Hints for Lab 13 Task F

Question: can an int array be divided into two subarrays with equal sum? An application is, given items with different values, can the items be divided among two people, each gets the same total value.

Example: given an array {1, 2, 3}, put 1 and 2 to a group and 3 to another, that is, array {1, 2, 3} can be divided. However, array {1, 2, 6} cannot be divided into two parts with equal sum.

The function header is bool divisible(int* price, int size);

Function divisible cannot call itself directly, that is, when we take one element off (without loss of generality, take the first one out), whether the remaining array is divisible has no relation with the divisibility of the original array. For example, {1, 2, 3} is divisible, but any of its subarrays {2, 3}, {1, 2}, {1, 3} is divisible. As another example, {6, 1, 1} is not divisible, but one of its subarray {1, 1} is divisible.

Idea: Since we want to test whether an array can be divided by equal-sum subarray or not, we will find some simple corner cases. For example, calculate the sum of the array using a recursive function, if the sum is odd, then return false. If the sum is even, how to solve this problem? The question is reduced to find out whether we can find out a subarray whose sum is half of the original total.

In general, we define a function bool subarraySum(int* arr, int size, int taget), for a given array of integers and a target, can we find a subarray whose sum is target? This problem can use recursion. Consider two possibilities:

(1) If the first item in the original array is not included in the chosen subarray, then the remaining array needs to come up a subarray whose sum is target.
(2) If the first item in the original array is included in the chosen subarray, then the remaining array needs to come up a subarray whose sum is target – arr[0].

The base cases are the following:

(1) If target is 0, no more items need to be put in the subarray, return true.
(2) Otherwise, when size is 1, if target equals arr[0], return true, otherwise, return false.

Let us work an example. Given array [2, 4, 6], whose half sum is (2 + 4 + 6) / 2 = 6, our goal is to find out whether there is a subarray in [2, 4, 6] whose total is 6. In fact, either [2, 4] or [6] has a total of 6. When calling subarraySum(int* arr, int size, int taget) **for the purpose of divisible(int *arr, int size)**, set target to be half sum of [2, 4, 6], that is, 6. In general, target can be any integer for subarraySum function.

The following is illustration of idea, not actual code.

call bool subarraySum(int *arr, int size, int target) on array $\boxed{2|4|6}$ to see whether there is a subarray whose total is 6, where 6 is half of the total of the original array $\boxed{2|4|6}$.

$$\text{bool subarraySum}(\underbrace{\boxed{2|4|6}}_{\text{arr}},\ \underbrace{3}_{\text{size of arr}},\ \underbrace{6}_{\text{target}})$$

The answer depends on the following:

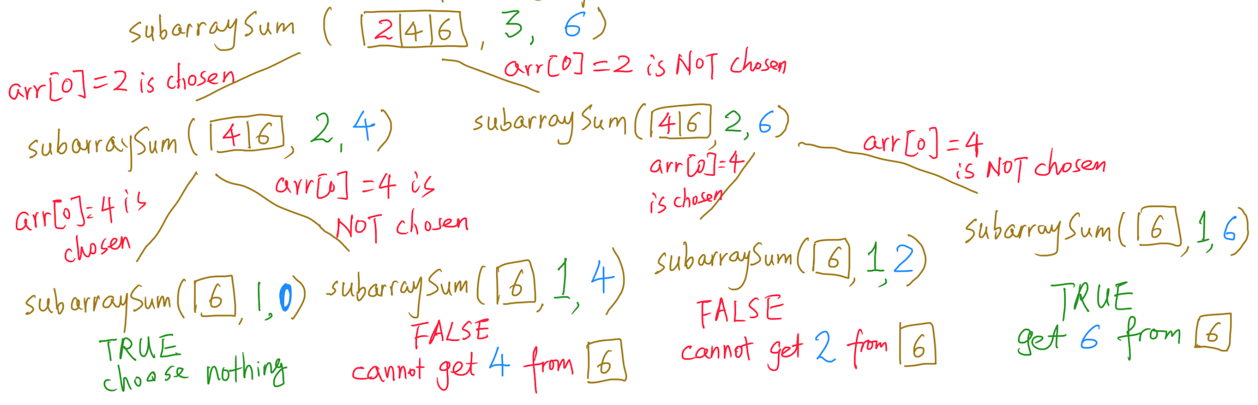
subarraySum ($\boxed{2|4|6}$, 3, 6)

(1) suppose $arr[0]^{=2}$ is chosen, can we find a subarray in the remaining array $\boxed{4|6}$ whose total is $6 - arr[0] = 6-2=4$?

That is, call subarraySum ($\boxed{4|6}$, $\underbrace{2}_{\text{size of subarray } \boxed{4|6}}$, 4).

(2) Suppose $arr[0]^{=2}$ is NOT chosen, can we find a subarray in the remaining array $\boxed{4|6}$ whose total is 6 ? That is, call subarraySum ($\boxed{4|6}$, 2, 6)

In short, we have the following figure



Here is an idea of bool subarraySum(int *arr, int size, int target)

```
bool subarraySum(int *arr, int size, int target)
{
     //Base cases: in what situations can we make conclusion?

    //When size > 1,
    //how to relate with cases when size -1?
}
```

In Lab 13 F, you may need to use sumArray(int *arr, int size) in Lab 13 C.

Lab 13 C, write a recursive function int sumArray(int *arr, int size)
arr is pointer to the first element arr[0] in array arr
arr + 1 is the pointer to arr[1],
arr + 2 is the pointer to arr[2],
...
In class time, I mistakenly wrote arr + sizeof(int), thinking it points to arr[1], I was wrong. Lesson learned: C++ compiler is smart, normally, given an array arr, then arr + i is the pointer pointing to arr[i], where 0 <= i < size of arr.