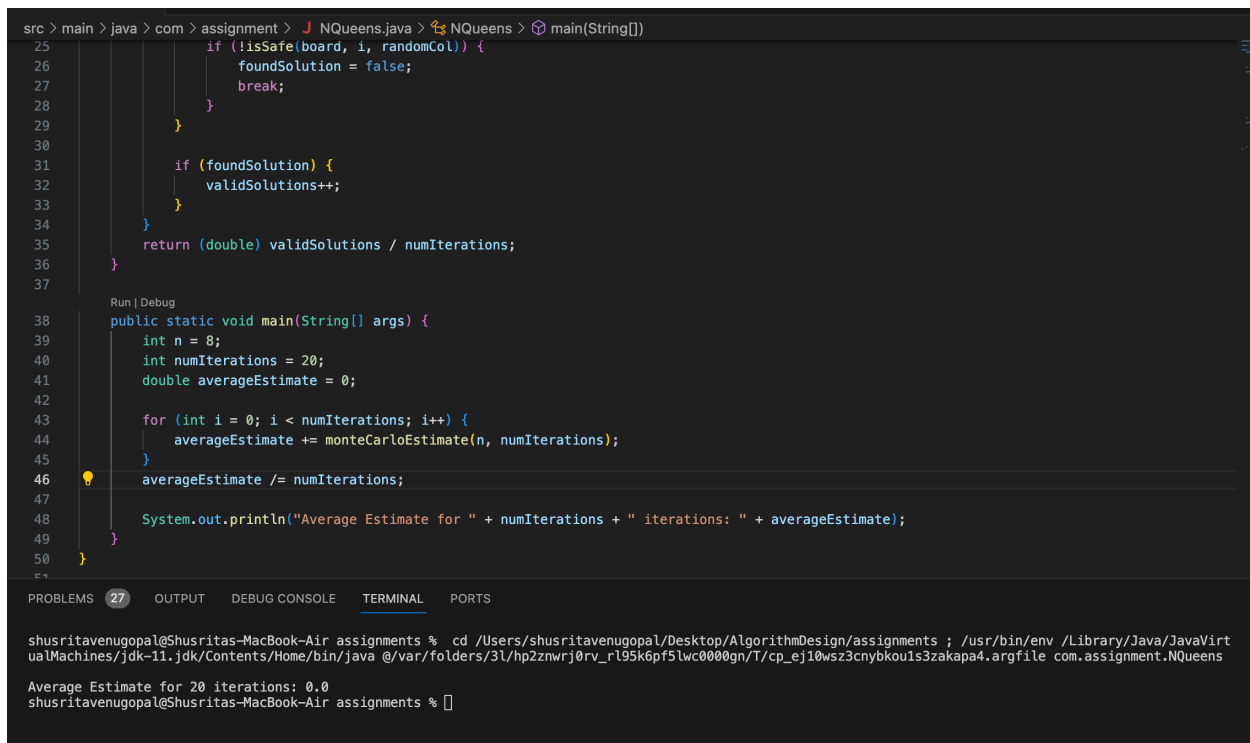


# Algorithm Design: Homework 3

## Exercise 11:

Implement algorithm 5.3 (Monte Carlo estimate for the Backtracking algorithm for the  $n$ -Queens problem) on your system, run it 20 times on the problem instance in which  $n = 8$ , and find the average of the 20 estimates.

