Design and Analysis of Algorithms

Algorithm Comparison

Contents

Algorithmic Comparison

Select a challenging Puzzle (Medium/Hard Level: #51 till #150) from the following book:

> **Algorithmic Puzzles**, *Anany V. Levitin and Maria Levitin*, Oxford University Press.

Algorithm Selection

Go through the solution and comments about your selected puzzle given in the book and design two algorithms using two different design techniques.

1.1 Problem Statement:
Book: Algorithmic Puzzles

Puzzle number: 63

Difficulty Level: Medium

1.2 Puzzle Statement:
Pluses and Minuses: The n consecutive integers from 1 to n are written in a row. Design an algorithm that puts signs “+” and “−” in front of them so that the expression obtained is equal to 0 or, if the task is impossible to do, returns the message “no solution.”

1.3 Basic Idea:
The main idea of the algorithm is to take consecutive numbers from 1 to a given number n and place either + or – in front of each number. These

arithmetic signs will be placed in a way that the sum equals to 0. The numbers from 1 to n can be divided into 2 subsets:

- The subset with + sign
- The subset with – sign

The sum of both the subsets should be equal only then the sum will be 0.

1.4 Greedy Approach:

In the greedy approach first step if to check whether the sum of the consecutive numbers is odd or even. If it is odd, it cannot be split into two equal subsets hence we cannot obtain zero. There is no solution in this case. If it is even, we divide the sum of the numbers by 2 to compute the half (sum/2). The sum of the subsets should be equal to this half. Start by computing the positive subset (that is, the subset that will be assigned + sign). We start with the largest number, check whether it is smaller than or equal to half of the sum and assign it a positive value. In this way we keep filling one subset (positive subset) with numbers so that its sum gets as close as possible to sum/2, the target subset sum. At every step, the algorithm tries to maximize the sum of the positive subset by adding the largest possible number that keeps the subset's sum under Sum/2. It makes the optimal choice hence it is a greedy approach.

The remaining numbers are assigned – sign.

Natural Language:

Input: a number n

Output: array of valid signs

Step 1: Add all the numbers from 1 to n together to find the total sum $S=1+2+\cdots+n \Rightarrow n(n+1)/2$. If S is odd, it’s impossible to split it into two equal parts, so there’s no solution.

Step 2: If S is even, calculate sum/2 this will be the target that the positive subset needs to reach.

Step 3: Start with the largest number n. Add it to the positive subset (assign +) if it keeps the sum under or equal to target. If not assign -.

Step 4: Once the positive subset is equal to target, the remaining numbers are assigned – sign.

Example:

n=7

Calculate S=1+2+3+4+5+6+7=28

Target for each group: S/2=14

Start with the largest number:

Add +7to the positive group. Positive sum = 7.

Move to the next largest number:

Add +6 to the positive group. Positive sum = 13.

Add the next number:

Add +1 to the positive group. Positive sum = 14 (target reached).

Assign remaining numbers to the negative group:

Numbers 2,3,4,5 go to the negative group.

Result:

Positive group: +7, +6,+1.

Negative group: −2,−3,−4,−5

Expression: +7+6+1−2−3−4−5=0

1.5 Brute-force approach:

The Brute-force approach will try all the possible combinations of + and – signs. It will try one combination and compute sum to check if it is equal to 0. It will repeat this for all combinations. Since each number can have 2 combinations, - or + hence the total number of combinations for n numbers will be 2^n.

Natural Language:

Base case: The recursive function processes one number at a time. When we reach the last number (i.e., when we have considered all numbers), we calculate the sum of the numbers with their assigned signs. If the sum equals 0, that means the current combination of "+" and "-" signs is valid.

Step 1: Start from the first number (1) and assign it a "+" sign.

Step 2: After assigning the "+" sign, move to the next number (2) and assign it a "+" sign. Continue this process for all numbers, recursively. Once the last number is reached and all numbers have been assigned "+" signs, the algorithm will compute the sum of the numbers and check if it equals 0. If it does, print this combination as a valid solution.

Step 3: if no valid solution, then backtrack and assign the current index – sign.

Step 4: Once the last number is reached, the base case is reached so the sum will be calculated.

Example:

We begin with the first number 1, and assign it a "+" sign.

The sign combination so far: +1

Now, move to the second number 2, and assign it a "+" sign.

The sign combination: +1 +2.

Now, move to the third number 3, and assign it a "+" sign.

The sign combination: +1 +2 +3.

Calculate the sum: 1+2+3=6 (this is not a valid solution).

Now, backtrack and assign a "-" sign to 3.

The sign combination: +1 +2 -3.

Calculate the sum: 1+2−3=0 (this is a valid solution algorithm will give this output).

Only for explanation purpose I have done further steps

After printing the valid solution, we backtrack to the second number **2**, and assign a "-" sign to 2:

The sign combination: +1 -2.

