RMIT University

Algorithm Design & Implementation

COSC2658 Data Structure & Algorithm

Lecturer: Dr. Minh Dinh
Author: Huynh Nguyen Nguyen
ID: s3694703

12-20-2020

Contents

List of Figures

List of Tables

1. Question 1: Search item in Split-Sorted Array

In this question, it is required to develop an efficient search algorithm on a special kind of array – split-sorted array. By definition, the split-sorted array contains two ascending-order sorted arrays which all elements in the first one are greater or equal to the biggest/the last element in the second array.

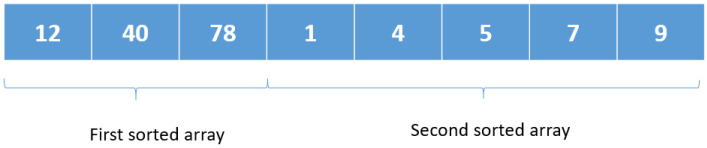


Figure 1. An example of Spit-sorted array

1.1. Algorithm Explanation

The most naïve way to search an element in an array is to traverse through all elements within an array and compare them to the target value; thus, producing O(n) in time complexity. This strategy is simple to develop and is applicable to all types of array. However, as our array is not a totally random array, rather than implementing such Brute Force algorithm, it is more ideal to take advantage of some specific features of the array to improve the software efficiency. As known, within the normal sorted array, a specific element can be found using Binary Search algorithm which offers a much greater improvement over the naïve one regarding time complexity – from O(n) to O(logn). Although the inspected array in this case is not wholly sorted, Binary Search algorithm can be taken as an inspiration to utilize the partly sorted characteristic of Split-Sorted array in searching for the target key.

In specific, according to the Binary Search algorithm, the pointer will be first initialized at the middle of the array; thus, dividing the array into two halves. Then, based on the comparison between the middle point and the target value, the traversal direction can be divided. However, in the split-sorted array, the most left element is not the smallest value, the most-right element is also not the largest value; thus, the normal comparison approach will not work.

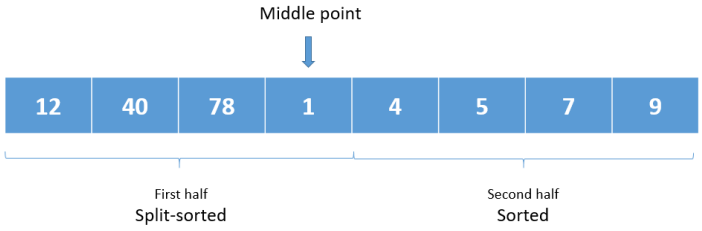


Figure 2. Split-sorted search algorithm illustration 1

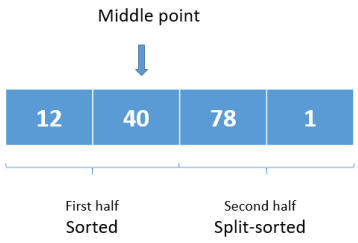


Figure 3. Split-sorted search algorithm illustration 2

On the other hand, when dividing a split-sorted array at the middle point, another pattern is emerged in which a fully sorted array and a split sorted array is created at two ends as illustrated. Once again dividing the small split-sorted array, we can again obtain exactly the same pattern. Since one of the sub array will be sorted, it is possible to know its range and decide whether the target lies on the sorted or split-sorted part.

Step 1: Get the middle position

Step 2: Compare the middle value to target. If true, return middle position as a result

Step 3: If the left-side array is sorted, get its range

- *If the target is within that range, move to left*
- *Else, move to right*

Step 4: If the right-side array is sorted, get its range

- *If the target is within that range, move to right*
- *Else, move to left*

Hence, the problem now lays upon the methodology to figure out which side of array has already been sorted. Because the sorting follows ascending order, the two-side elements of each sub array can be taken into comparison such that if the left element is smaller than the right one, the sub array is increasingly sorted as illustrated [1].

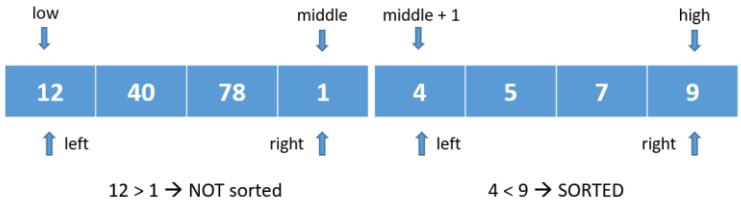


Figure 4. Split-sorted search algorithm illustration 3

However, this classification condition only works when all the elements in the list are unique. In case there are some duplicated values and thence the left and right element would probably share the same value, it does not have enough evidence to decide whether the sub array is sorted or not since the equivalence cannot show the sort order of elements. For example, in the Figure below, although two arrays have their elements on the edges equal to each other (12 = 12), one is sorted while the other is not.

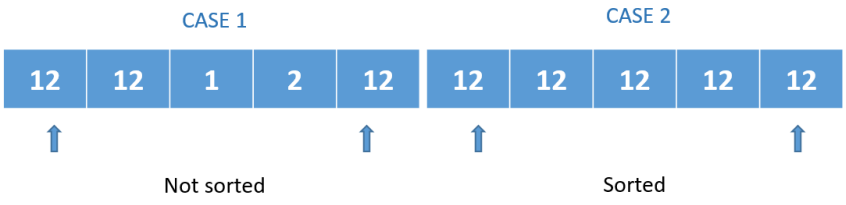


Figure 5. Split-sorted search algorithm illustration 4

As a resolution for this problem, whenever detecting the situation where the low, high and middle point have the same value, the pointers at the two side of the array would be moved inward until there is a difference between three points so that we can conclude the position of the sorted array [2].

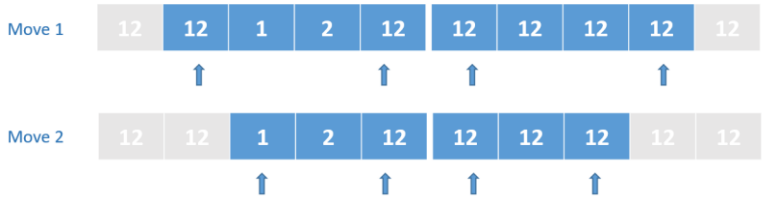


Figure 6. Split-sorted search algorithm illustration 5

The program will return in two situations:

1. **Found out the target position**: When the middle position matches with the target key, the program returns the middle position.
2. **Finished the array but there was no match**: The array is considered to be finished when
    - The low position is greater than high position.
    - The low position is equal to the high position and that position does not have the key

1.2. Software Implementation

To ensure the correctness of the algorithm such that all possible cases can be handled, it is ideal to perform automatic testing with random inputs instead of predefined one. Hence, in the software implementation for this question, we decide to generate test data randomly and execute each test case multiple times to measure runtime as well as ensure the program's correctness.

The following section is the documentation of how the algorithm and the test data is implemented or generated.

1.1.1. Supporting Functions and Algorithm Implementation

Step 1: Generating random sorted array

This function generates a number of random integers within pre-defined range according to the input parameter (min_value and max_value), and sorts them in ascending order using the in-built function of Python.

Table 1. Random sorted array generator function.

```
def random_sorted(size, max_value, min_value = 1):
    a = random.randint(min_value, max_value + 1, size)
    a.sort()
    return a
```

```
Test random sorted array generator
----------------------------------
Size: 3 & Value range: [1,100]
[24, 32, 58]

Size: 10 & Value range: [1,1000]
[5, 22, 110, 185, 294, 394, 504, 707, 868, 951]
```

Step 2: Split a sorted array at k position

To split a sorted array in an efficient way such that the magnitude of K is not affected the time complexity, we decide to write our own functions:

- **Reverse array**: To save space resources and have O(1) space complexity, the reverse operation is preformed directly inside the input array by swapping elements at two sides.

Table 2. Reverse array function

```
def my_reverse(arr, start=0, end=None):
    if end is None:
        end = len(arr)
    while (start < end ):
        arr[start], arr[end - 1] = arr[end - 1], arr[start]
        start += 1
        end -= 1
```

```
Test reverse array function
---------------------------------
Origin array:  [44, 65, 67, 81, 85]

Reverse array:  [85, 81, 67, 65, 44]
```

- **Split array**: By reversing the whole array then dividing it at K position and again reverse two parts, we obtain the rotated version of the initial array.

