CSE 331- HW2 Report

Explanation of C++ Code

There is a function as CheckSumPossibility and main in my code, CheckSumPossibility function take three arguments :

Code

▼

```
bool CheckSumPossibility(int num, int arr[], int size)
{
    // cout << num << " " <<size << " ";
  if (num == 0){
        // cout << "true"<<endl;
    return true;
    }
  if (size == 0){
        // cout << "false"<<endl;
    return false;
    }

  if (arr[size - 1] > num){
        // cout << "case:1"<<endl;
    return CheckSumPossibility(num, arr, size - 1);
    }
    // cout << "f1"<<endl;
    bool val = CheckSumPossibility(num, arr, size - 1 );
  if(! val){
    // cout << num << " " <<size << " ";
        // cout << "f2"<<endl;
    return CheckSumPossibility(num - arr[size - 1], arr, size - 1);
    }
    else
        return true;
}
```

Arguments of the CheckSumPossibility :

▼ *num*

- It's type is integer, it holds the value of the target sum value.

▼ arr

- It's type is integer array, it holds the array which is gonna be looked up.

▼ size

- It's type is integer, it holds the value of the non-looked up part of the array.

Return Value: This function returns a boolean value for the case of the desired sum value can be got with using the elements inside the given array. It returns true if the case is possible, returns false in the other case.

Main Structure and Functionality of CheckSumPossibility

There are 4 main if cases in the function. Which are:

```
if (num == 0)
if (size == 0)
if (arr[size - 1] > num)
if(! val)
```

and there are 2 types of recursive function calls

```
CheckSumPossibility(num, arr, size - 1 );
CheckSumPossibility(num - arr[size - 1], arr, size - 1);
```

For the sake of understandability I'm gonna start to explain from these function calls.

First call is used for 2 cases :

- If the last element of the array is bigger than desired sum value.
- If the last element of the array is not bigger than desired sum value.

Wait a minute, isn't it a contradiction then ? Not much because, in the first case if the last element is bigger than the desired value there is no need to make second function call, if we do this we get negative input for next function call, which will be spoil the function's operation.

For the second case, if you think deeply there are two options for a element of the array in this operation:

1. Element can be part of sum
2. Element can't be part of sum

Since we make recursive calls we need to prevent unwanted cases like above but when there is no situation like that we have to make function call for every element. In our case we made function call like the element of the array is not the part of the sum. After that if we didn't get the true output from next recursive calls we make the second function call in the above, which subtracts last element from desired sum. This trick also helps us to make less function calls, if the result case is equal to true it terminates itself. I talk a bit much about functions but since they are more clear now, let's talk about if statements:

1. If function reaches first if case it returns true because since I described above, desired sum gets substracted by the function if the are smaller than sum value. so in this case when desired sum equals 0, it means that we reached desired sum.
2. Function reaches second if cases when, num does not equal to 0 but not-looked up number of the array equals 0. That means all elements of the array are looked up but function didn't reach desired sum. So function returns false in this case.
3. Third if case controls the situation I mentioned above. If the last element of not-looked up part of the array is bigger than desired sum it skips that number and continues with next number.
4. Last case is used for the prevent making more function calls. With using this if statement we stop making recursive calls if the function got desired input in previous function calls.

Explanation of Assembly Code

If analyze the assembly code in simple sub topics:

1. Getting size and desired number input from user
2. Creating the array and getting input for every element from the user
3. Making function call

For the first part I added some terminal input to my program, because in debugging (especially in the first times) there was description and I got a bit confused because this reason and I made mistakes. So added them. In the first part program asks for size input after that is asks for the desired sum value.

```
#print m1
        la $a0 m1
        li $v0 4
        syscall

        #get size of the array
        li $v0 5
        syscall
  move $s0,$v0

  #print m2
        la $a0 m2
        li $v0 4
        syscall

        #get sum
        li $v0, 5
        syscall
        move $s1, $v0
```

