Homework 6: Subsets
Due: 7:00 PM, Oct 18, 2019

Contents

1 Bookends (Practice) 2

2 Subsets (13 Points) 3

3 Subset Sum (21 Points) 4

4 k-Subsets (21 Points) 5

5 k-Subset Sum (24 Points) 6

Objectives

By the end of this homework you will be able to:

1. write recursive combinatorial procedures
2. write recursive search procedures

How to Hand In

For this (and all) homework assignments, you should hand in answers for all the non-practice questions. For this homework specifically, this entails answering the Subsets, Subset Sum, k-Subsets, and k-Subset Sum questions.

In order to hand in your solutions to these problems, they must be stored in appropriately-named files. In particular, each should be named for the corresponding problem, as follows (e.g., subsets.rkt corresponds to Subsets):

- README.txt
- subsets.rkt
- subset-sum.rkt
- ksubsets.rkt
- ksubset-sum.rkt

For this assignment, all files you turn in that contain code must be Racket files, so they must end with extension .rkt
For this and every future assignment, you should also have a README.txt file whose first line contains only your Banner ID, and optionally with a message to the person grading explaining anything peculiar about the handin. For example:

README.txt:
B01234567
There’s nothing to say except that I’m turning in some files plus this README the way the instructions say that I should.

To hand in your solutions to these problems, you must upload them to Gradescope. Do not zip or compress them. If you re-submit your homework, you must re-submit all files.

Practice

1 Bookends (Practice)

Task: Write the procedure `bookends`, which consumes a list of data, `alod`, and produces a sublist of `alod` of length at least 2 whose first and last elements are equal to each other. If no such sublist exists, `bookends` produces the empty list.

Examples:

```
(bookends (list 1 2 -3 4 2 0))
=> (list 2 -3 4 2)

(bookends (list "a" "b" "b" "c"))
=> (list "b" "b")

(bookends (list true))
=> empty
```

Problems

Data Definition

A set $X$ is a collection of objects in no particular order, and with no repetition. A subset of the set $X$ is a set containing only elements of $X$, but not necessarily all of its elements. Note that a subset might contain nothing at all.

For example, $\{1,2,3\}$ is a set, and its subsets are:

$$\emptyset, \{1\}, \{2\}, \{3\}, \{1,2\}, \{1,3\}, \{2,3\}, \{1,2,3\}$$

In this homework, you’ll use lists to represent sets. The set $\{1,2,3\}$ can be represented by the list (list 1 2 3) or the list (list 3 1 2), or any of the other 4 combinations. The set of all subsets of $\{1,2,3\}$ can be represented by a list of lists as follows:
(define subsets-123
  (list empty
       (list 1)
       (list 2)
       (list 3)
       (list 1 2)
       (list 2 3)
       (list 1 3)
       (list 1 2 3)))

Remember, sets don’t necessarily have to be of integers! For instance, the set of all subsets of
(list "a" "b" "c") can be represented in a similar manner as above:

(define subsets-abc
  (list empty
       (list "a")
       (list "b")
       (list "c")
       (list "a" "b")
       (list "a" "c")
       (list "b" "c")
       (list "a" "b" "c")))

Don’t forget to include the proper data definitions in your design recipe for each problem!

Note: These problems are in many ways different from those we have done in CS17 so far. The
most similar problems are from Lab 5 (search). You may find it useful to refer to your solutions to
this lab. At the same time, Subsets questions are different - so do not try to “copy and edit” your
lab solutions.

Note: Remember that the empty list is a proper subset of every set and should be included in the
list of subsets that you generate.

2 Subsets (13 Points)

Task: Write a procedure, subsets, which consumes a set, set, represented as a list, and produces
the set of all subsets of set.

Note: One recursion diagram will suffice for the design recipe for all of these problems, although
writing many more on your own will help guide you to a solution in code. Please use an Original
Input that has a recursive call (i.e. do not use empty as your Original Input).

Note: The order of the subsets you produce need not match the order in our examples. Because of
this, the way to write a check-expect is by comparing your expected output to your actual output
using set-equal (you do not need a design recipe for set-equal!). You may use the set-equal
code from class by copy-pasting it at the top of your code. This code has been sent out in an email
and is posted on Piazza.
Note: All of the sets we’re going to be talking about are going to be ones containing atomic data (bool, num, string). We will not be using lists of functions or lists of lists because things get a little funky when we try to do comparison testing with these.

Testing Example:
(check-expect (set-equal (subsets (list 1))(list (list 1)empty))true)
This way, both outputs of (list (list 1)empty) and (list empty (list 1)) are valid.

Examples:

(subsets (list -3 4 -5))
=> (list empty
 (list -3)
 (list 4)
 (list -5)
 (list -3 4)
 (list 4 -5)
 (list -3 -5)
 (list -3 4 -5))

(subsets (list 1 2))
=> (list empty
 (list 1)
 (list 2)
 (list 1 2))

(subsets (list 1))
=> (list empty
 (list 1))

(subsets empty)
=> (list empty)

Just for Fun: Try out your procedure on sets of length 0 through 10.

