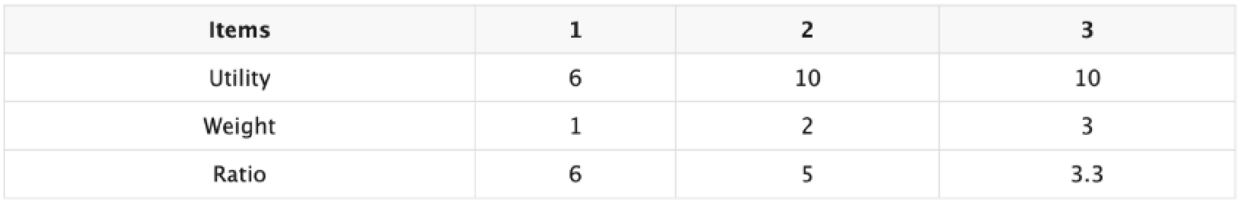
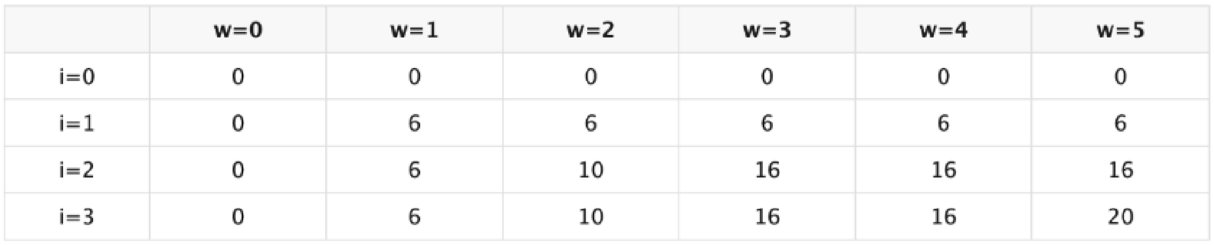
Hasna Zakiyyah

