s algorithm can have $O(|E|\log|V|)$ time complexity if it is implemented with binary heaps. **True**
- The lower-bound for all sorting algorithms based on comparisons is $O(n)$, where $n$ is the number of elements that must be sorted.
True/False questions.

- The complexity of the DFS is $O(|V| + |E|)$. **True**
- Dijkstra’s algorithm should be used on graphs with negative weights. **False**
- Bellman-Ford can find negative cycles in graphs. **True**
- There exists a searching algorithm based on comparisons with time complexity of $O(\log \log(n))$, which finds some number $x$ in an array of $n$ elements. **False**
- Prim’s algorithm can have $O(|E| \log |V|)$ time complexity if it is implemented with binary heaps. **True**
- The lower-bound for all sorting algorithms based on comparisons is $O(n)$, where $n$ is the number of elements that must be sorted. **False**
True/False questions.

- Connected components can’t share nodes (can’t intersect).
True/False questions.

- Connected components can’t share nodes (can’t intersect).
  True

- We use memoization to avoid repeated computations.
  True

- Knapsack problem with integer weights can be solved with $O(W)$ space complexity, where $W$ is the capacity of the knapsack.
  True

- Edge with maximum weight in a cut must belong to MST.
  False

- There can be cycles in meta-graphs (that are built from strongly connected components).
  False

- If the problem is $NP$, then we don’t know how to solve it in a polynomial time.
  False
True/False questions.

- Connected components can’t share nodes (can’t intersect).  **True**
- We use memoization to avoid repeated computations. **True**

Knapsack problem with integer weights can be solved with $O(W)$ space complexity, where $W$ is the capacity of the knapsack. **True**

Edge with maximum weight in a cut must belong to MST.  **False**

There can be cycles in meta-graphs (that are built from strongly connected components).  **False**

If the problem is $NP$, then we don’t know how to solve it in a polynomial time.  **False**
True/False questions.

- Connected components can’t share nodes (can’t intersect). **True**
- We use memoization to avoid repeated computations. **True**
True/False questions.

- Connected components can’t share nodes (can’t intersect).  **True**
- We use memoization to avoid repeated computations.  **True**
- Knapsack problem with integer weights can be solved with $O(W)$ space complexity, where $W$ is the capacity of the knapsack.
True/False questions.

- Connected components can’t share nodes (can’t intersect).  
  True
- We use memoization to avoid repeated computations.  True
- Knapsack problem with integer weights can be solved with $O(W)$ space complexity, where $W$ is the capacity of the knapsack.  True
True/False questions.

- Connected components can’t share nodes (can’t intersect).  **True**
- We use memoization to avoid repeated computations.  **True**
- Knapsack problem with integer weights can be solved with $O(W)$ space complexity, where $W$ is the capacity of the knapsack.  **True**
- Edge with maximum weight in a cut must belong to MST.  **False**
- There can be cycles in meta-graphs (that are built from strongly connected components).  **False**
- If the problem is $NP$, then we don’t know how to solve it in a polynomial time.  **False**
True/False questions.

- Connected components can’t share nodes (can’t intersect). **True**
- We use memoization to avoid repeated computations. **True**
- Knapsack problem with integer weights can be solved with $O(W)$ space complexity, where $W$ is the capacity of the knapsack. **True**
- Edge with maximum weight in a cut must belong to MST. **False**
True/False questions.

- Connected components can’t share nodes (can’t intersect). **True**
- We use memoization to avoid repeated computations. **True**
- Knapsack problem with integer weights can be solved with $O(W)$ space complexity, where $W$ is the capacity of the knapsack. **True**
- Edge with maximum weight in a cut must belong to MST. **False**
- There can be cycles in meta-graphs (that are built from strongly connected components).
True/False questions.

- Connected components can’t share nodes (can’t intersect). True
- We use memoization to avoid repeated computations. True
- Knapsack problem with integer weights can be solved with $O(W)$ space complexity, where $W$ is the capacity of the knapsack. True
- Edge with maximum weight in a cut must belong to MST. False
- There can be cycles in meta-graphs (that are built from strongly connected components) False
True/False questions.

- Connected components can’t share nodes (can’t intersect). **True**
- We use memoization to avoid repeated computations. **True**
- Knapsack problem with integer weights can be solved with $O(W)$ space complexity, where $W$ is the capacity of the knapsack. **True**
- Edge with maximum weight in a cut must belong to MST. **False**
- There can be cycles in meta-graphs (that are built from strongly connected components) **False**
- If the problem is $NP$, then we don’t know how to solve it in a polynomial time.
True/False questions.

