

SCHOOL OF COMPUTER SCIENCE

UNIVERSITI SAINS MALAYSIA

CPT212 : Design and Analysis of Algorithm

Semester 2, Academic Session: 2019/2020

Lecturer : Dr. Teh Je Sen

|  |  |  |
| --- | --- | --- |
| Name | Matric Number | Task Division |
| Chan Siang Sheng | 142413 | Problem Requirement, Algorithm & Pseudocode, Result & Discussion |
| Tang Wen Shuen | 142623 | Algorithm & Pseudocode, Result & Discussion |

**Table of Content**

|  |  |  |
| --- | --- | --- |
| No | Content | Page |
| 1 | 1. Problem Requirement | 2 |
| 2 | 1. Solution Approach    1. Algorithm of Big-O (*n*)    2. Pseudocode of Big-O (*n*)    3. General Idea of Big-O (*n*)    4. Algorithm of Big-O ()    5. Pseudocode of Big-O ()    6. General Idea of Big-O () | 3  7  7  8  9  9 |
| 3 | 1. Result and Discussion    1. Screenshots       1. Experiment 1       2. Experiment 2    2. Graphs       1. Experiment 1          1. Algorithm of Big-O (*n*)          2. Algorithm of Big-O ()          3. Algorithm of Big-O (*n*) vs Big-O ()          4. Summary of Experiment 1       2. Experiment 2          1. Algorithm of Big-O (*n*)          2. Algorithm of Big-O ()          3. Algorithm of Big-O (*n*) vs Big-O ()          4. Summary of Experiment 2    3. Experiment 1 vs Experiment 2 | 10  10  11  12  13  13  14  14  15  16  16 |
| 4 | References | 17 |

1. **Problem Requirement**

Problem:

Squares of a Sorted Array. An input of list consisting of only integers in ascending order. The integers can be in negative and positive, thus making the list ascending from negative integers to positive integers. The integers will be randomly generated from -*n* to *n* in the list where *n* is the size of the list. Print a list of squared of the input list in an ascending order.

Input:

*n* = input size of the list

Constraint:

1 *≤ n ≤* 100 000

-*n ≤ ar [i] ≤ n*

Output:

A list of squared of the input list in ascending order.

Sample Input:

*n* =10

*ar* = [-7, -6, -3, -2, 0, 1, 4, 5, 6, 8]

Expected Sample Output:

[0, 1, 4, 9, 16, 25, 36, 36, 49, 64]

Explanation:

1. 0 is the smallest integer after squaring 0 from the input list.

[0]

1. 1 is the second smallest integer after squaring 1 from the input list.

[0, 1]

1. 4 is the next smallest integer after squaring -2 from the input list.

[0, 1, 4]

1. 16 is the next smallest integer after squaring -3 from the input list.

[0, 1, 4, 9]

1. 16 is the next smallest integer after squaring 4 from the input list.

[0, 1, 4, 9, 16]

1. This repetition continues until 64 which is the square of 8 that is greatest in the input list.

[0, 1, 4, 9, 16, …, 64]

**2.0 Solution Approach**

In this problem, we have identified 2 algorithms to solve this question. One of the algorithms is O(*n*) and the other is O(). Both algorithms are written in Python 3 and programmed in .py format. The explanation of both algorithms will be done with the following sample input:

* *n* = 10
* *ar* = [-7, -6, -3, -2, 0, 1, 4, 5, 6, 8]