3 Subset Sum (21 Points)

Task: Write a procedure, subset-sum, which consumes a set of integers, weights, and a target weight, target, and produces true if there exists a subset of weights whose elements sum to target, and false otherwise. One recursion diagram will suffice.

Note: Each weight and the target can be positive, negative, or zero. Furthermore, the empty set has weight 0.

Hint: What should this procedure call return?

(subset-sum (list 1 2 3 4) 0)

Examples:

(subset-sum (list 0 2 -3 4) 1)
=> true
(subset-sum (list 1 2 3 4) -17)
=> false

(subset-sum (list 1 17 3 4) 21)
=> true

Note: There are many possible combinations of inputs - make sure to test extensively. You should test all of the edge cases, including testing the empty list, a one element list, and a multi-element list, all with varying target values. In your check-expects you should be testing at least 6 distinct cases.

Hint: One solution to this problem is the following:

(define (slow-subset-sum weights target)
  (member? target (map (lambda (alon) (foldr + 0 alon)) (subsets weights))))

This solution first computes all the subsets of weights, then sums each subset, and finally checks to see whether target is equal to any of the resulting sums. The disadvantage of this solution is that it wastes a whole lot of space computing all subsets, before checking to see whether any of them sum to target. There is a “better” solution, one that does not use subsets as a helper, and does not require as much space\(^1\) as slow-subset-sum. The idea underlying the better algorithm is similar to that used to compute subsets: consider whether each weight in weights is part of a subset that sums to target, or not.

Just for Fun: Try out your procedure on sets of size 10, 11, 12, and 20. Estimate how long it would take on a list of size 40.

4  \(k\)-Subsets (21 Points)

Task: Write a procedure, \(k\)-subsets, which consumes a set, set, and a natural number, \(k\), and produces a list of the subsets of set that are of size \(k\). One recursion diagram will suffice. Be sure to test extensively.

Note: As in subsets, the order of the subsets you produce need not match the order in our examples. Note also that \(k\) is a natural number; as such, it cannot be negative, but it can be very large.

Examples:

(k-subsets (list 1 2) 0)
=> (list empty)

\(^1\)We’ve only used operation-counting as a way to measure performance so far, but the amount of memory a procedure uses also matters in some cases. For our purposes, we can measure this as a combination of two things: (i) how deep our nested function-calling goes (do you have to recur 10 times or 100 times?), and (ii) The maximum, over the execution of a program, of the sum of the lengths of all lists in the program at any moment. When you compute all subsets of an \(n\)-element list, you’ve got a total of \(2^n\) subsets whose average length is about \(\frac{n}{2}\), so your “memory use” looks like \(\frac{n}{2} 2^n\). By contrast, if you compute and evaluate each subset and then move on to the next, you never have a list with more than \(n\) elements. Other approaches can have similar impact on space-use.
In the next problem, we refer to these subsets as \(k\)-element subsets.

**Hint:** One way to solve this problem is to generate all the subsets of \(set\), and then select those of size \(k\). This is not a good solution; for small \(k\), it wastes a large amount of time and space, as the majority of the subsets of \(set\) do not have \(k\) elements.

**More Hints:**

- If \(k\) is zero, then \(set\) has only one subset of size \(k\)—the empty set.
- Otherwise, if \(set\) is empty (and \(k\) is not equal to 0), then \(set\) has no subsets of size \(k\).
- Otherwise (i.e., if \(set\) is not empty and \(k\) is not equal to 0), there are at least two ways to construct subsets of \(set\) of size \(k\). We leave it to you to discover exactly how.

Once again, your solution should *not* use \(subsets\) as a helper.

5  

\(k\)-Subset Sum (24 Points)

**Task:** Write a procedure, \(k\)-subset-sum, which consumes a set of integers, \(weights\), a natural number, \(k\), and a target weight, \(target\), and produces true if there exists a \(k\)-element subset of \(weights\) whose elements sum to \(target\), and false otherwise. One recursion diagram will suffice. Be sure to test extensively.

**Note:** Once again, as in \(subset\)-sum, each weight and the target weight can be positive, negative, or zero. Also, as in \(k\)-subsets, \(k\) cannot be negative.

**Examples:**

\[(k\text{-subset-sum} \ (\text{list} \ 1 \ 2 \ 3 \ 4) \ 2 \ 8)\]
\[\Rightarrow \text{false}\]
(k-subset-sum (list 1 2 3 4) 3 8) => true

(k-subset-sum (list 1 2 3 4) 4 11) => false

(k-subset-sum (list -1 2 -3 4) 2 1) => true

(k-subset-sum (list -1 2 -3 4) 2 -4) => true

**Hint:** As above, your solution should *not* use subsets, or any of the other procedures you have already written, as a helper. Instead, try combining ideas from your k-subsets and subset-sum procedures.

Please let us know if you find any mistakes, inconsistencies, or confusing language in this or any other CS 17 document by filling out the anonymous feedback form: [http://cs.brown.edu/courses/csci0170/feedback](http://cs.brown.edu/courses/csci0170/feedback).