Group: 7	due: 18 October 2019

CSCI 432 Problem 3-1

Collaborators: *Peter Gifford, Kyle Brekke, Madison Henson, Ren Wall*

If 23 people are in a room, then the probability that at least two of them have the same birthday is at least one half. This is known as the birthday paradox, since the number 23 is probably much lower than you would expect. How many people do we need in order to have 50% probability that there are three people with the same birthday?

88 people

The probability P that there exists $W \geq 1$ three person birthday collision for any particular number of people n for any particular number of possible birthdays m can be calculated with the following function:

$$P(W \geq 1) = 1 - \sum_{i=0}^{n/2} \frac{m!n!}{i!(n-2i)!(m-n+i)!2^i m^n}$$

Using a simple algroithm that performs a modified binary search to find the lowest value of n for which $P(W \geq 1) \geq P_{\text{thresh}}$

---

procedure BINARY SEACH TRIPLE BIRTHDAY(number of days : m, probabilty threshold : P_{thresh})
    $L \leftarrow 3$
    $R \leftarrow 2m + 1$
    while $L < R$ **do**
        $n \leftarrow \text{floor}((L + R)/2)$
        $P_n = 1 - \sum_{i=0}^{n/2} \frac{m!n!}{i!(n-2i)!(m-n+i)!2^i m^n}$
        if $P_n < P_{\text{thresh}}$ **then**
            $L \leftarrow n + 1$
        else if $P_n \geq P_{\text{thresh}}$ **then**
            $R \leftarrow n$
        end if
    end while
    return L
end procedure

The algorithm has three components to its runtime. The outermost serarch while loop which has a complexity of $O(\log(m))$, the sum inside that loop which has a complexity of $O(m)$, and the inner value of the sum which has a complexity of $O(m(\log(m)\log(\log(m)))^2)$. The outermost loop has a decrementing function $D\colon \mathbb{X} \to \mathbb{N} \cup \{0\}$ defined $D(\mathbb{X}) = (L + R)/2$ which gives it a time complexity of $O(log(m)$ since initially $L + R = 2m - 2$. The sums complexity is $O(m)$ because $n \in \{3, 4, ..., 2m - 1\}$. The calculation of the inner value of the sum is bound by the calculation of $m!$, which if the factiorial is calculated with the fastest available algorithms is $O(m(\log(m)\log(\log(m)))^2)$.

The loop invarients of the main while loop are that $L \leq$ the minimum number of people requred for $P_n \geq P_{\text{thresh}}$, $R \geq$ the maximum number of people requred for $P_n \leq P_{\text{thresh}}$, and that $L \leq R$.

Group: 7		due: 18 October 2019
	CSCI 432 Problem 3-2	
Collaborators: *Peter Gifford, Kyle Brekke, Madison Henson, Ren Wall*		

Suppose we have a graph $G = (V, E)$ and three colors, and randomly assign a color each node (where each color is equally likely).

1. What is the probability that every edge has two different colors on assigned to its two nodes?

   Let $P(3-COLORING)$ be the probability that the graph is randomly colored such that no vertices share a connection with another vertex of the same color (henceforth referred to as a legal coloring).

   If G is any connected graph K_m for $m > 3$, then $P(3-COLORING)$ is 0. Additionally, if a clique K_m exists within G, then the probability of its being a three coloring is also 0. This is due to the fact that three colors will be insufficient to form a legal coloring in said cliques.

   If G is assumed to be a graph which contains no cliques with greater than 3 vertices, then it is possible to be legally colored. The probability that a graph of this type can be legally colored is contingent upon the arrangement of edges formed between each vertex, but can be calculated programmatically:

```
procedure threeColoringProbability(graph)
in: graph - a graph with |V| vertices and |E| edges
out: The probability the provided graph is randomly colored with a legal 3-coloring
    P_c ← 1
    for all vertices v in graph do
        if v is marked as colored then
            continue
        else
            if v is not connected to any other vertices in graph then
                Mark v as colored
            else
                Mark v as colored
                traverse and color each vertex accessible from v:
                    if The current vertex is connected to only one other vertex then
                        Mark the vertex as colored
                        P_c ← P_c * 2/3
                    end if
                    if No vertices connected to the current vertex are connected to each other then
                        Mark the current vertex as colored
                        P_c ← P_c * 2/3
                    end if
                    if Two or more of the connected vertices are also connected to each other then
                        Mark the current vertex
                        P_c ← P_c * 1/3
                    end if
                end traverse
            end if
        end if
    end for
    return P_c
end procedure
```

The algorithm operates in $\mathbf{O}(|V|^2)$ time. The algorithm functions such that it multiplies the base probability (1) by the probability of each vertex's coloring resulting in a legal 3-coloring, based off of the previously touched vertices. After completing, the algorithm should yield the probability of G being randomly colored with a legal 3-coloring. Once again though, this algorithm only works when the graph can be legally colored in the first place. Unfortunately, determining whether a graph can be 3-colored is $NP-Complete$.