- Connected components can’t share nodes (can’t intersect).  **True**
- We use memoization to avoid repeated computations.  **True**
- Knapsack problem with integer weights can be solved with $O(W)$ space complexity, where $W$ is the capacity of the knapsack.  **True**
- Edge with maximum weight in a cut must belong to MST.  **False**
- There can be cycles in meta-graphs (that are built from strongly connected components)  **False**
- If the problem is $NP$, then we don’t know how to solve it in a polynomial time.  **False**
Dynamic Programming

Dynamic programming (DP) is a powerful mathematical technique that breaks a problem into subproblems of smaller dimensions, solves these subproblems only once, memorizes the answer to each of them and then uses obtained results to get answers to subproblems of larger dimensions. Eventually, this leads to the solution of the initial problem. In order to get an optimal answer, each of the subproblems must be solved optimally.
Dynamic Programming

Steps in solving problems with DP (theoretical approach):

- Define a subproblem
- Find a recurrent relation
- Compute base cases

Steps in solving problems with DP (practical/programming approach):

- Compute base cases
- Solve subproblems optimally
- Solve an initial problem, using results for subproblems
Dynamic Programming

Steps in solving problems with DP (theoretical approach):
- Define a subproblem
- Find a recurrent relation
- Compute base cases

Steps in solving problems with DP (practical/programming approach):
- Compute base cases
- Solve subproblems optimally
- Solve an initial problem, using results for subproblems
Dynamic Programming

Steps in solving problems with DP (theoretical approach):
▶ Define a subproblem
▶ Find a recurrent relation
▶ Compute base cases

Steps in solving problems with DP (practical/programming approach):
▶ Compute base cases
▶ Solve subproblems optimally
▶ Solve an initial problem, using results for subproblems

A problem should depend on a set of parameters. These parameters define the dimension of the problem. Depending on these parameters, we can find subproblems to any problem.
Dynamic Programming. Example 1 (HW4.1)

**Statement.** There is a robot on the line at coordinate 1. In one move, the robot can get from coordinate \(i\) to any one of the coordinates \(i + 1, i + 2, \ldots, i + k\). Give an algorithm to determine how many distinct ways the robot can get from coordinate 1 to coordinate \(n\), where \(n\) and \(k\) are positive integer numbers and \(k \leq n\). Your algorithm must have \(O(n)\) time complexity and \(O(k)\) space complexity.
Dynamic Programming. Example 1 (HW4.1)

**Solution.** Let's define $f[i]$ as the number of different ways to get to $i$ from 1 by making steps of lengths 1, 2, ..., $k$.

Let's fix the last step $x$ in a path from 1 to $i$. Then, we have:

$$f[i] = \begin{cases} 
0 & \text{if } i \leq 1 \\
1 & \text{if } i = 2 \\
\sum_{x=1}^{\min(k,i-1)} f[i-x] & \text{if } i > 2
\end{cases}$$

The answer for this problem will be $f[n]$. This solution has $O(nk)$ time complexity and $O(n)$ space complexity. It can be optimized both in terms of time and space. Please see the solutions for HW4.
**Statement.** We have \( n \) items and a knapsack of integer capacity \( W > 0 \). Each item has a positive integer weight \( w_i \) and cost \( c_i \). Put items in the knapsack, such that their total weight doesn’t exceed \( W \) and their total cost is maximized, if:

1. We can take each item once
2. We can take each item as many times as we want
Dynamic Programming. Example 2 (Knapsack)

**Solution.** Let’s define $f[i][k]$ as the optimal cost of items in the knapsack if its capacity is $k$ and we can use only some subsets of the first $i$ items.

Let’s consider two cases: when we put item $i$ into the knapsack and when we don’t. Then, we have:

(if we can use each item only once)

$$f[i][k] = \begin{cases} 
0 & \text{if } i = 0 \text{ or } k = 0 \\
 f[i - 1][k] & \text{else if } k - w_i < 0 \\
 \max(f[i - 1][k], f[i - 1][k - w_i] + c_i) & \text{otherwise}
\end{cases}$$

The answer for the problem is $f[n][W]$. The time and space complexity is $O(nW)$.
Solution. Let’s define $f[i][k]$ as the optimal cost of items in the knapsack if its capacity is $k$ and we can use only some subsets of the first $i$ items.

