Homework 2: Backtracking and Dynamic Programming
(Part I: written questions)
DUE: 11:59pm Thur, Oct 13, 2022, on Blackboard

Academic Honesty:

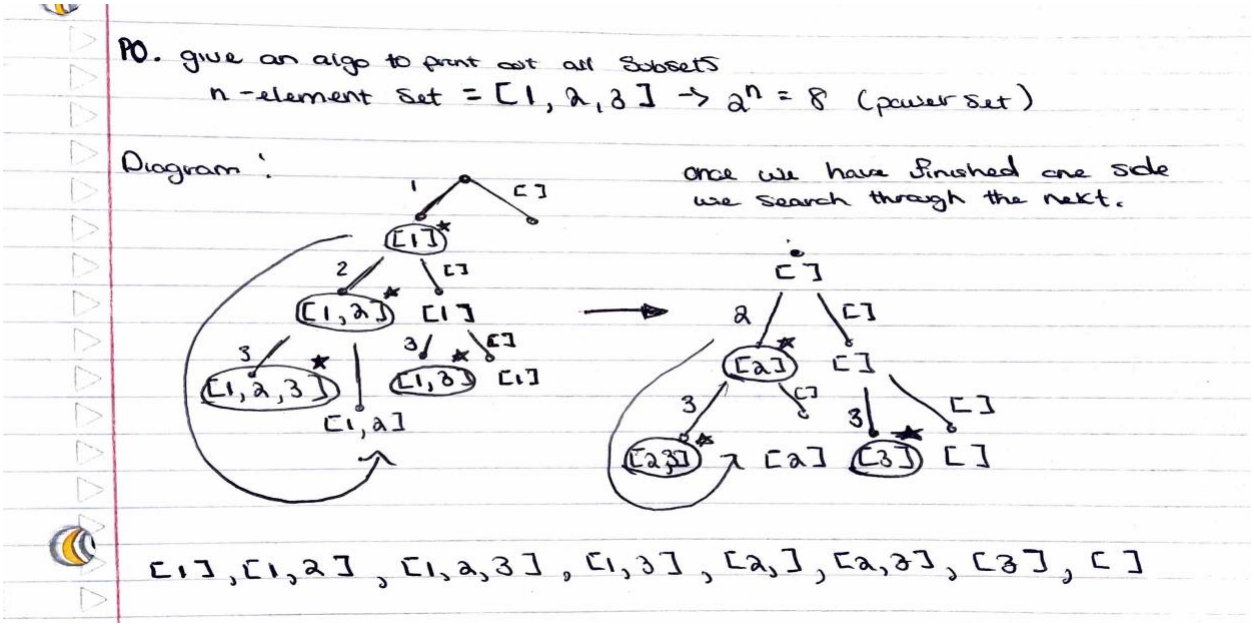
"I understand and agree to abide by the provisions in the University of Houston Undergraduate Academic Honesty Policy" and "This is my own work."

Backtracking:

P0 (15pt): Give an algorithm to print out all possible subsets of an n-element set, such as $\{1, 2, 3 \ldots, n\}$.

Our goal for this problem is to print out every possible subset of a given array[n] (think of this as a power set of the element set). We can get the number of subsets using 2^n which will give us the number of subsets depending on the n value. If we have 3 elements in our set, then we will have a total of 8 possible subsets ($2^n = 2^3 = 8$). We will use a brute force method of backtracking to solve for this. Our diagram will look like a tree, and we will use effectively be using DFS (depth first search) and printing each unique node as we go. [From Left to Right]

Visual Diagram Example:



Homework 2: Backtracking and Dynamic Programming

(Part I: written questions)

DUE: 11:59pm Thur, Oct 13, 2022, on Blackboard

Pseudocode:

```
# an array that points into an array
solveSubsets(array)
# subset will hold the current subsets throughout the recursion
subset = []
# start of recursion

solveDFS(i)
# base case
  if(i >= size[array])
    subsets.append(array[i])
    return

# recursive call
  # add subset
  subset.append(array[i])
  solveDFS(i+1)

  # skip subset
  subset.pop()
  solveDFS(i+1)

return subset
```

Proof and Time Complexity:

Finally, we say that the time complexity for this algorithm is $O(n * 2^n)$ since we generate a total number of 2^n as we stated earlier in the problem. This also doubles as our proof of correctness