1301164599

1. Dynamic Programming

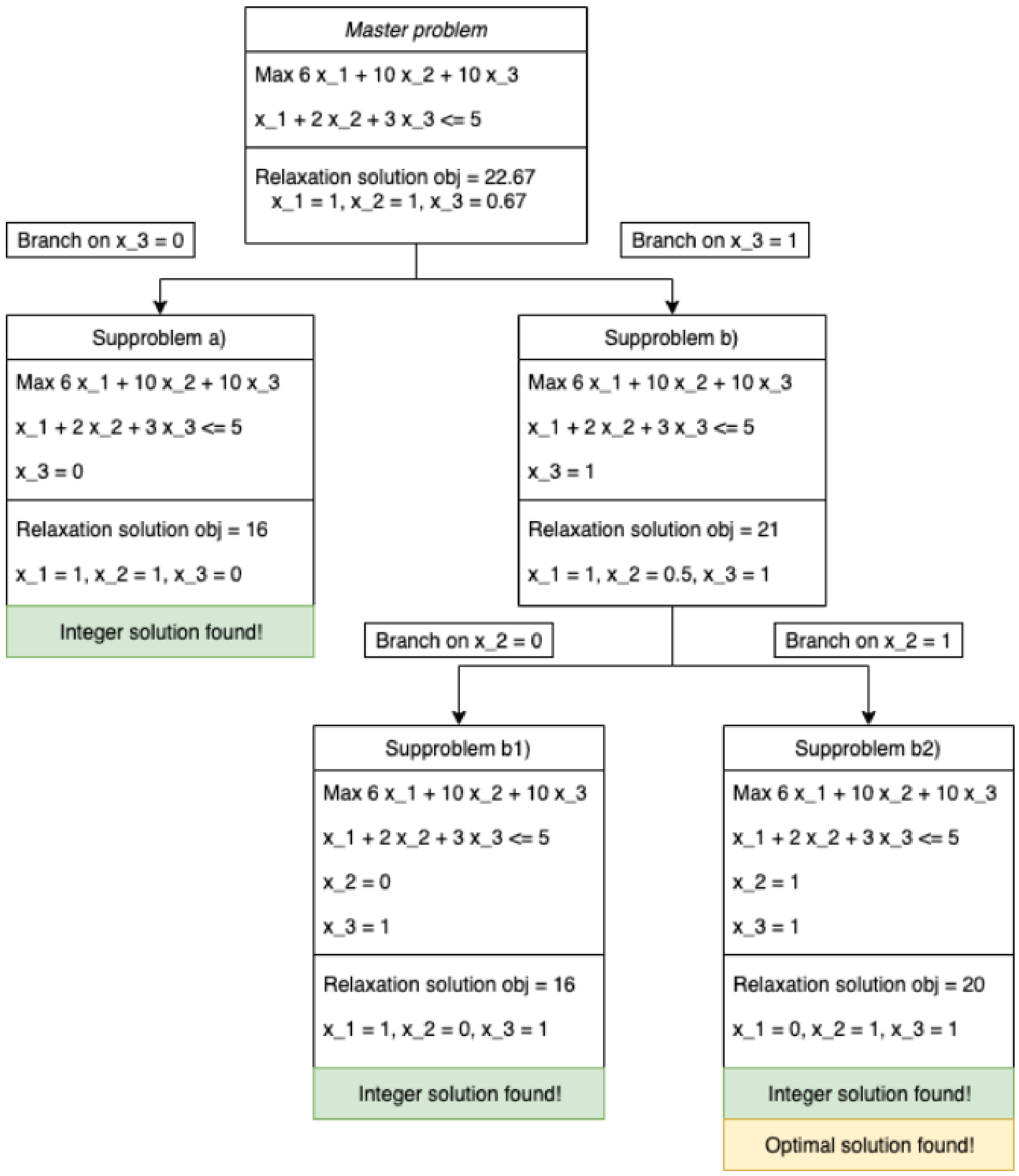


Items	1	2	3
Utility	6	10	10
Weight	1	2	3
Ratio	6	5	3.3

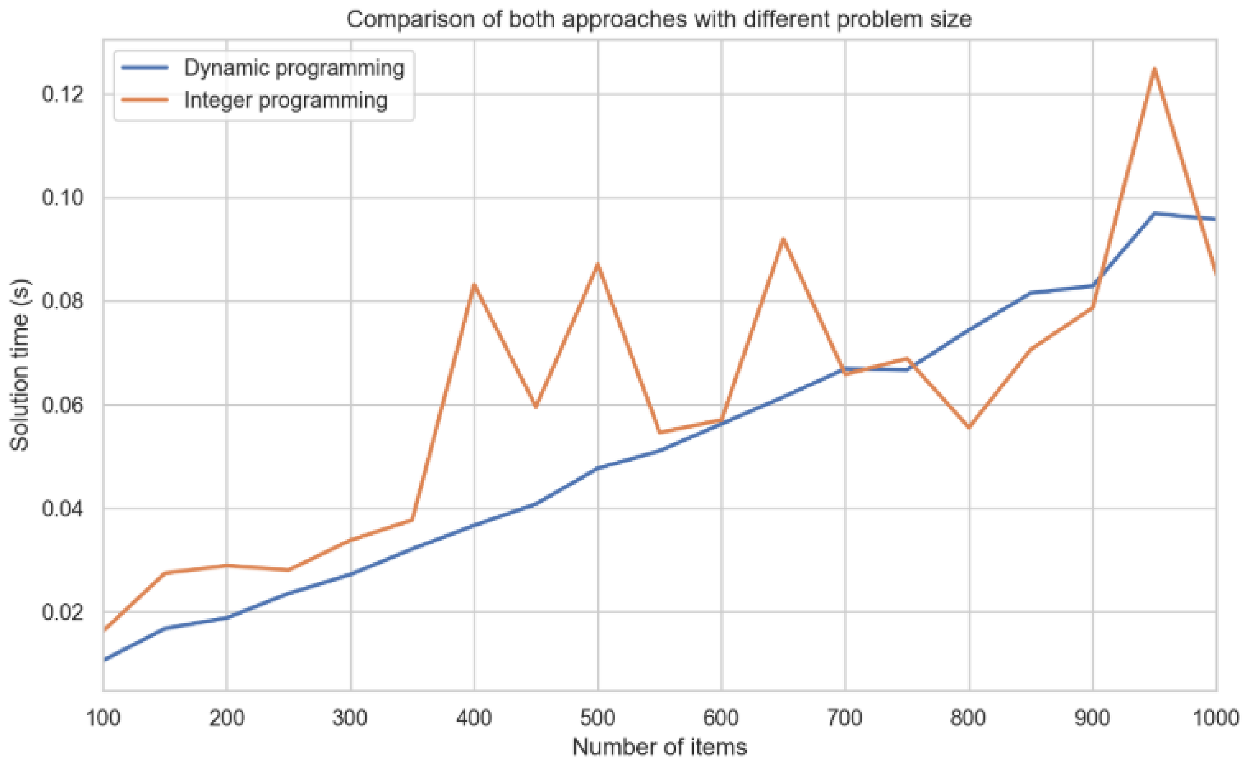


	w=0	w=1	w=2	w=3	w=4	w=5
i=0	0	0	0	0	0	0
i=1	0	6	6	6	6	6
i=2	0	6	10	16	16	16
i=3	0	6	10	16	16	20