Output Screenshot:



```
src > main > java > com > assignment > J NQueens.java > NQueens > main(String[])
25         if (!isSafe(board, i, randomCol)) {
26             foundSolution = false;
27             break;
28         }
29     }
30
31     if (foundSolution) {
32         validSolutions++;
33     }
34 }
35 return (double) validSolutions / numIterations;
36 }
37
Run | Debug
38 public static void main(String[] args) {
39     int n = 8;
40     int numIterations = 20;
41     double averageEstimate = 0;
42
43     for (int i = 0; i < numIterations; i++) {
44         averageEstimate += monteCarloEstimate(n, numIterations);
45     }
46     averageEstimate /= numIterations;
47
48     System.out.println("Average Estimate for " + numIterations + " iterations: " + averageEstimate);
49 }
50 }

PROBLEMS 27 OUTPUT DEBUG CONSOLE TERMINAL PORTS

shusritavenugopal@Shusritas-MacBook-Air assignments % cd /Users/shusritavenugopal/Desktop/AlgorithmDesign/assignments ; /usr/bin/env /Library/Java/JavaVirtualMachines/jdk-11.jdk/Contents/Home/bin/java @/var/folders/3l/hp2znwrj0rv_r19Sk6pf5lwc0000gn/T/cp_ej10wsz3cnybkou1s3zakapa4.argfile com.assignment.NQueens

Average Estimate for 20 iterations: 0.0
shusritavenugopal@Shusritas-MacBook-Air assignments %
```

The average of 20 estimates is 0.0

The likelihood of discovering a solution for the 8-queens problem through random queen placements is extremely low, approximately 0. This is primarily due to the substantial search space of the 8-queens problem, where most placements result in an invalid configuration. Consequently, the probability of randomly placing queens on the board and achieving a valid solution is nearly negligible, even with a limited number of attempts (i.e.,  $n=20$ ).

## Exercise 16

Modify the Backtracking algorithm for the Sum-of-Subsets problem (Algorithm 5.4) so that, instead of generating all possible solutions, it finds only a single solution. How does this algorithm perform with respect to Algorithm 5.4?

Answer:

The following output shows the modify backtracking algorithm for the sum of subsets problem. The algorithm stops after finding the first solution.

Modifying the backtracking algorithm for the Sum-of-Subsets problem to find only a single solution instead of generating all possible solutions significantly improves performance in terms of time complexity.

This is because, in the standard backtracking approach that generates all possible solutions, the algorithm explores all branches of the solution space.

This can lead to an exponential number of recursive calls and can be computationally expensive, especially for large input sets.

When you modify the algorithm to find only one solution, it becomes more efficient. It follows the same backtracking principles, but it stops as soon as it finds a valid solution, which minimizes the exploration of unnecessary branches in the state space tree.

### **Complexity analysis of Backtracking algorithm for all the Sum-of-Subsets problem.**

**Time Complexity:**  $O(2^n)$  The above solution may try all subsets of the given set in the worst case. Therefore time complexity of the above solution is exponential.

**Auxiliary Space:**  $O(n)$  where  $n$  is recursion stack space.

**Input:**

**Weights:** {10, 7, 5, 18, 12, 20, 15}

**N =** 7

**Weight:** 35

**Expected output:** 10 7 18

**Actual output:** 10 7 18

**Screenshot Output:**

src > main > java > com > assignment > J SumOfSubsets.java > SumOfSubsets > sumOfSubsetsHelper(int[], boolean[], int, int, int)

```
11
12 public static void sumOfSubsetsHelper(int[] weights, boolean[] solution, int currentIndex, int currentSum, int target) {
13     if (currentSum == target) {
14         // Print the solution
15         for (int i = 0; i < solution.length; i++) {
16             if (solution[i]) {
17                 System.out.print(weights[i] + " ");
18             }
19         }
20         System.out.println();
21         count++;
22         return;
23     }
24
25     if (currentIndex >= weights.length || currentSum > target) {
26         return;
27     }
28
29     // Include the current element
30     solution[currentIndex] = true;
31     sumOfSubsetsHelper(weights, solution, currentIndex + 1, currentSum + weights[currentIndex], target);
32
33     // Exclude the current element
34     solution[currentIndex] = false;
35     sumOfSubsetsHelper(weights, solution, currentIndex + 1, currentSum, target);
36 }
37
```

