Group: 7 Names: Peter Gifford, Kyle Brekke, Madison Henson, Ren Wall due: 18 October 2019

CSCI 432 Problem 3-1

Collaborators: *None*

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? As a reminder, when giving an algorithm as an answer, you are expected to give:

- A prose explanation of the problem and algorithm.
- Psuedocode.
- The decrementing function for any loop or recursion, or a runtime justification.
- Justification of why the runtime is linear.
- The loop invariant for any loops, with full justification.

Group: 7 Names: Peter Gifford, Kyle Brekke, Madison Henson, Ren Wall due: 18 October 2019

CSCI 432 Problem 3-2

Collaborators: *None*

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

Group: 7 Names: Peter Gifford, Kyle Brekke, Madison Henson, Ren Wall due: 18 October 2019

CSCI 432 Problem 3-3

Collaborators: *None*

CLRS, Question 15-6.

Group: 7 Names: Peter Gifford, Kyle Brekke, Madison Henson, Ren Wall due: 18 October 2019

CSCI 432 Problem 3-4

Collaborators: *None*

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 Names: Peter Gifford, Kyle Brekke, Madison Henson, Ren Wall due: 18 October 2019

CSCI 432 Problem 3-5

Collaborators: *None*

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 $D\colon \mathbb{X} \to \mathbb{N} \cup \{0\}$ 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 $D\colon \mathbb{X} \to \mathbb{N} \cup \{0\}$ 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.