Table 3. Split array function

```
def create_split_sorted(arr, k):
    my_reverse(arr)
    my_reverse(arr, end=k)
    my_reverse(arr, k)
```

```
Test split array function
---------------------------------
Origin array:  [5, 18, 19, 24, 32, 35, 74, 85]

Splitted array at index 3:  [35, 74, 85, 5, 18, 19, 24, 32]
```

Step 3: Define split sorted class

For convenience, the split-sorted object can be generated by inputting size, max value, the splitting position or inputting the self-defined array.

```
class SplitSorted:
    def __init__(self, size, max_value, split_pos):
        self.size = size
        self.split_pos = split_pos
        # Generate random sorted array
        self.arr = random_sorted(size, max_value)
        # Split array
        create_split_sorted(self.arr, split_pos)
        # To save search result
        self.searchResultDict = {}

    def __init__(self, arr):
        self.size = len(arr)
        self.arr = arr
        self.searchResultDict = {}
```

Figure 7. Code snipping of Split-Sorted class

Once the split-sorted array has been created, we can pass in the required key to be searched for. The function will return the target position and record all testing information including:

- Number of comparison operations made
- Number of recursive calls
- Execution time in microsecond

```
def search(self, key):
    if key not in self.searchResultDict:
        self.searchResultDict[key] = self.SearchResult(key)
    self.__count = 0
    self.__recursiveCount = 0
    start = timer()
    result = self.__search(key, 0, len(self.arr) - 1)
    self.searchResultDict[key].updateResult((timer() - start) * 1000000, self.__count, self.__recursiveCount)
    return result
```

Figure 8. Code snipping of public search function

Each search run will be grouped according to its input key and be saved into SearchResult object so that we can retrieve and calculate the average time taken to search for a specific key in a specific split-sorted array when running a test case multiple times.

```
class SearchResult:
    def __init__(self, key):
        self.key = key
        self.count = 0
        self.averageTime = 0
        self.results = []
    def updateResult(self, measured_time, count, recursiveCount):
        self.count = count
        self.recursiveCount = recursiveCount
        self.averageTime = (self.averageTime * len(self.results) + measured_time) / (len(self.results) + 1)
        self.results.append(measured_time)
```

Figure 9. Code snipping of Search Result class

Step 4: Implement search algorithm

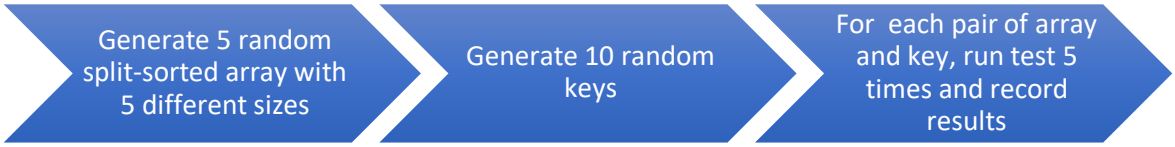
This function is a private method inside SplitSorted class to implement recursive calls. The explanation of the function has been discussed in the above section. Those "count" and "recursiveCount" variables are private attributes for keeping track of the program's complexity for further analysis.

```
def __search(self, key, l, h):
    self.__count += 1
    if l > h: # Return not found if there is nothing left
        return -1
    mid = (l + h) // 2

    self.__count += 1
    if self.arr[mid] == key: # Meet the target, return
        return mid

    self.__count += 2
    # SPECIAL CASE: loop until find out low, high and mid has difference
    while ((self.arr[l] == self.arr[mid]) and (self.arr[h] == self.arr[mid])):
        self.__count += 1
        if (l >= h): return -1 # Iterated through all elements, return not found
        l += 1
        h -= 1
        self.__count += 2

    self.__count += 1
    self.__recursiveCount += 1
    if self.arr[l] <= self.arr[mid]: # If the first array is sorted
        self.__count += 1
        if self.arr[l] <= key and key <= self.arr[mid]: # If sorted range has key
            return self.__search(key, l, mid-1) #  Move to sorted array
        return self.__search(key, mid + 1, h) # Otherwise, move to split-sorted array

    # If the first array is not sorted, the second must be sorted
    self.__count += 1
    if self.arr[mid] <= key and key <= self.arr[h]: # If sorted range has key
        return self.__search(key, mid + 1, h) # Move to sorted array
    return self.__search(key, l, mid-1) # Otherwise, move to split-sorted array
```

Figure 10. Code snipping of split-sorted array search function

1.1.2. Timing Functions Implementation

As been required, the timing process consists of:



Hence, for reusability, there are two functions: (1) to generate test data and (2) to run test cases.

```
def generate_test_data(min_value, max_value, arr_size):
  # Generate 10 random key
  split_sorteds = []
  keys = []
  i = 0
  while i < 10:
      key = randint(min_value, max_value) # Randomly create key
      if key not in keys: # If key is not exist, save. Otherwise, ignore
          keys.append(key)
          i += 1
  print("Keys to be searched: ", keys)

  for size in arr_size:
      print("Generating random array with size " + str(size) + ".......", end="")
      # Random select split position K
      split_pos = randint(0, size - 1)

      # Generate split sorted
      my_split_sorted = SplitSorted(size, max_value, split_pos)

      split_sorteds.append(my_split_sorted)
      print(" Finished")
  return (split_sorteds, keys)
```

Figure 11. Code snipping of generating test data function

```
def test_run(split_sorted_list, key_list, number_of_test):
  for split_sorted in split_sorted_list:

    table_data = []
    for key in key_list:
        for _ in range(number_of_test):
            split_sorted.search(key)
        search_result = split_sorted.searchResultDict[key]
        row_data = [split_sorted.size if key == key_list[0] else '', # Print size
                    split_sorted.split_pos if key == key_list[0] else '', # Print split position
                    key, # Print key
                    search_result.count, # Print number of comparison operation
                    search_result.recursiveCount, # Print number of recursive call
                    "%.3f" % search_result.averageTime] # Print average time of 5 runs
        table_data.append(row_data)

    table_data.append(['', '', '', '', '', '----------'])
    table_data.append(['', '', '', '', '', "%.3f" % split_sorted.averageTimeAllTestCases()])
    print(tabulate(table_data, headers=headers, tablefmt=tablefmt))
    print()
```

Figure 12. Code snipping of running test function

1.3. Time Measuring Result

```
# Test Parameter Configuration
min_value = 1
max_value = 1000
arr_size = [100000, 200000, 400000, 800000, 1600000]
number_of_test = 5
```

Figure 13. Time measuring configuration

After inputting these test configuration, we got the results as follows where:

- Array size: is the length of the split-sorted array that will be generated and tested
- Split position: is the K position where the generated sorted array is rotated
- Searched value: is the value that needs to be found
- Compare operations: is the numbers of comparisons that have been made during the searching
- Recursions: is the numbers of recursive calls (calling search method in the sub-divided array)
- Average time: is the average execution time after 5 runs with the same array and the same key. The execution time is recorded in microsecond.
- The bottom value in the "Average time" column is the average time for 10 test cases.

Array size	Split position	Searched value	Compare operations	Recursions	Average time (us)
100000	34951	415	50	8	13.701
		137	56	9	14.240
		281	62	10	16.696
		925	56	9	15.540
		375	50	8	12.616
		761	32	5	7.989
		919	62	10	15.921
		205	38	6	9.446
		437	56	9	15.124
		724	56	9	13.064

					14.075

Figure 14. Result table of array size 100,000

Array size	Split position	Searched value	Compare operations	Recursions	Average time (us)
200000	146545	415	56	9	18.296
		137	50	8	13.138
		281	50	8	12.179
		925	56	9	15.566
		375	50	8	13.169
		761	56	9	16.207
		919	56	9	14.686
		205	20	3	5.000
		437	50	8	14.461
		724	44	7	12.537

					14.259

Figure 15. Result table of array size 200,000

Array size	Split position	Searched value	Compare operations	Recursions	Average time (us)
400000	139178	415	56	9	16.214
		137	56	9	14.351
		281	56	9	13.747
		925	50	8	11.923
		375	44	7	10.997
		761	56	9	13.000
		919	32	5	7.980
		205	56	9	12.429
		437	56	9	12.599
		724	50	8	13.208

					13.547

Figure 16. Result table of array size 400,000

Array size	Split position	Searched value	Compare operations	Recursions	Average time (us)
800000	235816	415	50	8	15.760
		137	50	8	36.705
		281	56	9	14.271
		925	26	4	6.009
		375	56	9	12.902
		761	56	9	13.312
		919	50	8	12.384
		205	56	9	10.824
		437	44	7	9.106
		724	56	9	11.059

					15.148

Figure 17. Result table of array size 800,000

Array size	Split position	Searched value	Compare operations	Recursions	Average time (us)
1600000	1302900	415	50	8	15.270
		137	50	8	11.959
		281	56	9	12.834
		925	44	7	12.783
		375	56	9	13.506
		761	56	9	14.831
		919	50	8	12.280
		205	56	9	12.399
		437	56	9	12.625
		724	56	9	14.622

					16.226

Figure 18. Result table of array size 1,600,000

1.4. Empirical Analysis

1.1.3. Analysis on Required Test Cases

In the earlier section, we ran tests for arrays with size 100000, 200000, 400000, 800000 and 1600000. However, as can be easily observed, the execution time for those test cases had insignificantly differences although the time was measured in an exceedingly small timing unit (microsecond). In other word, for

those specific test cases, the size of the array does not have any impact upon the taken time. For re-verification, we ran one more test for the array size of 10,000,000; thus, gaining the same amount of time.

Array size	Split position	Searched value	Compare operations	Recursions	Average time (us)
10000000	4680993	539	56	9	22.526
		112	50	8	11.480
		458	44	7	9.205
		723	56	9	11.349
		651	50	8	10.126
		212	38	6	7.580
		511	50	8	9.882
		141	56	9	11.960
		322	56	9	11.367
		873	56	9	11.460

					11.693

Figure 19. Result table of array size 10,000,000

While we do admit that the next run tends to be faster than the last due to the reboot or RAM / cache access times compared to disk I / O, those factors cannot completely falsify the time measurement of the program. In this case, not only does the average measured time show the equivalence of time complexity, but the number of comparisons and recursion also shows the same. For all test cases, regardless of size, the maximum number of comparisons and recursion to be performed is 56 and 10, respectively. Hence, this is an evidence that our test cases are happened to enter a special circumstance; thus, cannot use to prove the actual time complexity for this algorithm to be O(logn).

