104291-Combinatorics-Algorithm-PA1-Q1

Contributors

`Zixuan Guan. guan09236@gtiit.edu.cn` who writes the codes, report and README.

`Houjian Qin. qin09376@gtiit.edu.cn` who gives constructive suggestions.

Algorithm Details

Algorithm for Generating and Sorting Subsets

Pseudo-code

Algorithm 1: Algorithm for generating and sorting subsets of an n-set using binary representation and sorting based on size and lexicographical order

Input: An integer n representing the size of the set
Output: A list of all subsets of the n-set $\{1, 2, \ldots, n\}$, sorted by size and lexicographically

```
Function Compare(a, b):
    Input: Two subsets a and b represented as arrays of integers
    Output: Integer value -1, 0, or 1 based on the comparison result
    sizeA ← the number of elements in a (a[0] contains size)
    sizeB ← the number of elements in b (b[0] contains size)
    if sizeA ≠ sizeB then
        return sizeA - sizeB
        // Sort by size
    else
        for index ← 1 to sizeA do
            if a[index] ≠ b[index] then
                return a[index] - b[index]
                // Sort lexicographically within the same size
            end
        end
        return 0
        // They are identical
    end
end
Function GenerateSubsets(n, subsets, pow_set_size):
    pow_set_size ← 2^n
    // Calculate the power set size
    subsets ← allocate memory for pow_set_size subsets
    index ← 0
    for i ← 0 to pow_set_size - 1 do
        subset ← empty set
        for j ← 0 to n − 1 do
            if i & (1 ≪ j) ≠ 0 then
                subset ← subset ∪ {j + 1}
            end
        end
        subsets[index] ← subset
        index ← index + 1
    end
    // Sort all subsets using the Compare function
    qsort(subsets, pow_set_size, sizeof(int*), Compare)
end
```

Natural Language Description of the Algorithm

Algorithm Overview: The algorithm serves to generate all possible subsets of a given set of integers and orders them in a structured way. This is crucial for applications that require detailed analysis of combinations such as optimization problems, probabilistic studies, and simulations.

Step-by-Step Process:

1. **Initialization and Setup:**
   - The function begins by calculating the total number of possible subsets, which is `2^n` for a set with `n` elements. This is because each element can independently be either in or out of a subset, leading to `2` choices per element.
   - An array (or a similar data structure) is prepared to hold these subsets.
2. **Generating Subsets Using Binary Representation:**
   - Each subset is represented by an integer from `0` to `2^n - 1`. The binary form of each integer directly maps to the presence or absence of elements in the subset.
   - For each integer, the algorithm iterates through its bits. The presence of a bit (i.e., if a bit is `1`) indicates the inclusion of the corresponding element from the set in the subset.
   - **Example:** For `n = 3`, the number `5` is represented as `101` in binary. This represents the subset `{1, 3}`, as the first and third elements of the set `{1, 2, 3}` are included in the subset.
3. **Efficient Subset Construction:**
   - The construction of subsets is efficiently handled through bitwise operations, which are known for their low computational overhead. The algorithm uses shifts and bitwise AND operations to decide which elements are included in each subset.
4. **Sorting of Subsets:**
   - **Primary Sorting by Size:** Once all subsets are generated, they are first sorted by their size. This is a natural way to organize data as it groups subsets by their complexity or extent, which can be crucial for analyzing properties that depend on subset size.
   - **Secondary Sorting by Lexicographical Order:** Within subsets of the same size, the next level of sorting is lexicographical. This involves comparing elements of each subset similarly to how words are sorted in a dictionary. This method ensures that the data is not only structured by size but also by the natural order of the elements within.
5. **Role of the Comparison Function:**
   - The comparison function is pivotal for sorting as it dictates the rules based on which subsets are ordered. It first checks subset sizes; if they match, it proceeds to compare the subsets element by element.
   - This dual-criterion comparison ensures that subsets are presented in a uniform and predictable order, making downstream processing more straightforward and efficient.
6. **Memory Management:**
   - Dynamic memory allocation is utilized to handle the storage needs of subsets, which vary depending on the number of elements `n`. Efficient memory management is critical to accommodate the exponential growth of the power set size without wasting resources.
   - Proper handling and cleanup of allocated memory prevent memory leaks, which can be a concern in programs dealing with such potentially large data structures.
7. **Practical Implications and Applications:**
   - **Combinatorial Analysis:** This algorithm is essential for exhaustive combinatorial analyses where all possible combinations of a set need to be considered.
   - **Machine Learning and Data Analysis:** In scenarios where feature selection or the exploration of variable combinations is necessary, the algorithm can systematically generate and sort all potential feature sets.
   - **Optimization and Decision Making:** In optimization tasks, especially those involving constraints that can be represented as subsets, having a structured way to enumerate and sort these subsets is invaluable.