Let’s consider two cases: when we put item $i$ into the knapsack and when we don’t. Then, we have:

\[
f[i][k] = \begin{cases}
0 & \text{if } i = 0 \text{ or } k = 0 \\
 f[i-1][k] & \text{else if } k - w_i < 0 \\
 \max(f[i-1][k], f[i][k-w_i] + c_i) & \text{otherwise}
\end{cases}
\]

The answer for the problem is $f[n][W]$. The time and space complexity is $O(nW)$.
Statement. You are given a sequence $S = (s_1, s_2, ..., s_n)$. Find the length of the largest sub-sequence which is a palindrome. (a palindrome is a sequence which equals its reversed self).
Solution. Let’s define $f[i][k]$ as the length of the longest sub-sequence in a sequence $(s_i, s_{i+1}, ..., s_k)$ which is a palindrome. Now, let’s consider two cases: when we include $s[i]$ and $s[k]$ (if it is possible) to the optimal sub-sequence and when we don’t. Then, we have:

$$f[i][k] = \begin{cases} 
0 & \text{if } i > k \\
1 & \text{else if } i = k \\
\max(f[i+1][k], f[i][k-1]) & \text{else if } s[i] \neq s[k] \\
f[i+1][k-1] + 2 & \text{otherwise}
\end{cases}$$

The answer for the problem is $f[1][n]$. The time and space complexity is $O(n^2)$. 
Statement. You are given a weighted DAG $G = (V, E)$. Find two nodes $(u, v)$, such that there is a path from $u$ to $v$ and the total weight of this path is maximized. You should output only the length of this path.
Dynamic Programming. Example 4 (heaviest path)

Let’s define $f[v]$ as the weight of the heaviest path that starts at node $v$. We process the nodes recursively and find $f[v]$ for each node only once. We must find $f[v]$ for every $v \in V$ and we can find them in any order. Then, we have:

$$f[v] = \begin{cases} 
0 & \text{if } \text{outdegree}[v] = 0 \\
\max(\max_{(v,u) \in E}(f[u] + w[v][u]), 0) & \text{otherwise}
\end{cases}$$

$f[u]$ is computed recursively first before finding the value for $f[v]$. The answer will be $\max_{v \in V} \{f[v]\}$. If we compute each value $f[v]$ only once, then the time complexity will be $O(|V| + |E|)$. 
Greedy algorithms

Greedy algorithms are a class of algorithms that make choices that lead to best locally optimal results. Such strategy does not always lead to the globally optimal outcome.

Greedy algorithms are often used in approximations. They also are quite common techniques to provide lower/upper bounds for branch-and-bound problems.

Classical examples for greedy algorithms are: Prim and Kruskal’s algorithms for finding minimum spanning trees, Dijkstra’s algorithm for finding shortest paths in graphs, interval scheduling and others.
Greedy algorithms. Prim’s algorithm review.

Prim’s algorithm builds MST iteratively. Initially, MST consists of one arbitrary node. At each step the algorithm greedily adds an edge with minimum weight that has exactly one of its ends in the current MST.

The local optimality here is the fact that the chosen edge has minimum weight among all the candidate edges. Using cut property we can prove, that these locally optimal steps will eventually lead the globally optimal answer, i.e. minimum spanning tree.

Prim’s algorithm can be implemented in \( \mathcal{O}(|V|^2) \) using adjacency lists and in \( \mathcal{O}(|E| \log |V|) \) using binary heaps.

We can use first implementation for dense graphs, while the second for sparse ones.
Greedy algorithms. Kruskal’s algorithm review.

Kruskal’s algorithm builds MST iteratively as well. We first sort all edges in an increasing order. Initially, ”MST” consists of $|V|$ nodes and zero edges. We try to add edges to the current MST in an increasing order of their weights if adding such edges do not lead to any cycles.

Again, the local optimality here is that we add edges with minimum possible weight.

Using cycle property, we can prove that these local optimal steps will lead to the global optimal answer as well.

Kruskal’s algorithm is implemented with disjoint-sets. Sorting edges takes $O(|E| \log |V|)$ time. Adding all edges one by one takes $O(|E|)$ time overall. Thus, overall time complexity is $O(|E| \log |V|)$.

We use Kruskal’s algorithm for sparse graphs. If edges are already sorted (or can be sorted in a linear time), then time complexity of the algorithm is $O(|E|)$. 