The for loop in the algorithm can be represented with the decrementing function $D_F = |V| - i$, where the starting value is the number of vertices in G, $|V|$. The traversal can also be represented using a similar decrementing function $D_T = |V'| - i$, where $|V'|$ is the total number of vertices accessible from v. As such, we can confirm that the algorithm eventually terminates.

The runtime invariant for the algorithm is that $0 < P_c <= 1$.

2. What is the expected number of edges that have different colors assigned to its two nodes?

   The number of edges we expect to have different colorings on each end can be represented as $|E| * P_c$, where $|E|$ is the number of edges in G.

Group: 7 due: 18 October 2019

CSCI 432 Problem 3-3

Collaborators: *Peter Gifford, Kyle Brekke, Madison Henson, Ren Wall*

---

procedure DFSBestParty
    S.push V_1
    while S is not empty **do**
        if S.peek has not been marked **then**
            S.peek.marked ← true
            if S.peek has left child **then**
                S.push S.peek.leftchild
            else if S.peek has right sibling **then**
                S.push S.peek.rightsibling
            end if
        else
            if S.peek has left child **then**
                if S.peek.leftchild.inbest ≥ S.peek.leftchild.outbest **then**
                    S.peek.inbest ← S.peek.leftchild.outbest + S.peek.cv
                    S.peek.outbest ← S.peek.leftchild.inbest
                else
                    S.peek.inbest ← S.peek.leftchild.outbest + S.peek.cv
                    S.peek.outbest ← S.peek.leftchild.outbest
                end if
            else
                S.peek.inbest ← S.peek.cv
                S.peek.outbest ← 0
            end if
            S.pop
        end if
    end while
    if V_1.inbest ≥ V_1.outbest **then**
        bestList.append V_1.name
        V_1.isin ← true
    else
        V_1.isin ← false
    end if
    Walk the tree again appending every node appending every node whose parent is not and and which has an inbest ¿ outbest **return** bestList
end procedure

This algorithm has a time complexity of $O(|V| + |E|)$ since it simply runs two depth first searches and thus has the same time complexity as DFS.

Group: 7		due: 18 October 2019
	CSCI 432 Problem 3-4	
Collaborators: *Peter Gifford, Kyle Brekke, Madison Henson, Ren Wall*		

For the Greedy make change algorithm described in class on 10/02, describe the problem and solution in your own words, including the use of pseudocode (with more details than what was written in class). Note: you do not need to give a loop invariant and the proof of termination/runtime complexity.

The Greedy make change algorithm is meant to create one of the optimal solutions for making change, that is, to create change using the lowest number of coins. The solution to implement the Greedy make change algorithm is to sort an array of each denomination the currency being used has from large to small, iterate through the number of denominations the currency you are using has, and for each of the current denominations that you are at in the array add as many to the solution that don't cause the solution to go beyond the value of change you want to create. After you have iterated through the loop the solution you will have created will have the minimum number of coins given the currency used is a currency that the greedy make change algorithm works for.

greedyMakeChange(changeValue, denominations =[d_1,..., d_k]))

sort denominations from largest to smallest.

for i = 1 to k

  add as many d[i] to the set solution without exceeding changeValue

endfor

return the set solution.

Group: 7 due: 18 October 2019

CSCI 432 Problem 3-5

Collaborators: *Peter Gifford, Kyle Brekke, Madison Henson, Ren Wall*

Suppose we have n items that we want to put in a knapsack of capacity W. The i-th item has weight w_i and value v_i. The knapsack can hold a total weight of W and we want to maximize the value of the items in the knapsack. The *0-1 knapsack problem* will assign each item one of two states: in the knapsack, or not in the knapsack. The *fractional knapsack problem* allows you to take a percentage of each item.

1. Give an $O(n \log n)$ greedy algorithm for the fractional knapsack problem.

   This is a greedy solution to the knapsack problem. To make a greedy strategy work we find the ratio of value to weight for every item and put in items with the best ratio (largest) until the bag is full.

