
Note that the problem is to generate \( m \)-element multisets that are subsets of a multiset \( S \) having \( n \) elements. To be concrete, you may assume that a multiset is represented as an (unordered) list of integers. Give your algorithm in pseudocode. Then, implement the algorithm in a programming language of your choice. Test your implementation on input

\[
\begin{align*}
m &= 3 \\
S &= \{4, 9, 16, 1, 4, 7\}.
\end{align*}
\]

Include the source of your implementation in your solution document. Also, include the solution you get for the test input.

A \( O \) bound on the worst-case time complexity of your algorithm is a function of \( n \) and \( m \). This is difficult to determine, so you are not required to do so. However, feel free to give it a try.

This problem is much like that of problem 4.3, except for the input and output of multisets instead of sets. In particular, we have to avoid duplicate submultisets in the output; duplicates can be eliminate from a list by sorting the list and comparing consecutive entries for equality. The worst-case time complexity will occur for input \( S \) that is a set, since there will be no duplicates and there will be \( \binom{n}{m} \) submultisets (subsets) in the output.

The problem is solved rather easily with recursion. See Figure 1. The first thing we do is assume that input \( S \) is sorted; this simplifies things later. The base cases for the recursion are straightforward. In the general case, break \( S \) into its first element \( f \) and the list \( r \) that is the rest of the elements. Two recursive calls get \( m - 1 \) element submultisets from \( r \) that can be joined with \( f \) to get \( m \) element submultisets, plus \( m \) element submultisets from \( r \) that do not contain \( f \) (unless \( f \) is duplicated in \( r \)). Putting all these submultisets together gives a list \( C \) of submultisets that may contain duplicates. These are eliminated before the list is returned. A Python 3 program that implements the pseudocode with more details is found in Figures 2 and 3. The output for the test input is found in Figure 4. Note that the sorting in \texttt{eliminate_duplicates} is not really necessary; you can check that the fact that \( S \) is initially sorted means that everything stays sorted throughout.

The worst-case time complexity is a function \( T(n, m) \). Without loss of generality, we can assume that \( n \geq m \geq 1 \). We use the bound \( \binom{n}{m} \leq n^m \), which is a fairly good bound when \( m \) is small but is fairly bad when \( m \) approaches \( n \). Still, there is a constant \( C > 0 \) such that we may write this inequality:

\[
T(n, m) \leq \begin{cases} 
Cn & \text{if } m = 1; \\
T(n, m - 1) + T(n - 1, m) + Cn^m & \text{if } m \geq 2.
\end{cases}
\]
The $n^m m$ term is meant to bound the number of submultisets $\binom{n}{m}$ times the size $m$ of each submultiset.

We need to show an upper bound on $T(n, m)$ that does not involve a recurrence, so we will show this crude upper bound

$$T(n, m) \leq C n^{m+2} m$$

by induction on $n$ and $m$. For the base cases, we have $m = 1$ and $n \geq m$, in which case we have

$$T(n, 1) \leq C n \leq C n^{1+2} \cdot 1$$

as needed.

For the general case, we need to show the upper bound for $n \geq m \geq 2$, and we assume that the upper bound holds for $n' \geq m' \geq 1$ whenever $n' \leq n$ or $m' \leq m$. By the recurrence, we know that

$$T(n, m) \leq T(n, m - 1) + T(n - 1, m) + C n^m m.$$

We bound each term separately, as follows:

$$T(n, m - 1) \leq C n^{m+1} m$$

$$T(n - 1, m) \leq C (\frac{n - 1}{n})^{m+2} m^{m+2} m = C (\frac{n - 1}{n})^{m+2} n^{m+2} m$$

$$C n^m m = C (\frac{1}{n^2}) n^{m+2} m$$

These bounds yield

$$T(n, m) \leq C (\frac{1}{n}) n^{m+2} m + C (\frac{n - 1}{n})^{m+2} n^{m+2} m + C (\frac{1}{n^2}) n^{m+2} m,$$