Explanation

Subset Generation Explanation The process of generating subsets of a set with `n` elements can be elegantly handled using binary representation. Each element of the set corresponds to a bit in a binary number. There are `2^n` possible combinations of these bits, where `n` is the number of elements in the set. Each combination represents a unique subset, derived directly from the binary representation of the numbers from 0 to `2^n - 1`.

- **Binary Mapping:** Consider a set with elements `{1, 2, 3}`. The binary numbers ranging from `000` to `111` in binary (which equals 0 to 7 in decimal) represent all possible subsets:
  - `000` → `{}` (no elements selected)
  - `001` → `{3}` (only the third element is selected)
  - `010` → `{2}` (only the second element is selected)
  - `100` → `{1}` (only the first element is selected)
  - And so on up to `111` → `{1, 2, 3}` (all elements selected)
- **Subset Construction:** Each bit position (from right to left, or least significant to most significant) in a binary number corresponds to the respective position of an element in the set. If a bit is `1`, the corresponding element is included in the subset; if `0`, it is excluded. This direct mapping from bits to elements simplifies the generation process and ensures that all possible subsets are considered without repetition.
- **Efficiency:** Using bitwise operations to generate subsets is computationally efficient. The number of operations required is directly proportional to `2^n`, making it feasible to handle reasonably large sets. Additionally, bitwise operations are among the fastest operations in modern computing, providing a significant speed advantage.

Sorting Explanation Once all subsets are generated, they must be sorted. This is done in two major steps: first by the size of the subset, and then by lexicographical order within subsets of the same size. This sorting method is chosen for its logical progression and practical utility in many computational scenarios.

- **Sorting by Size:** The subsets are first sorted based on their size (the number of elements they contain). This is a straightforward comparison where a subset containing fewer elements is considered "smaller" than one with more elements. Sorting by size helps in segmenting the data into manageable chunks, which is particularly useful in applications like data mining where the size of the subset might indicate its utility or significance.
- **Lexicographical Sorting:** For subsets of equal size, a secondary sorting is applied based on lexicographical order. This is akin to alphabetical ordering in dictionaries but applied to numerical elements:
  - Compare the first elements of each subset; the subset with the smaller first element comes first.
  - If the first elements are equal, compare the second elements, and so on.
  - This ordering is natural to humans and is often used in combinatorial and mathematical applications where ordered data are essential for algorithms, such as searching and pattern recognition.
- **Application:** This dual-layered sorting mechanism makes the analysis and manipulation of subsets easier and more intuitive. For example, finding all subsets of a specific size or the smallest subset containing a particular element becomes much more straightforward.

Detailed Runs

Provide a detailed example using specific values to illustrate the process:

This shows how to run the codes. First enter the directory which contains `Makefile` and `order.c`. Then input `make`.

```
make
```

After that use `./order` command to run the program.

```
./order
```

Then you can choose a functionality as you want. In our case the first functionality is our goal so we choose 1. Then it requires we enter the size of the n-set. We enter 4. Then we got the result as desired. And the program return to the menu.

```
> ./order

Menu:
1. Generate and print all subsets/rank
2. Find and print the rank of a given subset
3. Find and print a subset given its rank
4. Find and print the successor of a given subset
5. Exit
Enter your choice: 1
Enter the size of the set (n): 4
Rank 0: {}
Rank 1: {1}
Rank 2: {2}
Rank 3: {3}
Rank 4: {4}
Rank 5: {1, 2}
Rank 6: {1, 3}
Rank 7: {1, 4}
Rank 8: {2, 3}
Rank 9: {2, 4}
Rank 10: {3, 4}
Rank 11: {1, 2, 3}
Rank 12: {1, 2, 4}
Rank 13: {1, 3, 4}
Rank 14: {2, 3, 4}
Rank 15: {1, 2, 3, 4}

Menu:
1. Generate and print all subsets/rank
2. Find and print the rank of a given subset
3. Find and print a subset given its rank
4. Find and print the successor of a given subset
5. Exit
Enter your choice:
```