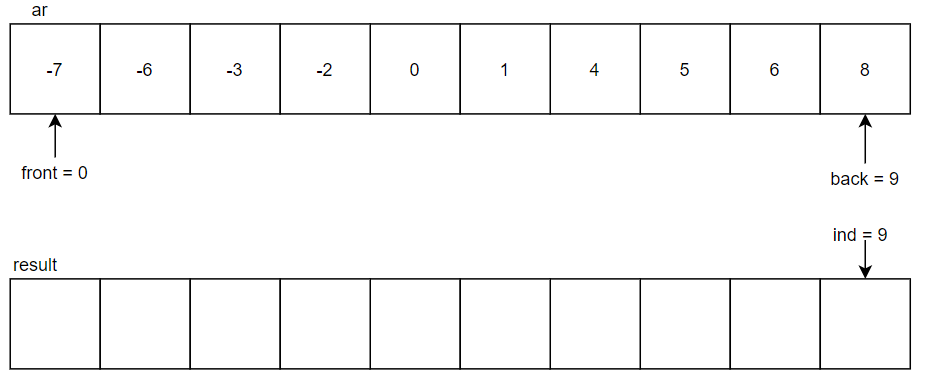
**2.1 Algorithm of Big-O (*n*)**

This algorithm will be assisted with 3 variables as well as 1 list for output as stated below:

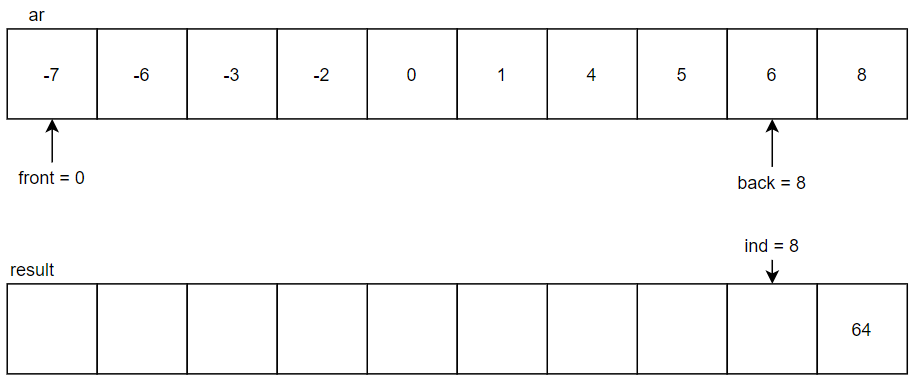
* *front*, an integer variable that point at the index of first element from the input list that has yet to be square
* *back*, an integer variable that point at the index of last element from the input list that has yet to be square
* *ind*, an integer variable that point at the last index of output list that to be inserted by integer after squaring it.
* *result*, an output list with same input size or length as input list

Steps:

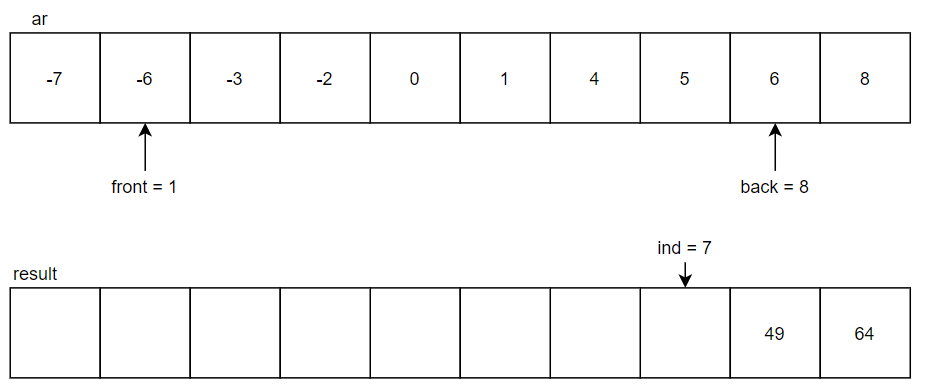
1. The initial state of algorithm before entering first cycle of while loop is shown as below:



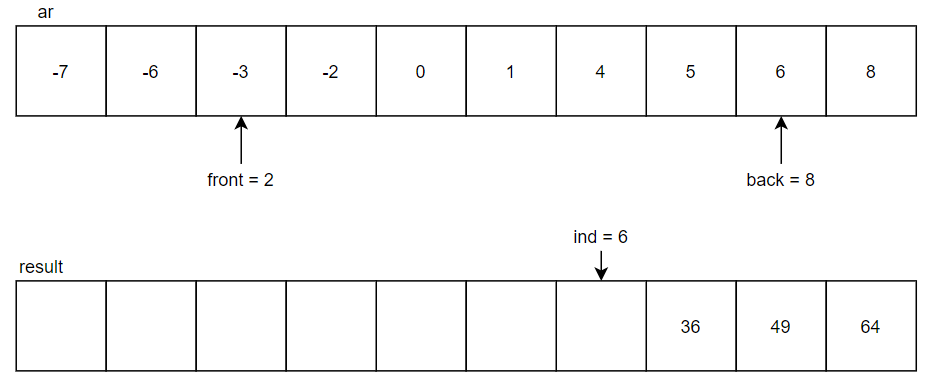
1. In the 1st cycle of while loop, -7 in the index 0 will be taken with absolute and compares with 8 in the index 9. Since 8 is greater than 7, 8 will be squared to 64 and put into the index 9 of the result list. The ***back*** will be decremented to point at index 8 and ***ind*** will be decremented to point at index 8 while there is no change to ***front***.



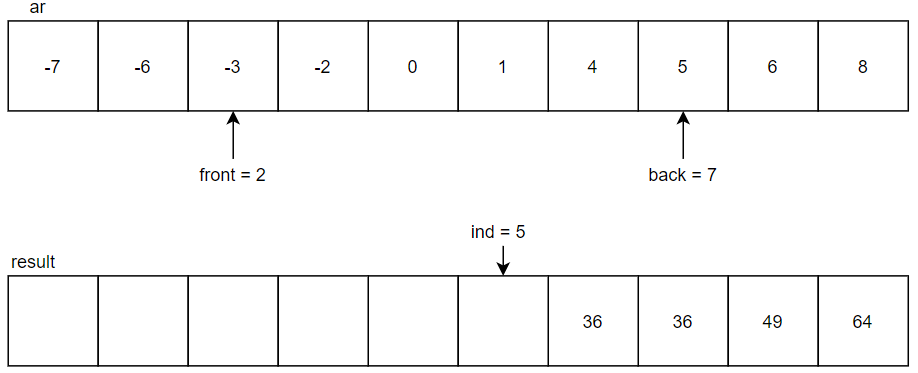
1. In the 2nd cycle, -7 in the index 0 will be taken with absolute and compares with 6 in the index 8. Since 7 is greater than 6, 7 will be squared to 49 and put into the index 8 of the result list. The ***front*** will be incremented to point at index 1 and ***ind*** will be decremented to point at index 7, while ***back*** will remain no change.



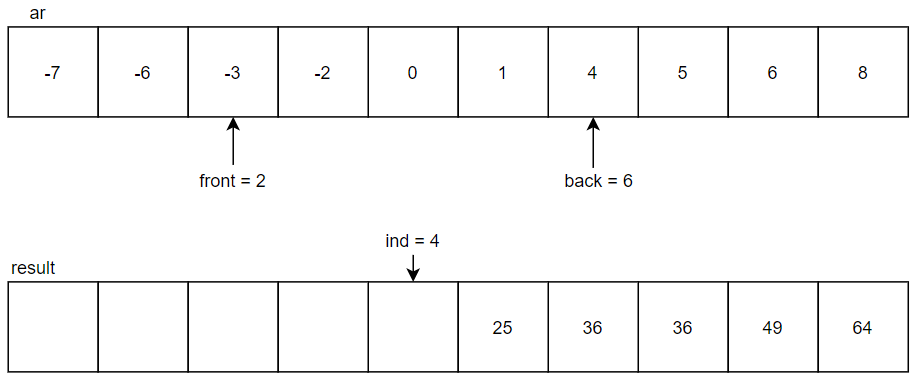
1. In the next cycle, -6 in the index 1 will be absolute and compares with 6 in the index 8. Since both are 6, 6 in index 1 will be squared to 36 and put into the index 7 of the result list. The***front*** will be incremented to point at index 2 and ***ind*** will be decremented to point at index 6.