The reason behind this strange behavior lies in the test inputs. As shown, the range of values is not enough for the size of the array. Generating random values too many times in a range of values too small will lead to an even distribution of values in an array. The distributions of generated arrays are plotted in Figure below.

Table 4. Time-testing configuration

INPUT
Value range: from 1 to 1000 Sizes: 100000, 200000, 400000, 800000 and 1600000

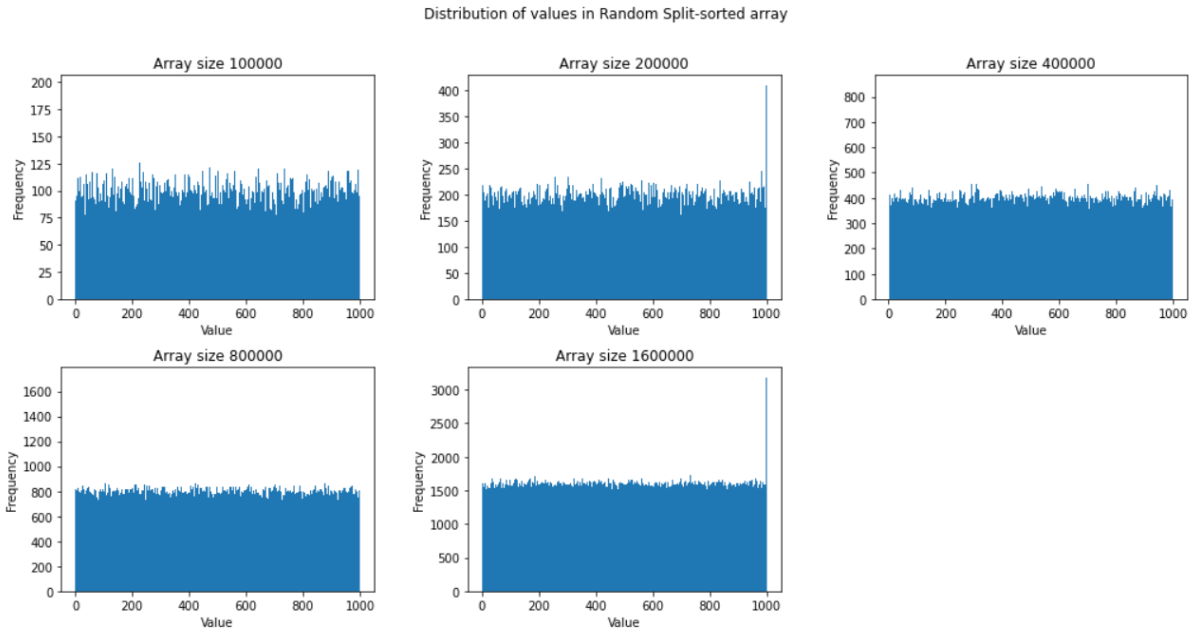


Figure 20. The histograms for distribution of values in Random split-sorted array

The bigger the array size is, the more uniform the distribution is. As a result, the bigger the array size is, the more frequently a specific value appears in that array. For example, each value in array size 1,600,000 appears 1,600 times or each value in array size 800,000 appears 800 times. It is a reasonable pattern since we have 1000 possible values within each array.

$$number\ of\ occurrences = \frac{size}{1000}$$

Since the number of occurrences of a value is proportional to the array size, the greater the number of occurrences in the array size, the greater the chance of finding that value because we can return any position as long as it fits the target key. As a result, when the array contains a significant length relative to the range of values, the time complexity of the program is not dependent on the size but on the range of values*. Thus, because our test cases have the same input range, they share the same time complexity. However, the logarithmic growth-rate can be observed through the number of recursions:

$$number\ of\ recursions\ \approx 10 \approx \log_2 N = \log_2 1000 \approx 9.9$$

Where N is the number of unique values in the array. As we have min value is 1, max value is 1000, there will 1000 unique values.

*Note: the exception for this statement will be mentioned in the worst-case scenario.

1.1.4. Analysis on Worst Cases

As the above test cases appear to be in the same situation, they have not covered all possible cases for this algorithm. Hence, this part aims to re-run the test so as to achieve a stronger evidence about the time complexity of the algorithm.

Test case 1: The searched value is not available in the array

In the previous test scenario, when we entered a small range of values and large input size, the runtime was not affected by the array size in finding the key that existed due to the increase in the frequency of a

specific value with respect to array size. In the opposite case, the key is out of scope or not available in the array, the program will have to end the entire binary search to find the absence of the target key. Hence, this test will input the non-existed key so as to get rid of the unusual impact of the uniform distribution of values and observe the normal program behavior.

Test design: This test scenario is a modified version of the previous one as we keep the running test for array size from 100,000 to 1,600,000 and value range from 1 to 1000, each test runs for 5 times. However, we insert more array sizes for plotting purpose and adjust the searched value to be larger than the maximum value.

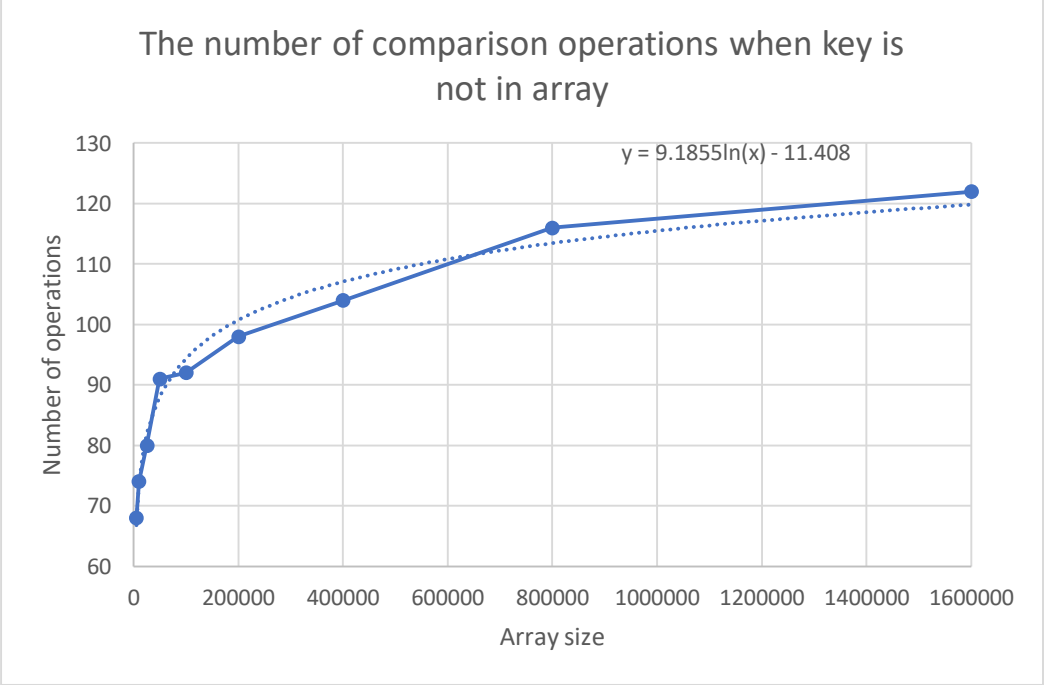
Result:

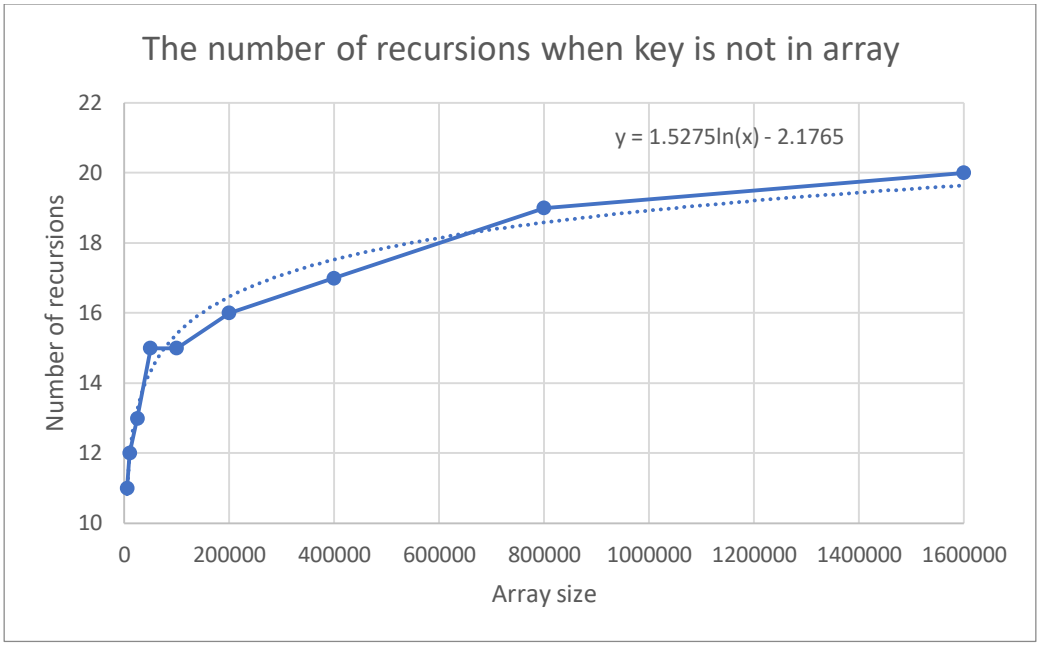
Array size	Split position	Searched value	Compare operations	Recursions	Average time (us)
5000	2670	1001	68	11	15.020
10000	7178	1001	74	12	16.045
25000	12999	1001	80	13	18.635
50000	34395	1001	91	15	20.696
100000	82829	1001	92	15	21.982
200000	69158	1001	98	16	22.487
400000	178466	1001	104	17	25.162
800000	310662	1001	116	19	27.491
1600000	1136764	1001	122	20	31.424