Backtracking


Master problem
Max 6 x_1 + 10 x_2 + 10 x_3
x_1 + 2 x_2 + 3 x_3 <= 5
Relaxation solution obj = 22.67
x_1 = 1, x_2 = 1, x_3 = 0.67
Branch on x_3 = 0
Branch on x_3 = 1
Supproblem a)
Max 6 x_1 + 10 x_2 + 10 x_3
x_1 + 2 x_2 + 3 x_3 <= 5
x_3 = 0
Relaxation solution obj = 16
x_1 = 1, x_2 = 1, x_3 = 0
Integer solution found!
Supproblem b)
Max 6 x_1 + 10 x_2 + 10 x_3
x_1 + 2 x_2 + 3 x_3 <= 5
x_3 = 1
Relaxation solution obj = 21
x_1 = 1, x_2 = 0.5, x_3 = 1
Branch on x_2 = 0
Branch on x_2 = 1
Supproblem b1)
Max 6 x_1 + 10 x_2 + 10 x_3
x_1 + 2 x_2 + 3 x_3 <= 5
x_2 = 0
x_3 = 1
Relaxation solution obj = 16
x_1 = 1, x_2 = 0, x_3 = 1
Integer solution found!
Supproblem b2)
Max 6 x_1 + 10 x_2 + 10 x_3
x_1 + 2 x_2 + 3 x_3 <= 5
x_2 = 1
x_3 = 1
Relaxation solution obj = 20
x_1 = 0, x_2 = 1, x_3 = 1
Integer solution found!
Optimal solution found!

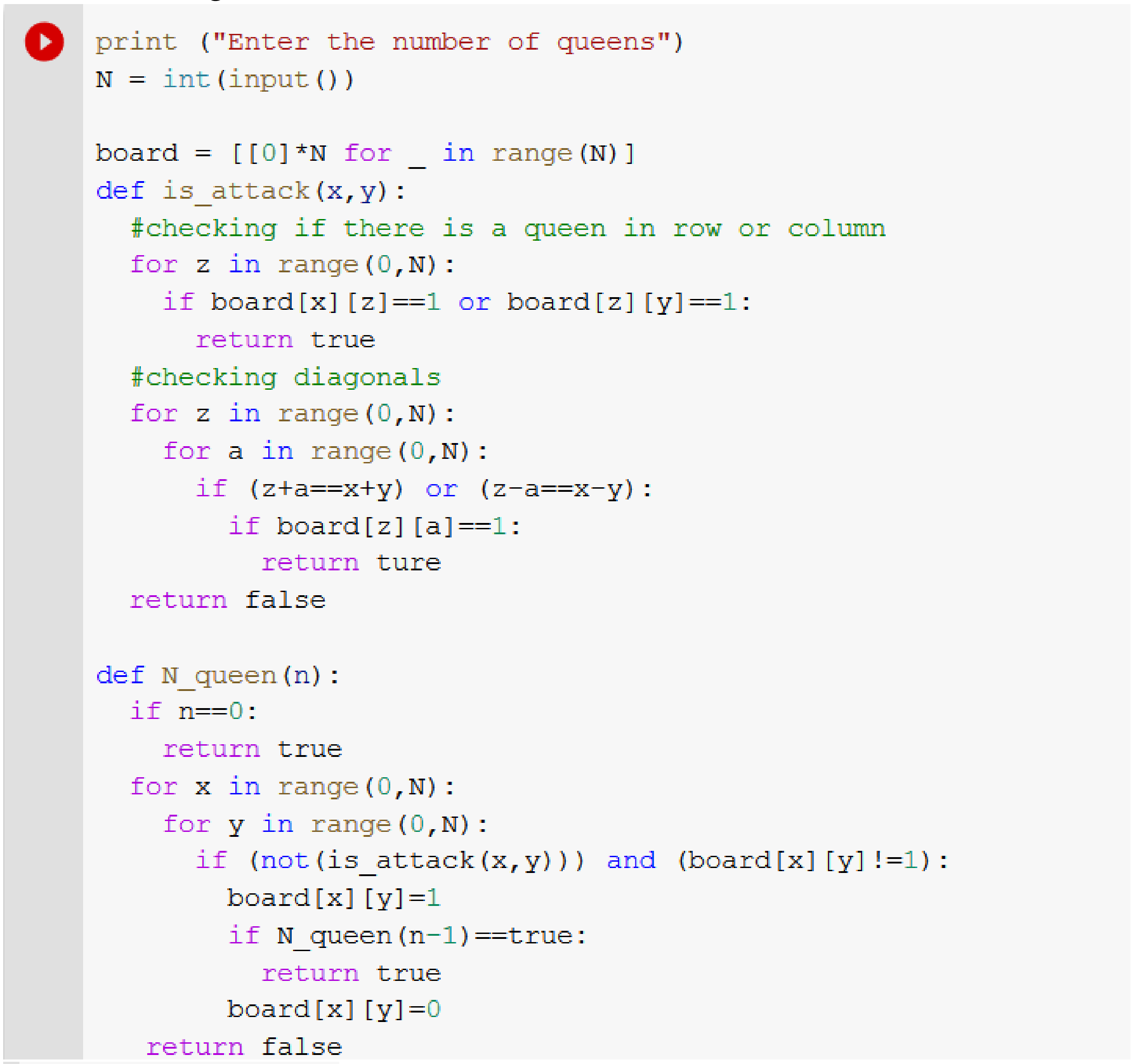
Dynamic programming exploit the problem structure and builds towards the optimal solution from smaller problems recursively, while integer programming recursively partitions the problem space to smaller trunks, and use estimated bounds to discard uninteresting solution partitions to accelerate the search. Dynamic programming is like a super smart enumeration, and it avoids unnecessary computations by always building upon simpler problem's optimal solutions. Backtracking does not necessarily work exclusively within the solution set. Rather, it uses the information gained from solving the relaxations to refine the lower and upper bounds, and work towards closing the gap between the bounds. The bounds can help eliminate parts of the solution space that does not contain better solutions, which is why branch-and-bound can be very efficient. Dynamic programming is great when the problem structure is nice, and the solution set is moderate. Backtracking can be very efficient if you have efficient ways to compute quality lower and upper bounds on the solutions.