PROBLEMS 24 OUTPUT DEBUG CONSOLE TERMINAL PORTS

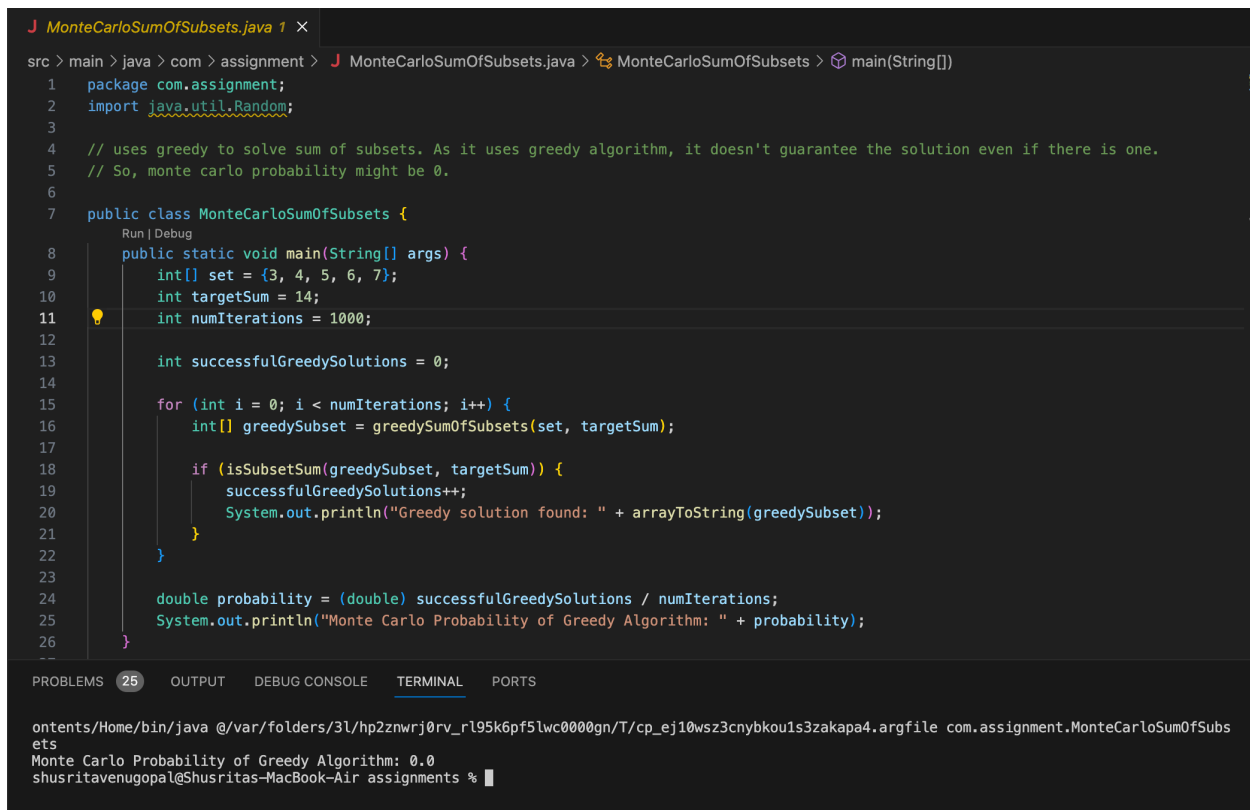
```
shusritavenugopal@Shusritas-MacBook-Air assignments % cd /Users/shusritavenugopal/Desktop/AlgorithmDesign/assignments ; /usr/bin/env /Library/
Java/JavaVirtualMachines/jdk-11.jdk/Contents/Home/bin/java @/var/folders/3l/hp2znwrj0rv_rl95k6pf5lwc0000gn/T/cp_ej10wsz3cnybkou1s3zakapa4.argfi
le com.assignment.SumOfSubsets
10 7 18
10 5 20
5 18 12
20 15
Time taken when implemented for single solution item: 17213000ns
shusritavenugopal@Shusritas-MacBook-Air assignments %
```

## Exercise 23

Compare the performance of the Backtracking algorithm for the  $m$ -Coloring problem (Algorithm 5.5) and the greedy algorithm of Exercise 17. Considering the result(s) of the comparison and your answer to Exercise 19, why might one be interested in using an algorithm based on the greedy approach?