Source: https://www.geeksforgeeks.org/backtracking-to-find-all-subsets/

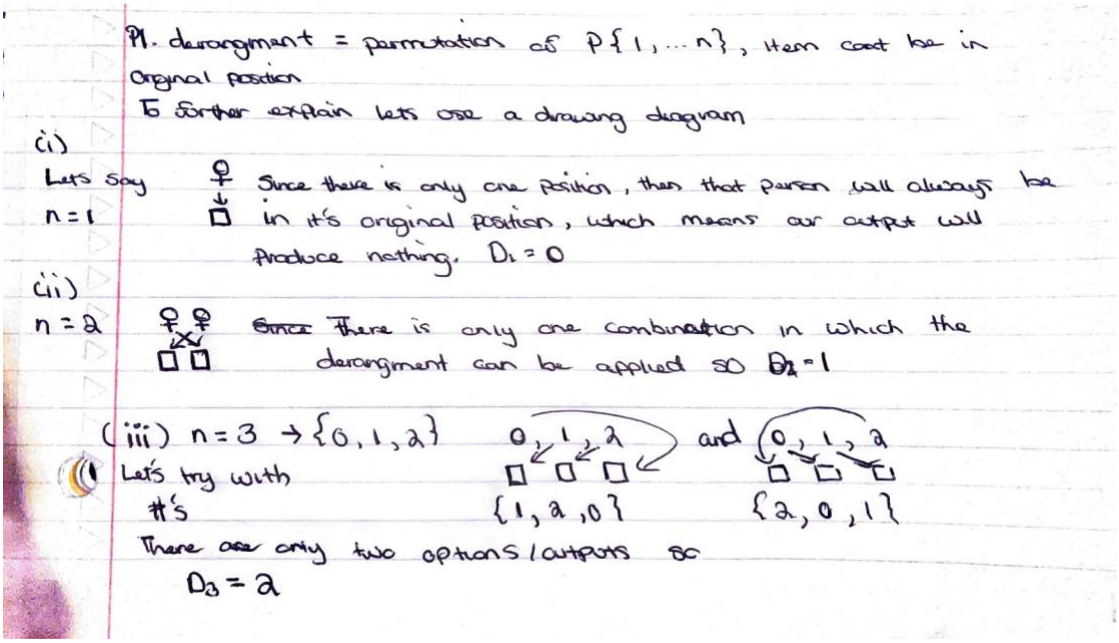
P1 (15pt): A *derangement* is a permutation p of $\{1, \ldots, n\}$ such that no item is in its proper position; i.e. $p_i \neq i$ for all $1 \leqslant i \leqslant n$. For example, $p = \{3, 1, 2\}$ is a derangement of $\{1,2,3\}$, whereas $p = \{3, 2, 1\}$ is not.

Write an efficient backtracking program that prints out all derangements of $\{1, \ldots, n\}$.

Using the description of what a derangement, (a permutation of an $p = \{1, \ldots n\}$ where no element appears in its proper/original position. To show this let's use a diagram.

Homework 2: Backtracking and Dynamic Programming
(Part I: written questions)
DUE: 11:59pm Thur, Oct 13, 2022, on Blackboard

Visual Diagram:



Using the concept/diagram above we can come up with an algorithm like P0. Now let's create an algorithm to return all derangements. Our code can basically be explained and split into three different parts. Our choice, constraints, and our goal. Our choice entails how we will solve our problem (using two arrays we will compare their values and positions to determine a derangement). Our constraint (our values cannot be in the original position or a taken position). Our goal is to return all derangements through a recursive backtracking algorithm.

Proof of Correctness:

Again, we must first understand what a derangement is. We have n (distinct objects) which we must arrange into n places (positions). For the first position we have n choices, the second we have n-1 and the next n-2 and so on. **So, the total in which we can arrange these objects into a list is $n!$ *ways.***

Source: https://www.youtube.com/watch?v=_2QTyYyAbSI

Homework 2: Backtracking and Dynamic Programming
(Part I: written questions)
DUE: 11:59pm Thur, Oct 13, 2022, on Blackboard

Source: https://www.youtube.com/watch?v=Zq4upTEaQyM (Helped structure my program)

Pseudocode:

```
# we will compare two arrays (people and positions)
# nums is our solution array
derangements(a[], nums: List[int])

# base cases
if nums == 1 return 0
if nums == 2. return 1

# recursive call
else
  for(i = 0; i < a.size]; i++)
    if(nums != original position && nums != used position)
      nums.append(a[i])
      derangements(a[], nums)
      nums.pop()
  return nums
```

While not being the fastest in terms of time complexity compared to dynamic programming. We can safely say that the algorithm will get accomplished reliably.