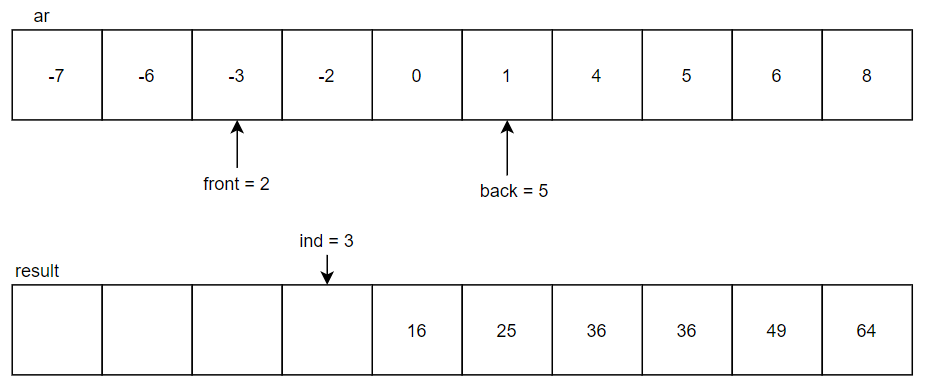
1. In the next cycle, -3 in the index 2 will be absolute and compares with 6 in the index 8. Since 6 is greater than 3, 6 will be squared to 36 and put into the index 6 of the result list. The ***back*** will be decremented to point at index 7 and ***ind*** will be decremented to point at index.



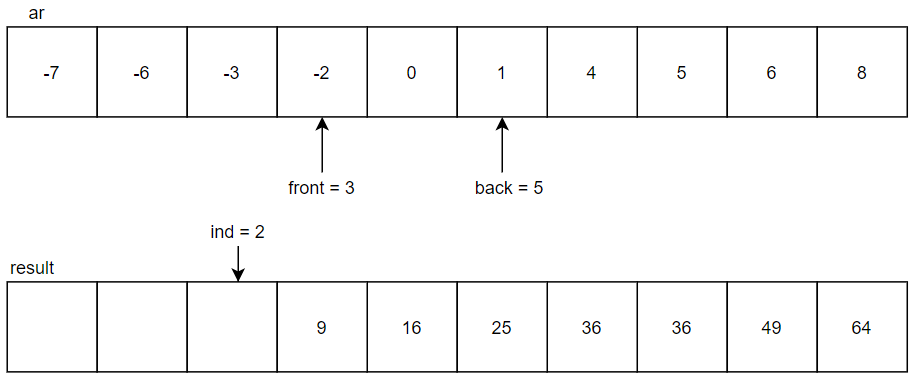
1. In the next cycle, -3 in the index 2 will be absolute and compares with 5 in the index 7. Since 5 is greater than 3, 5 will be squared to 25 and put into the index 5 of the result list. The ***back*** will be decremented to point at index 6 and ***ind*** will be decremented to point at index 4.



