Assignment 7

Algorithms, Spring 2024

Honor code: *Work on this assignment alone or with one partner (highly encouraged). Partner policy: You and your partner will work together on the assignment throughout the whole process, you will write it and review it together, and will submit one assignment. You can talk to anyone in the class (collaboration level 1). It is not allowed to search online for the specific problems in this assignment—doing so violates academic honesty for the class.*

Load balancing: You have been hired to design algorithms for optimizing system performance. Your input is an array $A[]$ where $A[i]$ represents the running time of job i; jobs do not have specific start and end times, but they can be started at any time[1]. The running times are integers. The problem is to determine whether there is a subset S of A such that the elements in S sum up precisely to the same amount as the sum of the elements not in S.

Examples:

```
a = []
True, subset=[]
a = [1]
False
a = [1,2,1]
True, subset=[1,1]
a=[1, 2, 3, 5]
False
a = [1, 5, 11, 5]
True, subset=[1, 5, 5]
```

You'll solve this problem in three steps:

1. **Recursive:** Come up with a recursive formulation for this problem (without dynamic programming). To do this you'll want to write a wrapper function, called *haseqs*, which has one parameter, the array of values *vals*, and returns True if *vals* can be partitioned into two subsets such that their sums are equal, and false otherwise.

   ```
   def haseqs(vals):
       """PARAMETER vals: an array of values, all positive integers
   ```

[1]This is a different scenario than in the interval scheduling /activity selection problem

```
    RETURN: true if vals can be partitioned into two subsets such that the
    sum of the elements in both subsets is equal. False otherwise.
    """
    return subset_sum(vals, ...)
```

This wrapper function will call your recursive function (called *subset_sum* above, but feel free to give it any name you like). This recursive function is where the work to answer the question will happen. It's up to you how you define the parameters for this function (you'll probably want to have it return a boolean).

Once both functions are implemented, you should be able to test various arrays like so:

```
vals1=[1,5,11, 5]
vals2=[1, 2, 3, 5]
vals3=[1, 1, 1, 1]
haseqs(vals1)
True
haseqs(vals2)
False
haseqs(vals3)
True
```

2. **Dynamic programming:** Turn your recursive solution from (1) into a DP solution with a table. Again, you will start by writing a wrapper function called *haseqs_dp*, which has one parameter, the array of values *vals*, and returns true if *vals* can be partitioned into two subsets such that their sums are equal, and false otherwise.

```
def haseqs_dp(vals):
  """PARAMETER vals: an array of values, all positive integers

    RETURN: true if vals can be partitioned into two subsets such that the
    sum of the elements in both subsets is equal. False otherwise.
    """
    return subset_sum_dp(vals, ...)
```

This wrapper function will call your function (called *subset_sum_dp* above, but feel free to give it any name you like). This function is where the work to answer the question will happen. It's up to you how you define the parameters for this function, and whether you use recursive or iterative dynamic programming.

Once both functions are implemented, you should be able to test various arrays like so:

```
vals1=[1,5,11, 5]
vals2=[1, 2, 3, 5]
vals3=[1, 1, 1, 1]
haseqs_dp(vals1)
True
haseqs_dp(vals2)
False
```

Specifically you will be able to test largest arrays because it is efficient (unlike your function *haseqs* which is exponential and will be too slow).

3. **Full solution:** Extend your DP solution from (2) to find and print the actual subset. Write a function which we'll call *haseqs_full*, which has one parameter, the array of values *vals*. Its return value is the following: if *vals* can be partitioned into two subsets of equal sum, it returns one of these subsets (does not matter which one); otherwise, if *vals* cannot be partitioned into two such subsets, it returns an empty array []. Once implemented, you could test it like so:

```
vals1=[1,5,11,5]
haseqs_full(vals1)
[1, 5, 5]
vals2=[1,2]
haseqs_full(vals2)
[]
```

To submit: use Gradescope. Your file shoud be called `assignment7.py`. Note that the autograder expects a file with precisely this name, so if you submit a file with a different name it will not see it. The file should contain the following three functions (and possibly more): `haseqs, haseqs_dp, haseqs_full`, each one taking an array of values as parameter. The first two return a booolean, the third one a set. See the specs above. The autograder will run these functions on a bunch of test cases (which are invisible to you) and you will see the score. You can submit as many times as you want. The maximum score is 30.