Now choose 1 again and enter 5 for the size of n-set.

```
Menu:
1. Generate and print all subsets/rank
2. Find and print the rank of a given subset
3. Find and print a subset given its rank
4. Find and print the successor of a given subset
5. Exit
Enter your choice: 1
Enter the size of the set (n): 5
Rank 0: {}
Rank 1: {1}
Rank 2: {2}
Rank 3: {3}
Rank 4: {4}
Rank 5: {5}
Rank 6: {1, 2}
Rank 7: {1, 3}
Rank 8: {1, 4}
```

```
Rank 9: {1, 5}
Rank 10: {2, 3}
Rank 11: {2, 4}
Rank 12: {2, 5}
Rank 13: {3, 4}
Rank 14: {3, 5}
Rank 15: {4, 5}
Rank 16: {1, 2, 3}
Rank 17: {1, 2, 4}
Rank 18: {1, 2, 5}
Rank 19: {1, 3, 4}
Rank 20: {1, 3, 5}
Rank 21: {1, 4, 5}
Rank 22: {2, 3, 4}
Rank 23: {2, 3, 5}
Rank 24: {2, 4, 5}
Rank 25: {3, 4, 5}
Rank 26: {1, 2, 3, 4}
Rank 27: {1, 2, 3, 5}
Rank 28: {1, 2, 4, 5}
Rank 29: {1, 3, 4, 5}
Rank 30: {2, 3, 4, 5}
Rank 31: {1, 2, 3, 4, 5}

Menu:
1. Generate and print all subsets/rank
2. Find and print the rank of a given subset
3. Find and print a subset given its rank
4. Find and print the successor of a given subset
5. Exit
Enter your choice:
```

Now choose 1 again and enter 6 for the size of n-set.

```
Menu:
1. Generate and print all subsets/rank
2. Find and print the rank of a given subset
3. Find and print a subset given its rank
4. Find and print the successor of a given subset
5. Exit
Enter your choice: 1
Enter the size of the set (n): 6
Rank 0: {}
Rank 1: {1}
Rank 2: {2}
Rank 3: {3}
Rank 4: {4}
Rank 5: {5}
Rank 6: {6}
Rank 7: {1, 2}
Rank 8: {1, 3}
Rank 9: {1, 4}
Rank 10: {1, 5}
Rank 11: {1, 6}
```

```
Rank 12: {2, 3}
Rank 13: {2, 4}
Rank 14: {2, 5}
Rank 15: {2, 6}
Rank 16: {3, 4}
Rank 17: {3, 5}
Rank 18: {3, 6}
Rank 19: {4, 5}
Rank 20: {4, 6}
Rank 21: {5, 6}
Rank 22: {1, 2, 3}
Rank 23: {1, 2, 4}
Rank 24: {1, 2, 5}
Rank 25: {1, 2, 6}
Rank 26: {1, 3, 4}
Rank 27: {1, 3, 5}
Rank 28: {1, 3, 6}
Rank 29: {1, 4, 5}
Rank 30: {1, 4, 6}
Rank 31: {1, 5, 6}
Rank 32: {2, 3, 4}
Rank 33: {2, 3, 5}
Rank 34: {2, 3, 6}
Rank 35: {2, 4, 5}
Rank 36: {2, 4, 6}
Rank 37: {2, 5, 6}
Rank 38: {3, 4, 5}
Rank 39: {3, 4, 6}
Rank 40: {3, 5, 6}
Rank 41: {4, 5, 6}
Rank 42: {1, 2, 3, 4}
Rank 43: {1, 2, 3, 5}
Rank 44: {1, 2, 3, 6}
Rank 45: {1, 2, 4, 5}
Rank 46: {1, 2, 4, 6}
Rank 47: {1, 2, 5, 6}
Rank 48: {1, 3, 4, 5}
Rank 49: {1, 3, 4, 6}
Rank 50: {1, 3, 5, 6}
Rank 51: {1, 4, 5, 6}
Rank 52: {2, 3, 4, 5}
Rank 53: {2, 3, 4, 6}
Rank 54: {2, 3, 5, 6}
Rank 55: {2, 4, 5, 6}
Rank 56: {3, 4, 5, 6}
Rank 57: {1, 2, 3, 4, 5}
Rank 58: {1, 2, 3, 4, 6}
Rank 59: {1, 2, 3, 5, 6}
Rank 60: {1, 2, 4, 5, 6}
Rank 61: {1, 3, 4, 5, 6}
Rank 62: {2, 3, 4, 5, 6}
Rank 63: {1, 2, 3, 4, 5, 6}
```

```
Menu:
1. Generate and print all subsets/rank
2. Find and print the rank of a given subset
3. Find and print a subset given its rank
4. Find and print the successor of a given subset
5. Exit
Enter your choice:
```