Extra Information: (If we want a faster algorithm than backtracking)

Let's say we must find the element of n objects = D_n *where we have n people and n assigned positions. We need to arrange in a fashion where these numbers can't go into their original positions. To do that we will say n-1 which will index it to the next possible position.*

People = $\{p_1, p_2, p_k, p_n\}$ *and Positions =* $\{pos_1, pos_2, pos_k, pos_n\}$

(p_k *is assigned to* pos_1) *which is the* $0th$ *element. Well, where does the element that was originally residing on the* 0^{th} *element go? It will go to where* p_k *was originally positioned which is* $n - 2 = D_{n-2}$. *Well now that the remaining p values can't go on the edge cases (first and last element), they can go n-1 =* D_{n-1}. *From this we get the recursion formula*

$$D_n = (n-1)(D_{n-1} + D_{n-2})$$

Homework 2: Backtracking and Dynamic Programming
(Part I: written questions)
DUE: 11:59pm Thur, Oct 13, 2022, on Blackboard

If we want to see a more efficient way of doing this problem rather than backtracking.

Source: https://www.geeksforgeeks.org/count-derangements-permutation-such-that-no-element-appears-in-its-original-position/

Dynamic Programming:

Dynamic Programming:

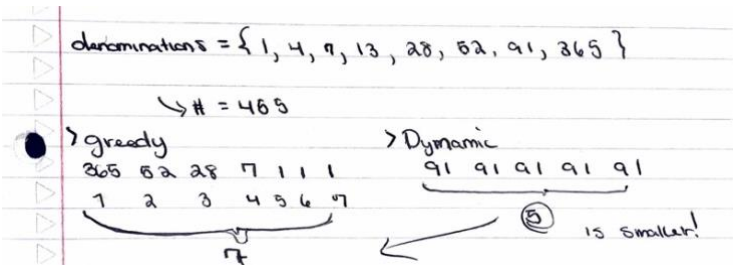
P3 (20pt): In a strange country, the currency is available in the following denominations: \$1, \$4, \$7, \$13, \$28, \$52, \$91, \$365. Find the minimum bills that add up to a given sum \$$k$.

(a) The greedy change algorithm repeatedly takes the largest bill that does not exceed the target amount. For example, to make \$122 using the greedy algorithm, we first take a \$91 bill, then a \$28 bill, and finally three \$1 bills. Give an example where this greedy algorithm uses more bills than the minimum possible. [Hint: It may be easier to write a small program than to work this out by hand.]

(b) Describe and analyze a recursive algorithm that computes, given an integer k, the minimum number of bills needed to make \$$k$. (Don't worry about making your algorithm fast; just make sure it's correct.)

(c) Describe a dynamic programming algorithm that computes, given an integer k, the minimum number of bills needed to make \$$k$. (This one needs to be fast.)

(a) Let's say that we had an array of denominations D{\$11, \$9, \$5, \$1} and were given a task to find the minimum number of denominations used to add up to N. In this case our N value is 14. Our greedy algorithm will take the largest value denomination which is ($N >$ $D[i]$). In this case our largest number that fits into the N is the first array value 11. After that 9 and 5 can't fit into N so we're forced to use three 1's. However, there is a more efficient solution to this would be to add the second and third array values ($9 + 5$) which add up to 14 in less additions and uses the minumum number of bills. An example of a number for this problem's denomination is **455.**



Homework 2: Backtracking and Dynamic Programming
(Part I: written questions)
DUE: 11:59pm Thur, Oct 13, 2022, on Blackboard

(b) **Texts**:

A recursive algorithm we could use that would yield us any possible solution is a greedy algorithm. This algorithm does not worry about if it's the best possible solution (time complexity is not of importance). All it needs to do is prove that it can solve problem at hand. Following the prompt, the given denominations are:

D{$1, $4, $7, $13, $28, $52, $91, $365} and N = $(x) | x = any positive integer

Like we mentioned in (a), our greedy algorithmth will take the largest bill that does not exceed N. It will take the greatest value first in D{} that is $N > bill[i]$ and will keep subtracting denominators from N until $N - bills[i] \geq 0$.

Proof of Correctness:

- Base Case: If $N == 0$, $return\ 0$;
- $minBill(int\ bill[], m, N)\ where\ N > bill[i]\ then\ recurse,\ return\ minCount$

Psuedocode 1:

Recursive: O(n^2): Would traverse through two level's of tree in the worst case. The worst case would be if $N < bill[i]$

```
# instead of a cache, array of bills)
# bills[i] = {$1, $4...}
int leastBills(bills[i], m, N)

# base case
if N == 0 return 0

# set all to infinity or -1
numBill = INT_MAX
# recursive call
for (m-1,... 1)
  if N >= bills[i]
  # subtract N by the largest possible bill
    count = leastBills(bill, m, N - bills[i])
    if(1 + count < numBill)
      numBill = 1 + count

return numBill
```

Homework 2: Backtracking and Dynamic Programming
(Part I: written questions)
DUE: 11:59pm Thur, Oct 13, 2022, on Blackboard

