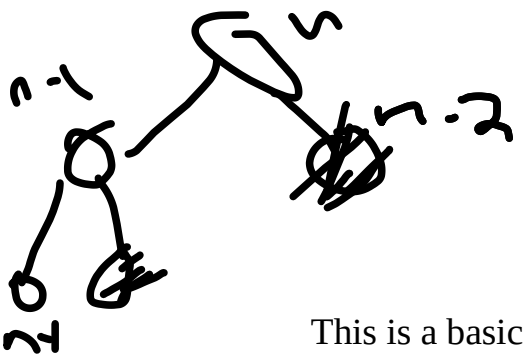
Joshua Gracie

COT 5405 HW4 Questions

1. Create an optimal time algorithm to determine the nth Fibonacci number.



First, create basic algorithm to solve:

If(n = 0 or 1) return 1

$Fib = Fib(n-1) + Fib(n-2)$

Return Fib

This is a basic recursive algorithm that solves Fibonacci in $O(2^n)$.

We can improve using memorization:

Create an array A of size n to store the Fibonacci numbers setting all values to negative infinity to start

Then $A_0 = 1 \; and \; A_1 = 1$:

If $A_n ! = -\infty$ return A_n

$Fib = Fib(n-1) + Fib(n-2)$

Set $A_n = Fib$ and return

This will now solve Fibonacci in $O(n)$. But we can still improve

Dynamic Programming Solution:

Create an array A to store the Fibonacci numbers and set $A_0 = 1 \; and \; A_1 = 1$

For i = 0 to n:

$$A_i = A_{i-1} + A_{i-2}$$ O(n)

This now removes all unnecessary recursions from our algorithm. But there is still one more improvement we can make

Optimized Space Complexity:

Create two variables $F_0 = 1$ *and* $F_1 = 1$

For i = 0 to n:

Fib = $F_0 + F_1$

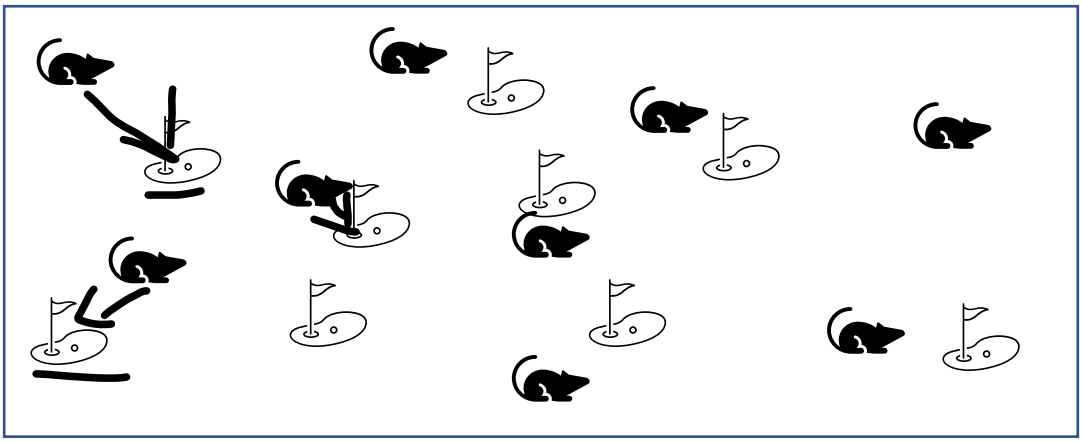
$F_1 = F_0$ *and* $F_0 = Fib$

F_0 n-1

F_1 n-2

This solves Fibonacci in $O(n)$ time with $O(1)$ space.

2. Assign Mice to Holes https://www.geeksforgeeks.org/assign-mice-holes/. Given N mice and N holes, were mice are scattered around the holes, assign mice to the holes such that the time it takes the last mouse to reach its hole is minimized. Mice can only move to the right by 1 or left by 1 per minute.



We can solve this with a greedy approach. Since the mice are scattered around the holes at certain positions, we can simply put mice in the hole closest to them in sorted order. This would minimize the time it would take each mouse to reach its final hole.

Sorting Approach

First, sort the mice in order of their positions.

Second, sort the holes in order of their positions.

Then, create an array to hold the time values for each mouse in FinalTime

For i = 0 to N:

FinalTime[i] = |Mouse[i] – Hole[i]|

Return the maximum of FinalTime.

This algorithm runs in $O(nlogn)$ time. But the tight bound is $\Theta(2nlogn + n)$. We can do better.

Binary Search Approach

Foreach mouse:

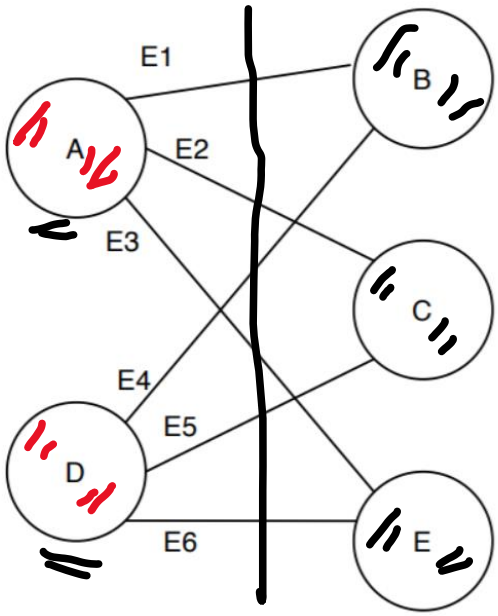
Use modified binary search to find the closes hole

Record the time in FinalTime[i]

Return the max of FinalTime

This new algorithm runs in $O(nlogn)$ same as before, but now also runs in $\Theta(nlogn)$ as well.