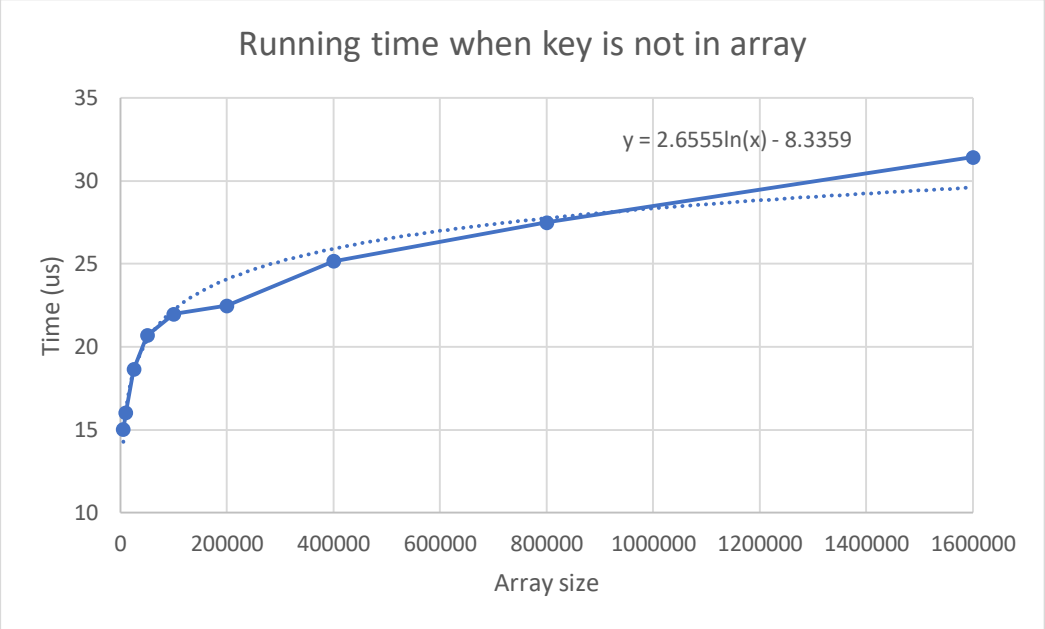
Figure 21. Timing result when the target is not in the array

As shown, unlike the constant pattern previously, there is an increase in the comparison operation counts, the recursion counts as well as the average time. Therefore, it is a prove that the array size has a considerably impact on the time complexity when the key is not existed in the array.

Graph:







Three graphs are presented the numbers of operations, recursions and time with respect to different input sizes. As can be seen, after fitting data using regression, all three graphs show that the algorithm has grown in the logarithmic rate O(logn) on this worst-case scenario.

Test case 2: The array contains only one unique value which is not the key

As been discussed, our algorithm works best when the array has only unique value. In case there are some duplicates, the program will move two outer pointers inward until the middle and two outer pointers have no longer the same. This action causes O(n) in time based on the theoretical analysis which is even worse than the test case 1.

Test design: This test runs with 9 test cases. Each case will be for different array size and be run 5 times to take the average result. All tested arrays will have one unique value – 1. And the input key is 2.

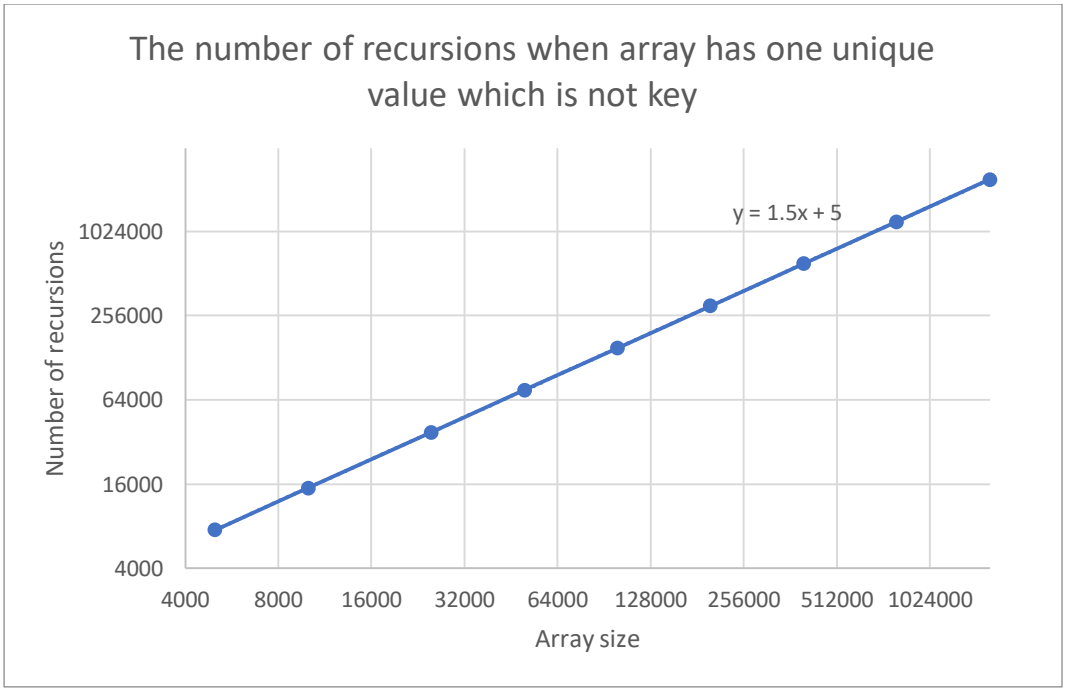
Result:

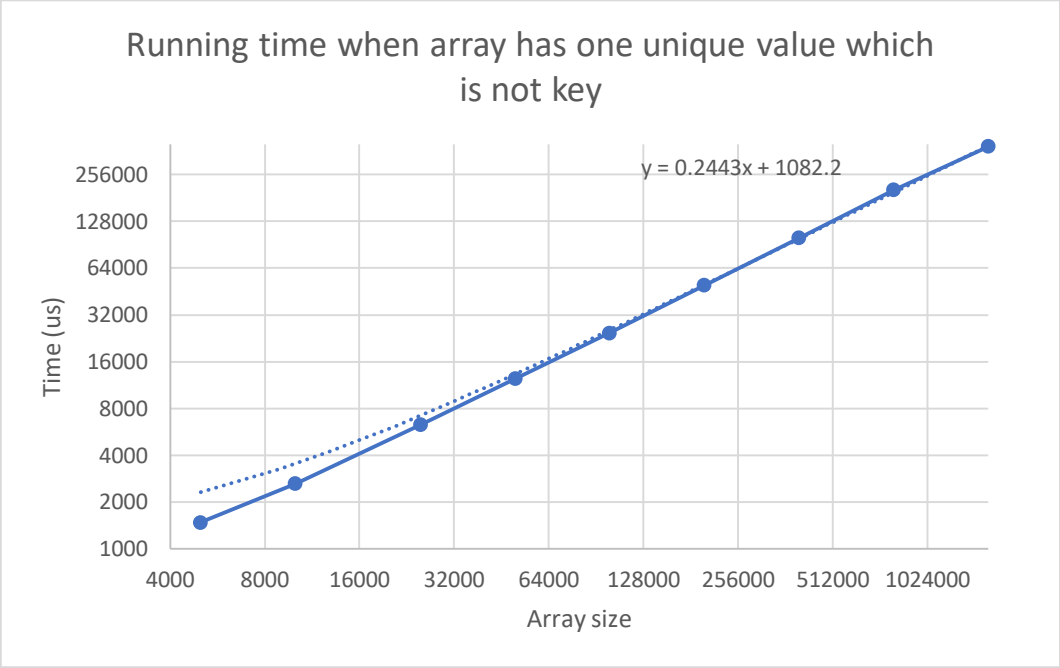
Array size	Split position	Searched value	Compare operations	Recursions	Average time (us)
5000	1385	2	7505	0	1476.756
10000	5450	2	15005	0	2621.413
25000	13091	2	37505	0	6285.697
50000	8488	2	75005	0	12415.573
100000	87671	2	150005	0	24519.385
200000	163684	2	300005	0	49549.354
400000	298723	2	600005	0	100286.614
800000	551839	2	1200005	0	203831.880
1600000	1002036	2	2400005	0	388145.427

Figure 22. Timing result when searching for non-exist key in one-value array

For larger array sizes, the number of comparisons or execution time is significantly greater. The order of growth will be plotted in the line chart below. On the other hand, there is no recursive call in all cases which can be explained by having a formal analysis for software implementations. Since all values in the array are equal, it is not possible to compare and make decisions about the direction of binary search; therefore, the program has entered the loop until it can detect any discrepancies. Since the array has only a single value, the program loops through all elements and breaks the function without recursively calling Binary Search.