(c) Texts:

For a dynamic program, it must give us the best possible solution throughout the entire recursive tree. We are looking for the minimum number of bills. Unlike the greedy algorithm, we wont intuitavely search for the largest number in D{} or in this case bill[]. We will recurse through all the possibilities/subproblems while saving our minCount's in a separate array/table which we will later compare to one another. Our dynamic program will use a bottom-up approach. We will also for simplicity sake have our bill[] in descending order. {$365, $91, $52, …}

Proof of Correctness:

- Base Case: If $N == 0$, $return\ 0$;
- $If\ N > 0$

$$minBills(bill[0, m-1, N] = min\ (1 + minBills(N - bills[i])$$

Psuedocode: Similar to the one above however we save the partial solutions and compare them

```
int leastBills(bills[i], m, N)

int table[N+1]
# base case
table[0] = 0

# recursive call
for(i = 1, ... as <= N)
  for(j = 0... m-1)
    if N >= bills[i]
    sub = table[i - bills[i]]
    if 1 + sub < table[i]
       table = 1 + sub
return table[N]
```

As we can see, this solution is somewhat similar to out greedy algorithm as we did above with the added subproblem/partial solutions saved into an array. The time complexity of this algorithm is $O(m * N)$

Source: https://www.enjoyalgorithms.com/blog/minimum-coin-change

---

Programming Questions 1-3

Problem 1: Backtracking

This problem is very similar to the N-Queens backtracking problem with an added constraint of '. ' being a blank region (cannot place). We also don't have to account for a diagonal constraint. Backtracking problems are all similar in nature and the following source helped.

Source: https://www.interviewbit.com/blog/8-queens-problem/

I discussed this problem with classmate Kabeer Ali.

ID: 1874349

Submission ID: 8071d129-8083-4c6e-9e4b-cdf67264275d

Please save the submission ID if you want to later retrieve it, or include it in your homework.

PSID: 1874349
Submission ID: 8071d129-8083-4c6e-9e4b-cdf67264275d

2022-10-17 16:26:31.535235893 -0500 CDT

Compilation error msg:

```
: In function 'int main(int, char**)':
:49:11: warning: ignoring return value of 'int fscanf(FILE*, const char*, ...)', declared with attribute warn_unused_result [-Wunused-result]
```

takes 418.111446ms

---

test 0 PASS

test 1 PASS

test 2 PASS

test 3 PASS

test 4 PASS

test 5 PASS

test 6 PASS

takes 19.641279ms

Homework 2: Backtracking and Dynamic Programming
(Part I: written questions)
DUE: 11:59pm Thur, Oct 13, 2022, on Blackboard

Problem 2: Divide & Conquer

To solve this problem optimally we will have to get two optimal indexes of array []. We will split the array in half, the left index i and right index j. We find Lmin[i] and Rmax[j]. After this, we traverse both sides of array from left to right and while traversing, if we see that Lmin[i] is greater than Rmax[j], then i++. During this we will also be recusing to get the max distance between the left and right. We will then compare the two constructs to determine the max b/w them. I discussed this problem with Kabeer Ali.

ID: 1874349

Submission ID: 52c3a431-fddf-42de-83b4-c616528cfcf5

Source: https://www.geeksforgeeks.org/given-an-array-arr-find-the-maximum-j-i-such-that-arrj-arri/

Please save the submission ID if you want to later retrieve it, or include it in your homework.

PSID: 1874349
Submission ID: 52c3a431-fddf-42de-83b4-c616528cfcf5

2022-10-17 19:30:56.603800951 -0500 CDT

Compilation error msg:

```
: In function 'int main(int, char**)':
:54:11: warning: ignoring return value of 'int fscanf(FILE*, const char*, ...)', declared with attribute warn_unused_result [-Wunused-result]
```

takes 379.911851ms

---

test 0 PASS

test 1 PASS

test 2 PASS

test 3 PASS

test 4 PASS

test 5 PASS

test 6 PASS

test 7 PASS

test 8 PASS

takes 526.514212ms

Homework 2: Backtracking and Dynamic Programming
(Part I: written questions)
DUE: 11:59pm Thur, Oct 13, 2022, on Blackboard

Problem 3: Dynamic Programming (INCOMPLETE)

For this problem, our objective is to find the minimum path from the top left corner to the bottom right corner. We will do this recursively with all possible combinations and choose the one with the minimum cost. There are plenty of ways to do this, like using a modified version of Dijkstra's, however a dynamic bottom-up approach can be done as well.

Source: https://algorithms.tutorialhorizon.com/dynamic-programming-minimum-cost-path-problem/