Delving into O(1) Operations and O(n) Sorting using Bitmap

Efficiently sorting and manipulating large datasets is a fundamental challenge in computer science, with traditional data structures such as arrays, linked lists, and trees offering trade-offs between speed and memory usage. This paper introduces a practical approach using **bitmap**, which achieves **O(n)** sorting and **O(1)** for all key element-wise operations, providing the smallest possible time complexity in theory.

1. Bitmap: Trivial O(n) Sorting for Positive Integers

For a dataset containing positive integers, sorting using bitmap is inherently **O(n)**. This is achieved by directly converting each integer into its corresponding bit position within the bitmap, a trivial operation due to the direct mapping:

- Given an integer x, the bit at position x is set to 1.
- Sorting becomes implicit, as extracting the set bits in order naturally yields a sorted sequence.

Thus, the insertion of each element occurs in **O(1)** time, and a full traversal to read the sorted values is **O(n)**, making the overall sorting process linear.

2. Limitations of Traditional Sorted Data Structures

While traditional data structures provide various efficiencies, they fall short of achieving **O(1)** for all operations:

2.1 Sorted Arrays

- **Random access:** O(1)
- **Insertion/Deletion:** O(n) due to shifting elements
- **Traversal:** O(n) which is O(1) element-wise

2.2 Linked Lists

- **Random access:** O(n) due to traversing elements
- **Insertion/Deletion:** O(1) if the location is known
- **Traversal:** O(n) which is O(1) element-wise

2.3 B-trees and Balanced Trees

- **Random access:** O(log n)
- **Insertion/Deletion:** O(log n)
- **Traversal:** O(n) which is O(1) element-wise

These structures excel in specific areas but fail to achieve constant time for all key operations.

3. The Core Tweak: Enabling O(1) Find Next with Sparse Data Layouts

A major challenge in sorted data structures, especially when dealing with sparse datasets, is efficiently finding the **next element** without scanning step by step. This approach overcomes the challenge by leveraging the **constant cost of bitwise operations**:

3.1 Explanation of the Find Next Operation

Given the current bit position p, we can find the **next set bit** using the following steps:

1. **Mask lower bits:**
   Create a mask that clears all bits **below and including** position p:

$$\text{mask} = (1 \ll (p+1)) - 1$$

Clear the lower bits:

$$\text{masked\_bitmap} = \text{bitmap} \,\&\, \sim \text{mask}$$

2. **Isolate the least significant set bit:**
   Use the bitwise trick to isolate the next set bit:

$$\text{next\_bit} = \text{masked\_bitmap} \,\&\, -\text{masked\_bitmap}$$

3. **Determine the position of the next set bit:**
   The position of the next set bit can be found using:

$$\text{position} = \log_2(\text{next\_bit}) = \text{bit\_length}(\text{next\_bit}) - 1$$

This process ensures that the next set bit is found in constant time **O(1)** without needing sequential traversal, unlike traditional data structures.

Some may wonder if the bitwise operation should **O(1)** only with size of CPU register. I am not sure how the Big O community, if there is such a thing, could care less about the size of numbers when they constantly ignore the size of strings, which are really an array of byte object, but I understand it is indeed **O(1)** with respect the size of the bitmap as it can be split into arrays of CPU register size, and each can be accessed by the offsets calculated from the given integer within **O(1)**. I am not sure of the exact implementation of bitwise operations in Python, but the test below backs it up as it does not show the access speed does not grow with the size of integers,

4. Managing Space Efficiency: The Two-Layer Bitmap Approach

One challenge of using a bitmap structure is the potentially high space usage. The required space is proportional to the range of integers, specifically **range/8 bytes**. For large ranges, this can become costly. To overcome this, we introduce a **two-layer bitmap approach**:

4.1 Base Layer and Second Layer

- **Base layer:** Contains a bitmap of size $\sqrt{\text{range}}/8$ bytes, representing summaries of the second layer.
- **Second layer:** Only dynamically created where actual values exist, further reducing the memory overhead.