which we need to bound by $C n^{m+2} m$. Factoring out the common terms, we need

$$\frac{1}{n} + \frac{1}{n^2} + (\frac{n - 1}{n})^{m+2} \leq 1$$

for $n \geq m \geq 2$. If we fix $n$ in the inequality, we note that the left-hand side decreases as $m$ increases. So, if we can show the inequality for $m = 2$, then it will be true for all $n \geq m \geq 2$. Fix $m = 2$ and $n \geq m$. The inequality can be manipulated as follows:

$$\frac{1}{n} + \frac{1}{n^2} + (\frac{n - 1}{n})^4 \leq 1$$

$$n^3 + n^2 + (n - 1)^4 \leq n^4$$

$$n^3 + n^2 + n^4 - 4n^3 + 6n^2 - 4n + 1 \leq n^4$$

$$-3n^3 + 7n^2 - 4n + 1 \leq 0.$$
Submultisets($S,m$)

1. $S$ is a list of integers, representing a multiset; must be sorted
2. $m$ is the cardinality of the submultisets to return

3. Base cases
   4. if $|S| < m$ or $m <= 0$
   5. then return $[]$  \(\triangleright\) empty list represents empty set
6. if $m == 1$
7. then return non-redundant list of singleton lists from elements of $S$

8. General case
9. $f = S[1]$  \(\triangleright\) First of list
10. $r = S[2..|S|]$  \(\triangleright\) Rest of list
11. $A = \text{Submultisets}(r, m - 1)$
12. $B = \text{Submultisets}(r, m)$
13. $C = []$  \(\triangleright\) Accumulate all submultisets in a new list
14. for $a$ in $A$
15. do Append $[f] + a$ to $C$
16. Extend $C$ by $B$
17. return non-redundant list from $C$

Figure 1: Pseudocode for the submultisets algorithm.

It is easy to check that the inequality is true for $n = 2$. Then, we just need to show that $-3n^3 + 7n^2 - 4n + 1$ is decreasing for $n \geq 2$. Take the first two derivatives of the polynomial:

\[
\frac{d(-3n^3 + 7n^2 - 4n + 1)}{dn} = -9n^2 + 14n - 4 \\
\frac{d(-9n^2 + 14n - 4)}{dn} = -18n + 14.
\]

The first derivative $-9n^2 + 14n - 4$ plots as an upside-down parabola; we want to show that it is negative for $n \geq 2$, so we need its derivative $-18n + 14$ to be zero before 2. It is zero when $n = 14/18 < 2$, so we have shown that the inequality

\[
\frac{1}{n} + \frac{1}{n^2} + \left(\frac{n - 1}{n}\right)^{m+2} \leq 1
\]

holds for $n \geq m \geq 2$. By the principle of mathematical induction, the desired inequality is true for all $n \geq m \geq 1$. Moreover, $T(n,m)$ is $O(n^{m+2}m)$. 

---

---
#!/usr/bin/env python3
# Copyright (c) 2015 Lenwood S. Heath. All rights reserved.
# This program or module is free software: you can redistribute it and/or
# modify it under the terms of the GNU General Public License as published
# by the Free Software Foundation, either version 3 of the License, or
# (at your option) any later version. It is provided for educational
# purposes and is distributed in the hope that it will be useful, but
# WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
# General Public License for more details.

"""
This program reads a number m and a list of integers from standard input.
It outputs to standard output all the submultisets of the list of size m.
"""

import sys

m = int(sys.stdin.readline().rstrip()) # Retrieve m
if m < 1:
    raise ValueError("m is not at least 1")

S = [] # Start with the empty multiset
for line in sys.stdin:
    S.append(int(line.rstrip())) # Add next number to multiset
S.sort() # Need S to be sorted