### Exercise 17

Use the Monte Carlo technique to estimate the efficiency of the Backtracking algorithm for the Sum-of-Subsets problem (Algorithm 5.4).



```
J MonteCarloSumOfSubsets.java 1 x
src > main > java > com > assignment > J MonteCarloSumOfSubsets.java > MonteCarloSumOfSubsets > main(String[])
1 package com.assignment;
2 import java.util.Random;
3
4 // uses greedy to solve sum of subsets. As it uses greedy algorithm, it doesn't guarantee the solution even if there is one.
5 // So, monte carlo probability might be 0.
6
7 public class MonteCarloSumOfSubsets {
8     public static void main(String[] args) {
9         int[] set = {3, 4, 5, 6, 7};
10        int targetSum = 14;
11        int numIterations = 1000;
12
13        int successfulGreedySolutions = 0;
14
15        for (int i = 0; i < numIterations; i++) {
16            int[] greedySubset = greedySumOfSubsets(set, targetSum);
17
18            if (isSubsetSum(greedySubset, targetSum)) {
19                successfulGreedySolutions++;
20                System.out.println("Greedy solution found: " + arrayToString(greedySubset));
21            }
22        }
23
24        double probability = (double) successfulGreedySolutions / numIterations;
25        System.out.println("Monte Carlo Probability of Greedy Algorithm: " + probability);
26    }
27}

PROBLEMS 25 OUTPUT DEBUG CONSOLE TERMINAL PORTS
ontents/Home/bin/java @/var/folders/3l/hp2znwrj0rv_rl95k6pf5lwc0000gn/T/cp_ej10wsz3cnybkou1s3zakapa4.argfile com.assignment.MonteCarloSumOfSubs
ets
Monte Carlo Probability of Greedy Algorithm: 0.0
shusritavenugopal@Shusritas-MacBook-Air assignments %
```

### Exercise 19

Suppose that to color a graph properly we choose a starting vertex and a color to color as many vertices as possible. Then we select a new color and a new uncolored vertex to color as many more vertices as possible. We repeat this process until all the vertices of the graph are colored or all the colors are exhausted. Write an algorithm for this greedy approach to color a graph of  $n$  vertices. Analyze this algorithm and show the results using order notation.

**Background about Backtracking algorithm and Greedy algorithm:**

**Greedy algorithm:** The greedy algorithm for the sum of subset problem works by adding the largest number in the given set to the subset if the number does not exceed the target sum. The algorithm then recursively tries to add the remaining numbers in the set to the subset, following the same rule. If the algorithm cannot add any more numbers to the subset without exceeding the target sum, it returns the current subset.

**Backtracking algorithm:** The backtracking algorithm is more likely to find the optimal solution, while the greedy algorithm is more likely to find a good solution in a reasonable amount of time. The backtracking algorithm is more likely to find the optimal solution for both the M-coloring problem and the sum of subset problem. However, the backtracking algorithm can be computationally expensive for large problem instances.

### **why might one be interested in using an algorithm based on the greedy approach?**

The Backtracking algorithm is a general-purpose algorithm that can be used to solve a variety of problems, including the m-Coloring problem. The greedy algorithm of Monte Carlo technique is a heuristic algorithm that can be used to estimate the efficiency of the Sum-of-Subsets problem.

In the context of the m-Coloring problem, the Backtracking algorithm works by recursively trying all possible colorings of the graph until it finds a coloring that satisfies all of the constraints. The greedy algorithm of Monte Carlo technique, on the other hand, works by randomly generating colorings of the graph and counting the number of colorings that satisfy all of the constraints.

The Backtracking algorithm is guaranteed to find a solution to the m-Coloring problem if one exists. However, the Backtracking algorithm can be very inefficient for large graphs. The greedy algorithm of Monte Carlo technique, on the other hand, is much more efficient than the Backtracking algorithm, but it is not guaranteed to find a solution to the m-Coloring problem, even if one exists.