Graph:





The two charts prove that in this worse-case scenario, instead of executing the Binary Search which produces O(logn), the program timing result grows in linear order O(n).

1.5. Unit Testing (Accuracy Testing)

The testing process is carried out to ensure the correctness of the discussed algorithm. There will be four test scenarios be tested.

Test scenario 1: Array is rotated at index 0 – Array is sorted

Table 5. Summary table when testing the array being rotated at index 0

NO	DESCRIPTION	INPUT	KEY	EXPECTED RESULT	RESULT
1	Test if the algorithm outputs correctly when searching in a sorted array and target key is the first element	Array size is 10 Max value is 100 Split position is 0	The first element's value	Return the index which has the same value as the first one	Pass
2	Test if the algorithm outputs correctly when searching in a sorted array and target key is the last element	Array size is 10 Max value is 100 Split position is 2	The last element's value	Return the index which has the same value as the last one	Pass
3	Test if the algorithm outputs correctly when searching in a sorted array with many duplicates	Array size is 1000 Max value is 10 Split position is 2 Search	The second element's value	Return the index which has the same value as the second one	Pass

```python
# Test when the split occurred at position 0
def test_split_at_0(self):
    # Create array
    my_split_sorted = SplitSorted(size = 10, max_value = 100, split_pos = 0)
    # Key at first
    value = my_split_sorted.arr[1]
    search_result = my_split_sorted.search(value)
    self.assertEqual(my_split_sorted.arr[search_result], value)

    # Key at last
    value = my_split_sorted.arr[9]
    search_result = my_split_sorted.search(value)
    self.assertEqual(my_split_sorted.arr[search_result], value)

    # Many Duplicates Array
    my_split_sorted = SplitSorted(size = 1000, max_value = 10, split_pos = 0)
    value = my_split_sorted.arr[1]
    search_result = my_split_sorted.search(value)
    self.assertEqual(my_split_sorted.arr[search_result], value)
```

Figure 23. Code snipping of test when array is split at 0

Test scenario 2: The target key is at the beginning of the array

Table 6. Summary table when searching for first element in the array

NO	DESCRIPTION	INPUT	KEY	EXPECTED RESULT	RESULT
1	Test if the algorithm produces the correct result when it searches for the target key that is the first element	Array size is 10 Max value is 100 Split position is 5	The last element's value	Return the index which has the same value as the last one	Pass
2	Test if the algorithm produces the correct result when it searches for the target key that is the first element in the array with many duplicates	Array size is 1000 Max value is 10 Split position is 90	The last element's value	Return the index which has the same value as the last one	Pass
3	Test if the algorithm produces the correct result when it searches for the target key that is the first element in the array with only one value	Array size is 10 Max value is 1 Split position is 4	The last element's value	Return the index which has the same value as the last one which must not be -1	Pass

```python
# Test when the required number is at the first position in array
def test_key_at_first(self):
    # Normal array
    my_split_sorted = SplitSorted(size = 10, max_value = 100, split_pos = 5)
    value = my_split_sorted.arr[0]
    search_result = my_split_sorted.search(value)
    self.assertEqual(my_split_sorted.arr[search_result], value)

    # Many Duplicates Array
    my_split_sorted = SplitSorted(size = 1000, max_value = 10, split_pos = 90)
    value = my_split_sorted.arr[0]
    search_result = my_split_sorted.search(value)
    self.assertEqual(my_split_sorted.arr[search_result], value)

    # One-value array
    my_split_sorted = SplitSorted(size = 10, max_value = 1, split_pos = 4)
    value = my_split_sorted.arr[0]
    search_result = my_split_sorted.search(value)
    self.assertEqual(my_split_sorted.arr[search_result], value)
```

Figure 24. Code snipping of test when key is the first element

Test scenario 3: The target key is at the end of the array

Table 7. Summary table when searching for the last element in array

NO	DESCRIPTION	INPUT	KEY	EXPECTED RESULT	RESULT
1	Test if the algorithm produces the correct result when it searches for the target key that is the last element	Array size is 10 Max value is 100 Split position is 5	The first element's value	Return the index which has the same value as the first one	Pass
2	Test if the algorithm produces the correct result when it searches for the target key that is the last element in the array with many duplicates	Array size is 1000 Max value is 10 Split position is 90	The first element's value	Return the index which has the same value as the first one	Pass
3	Test if the algorithm produces the correct result when it searches for the target key that is the last element in the array with only one value	Array size is 10 Max value is 1 Split position is 4	The first element's value – 1	Return the index which has the same value as the first one which must not be -1	Pass

```
# Test when the required number is at the last position in array
def test_key_at_last(self):
     # Normal array
    my_split_sorted = SplitSorted(size = 10, max_value = 100, split_pos = 5)
    value = my_split_sorted.arr[9]
    search_result = my_split_sorted.search(value)
    self.assertEqual(my_split_sorted.arr[search_result], value)

    # Many Duplicates Array
    my_split_sorted = SplitSorted(size = 1000, max_value = 10, split_pos = 90)
    value = my_split_sorted.arr[999]
    search_result = my_split_sorted.search(value)
    self.assertEqual(my_split_sorted.arr[search_result], value)

    # One-value array
    my_split_sorted = SplitSorted(size = 10, max_value = 1, split_pos = 4)
    value = my_split_sorted.arr[9]
    search_result = my_split_sorted.search(value)
    self.assertEqual(my_split_sorted.arr[search_result], value)
```

Figure 25. Code snipping of test when key is the last element

Test scenario 4: The target key is not in the array

Table 8. Summary table when searching for key which is not in the array

NO	DESCRIPTION	INPUT	KEY	EXPECTED RESULT	RESULT
1	Test if the algorithm produces not-found signal when it searches for the target key that is not in a split-sorted array	Array size is 10 Max value is 100 Split position is 2	101	The program returns -1 as not-found signal	Pass
2	Test if the algorithm produces not-found signal when it searches for the target key that is not in a fully sorted array	Array size is 20 Max value is 100 Split position is 0	101	The program returns -1 as not-found signal	Pass
3	Test if the algorithm produces not-found signal when it searches for the target key that is not in a split-sorted array with many duplicates	Array size is 1000 Max value is 10 Split position is 4	101	The program returns -1 as not-found signal	Pass

```
# Test when the required number is not in array
def test_key_not_available(self):
    # Split-sorted array
    my_split_sorted = SplitSorted(size = 10, max_value = 100, split_pos = 2)
    search_result = my_split_sorted.search(101)
    self.assertEqual(search_result, -1)

    # Sorted array
    my_split_sorted = SplitSorted(size = 20, max_value = 100, split_pos = 0)
    search_result = my_split_sorted.search(101)
    self.assertEqual(search_result, -1)

    # Many Duplicates array
    my_split_sorted = SplitSorted(size = 1000, max_value = 10, split_pos = 2)
    search_result = my_split_sorted.search(101)
    self.assertEqual(search_result, -1)
```

Figure 26. Code snipping of test when key is not in the array

In sum up, all 12 test cases within 4 different scenarios are passed which has guaranteed the correctness of the algorithm as well as its software implementation in performing and outputting correct results as expected.

```
test_key_at_first (__main__.TestSearchMethod) ... ok
test_key_at_last (__main__.TestSearchMethod) ... ok
test_key_not_available (__main__.TestSearchMethod) ... ok
test_split_at_0 (__main__.TestSearchMethod) ... ok

----------------------------------------------------------------------
Ran 4 tests in 0.017s

OK
```

Figure 27. Test result for all scenarios

2. Question 2

2.1. Find pair whose sum is closest to zero in sorted array [3]

Algorithm Explanation

As been required to find the pair of integers whose sum is closest to zero, it may opt to loop through the array and try to sum up pair of some integers. To be the closest-to-zero sum, the calculated sum must have the smallest absolute value. Moreover, within the sorted array of positive and negative integers, it is reasonable to start the iteration at two ends of the array such that the negative and position integer can cancel out each other. In each iteration, we sum up the two integers and compare the absolute value of calculated sum to the minimal sum; if the calculated sum is smaller, we have the new min sum. In addition, based on that sum of those two integers, we can figure out which part is greater than others in absolute value. As calling two moving pointers as left and right pointers, if their sum is less than 0, the sum is made up by a larger negative term and a smaller positive term when comparing based on their absolute value; thus, in this case, to get the sum closer to 0, we should move the left pointer inward / closer to 0 or increase the left pointer's position by 1. In other case where the sum is greater than 0, the right pointer should move inward by 1 position; thus, decreasing the right element. Nevertheless, in the best case, if the calculated sum is equal to 0, we can return the result and end the program immediately without finishing the whole array.