For the second part I used sbrk feature of the SPIM System Calls, I shifted the size value 2 times and assigned this value to $a0 then called sbrk. So in the end system gave me an array which is dynamically allocated according to user input.

```
#allocate space
        sll $a0 $s0 2   #number of bytes now in $a0
        li  $v0 9 #sbrk call => (int*)malloc(sizeof(int)*$s0)
```

```
        syscall

        move $s2, $v0 #$s2 = arr
```

Then to assign user inputs to the element of the array, I used a function which I named as loop (not very creative name huh). This function gets an input from user making syscall. Then it assigns the given value into array with sw instruction and increments the pointer of the array, checks that pointer doesn't exceed the end of the array if it is not it calls loop function again, if it is, it stops.

```
#assign values to elements of array
  move $t0,$s2        # $t0 = &arr
  sll $t1,$s0,2       # (size-1)*4
  add $t1,$t0,$t1     # $t1 = &arr[size-1] = arr + (size-1)*4
loop:
        li $v0, 5       #get input from user
        syscall
        sw $v0, ($t0)   # store input to arr+(n)*4
  addi $t0, $t0, 4    # ptr = arr+4
  bne $t0, $t1, loop  # continue if ptr<arr + (size-1)*4
```

In the end we come up to big boss assembly implementation of the CheckSumPossibility. There 8 sub topics we are gonna talk about in here, which are:

1. CheckSumPossibility (Yes the function's itself what a surprise huh?)
2. case_1
3. case_2
4. last_return
5. size_1
6. last
7. true
8. false

In the star of the main I store all of the argument inside stack since there are too many and I made this function in recursive. After that I check the possibilty of end cases which are second from end in the my sub topic list, in these two I get the proper $ra value and resize the stack as it was before function call.

If the case is not these two, function continues the to check the case of if last element is bigger than desired num if it is, function passes through the case_1 if it is not it pases the case 2. In case_1 it calls the CheckSumPossibility with decremented size. In case_2 it makes this call but after that it makes CheckSumPossibility call with desired value such that the last element is subtracted by it and with decremented size value. I use for size_1 for function call it only decrements the size by 1. size_1 assigns true values to the arguments and calls CheckSumPossibility . In last it changes the arguments according to the other function call to explain in few words. As a last detail after calling last it returns to last_return and last_return resizes the stack and jump back to where it belongs.

Test Cases

C++

```
bilalbayrakdar@EKS-L0340216:~/code/331$ ./cpp_part1 c
10
881
12
32
54
56
73
74
232
647
234
24
Possible!
```

```
bilalbayrakdar@EKS-L0340216:~/code/331$ ./cpp_part1 c
8
41
13
57
36
765
234
133
63
65
Not possible!
```

```
bilalbayrakdar@EKS-L0340216:~/code/331$ ./cpp_part1 c
8
41
13
57
36
765
234
133
63
65
Not possible!
```

```
bilalbayrakdar@EKS-L0340216:~/code/331$ ./cpp_part1 c
8
41
13
57
36
765
234
133
63
65
Not possible!
```

```
bilalbayrakdar@EKS-L0340216:~/code/331$ ./cpp_part1 c
9
31
31
42
64
46
92
45
637
24
46
Possible!
```

```
bilalbayrakdar@EKS-L0340216:~/code/331$ ./cpp_part1 c
10
881
12
32
54
56
73
74
232
647
234
24
Possible!
```

Assembly

```
Enter the size: 10
Enter the sum: 881
12
32
54
56
74
73
232
647
234
24
Possible!
```

```
Enter the size: 8
Enter the sum: 41
13
57
36
765
234
133
63
65
Not possible!
```

```
Enter the size: 8
Enter the sum: 524
41
64
868
24
6547
23
0
13
Not possible!
-- program is finished running --
```

```
Enter the size: 9
Enter the sum: 172
41
23
56
9
44
145
63
94
56
Possible!
-- program is finished running --
```

```
Enter the size: 9
Enter the sum: 31
31
42
64
46
92
45
637
24
46
Possible!
-- program is finished running --
```

```
Enter the size: 8
Enter the sum: 15
12
42
64
76
90
48
285
663
Not possible!
-- program is finished running --
```