One reason why someone might be interested in using an algorithm based on the greedy approach is that greedy algorithms are often very efficient. Greedy algorithms make decisions based on the current state of the problem, without looking ahead to future states. This can make greedy algorithms very fast, especially for large problems.

- The greedy algorithm is more likely to find a good solution in a reasonable amount of time for both the M-coloring problem and the sum of subset problem. However, the greedy algorithm is not guaranteed to find the optimal solution.
- Greedy algorithms are typically very simple to implement and understand.
- Greedy algorithms are often very efficient, especially for large problem instances.
- Greedy algorithms can be used to find approximate solutions to problems that are difficult or impossible to solve optimally.

- If the choices are made in a suboptimal order, the algorithm may find a suboptimal solution. To avoid this, it is important to carefully consider the order in which the choices are made.

Greedy Algorithm is chosen over backtracking in these cases:

- When the problem is too large for an exhaustive search algorithm.
- When the problem is too complex for an optimal algorithm to be found.
- When the problem is time-sensitive and a good solution is needed quickly.

Algorithm	Problem	Time Complexity	Guaranteed to find a solution?
Backtracking Algorithm	m-coloring problem	$O(m^v)$	YES
Greedy Algorithm	Sum of Subsets	$O(2^n)$	NO
Greedy Algorithm	Color the graph	$O(V^2 + E)$ , in worst case	NO

## Exercise 30

Analyze the Backtracking algorithm for the Hamiltonian Circuits problem (Algorithm 5.6) and show the worst-case complexity using order notation.

Answer:

The Hamiltonian Circuits problem is a decision problem in graph theory. It asks whether a given graph contains a Hamiltonian circuit. A Hamiltonian circuit is a cycle in the graph that visits every vertex exactly once and returns to the starting vertex.

The Hamiltonian Circuits problem is a well-known NP-complete problem. This means that there is no known polynomial-time algorithm for solving the problem. However, there are a number of approximation algorithms and heuristics that can be used to find Hamiltonian circuits in a reasonable amount of time.

Here is an example of a Hamiltonian circuit problem:

A → B → C → D → E → A

This graph contains a Hamiltonian circuit, which is the path A → B → C → D → E → A.

The Hamiltonian Circuits problem can be solved using a variety of algorithms, including:

**Backtracking:** The backtracking algorithm recursively tries all possible paths through the graph.

**Branch and bound:** The branch and bound algorithm recursively explores the space of possible solutions, pruning suboptimal solutions along the way.

**Heuristics:** Heuristics are greedy algorithms that use a heuristic function to guide the search for a Hamiltonian circuit.

The worst-case complexity of the backtracking algorithm for the Hamiltonian Circuits problem is  $O(n!)$ , where  $n$  is the number of vertices in the graph. This is because the backtracking algorithm recursively tries all possible paths through the graph, and there are a total of  $n!$  possible paths.

At each step of the algorithm, the algorithm must check if the current path is a Hamiltonian circuit. This takes  $O(n)$  time, since the algorithm needs to check if all of the vertices in the graph are in the path and if the path starts and ends at the same vertex.

If the current path is not a Hamiltonian circuit, the algorithm must branch and explore all possible extensions of the path. This can lead to  $O(n)$  new paths to explore.

The algorithm terminates when it finds a Hamiltonian circuit or when it has explored all possible paths.

The example of a graph where the backtracking algorithm will take  $O(n!)$  time to complete:

A  $\rightarrow$  B  $\rightarrow$  C  $\rightarrow$  D  $\rightarrow$  E  $\rightarrow$  A

This graph has 5 vertices, so there are a total of  $5! = 120$  possible paths through the graph. The backtracking algorithm will have to explore all of these paths before finding the Hamiltonian circuit A  $\rightarrow$  B  $\rightarrow$  C  $\rightarrow$  D  $\rightarrow$  E  $\rightarrow$  A.

In practice, the backtracking algorithm is often able to find a Hamiltonian circuit much faster than  $O(n!)$  time. This is because the algorithm can prune many of the possible paths without exploring them. However, the worst-case complexity of the algorithm is still  $O(n!)$ .