Analytical Analysis

Table 9. Code analysis for finding pair program

PSEUDOCODE	COST	OCCURRENCES
Initialize 2 pointers at the left and right of the array	c_1	1
WHILE (Left pointer does not exceed right pointer):	c_2	N + 1
Sum elements at two pointers	c_3	N
If (current min sum is greater than the calculated sum):	c_4	N
Update the calculated sum to be the min sum	c_5	N
Save 2 pointers' position	c_6	N
If the calculate sum is equal to 0:	c_7	N
Break loop	c_8	1
If the calculated sum is greater than 0:	c_9	N
Increase left pointer	c_{10}	N
Else:		
Decrease right pointer	c_{11}	N
Return the positions of two terms made up the min sum	c_{12}	1

The total execution time for this program is

$$total\ time \leq c_1 + c_2 * (N+1) + (c_3 + c_4 + c_5 + c_6 + c_7) * N + c_8 + c_9 * N + \max(c_{10} + c_{11}) * N + c_{12}$$

Thus, total time is in form of: $a * N + b$. In other word, finding the pair whose sum is closest to 0 takes a linear growth-rate ***O(N)*** with respect to the length the of array in average and worst-case scenario.

However, in the best case when the first pair of numbers has zero as its sum, the program break in the first loop. Hence, the total execution time will be: $total\ time = c_1 + c_2 + c_3 + c_4 + c_5 + c_6 + c_7 + c_8 + c_{12}$. The time complexity is ***O(1).***

In sum up, the time complexity for this algorithm is:

Table 10. Time complexity of finding pair program

CASE	TIME COMPLEXITY
Best	O(1)
Average	O(n)
Worst	O(n)

2.2. Find the minimum number of gates required at any point in time [4]

Algorithm Explanation

Given the list of arrival times and departure times, we can have a simulation for the process of aircraft taking off and landing. During the simulation, we track the number of aircraft in the airport at the same time by reducing the count when an aircraft takes off and increasing when an aircraft lands in an airport. Therefore, it can be figured out that at the most crowded times, how many aircraft will be parked at the airport which is also the minimum number of gates required for the airport to serve all landing schedules.

For software implementations, the arrival and departure lists will initially be sorted so that they can be processed chronologically. For each list, there will be a pointer to keep track of the current index. Since we simulate parking at the airport in chronological order, if the arrival time is less than the departure time, it is the time the plane lands at the airport; thus, we increase the number of planes, update the maximum number of airplanes if necessary and finish the landing simulation by moving the cursor to the next arrival time. In other case, we do a takeoff simulation by reducing the number of planes. The program will end when there is no more arrivals which means the arrival list is finished.

Analytical Analysis

This algorithm consists of two parts:

- **Sorting the arrival and departure list**: If using merge sort, the time complexity to sort a list is *O(n*logn)* with respect to the length of the list.
- **Counting the maximum number of airplanes parking at a time**:

We assume that:

- The arrival and departure list has the same size – N.
- The arrival and departure list contains time on the same day.

Table 11. Code analysis for finding minimum number of gates

PSEUDOCODE	COST	OCCURRENCE
Initialize airplane to count the number of airplanes at present	c_1	1
Initialize max airplane to store the maximum number of airplanes been counted	c_2	1
LOOP through all times in arrival list:	c_3	Best: N. Worst: 2N
If current arrival time is less than departure time: (LANDING)	c_4	N
Point to another arrival time	c_5	N
Increase the count of current airplanes	c_6	N
If airplane is greater than max airplane:	c_7	N
Update max airplane to airplane	c_8	Worst: N
Else: (TAKING OFF)		
Point to another departure time	c_9	N
Decrease the count of current airplanes	c_{10}	N
Return the maximum count of airplanes	c_{11}	1

Although the condition to end the program depends on the arrival list, we have to do the traverse through both departure and arrival list. Therefore, in the best scenario where all the arrival time are less than departure time, the program only loop for N times. In worst case where the landing and taking-off procedures happen alternately, the program must complete both lists which causes 2N times of looping.

In general, the formula for the total execution time of this count implementation is

$$total\ time \leq c_1 + c_2 + c_3 * 2N + (c_4 + c_5 + c_6 + c_7 + c_8 + c_9 + c_{10}) * N + c_{11}$$

In worst case: $total\ time = c_1 + c_2 + c_3 * 2N + (c_4 + c_5 + c_6 + c_7 + c_8 + c_9 + c_{10}) * N + c_{11}$

In best case: $total\ time = c_1 + c_2 + c_3 * N + (c_4 + c_5 + c_6 + c_7 + c_8) * N + c_{11}$

Thus, in all cases, the total time is in form of: $a * N + b$ which is O(n) where N is the number of airplanes landing or taking off in a day.

In conclusion, to find the minimum number of gates required, it takes $N + N * \log(N)$ times where N is the number of airplanes; thus, producing a ***O(n*logn)*** time complexity.

3. Question 3: Smart Binary Search Vs Classical Binary Search

3.1. Software Implementation

Generating random sorted list

For this function, we re-use the function mentioned in Question 1 which is:

```
def random_sorted(size, max_value, min_value = 1):
    a = random.randint(min_value, max_value + 1, size)
    a.sort()
    return a
```

Figure 28. Code snipping for random sorted array generator

Binary Search

```
def binary_search(arr, key, lo, hi):
    if lo > hi:
        return -1

    # Calculate the middle point
    mid = (lo + hi) // 2
    if key == arr[mid]: # If the mid point is the target
        return mid

    if key < arr[mid]: # If target is smaller => move to left subarray
        return binary_search(arr, key, lo, mid-1)
    else: # Else, move to right subarray
        return binary_search(arr, key, mid+1, hi)
```

Figure 29. Code snipping for binary search

Binary Search algorithm operates based on recursion concept. According to the comparison between the target value and the middle point, the traversal direction can be determined; thus, other half of the array will eliminate. Hence, if calling k is the number of recursions, in the worst case the program will be called recursively n times until the length of the array is 1.

$$\frac{n}{2^k} = 1 \Longrightarrow k = \log_2 n$$

Therefore, using the regular Binary Search for finding a particular item in a sorted array, the time complexity it takes is **O(logn)**.

Smart Binary Search

Smart Binary Search is the modified version of Binary Search. However, rather than pointing and checking the middle element at any time and for any kinds of array, the Smart Binary Search goes to the position closest to the target which is computed using the formula:

$$pos = low + (key - value_{low}) * \frac{high - low}{value_{high} - value_{low}} \quad (1)$$

This formula is applicable when handling a uniformly distributed array which every value within that array has the same frequency to appear. Assume that the numerical values in the array grow in linear fashion, the above equation can be explained as follows. As changes in y is proportional to changes to x, we have the generic equation as:

$$\Delta y = rate\ of\ change * \Delta x$$

Hence, $rate\ of\ change = \frac{\Delta y}{\Delta x} = \frac{value_{high} - value_{low}}{high - low}$

$$= \frac{value_{target} - value_{low}}{target - low} = \frac{key - value_{low}}{pos - low}$$ [5]

As a result, $\frac{value_{high} - value_{low}}{high - low} = \frac{key - value_{low}}{pos - low}$. By re-structuring this equation, we will get the formula (1) which estimates the position of target key.

The software implementation for this algorithm is pretty similar to the Binary Search in using recursion function. However, there are three adjustments:

- The array is divided from the position computed from mentioned formula instead of from the middle position.
- The recursion will end when the elements at two sides of the array are equal and key is not in [low, high] range. This condition is to ensure that the formula will not output out of range index or negative position.
- Before returning the result, we have to check the equality of the low element and key. This is designated for the case when the high and low elements are the same; thus, the recursion must end without comparing them to the target input. In this situation, as the low and high elements are the same, all elements in middle will also contain the similar value. Therefore, there is no calculation for the new position required to perform the comparison.

```python
def smart_binary_search(arr, key, lo, hi):

    # Make sure the low position is smaller than high position
    # key must be in [low, high] range
    if (arr[hi] != arr[lo] and arr[lo] <= key <= arr[hi]):
        # Calculate position (use uniform distribution property)
        pos = lo + ((hi - lo) // (arr[hi] - arr[lo]) *  (key - arr[lo]))

        # If new position is target
        if arr[pos] == key:
            return pos

        elif arr[pos] < key: # If key is larger than element
            return smart_binary_search(arr, key, pos + 1, hi) # Key is in right subarray

        if arr[pos] > key: # If key is smaller than element
            return smart_binary_search(arr, key, lo, pos - 1) # Key is in left subarray

    # If both of low and high element is key
    if arr[lo] == key:
        return lo
    return -1
```

Figure 30. Code snipping for smart binary search

3.2. Empirical Analysis

Test data evaluation

To ensure the correctness of the analysis, it is necessary to randomly generate arrays which are uniformly distributed. The above histograms present the frequency in appearance of each value ranging from 1 to 1000 in different randomly generated array. As can be seen, the bigger the array size is, the more uniform the distribution is. Hence, for more accurate test result, our input size will be from 100,000.