---

procedure GREEDYKNAPSACK(weight, value, capacity)
in: weight - list of weights of items, value - list of associated values for items, capacity - capacity of weights the sack can hold
out: list of best items to add to knapsack
 $addedWeight \leftarrow 0$
 $itemNum \leftarrow []$
 $iter \leftarrow 0$
 $proportions \leftarrow []$
 for $i \leftarrow 0, i < weight.length, i \leftarrow i + 1$ **do**
  $proportions.add((i, value[i]/weight[i]))$
 end for
 $proportion \leftarrow proportions.sortLowToHigh$ ▷ Uses Mergesort to sort high to low based on proportion
 while $addedWeight \leq capacity$ **do**
  $addedWeight \leftarrow weight[proportion[iter][0]] + addedWeight$
  $itemNum[iter] \leftarrow propotion[iter][0]$ ▷ Adds the i value given to object to list of values to assign to knapsack
  $iter \leftarrow iter + 1$
 end while
 return itemNum
end procedure

Decrementing Functions:
Let $\mathbb{X}$ denote the state space of the algorithm. We define the function $D \colon \mathbb{X} \to \mathbb{N} \cup \{0\}$ by $D(\mathbb{X}) = length(weight) - i$
Each time through the first for loop, i increases by one which will bring it closer and closer to the length of weight and therefore it will eventually equal the length of weight which will break the loop.
Let $\mathbb{X}$ denote the state space of the algorithm. We define the function $D \colon \mathbb{X} \to \mathbb{N} \cup \{0\}$ by $D(\mathbb{X}) = capacity - addedWeight$
Each time through the loop a new item weight from proportion is added to addedWeight. This will increase its value and bring it close to capacity in each loop and therefore will break the loop once it is equal to or greater than capacity. This assumes that there are enough items given in the problem to over fill the knapsack.

*There are recursive iterations found in the sorting function that is assumed to use merge sort. These have been previously proven to work and terminate and therefore I have left them out.

Justification of linear run time: The total runtime for this algorithm is not linear because of the sort, but the other elements of it are. The for loop goes through every element in the weight list and does not have any loops within it. Then the following while loop goes through at max as many items as there are in weight because weight was used in the for loop to establish the list of items

Loop Invariants:
For loop - proportions = (0, value[0]/weight[0])...(i, value[i]/weight[i])
While Loop - addedWeight = weight[0]...weight[proportion[iter][0], itemNum = proportion[0][0]...proportion[iter][0]

2. Give an $O(nW)$ time algorithm that uses dynamic programming to solve the 0-1 knapsack problem.

This is a dynamic programming solution. It functions by building up a table of values so that they do not have to be computed twice. Using this we can cut out all the extra computations and build up to the best solution with few extra steps. While there are two for loops they use two finite values that do not grow with the increased options of items to put in the sack which is very efficient.

---

procedure LINEARKNAPSACK(weight, value, capacity)
in: weight - list of weights of items, value - list of associated values for items, capacity - capacity of weights the sack can hold
out: Best possible value that can be fit in the bag.
$len \leftarrow weight.length$
$grid \leftarrow [len][capacity]$
for $i \leftarrow 0, i < len + 1, i \leftarrow i + 1$ **do**
for $j \leftarrow 0, j < capacity + 1, j \leftarrow j + 1$ **do**
if i=0 or j=0 **then** ▷ Ignores first row so it has all 0s and no negative index
$grid[i][j] \leftarrow 0$
else if $weight[i-1] \leq w$ **then**
$grid[i][j] \leftarrow max(value[i-1] + grid[i-1][j - weight[i-1]], grid[i-1][j])$ ▷ Fills grid spot with either the value being checked plus he best value of the last spot that is allowed weight wise or the previous best if it is better
else
$grid[i][j] \leftarrow grid[i-1][j]$ ▷ Gets the previous best
end if
end for
end for
return grid[len][capacity]
end procedure

Decrementing Functions:
Let $\mathbb{X}$ denote the state space of the algorithm. We define the function $D: \mathbb{X} \to \mathbb{N} \cup \{0\}$ by $D(\mathbb{X}) = (len + 1) - i$
As i increases by one each time through the loop it will get closer to len and eventually equal it terminating the loop. Let $\mathbb{X}$ denote the state space of the algorithm. We define the function $D: \mathbb{X} \to \mathbb{N} \cup \{0\}$ by $D(\mathbb{X}) = (capacity + 1) - j$
As j increases by one each time through the loop it will get closer to capacity and eventually equal it terminating the loop.

Justification of linear run time:
The loops in this both go to different values despite the fact that they are nested, this makes the runtime O(len*capacity) because for every time it goes through len it must run through the entirety of capacity. Loop Invariants:
for loop base on len: Rows indexed 0 to i of grid are filled with values.
for loop base on capacity: Value index j of row index i is filled with a value.