From the above plot, it can be observed that for small to moderate size problems, dynamic programming approach is very competitive against integer programming (backtracking) approach. As the size of problem increase, the solution time of both algorithms increases. With the experiment setup, it seems there is no clear advantage of one algorithm to the other.

2. Efficient can be defined in terms either of number of colors or computation complexity. If we consider least number of colors required then,
   Greedy: may not always generate optimal solution, it depends on ordering of vertex chosen.
   Backtracking: always give solution how graph can be colored with given 'm' colors if possible.
3. List applications of m-coloring problem :
   a. Making schedule or time table for example make a schedule for exam in the university. We have list different subjects and students enrolled in every subject
   b. Mobile radio frequency assignment like when frequencies are asiigned to towers, frequencies assigned to all towers at the same location must be different.
   c. Sudoku is also a variation of Graph coloring problem where every cell represents a vertex. There is an edge between two vertices if they are in same row or same column or same block.
   d. Register allocation, process of assigning a large number of target program variables into a small number of CPU registers.

e. Bipartite graphs is to check is the graph bipartite or not by coloring the graph using two colors.
f. Map coloring is geographical maps of countries or states where no two adjacent cities cannot be assigned same color.

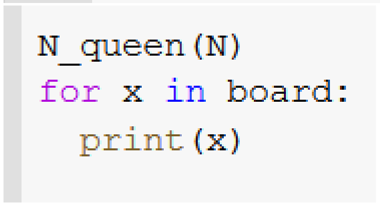


```
print ("Enter the number of queens")
N = int(input())

board = [[0]*N for _ in range(N)]
def is_attack(x,y):
  #checking if there is a queen in row or column
  for z in range(0,N):
    if board[x][z]==1 or board[z][y]==1:
      return true
  #checking diagonals
  for z in range(0,N):
    for a in range(0,N):
      if (z+a==x+y) or (z-a==x-y):
        if board[z][a]==1:
          return ture
  return false

def N_queen(n):
  if n==0:
    return true
  for x in range(0,N):
    for y in range(0,N):
      if (not(is_attack(x,y))) and (board[x][y]!=1):
        board[x][y]=1
        if N_queen(n-1)==true:
          return true
        board[x][y]=0
   return false
```

4.



```
N_queen(N)
for x in board:
  print(x)
```