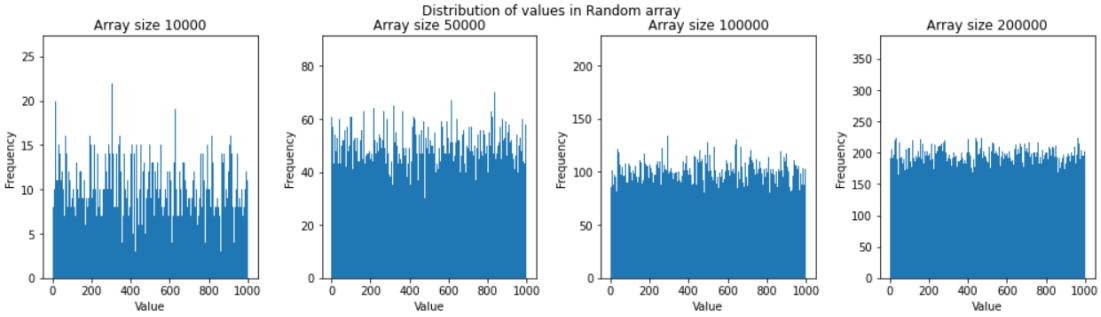


Figure 31. The histograms for distribution of values in Random array

Test description

We create four test scenarios which are:

- Search key which has the middle value
- Search key which is at the beginning of the array
- Search key which is at the end of the array
- Search key which is not available in the array

For each test scenario, we run 5 test cases corresponding to different array sizes. Each test case will be run 5 times repeatedly to get the average time consumed. The input to generate the tested arrays will be:

Table 12. SBS vs BS Test scenario configuration

INPUT
Value range: from 1 to 1000 Sizes: 100000, 200000, 400000, 800000 and 1600000

Test scenario 1 result: Search key which has the middle value

Array size	Key	BS time (us)	Smart BS time (us)
100000	500	0.692	2.207
200000	500	0.626	2.096
400000	500	0.645	2.204
800000	500	0.623	2.284
1600000	500	5.235	1.248

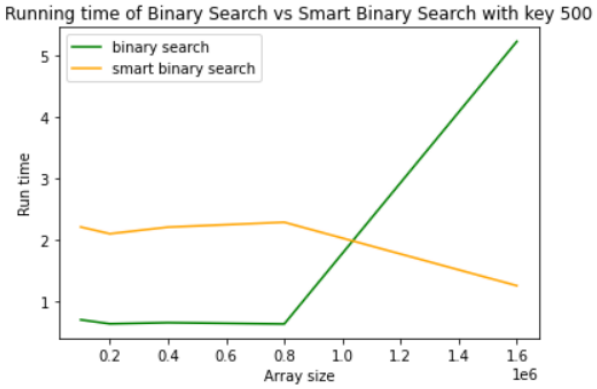


Figure 32. SBS vs BS - Search middle result

As illustrated, the Smart Binary search shows its stability when handling arrays with different lengths. On the other hand, since the Binary Search always get to the middle value by default, its performance in this test case is slightly better than one of Smart Binary Search. However, with the array length of 1,600,000, the Smart Binary Search is faster than the classical one. In general, when finding the value placed in the middle of the array, the performances of two methods have insignificant difference for comparison

Test scenario 2 & 3 result: Search key which is at the beginning and at the end of the array

Array size	Key	BS time (us)	Smart BS time (us)
100000	1	5.370	1.294
200000	1	5.401	1.223
400000	1	6.288	1.255
800000	1	5.238	1.373
1600000	1	5.628	1.261

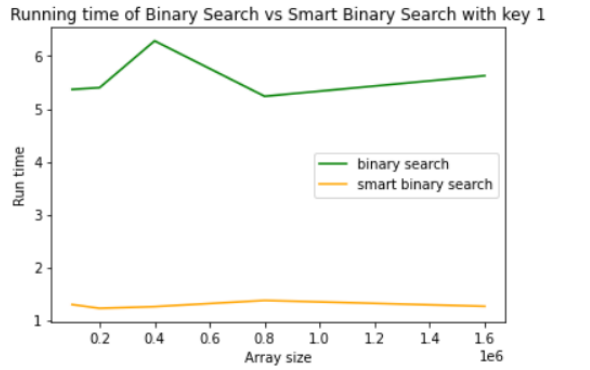


Figure 33. SBS vs BS – Search key at first

Array size	Key	BS time (us)	Smart BS time (us)
100000	999	3.650	1.966
200000	999	4.615	2.511
400000	999	3.782	1.813
800000	999	3.468	0.945
1600000	999	3.616	1.006

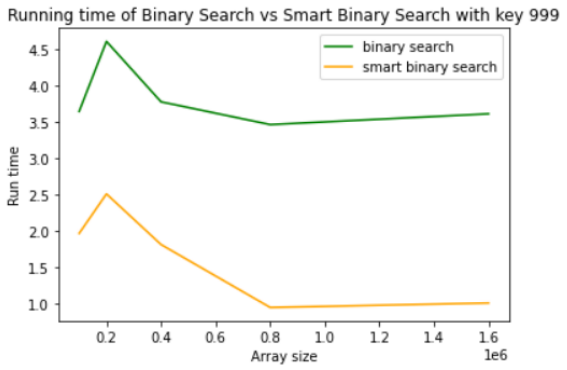


Figure 34. SBS vs BS – Search key at last

Finding the top and bottom element can show a significant difference in the performance of the two methods. In both cases, Smart Binary Search performs better in terms of speed. While Binary Search takes about 3.5 to 6 microseconds, Smart Binary Search improves efficiency when returning results in 1 microsecond. However, it is clear that both methods are not affected by the increase in array size.

Test scenario 4 result: Search key which is not available in the array

Array size	Key	BS time (us)	Smart BS time (us)
100000	1001	8.818	0.789
200000	1001	9.775	2.640
400000	1001	10.517	0.836
800000	1001	12.831	0.893
1600000	1001	12.215	0.858

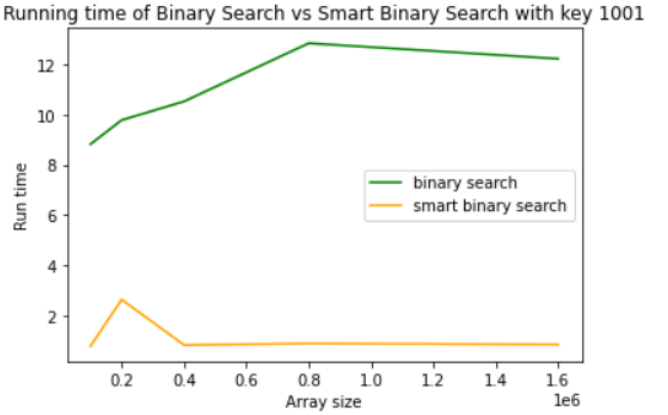


Figure 35. SBS vs BS - Search key not exist

In this test, the Smart Binary Search also executes in faster speed. However, unlike the previous scenario, while the timing result of Smart Binary Search stays constant throughout different lengths, the time complexity of Binary Search tends to increase with respect to array size.

In conclusion, as a result from those test run, the Smart Binary Search is proved to outperform the normal Binary Search in dealing with Uniformly Distributed Array in all cases, excepting the case when the target value is at the middle position which is the best-case scenario of Binary Search.

4. Question 4: Arranging items in the shopping bag

4.1. Among other orderings, which is safe and which is unsafe?

As in this question specification, it is required to order objects in vertical direction such that the strength of one at the bottom must be greater than the total weights of all objects on top of its so that it does not squash. Therefore, the general condition to arrange items is:

$$s_i \geq \sum_{i-1}^{1} w$$

Where: s_i is the strength value of an item

$\sum_{i-1}^{1} w$ is the total weights of items that sit on top of the current item.

For example, given three objects with corresponding weights and strengths as in the table:

Table 13.Bread-Potatoes-Apple information

OBJECT	WEIGHT	STRENGTH
Bread	4	4
Potatoes	12	9
Apple	5	6

Since there are three objects that need to be sorted, there is $3! = 6$ placement possibilities. As the requirement mentioned two cases in which Apple -> Bread -> Potatoes is a safe order while Bread -> Apple -> Potatoes is not unsafe order. The question now lays upon whether there is any safe order left in other four orderings. However, the potatoes have weight as 12 which is greater than two other objects' strengths (4 and 6); hence, there is no safe orderings that Potatoes can sit on top of Bread or Apple. The detail explanation is presented as follows.

Case 1:

Table 14. Object Arrangement 1

POSITION	OBJECT	WEIGHT	STRENGTH
Top	Bread	4	4
Middle	Potatoes	12	9
Bottom	Apple	5	6

This is an unsafe arrangement as Apple with a strength of 6 cannot carry both Potatoes and Bread with weight of 12 + 4 = 16.

Case 2:

Table 15. Object Arrangement 2

POSITION	OBJECT	WEIGHT	STRENGTH
Top	Apple	5	6
Middle	Potatoes	12	9
Bottom	Bread	4	4

This is an unsafe arrangement as Bread with a strength of 4 cannot carry both Potatoes and Apple with weight of 12 + 5 = 17.