Now, move to the third number 3, and assign a "+" sign to 3.

The sign combination: +1 -2 +3.

Calculate the sum: 1−2+3=2 (not a valid solution).

Backtrack and assign a "-" sign to 3.

The sign combination: +1 -2 -3.

Calculate the sum: 1−2−3=−4 (not a valid solution).

After finishing with number 2, backtrack to number 1 and assign a "-" sign to 1:

The sign combination: -1.

Now, move to number 2 and assign a "+" sign to 2.

The sign combination: -1 +2.

Move to number 3 and assign a "+" sign to 3.

The sign combination: -1 +2 +3.

Calculate the sum: $-1+2+3=4$ (not a valid solution).

Backtrack and assign a "-" sign to 3.

The sign combination: -1 +2 -3.

Calculate the sum: $-1+2-3=-2$ (not a valid solution).

Backtrack to number 2 and assign a "-" sign to 2:

The sign combination: -1 -2.

Move to number 3 and assign a "+" sign to 3.

The sign combination: -1 -2 +3.

Calculate the sum: $-1-2+3=0$ (valid solution).

Pseudo Code

Represent your algorithms using Pseudo code (use the syntax introduced in the class)

2.1 Pseudo code for Greedy Approach:

Algorithm PlusMinus(n)

Input: n integer

Output: signs array with the valid combination

```
    # Step 1: Calculate the total sum
    totalSum <- (n * (n + 1)) / 2


    # Step 2: Check if the total sum is odd
    if totalSum % 2 != 0:
```

```
    return "No solution"   # Odd sum cannot be divided into two equal
subsets


  # Step 5: Handle cases based on the totalSum
  target <- totalSum / 2   # The target sum for one subset


  currentSum = 0
  for i from n down to 1
    if currentSum + i <= target
      signs <- "+"   # Add this number to the "positive" subset
      currentSum += i
    else
      signs <- "-"   # Add this number to the "negative" subset
  return signs
```

2.2 Pseudo Code for Brute Force:

Algorithm generateCombinations(n, index, signs)

Input: n which represents consecutive number of integers that is if n=3 it means 1,2,3 then index which of sign array initially 0 then the vector array signs [] => initial state

Output : It prints put the valid combination of plus and minus signs for which the array results in 0 that is for n = 3 we will have +1,+2,-3 as our ouput

if index = n:

```
        sum <- 0
        for i from 0 to n-1
            if signs[i] = '-':
                sum <- sum - (i + 1)
            else
                sum <- sum + (i + 1)
        if sum = 0:
            for i from 0 to n-1:
                print signs[i] + (i + 1)
            print "Valid combination"
        return
    signs[index] <-  '+'
    generateCombinations(n, index + 1, signs)
    signs[index] <- '-'
    generateCombinations(n, index + 1, signs)
```

Asymptotic Analysis

Analyze the asymptotic upper, lower and tight bound for the worst-case and best-case Time and space complexity of your algorithms using Theoretical Approach. Represent your analysis using asymptotic notations.

3.1 Anaysis for Greedy:

The for loop runs from n down to 1, which means it iterates n times.

Rest of the operation stake constant time.

Hence the time complexity is O(n).

The best-case complexity still requires the loop to iterate through all numbers, and each iteration performs a constant amount of work (comparing and updating the sum).

Thus, the best-case time complexity is: O(n).

Space complexity: the space complexity of the algorithm remains O(n) as it uses one “signs” array to store n value.

3.2 Analysis for Brute Force:
When the index = n, the algorithm calculates the sum of the signs array. This involves looping through the entire array of size n and performing simple arithmetic operations (addition or subtraction).

This loop takes O(n) time since it iterates over n elements.

The total number of recursive calls is 2^n, and each recursive call involves O(n) work to compute the sum so the worst case time complexity is : O(n·2n)

For best case, the algorithm still needs to go through all 2^n combinations and check the sum so it will be the same.

Space complexity: the space complexity of the algorithm remains O(n) as it uses one “signs” array to store n value.

Comparison Table

Compare both the algorithms using a comparison table.

	Brute Force (generateCombinations Algo)	Greedy (PlusMinus Algorithm)
Approach	Recursive Backtracking: Generates all possible sign combinations recursively.	Greedy Approach: assigns "+" or "-" based on a target sum.
Time Complexity	O(n·2n)	O(n)
Space Complexity	O(n) for signs array.	O(n)for sig ns array.
Efficiency	Less efficient because it explores all possible sign combinations.	More efficient, only tries to find a valid partition based on sum constraints.
N=3	O(3·2^3)=O(3·8)=O(24)	O(3)
N=20	O(20·2^20)=O(20·1,048,576)=O(20,971,520)	O(20)

Conclusion:

Given the analysis, the brute force method becomes impractical for larger values of n due to its O(n·2n) complexity, while the greedy approach remains efficient with O(n).