Big-$\mathcal{O}$ Practice Problem Solution

Emilie Menard Barnard
Foothill College

Problem Statement

You are given a vector of exactly ten random integers between the values of between 1 and 100, inclusive. There are no other data values in the vector. Develop two algorithms with different time complexities that determine if there is a pair of numbers within the vector that sum to 100.

Solutions

There are multiple ways to approach the same problem. I will cover two of the most common algorithms to solve this problem. Feel free to contact me if you developed a different algorithm and want to check its correctness.

Solution 1

This straight-forward algorithm compares each of the ten numbers to the other nine numbers.

Algorithm Outline

For every number x in the vector:
  Take every *other* number in the vector, one at a time, say y:
    And check if $x + y = 100$
      If it does, we have a match!
If we didn't find a match, then we know there are no two numbers within the vector that sum to 100.

C++ Code

In C++, this would look as follows:

```
bool doNumsSum1(vector<int> ourVector) {
     bool twoNumsSumTo100 = false;
     for (int i = 0; i < ourVector.size(); i++) {
          for (int j = 0; j < ourVector.size(); j++) {
               if (ourVector[i] + ourVector[j] == 100 && i != j) {
                    twoNumsSumTo100 = true;
               }
          }
     }
     return twoNumsSumTo100;
}
```

Algorithmic Complexity

This algorithm runs in $\boxed{\mathcal{O}(n^2)}$ time because of the nested loops. Each of the n times the first i loop runs, the second j loop also runs n times, where n is the length of the vector. Thus we have $\mathcal{O}(n \cdot n) = \mathcal{O}(n^2)$.

Solution 2

To find a more efficient solution, we first examine solution 1 to find something we can improve. Let's consider how many pairs of numbers we check in solution 1. For each of the 10 numbers, it and 9 other numbers can make a pair. This is $10 * 9 = 90$ pairs total.

With the vector `vector<int> exampleVector2 = {39, 95, 80, 66, 44, 64, 66, 7, 68, 66};`, for example, we have the following pairs:

39, 95	39, 66	95, 80
39, 80	39, 7	95, 66
39, 66	39, 68	95, 44
39, 44	39, 66	95, 64
39, 64	**95, 39**	etc.

Each pair of numbers will be checked twice. This is unnecessary.

This second algorithm ensures that each pair of numbers is only checked once by sorting the vector first.

Algorithm Outline

First sort the vector.
Then set $a = 0$ and $b = 9$.
As long as a is less than b, do the following:
  Let x be the number at position a in the vector.
  Let y be the number at position b in the vector.
  If $x + y = 100$, we have a match!
  If $x + y > 100$, subtract 1 from b.
  If $x + y < 100$, add 1 to a.
If we didn't find a match, then we know there are no two numbers within the vector that sum to 100.

C++ Code

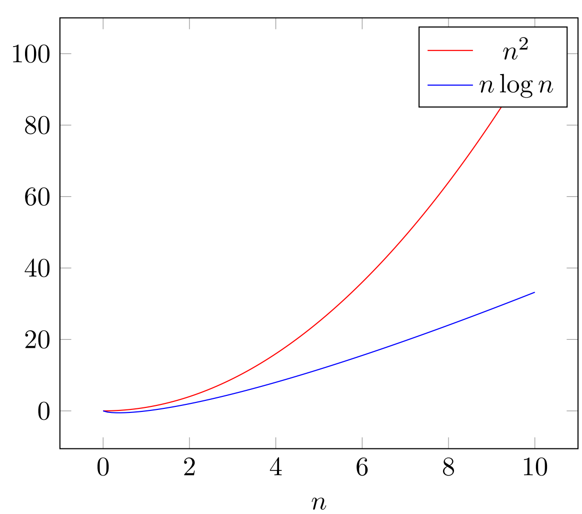
```
bool doNumsSum2(vector<int> ourVector) {
     sort(ourVector.begin(), ourVector.end());
     bool twoNumsSumTo100 = false;
     int leftIndex = 0;
     int rightIndex = 9;
     while (leftIndex < rightIndex) {
          if (ourVector[leftIndex] + ourVector[rightIndex] == 100) {
               twoNumsSumTo100 = true;
               break;
          }
          else if (ourVector[leftIndex] + ourVector[rightIndex] > 100) {
               rightIndex--;
          }
          else {
               leftIndex++;
          }
     }
     return twoNumsSumTo100;
}
```

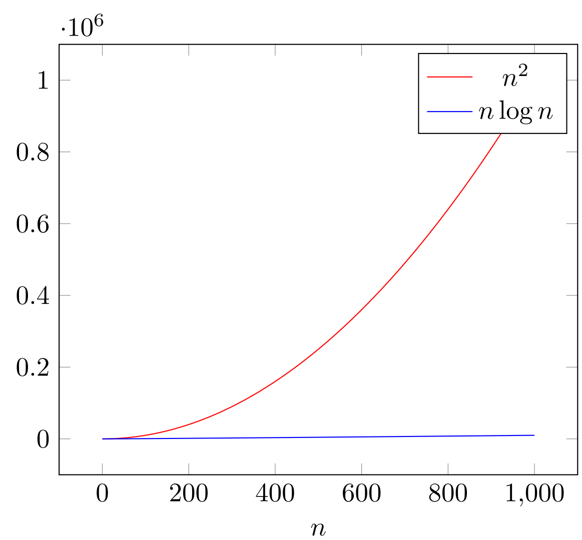
Algorithmic Complexity

Sorting takes $\mathcal{O}(n \log n)$ time, as explained here. After, the vector is sorted, we then perform a loop which iterates n times in the worst-case scenario, where n is the length of the vector. Thus the loop takes $\mathcal{O}(n)$ time. Since these operations are performed one after another and are not nested, we calculate the algorithmic complexity of the algorithm as follows: $\mathcal{O}(n \log n) + \mathcal{O}(n) = \mathcal{O}(n \log n)$, since $\mathcal{O}(n \log n)$ much faster than $\mathcal{O}(n)$. Thus, the algorithmic complexity of solution 2 is $\boxed{\mathcal{O}(n \log n)}$.

Solution 1 vs. Solution 2

Intuitively we can see that solution 2 is superior to solution 1 since it doesn't check as many pairs of numbers. Comparing Big-$\mathcal{O}$ runtimes of the two confirms this hypothesis, as $\mathcal{O}(n^2)$ outgrows $\mathcal{O}(n \log n)$ overtime. This is shown in the charts below:





Solution 3?

There is an even more efficient solution to this problem which I have not covered here. Can you figure it out? ☺