Case 3:

Table 16. Object Arrangement 3

POSITION	OBJECT	WEIGHT	STRENGTH
Top	Potatoes	12	9
Middle	Bread	4	4
Bottom	Apple	5	6

This is an unsafe arrangement as Bread with a strength of 4 cannot carry Potatoes with weight of 12 and Apple with a strength of 6 cannot carry both Potatoes and Bread with weight of 12 + 4 = 16.

Case 4:

Table 17. Object Arrangement 4

POSITION	OBJECT	WEIGHT	STRENGTH
Top	Potatoes	12	9
Middle	Apple	5	6
Bottom	Bread	4	4

This is an unsafe arrangement as Apple with a strength of 6 cannot carry Potatoes with weight of 12 and Bread with a strength of 4 cannot carry both Potatoes and Apple with weight of 12 + 5 = 17.

In sum up, the other four orderings are all unsafe orders.

4.2. Will ordering the heaviest item at the bottom always produce a safe arrangement?

As in the previous example, it can be seen that the only safe arrangement is to place Apples, Bread, and Potatoes from top to bottom, resulting in an increasing order of weight. Hence, to find out whether placing items in the order of their weight can always create a safe arrangement without squashing any items, let's invent two different items and place the heavier at the bottom as an example.

Table 18. Arrangement in weight order

POSITION	OBJECT	WEIGHT	STRENGTH
Top	Item 1	10	12
Bottom	Item 2	12	9

From this example, it is shown that packing the items by weight order is not the absolute arranging methodology since the Item 2 will be squashed when it just can handle weight up to 9 but the Item 1's weight is 10. As a conclusion, it is reasonable that if only taking the order of weight into consideration, we cannot get the solution for this problem that can apply into all cases since the problem requires to sort items according to two distinct categories – weight and strength – concurrently.

4.3. Will ordering the strongest item at the bottom always produce a safe arrangement?

Similar to the question above, let's invent two different items and place the stronger at the bottom as an example.

Table 19. Arrangement in strength order

POSITION	OBJECT	WEIGHT	STRENGTH
Top	Item 1	13	10
Bottom	Item 2	12	12

This is an unsafe arrangement as Item 2 with a strength of 12 cannot carry Item 1 with weight of 13. Hence, like the weight sorting, using the strength grading approach will not fulfill all possible cases. Rather than focusing merely on one attribute, it is more ideal to come up with a strategy that makes use of two given attributes in conjunction when deciding how to do the arrangement properly.

4.4. When item j sits directly on item i safely and (weight of j) – (strength of i) ≥ (weight of if) – (strength of j). Can we swap items i and j and still remain the safe arrangement?

As discussed, in order to arrange items in a safe order, every item must follow this condition:

$$s_i \geq \sum_{i-1}^{1} w$$

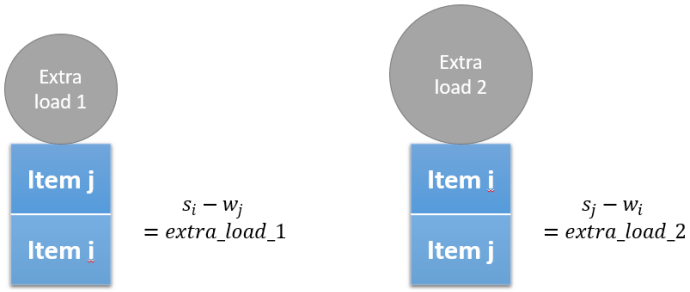
Therefore, if we subtract to get the difference between the strength of the item s_i and the weight it is currently carrying $\sum_{i-1}^{1} w$, we can calculate how much load the item can further process. When the ordering is safe which means $s_i - \sum_{i-1}^{1} w = extra\_load \geq 0$, the remain load ranges from 0 to infinity. Otherwise, if $s_i - \sum_{i-1}^{1} w = extra\_load < 0$, the arrangement is unsafe.

In our case, called:

s_i and w_i is the strength and weight of item i.

s_j and w_j is the strength and weight of item j.

We are given the hypothesis $w_j - s_i \geq w_i - s_j$ which can be re-written as $s_i - w_j \leq s_j - w_i$ (1). According to above discussion, $s_i - w_j$ can be understand as the remaining load that item i can further handle when we place item j on top of item i. Similarly, $s_j - w_i$ is the remaining load when we place item i on top of item j. Therefore, the hypothesis shows that if we put item j at the bottom, we can put more load than when item i was at the bottom.



The question states that item j can sit directly on item i safely. Therefore,

$$s_i \geq w_j \text{ and } s_i - w_j = extra\_load\_1 \geq 0 \text{ (2)}$$

From two above mathematical statement, we can derive that:

$$(1)(2) \Rightarrow s_j - w_i = extra\_load\_2 \geq 0 \text{ (3)}$$

Therefore, item j has enough strength to carry item i without being squashed

Furthermore, in generic situation, the two items i and j can be at the middle of the arrangement. Therefore, there will be some items at the top which require i and j to carry and those at the bottom which must be able to handle item i and j weight. However, since the total weight $w_i + w_j$ is unchanged when swapping two items, the bottom part can be ignored.

Called: $\sum w$ is the total weight of all items sitting on top of item i and j

As been given that item j can sit directly on item i safely, the strength of item i must be big enough to carry the weight of all above items and item j. Therefore,

$$s_i - w_j = extra\_load\_1 \geq \sum w$$

Also, as $s_i - w_j \leq s_j - w_i$, we have:

$$s_j - w_i = extra\_load\_2 \geq \sum w \rightarrow s_j \geq \sum w + w_i$$

In conclusion, for all cases, when item j sits directly on item i safely and (weight of j) – (strength of i) ≥ (weight of i) – (strength of j), swapping items i and j still remains the safe arrangement.

4.5. Practical arranging method

Algorithm Inspiration

As a summary of the lessons drawn previously,

- In section 4.2 and 4.3, the method of arranging items by taking one attribute either weight or strength order can sometimes produce an inaccurate arrangement. Therefore, it opts for combining both weight and strength values in sorting items

- In section 4.4, it was pointed out that when $s_i - w_j \leq s_j - w_i$, the positions of item i and j can be safely interchanged. Also, while carrying another item, the item j can bear more weight compared to the item i as $extra\_load\_2 > extra\_load\_1$. Therefore, among two items that can be interchangeable, it is reasonable to put the item that can further bear more load underneath which is item j in this case. As can be seen from the re-arranged version of given formula $strength_j + weight_j \geq strength_i + weight_i$, the sum of strength and weight of item j is greater and one of item i. As a result, it opts to sort items in $(weight + strength)$ order

Algorithm Description

The item list will be sorted based on the sum of weight and strength in reverse order such that the first element is the one with the greatest value. Then, we add that greatest item to the end of the shopping bag. While iterating through all items, we keep track of the remaining strength that the items underneath can bear. Whenever detecting an item with weight exceed the limit of below items, the program will exit and return None; thus, indicating that there is no possible arrangement for the input list. Otherwise, when the weight is acceptable, item will kept be inserted to the top of the bag. Concurrently, the minimum strength will be updated because as adding more items, the load capacity will be reduced. In addition, in case the new inserted item has small strength, the minimum strength will also be adjusted.

```
def arrange_items(items):
  items.sort(key=lambda x: x.sum, reverse=True)

  bag = [items[0]]
  min_strength = items[0].strength

  for item in items[1:]:
    if (min_strength >= item.weight):
      bag.insert(0, item)
      min_strength -= item.weight
      if (item.strength < min_strength):
        min_strength = item.strength
    else:
      return None
  return bag
```

Figure 36. Code snipping for arranging items function

```
item_list = [Item(4, 3), Item(3, 6), Item(1, 9), Ite
bag = arrange_items(item_list)
if (bag is None):
  print("No possible arrangement")
else:
  for item in bag:
    item.to_string()
```

```
Item: 2 2
Item: 4 3
Item: 3 6
Item: 1 9
```

Figure 37. Arranging program test 1

```
item_list = [Item(4, 3), Item(3, 6), Item(1, 8), Item(2, 2)]
bag = arrange_items(item_list)
if (bag is None):
  print("No possible arrangement")
else:
  for item in bag:
    item.to_string()
```

```
No possible arrangement
```

Figure 38. Arranging program test 2

5. Reference

[1] "Search an element in a sorted and rotated array", GeeksforGeeks, 2020. [Online]. Available: https://www.geeksforgeeks.org/search-an-element-in-a-sorted-and-pivoted-array/.

[2] "Search an element in a sorted and rotated array with duplicates", GeeksforGeeks, 2020. [Online]. Available: https://www.geeksforgeeks.org/search-an-element-in-a-sorted-and-rotated-array-with-duplicates/.

[3] "Given a sorted array and a number x, find the pair in array whose sum is closest to x", GeeksforGeeks, 2020. [Online]. Available: https://www.geeksforgeeks.org/given-sorted-array-number-x-find-pair-array-whose-sum-closest-x/.

[4] "Minimum Number of Platforms Required for a Railway/Bus Station", GeeksforGeeks, 2020. [Online]. Available: https://www.geeksforgeeks.org/minimum-number-platforms-required-railwaybus-station/.

[5] "Interpolation Search", GeeksforGeeks, 2020. [Online]. Available: https://www.geeksforgeeks.org/interpolation-search/