4.2 Dynamic Creation of Second-Layer Bitmaps

- When a value is inserted, the base layer is updated and a second-layer bitmap is created only when necessary.
- For sparse datasets, this optimization drastically reduces memory usage while retaining **O(1)** operations.

Note: While this implementation uses a two-layer structure, other implementations can extend the bitmap layers to more than two, depending on the data sparsity and space efficiency requirements. This flexibility allows the structure to scale efficiently even for extremely sparse or dense datasets.

5. Handling Duplicates Using Hash Tables

Bitmap structures natively do not support duplicate values because each bit can only be set once. This limitation is addressed by using a **hash table** to track the count of duplicates for each bit:

- When inserting a value that already exists, the hash table increments the count.
- When deleting a value, the count is decremented, and the bit is cleared when the count reaches zero.
- The hash table ensures that the duplicate handling remains within **O(1)** time complexity.

6. Extending to Floating-Point and Negative Numbers

Bitmap inherently works with positive integers. Handling **floating-point** and **negative** numbers requires transformations to map them to integers:

6.1 Handling Floating-Point Numbers

- Trim lower digits based on a specified precision and multiply by a power of 10 to convert the value into an integer.
- For example, with a precision of 2 decimal places, 3.14 becomes 314.

6.2 Handling Negative Numbers

- Identify the smallest negative value in the dataset and shift all values to be non-negative by adding the absolute value of this minimum.
- For example, if the smallest value is -10, each value is increased by 10.

6.3 Reverting the Transformation

- When accessing or outputting the values, reverse the transformation to restore the original floating-point or negative values.
- These operations maintain **O(1)** complexity.

7. Experimental Results and Performance

7.1 Sorting Performance (Bitmap)

Our sorting test highlights the efficiency of bitmap hashing in achieving **O(n)** sorting, where most traditional algorithms, such as quicksort and mergesort, operate at **O(n \log n)**. The following table demonstrates the sorting time across various data sizes:

Size	Sorting Time (s)	Sorted Elements
100	0.000010	10
10,000	0.001073	1,000
1,000,000	0.079612	100,000

- **Explanation:** The sorting process simply involves reading the set bits in the correct order using bitwise traversal, avoiding any comparisons. The results reflect the **linear growth** expected for an **O(n)** sorting mechanism.

7.2 Preliminary Experimental Results

Our preliminary experimental results demonstrate the efficiency of this approach across various scenarios:

Metric	Size 100	Size 10,000	Size 1,000,000	Change Rate 1-2	Change Rate 2-3
Test Data Size	10	1,000	100,000	100.0	100.0
Sort Time (s)	0.000016	0.000495	0.059559	30.9375	119.336
Contains Time (s)	0.000	0.000	0.000	NaN	NaN
Next Time (s)	0.000	0.000	0.000	NaN	NaN
Previous Time (s)	0.000	0.000	0.000	NaN	NaN

Metric	Size 100	Size 10,000	Size 1,000,000	Change Rate 1-2	Change Rate 2-3
Traversal Time (s)	0.000012	0.000389	0.066529	32.4167	170.975
Reverse Traversal Time (s)	0.000011	0.000373	0.065000	32.9091	173.264

8. Advantages and Trade-offs

8.1 Advantages

- **O(1) Complexity:** All key operations achieve O(1) performance, a unique feature not seen in other sorted structures.
- **Simplicity:** The implementation is straightforward, relying on bitwise operations instead of complex tree rotations or balancing.

9. Applications

This structure is well-suited for applications where fast access, insertion, and traversal are critical, such as:

- **Real-time indexing**
- **In-memory caching systems**
- **Database search optimizations**

10. Conclusion

The introduced two-layer bitmap structure achieves O(1) time complexity for all key element-wise operations and O(n) for sorting, representing the smallest possible time costs in theory. By overcoming challenges related to duplicates, floating-point values, and memory usage, this approach should be most effective in scenarios involving non-duplicate integers, but it remains highly effective for handling duplicates, floating-point values, and other numeric types through efficient constant-time operations. With its simplicity and efficiency, it offers a strong alternative to traditional sorted structures like arrays and B-trees while preserving performance even in extended use cases involving diverse numeric types.