Now we get what we want. Other choices are also compiled and usable. However, it's not our main goal. If you are curiosity, you can check it out. Now we exit the program by choose 5 and clean the files.

```
Menu:
1. Generate and print all subsets/rank
2. Find and print the rank of a given subset
3. Find and print a subset given its rank
4. Find and print the successor of a given subset
5. Exit
Enter your choice: 5
Exiting program.

> make clean
del /F /Q order.exe
```

Note: I have tested it on mac so it's fine for Linux to run it.

Why the Algorithm Works

Functions and Their Roles

1. `compare` **Function:**
   - **Purpose:** This function is used by the `qsort` standard library function to sort subsets. It orders subsets primarily by size and secondarily by lexicographical order within the same size.
   - **Mechanism:**
     - It accepts two generic pointers, casts them to arrays of integers, and first compares the subsets by their sizes (stored in the first element of each array).
     - If sizes are equal, it proceeds to compare elements one by one, thus implementing lexicographical ordering.
   - **Effectiveness:** Sorting by size helps in organizing subsets by their cardinality, which is intuitive and useful for many computational tasks. Lexicographical ordering within subsets of the same size ensures a consistent and predictable order, useful for algorithms that may require ordered input (like binary search).
2. `generateSubsets` **Function:**
   - **Purpose:** Generates all subsets of a given set of `n` elements and sorts them.
   - **Mechanism:**
     - It calculates the power set size as 2^n (every element can either be in a subset or not, leading to two choices per element).
     - Allocates memory for these subsets and iterates over a range from 0 to $2^n - 1$. Each number in this range represents a subset, where the binary representation of the number indicates the presence (`1`) or absence (`0`) of each element in the subset.
     - The subsets are then sorted using the `qsort` function and the previously defined `compare` function.
   - **Effectiveness:** This method of generating subsets using binary numbers is extremely efficient as it leverages bitwise operations that are computationally inexpensive. Sorting the subsets immediately after generation prepares them for any subsequent operations that may require ordered data.

More Information

1. **Binary Representation for Subsets:**
   - Each subset of a set corresponds uniquely to a binary number. For instance, the set `{1, 2, 3}` has a power set size of $2^3 = 8$. The binary numbers `000` to `111` correspond to subsets from `{}` to `{1, 2, 3}`. This is a straightforward and mathematically sound method to enumerate all possible combinations of a set.
2. **Sorting by Size and Lexicographically:**
   - Sorting subsets by size allows for structured access and analysis, for example in algorithms that might prioritize smaller or larger subsets.
   - Lexicographical sorting within subsets of the same size provides a natural ordering that mirrors how words are ordered in a dictionary. This can be crucial for algorithms that perform searches or comparisons among subsets.
3. **Efficient Memory Usage and Error Handling:**
   - The algorithm allocates memory dynamically for the subsets, which means it can handle sets of varying sizes up to the limits of available memory.
   - By using standard library functions like `qsort`, the implementation leverages highly optimized algorithms that are part of the C standard library, ensuring both reliability and efficiency.