3. Determine if a graph is Bipartite (graph is split into two disjoint and independent sets where each vertex only connects to members of the other set) (https://www.baeldung.com/cs/graphs-bipartite)



This can be solved like BFS by assigning colors to each vertex. If a vertex connects to another vertex of the same color, then the graph is not bipartite.

Assign a red color to the starting vertex

Find the neighbors of the starting vertex and assign a blue color

Find the neighbor's neighbor and assign a red color

Continue this process until all the vertices in the graph assigned a color

During this process, if a neighbor vertex and the current vertex have the same color then the algorithm terminates. The algorithm returns that the graph is not a bipartite graph

4. Create an algorithm to find all the Egyptian fractions of a given fraction. (https://www.geeksforgeeks.org/greedy-algorithm-egyptian-fraction/)

Every positive fraction can be represented by unit fractions of the type $\frac{1}{n}$. These unit fractions are called Egyptian fractions. The goal is to come up with an algorithm to find all the Egyptian fractions of a given fraction, that is, find all the unit fractions that sum up to make the original fraction.

Examples:

$Original: \frac{2}{3} \quad Egyptian\ Fractions: \frac{1}{2} + \frac{1}{6}$

$Original: \frac{12}{13} \quad Egyptian\ Fractions: \frac{1}{2} + \frac{1}{3} + \frac{1}{12} + \frac{1}{156}$

We can solve this with a greedy solution by finding the largest Egyptian fraction (EF) possible, subtracting it from the original, and continuously finding the next largest EF.

We can determine the largest Egyptian fraction by taking the ceiling of the denominator over the numerator.

Example:

$Original: \frac{2}{3} \quad Largest\ EF: \left\lceil \frac{3}{2} \right\rceil \Rightarrow \frac{1}{2}$

Therefore, our algorithm will be:

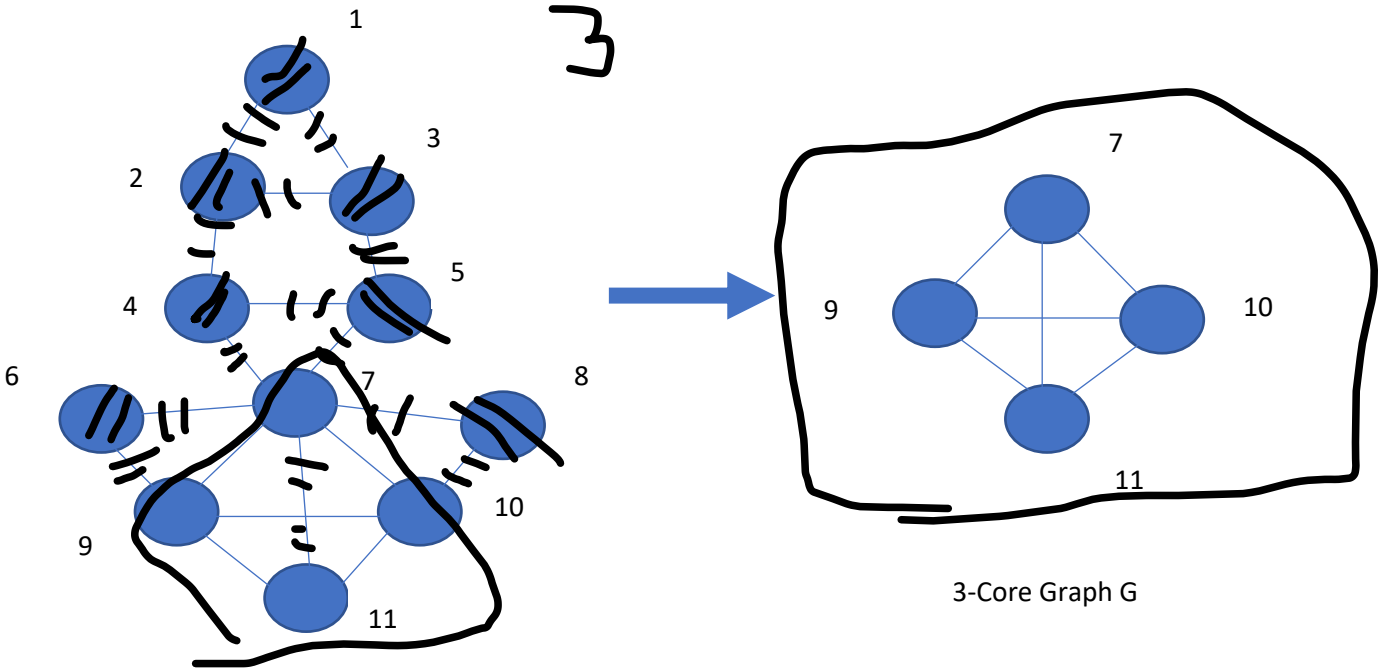
Calculate the ceiling of F` (denominator of F over numerator of F)

Subtract F by $\frac{1}{ceiling\ of\ F`}$

Repeat until final numerator is 1

5. Given a graph G and integer K, create an algorithm to find a K-Core graph of G (https://www.geeksforgeeks.org/find-k-cores-graph/)

A K-Core graph is a graph in which all the vertices have a degree of K or more. It is important to remember that when removing a vertex, the degrees of its neighbors must be reduced by one as well. Those neighbors may drop below K and therefore must be removed as well.



Graph G

3-Core Graph G

We can use DFS to solve this by iterating through the graph and deleting vertices with a degree less than K. We will repeat that until no vertices have been deleted.

Algorithm:

Run DFS on graph G:

If current vertex has a degree less than K, delete it.

Decrement the degree of neighbor vertices

Repeat until no vertex has been deleted