Figure 2: Python program for the submultisets algorithm (Part 1).
# Recursive function to find submultisets; S_prime must be sorted

```python
def submultisets(S_prime, number):
    if len(S_prime) < number or number <= 0:
        return []  # No submultisets of size number
    if number == 1:
        # Bottom of recursion; singleton
        singletons = []
        for elements in S_prime:
            singletons.append([elements])
        return eliminate_duplicates(singletons)
    first = S_prime[0]  # First of multiset
    rest = S_prime[1:]  # Rest of multiset
    smaller = submultisets(rest, number-1)  # Look for smaller submultisets
    larger = submultisets(rest, number)  # Look for submultiset on a smaller multiset
    list_of_submultisets = []
    for a_submultiset in smaller:
        temp_list = [first]
        temp_list.extend(a_submultiset)
        list_of_submultisets.append(temp_list)
    list_of_submultisets.extend(larger)  # Include larger submultisets
    return eliminate_duplicates(list_of_submultisets)
```

```python
def eliminate_duplicates(list_of_lists):
    list_of_lists.sort()  # Sorting makes finding duplicates easy
    trimmed_list_of_lists = []
    prior_list = None
    for a_list in list_of_lists:
        if prior_list is not None and a_list == prior_list:
            continue
        trimmed_list_of_lists.append(a_list)
        prior_list = a_list
    return trimmed_list_of_lists
```

# Top level call to recursive function

```python
all_submultisets = submultisets(S, m)
```

```python
print("All submultisets of size {0} are: ".format(m))
for a_submultiset in all_submultisets:
    print(a_submultiset)
```

Figure 3: Python program for the submultisets algorithm (Part 2).
All submultisets of size 3 are:
[1, 4, 4]
[1, 4, 7]
[1, 4, 9]
[1, 4, 16]
[1, 7, 9]
[1, 7, 16]
[1, 9, 16]
[4, 4, 7]
[4, 4, 9]
[4, 4, 16]
[4, 7, 9]
[4, 7, 16]
[4, 9, 16]
[7, 9, 16]

Figure 4: Output submultisets for $m = 3$ and $S = \{4, 9, 16, 1, 4, 7\}$. 
FirstMatch($T, s$)
1 \[ T = t_1t_2 \cdots t_n \]
2 \[ s = p_1p_2 \cdots p_m \]
3 for $i \leftarrow 1$ to $n - m + 1$
4 \[ \text{match} \leftarrow \text{True} \]
5 for $j \leftarrow 1$ to $m$
6 if $t_{i+j-1} \neq p_j$
7 \[ \text{then} \text{match} \leftarrow \text{False} \]
8 \[ \text{break} \quad \triangleright \text{break from inner for loop} \]
9 if match
10 \[ \text{return} \text{“Match at position } i \text{”} \]
11 \[ \text{return} \text{“No match”} \]

Figure 5: Pseudocode for the first text match algorithm.

Give pseudocode for your algorithm, along with an English explanation of how it works.
Determine a $O$ bound on its worst-case time complexity as a function of the lengths of $T$ and $s$. (Use $|T|$ and $|s|$ for these lengths.)

The exhaustive search approach steps through every possible position $i$ in $T$ where there could be a match and checks for a copy of $s$ beginning at position $i$. At the first match, it returns $i$; if no match is found, it returns that fact. See Figure 5.

The worst-case occurs when no match is found and the break statement is taken on the comparison to $p_m$. This can happen, for example, if $T = a^n = aa \cdots a$ and $s = a^{m-1}b = aa \cdots ab$. Each execution of the inner loop then takes $O(m)$ time. Since the outer loop is executed $O(n - m + 1)$ times, the asymptotic worst-case time complexity is $O(m(n - m + 1))$ or $O(|s|(|T| - |s| + 1))$. Indeed, the time complexity is $\Theta(|s|(|T| - |s| + 1))$, since the worst-case takes that amount of time.

Give pseudocode for your algorithm, along with an English explanation of how it works.
Determine a $O$ bound on its worst-case time complexity as a function of the lengths of $T$ and $s$ and of $k$.

The algorithm here is much like the previous algorithm for an exact text match. See Figure 6. Instead of breaking on a mismatch, we add one to the computed Hamming distance. After the inner for loop, we just check whether the Hamming distance computed is less than $k$ and report it if it is.
The time complexity analysis is the same as before. The worst-case time complexity happens, for example, when $T = a^n$, $s = a^{m-1}b$, and $k = 0$. The resulting time complexity is $\Theta(|s|(|T| - |s| + 1))$, as before.