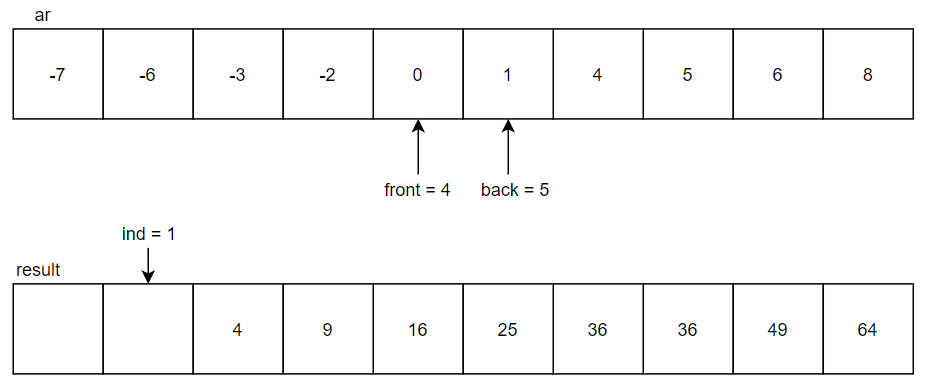
1. In the next cycle, -3 in the index 2 will be absolute and compares with 4 in the index 6. Since 4 is greater than 3, 4 will be squared to 16 and put into the index 4 of the result list. The ***back*** will be decremented to point at index 5 and***ind*** will be decremented to point at index 3.



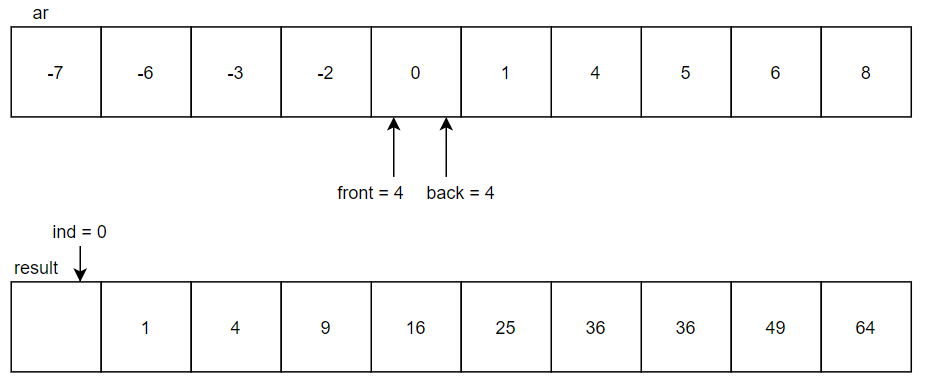
1. In the next cycle, -3 in the index 2 will be absolute and compares with 1 in the index 5. Since 3 is greater than 1, 3 will be squared to 9 and put into the index 3 of the result list. The***front*** will be incremented to point at index 3 and ***ind*** will be decremented to point at index 2.



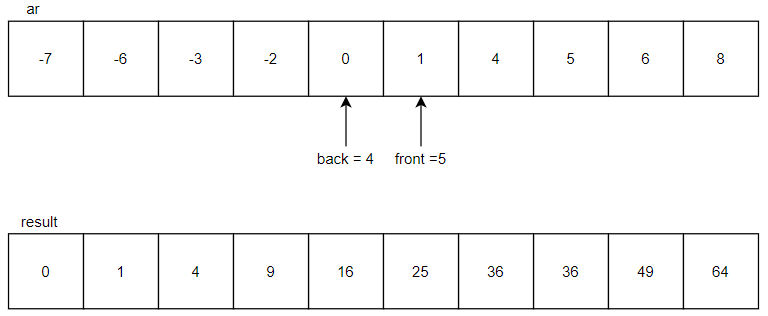
1. In the next cycle, -2 in the index 3 will be absolute and compares with 1 in the index 5. Since 2 is greater than 1, 2 will be squared to 4 and put into the index 2 of the result list. The ***front*** will be incremented to point at index 4 and ***ind*** will be decremented to point at index 1.



1. In the next cycle, 0 in the index 4 will be compared with 1 in the index 5. Since 1 is greater than 0, 1 will be squared to 1 and put into the index 1 of the result list. The ***back*** will be decremented to point at index 4 and***ind*** will be decremented to point at index 0.



1. In the next cycle, 0 in the index 4 will be compared with 0 in the index 4. Since both are 0, 0 will be squared to 0 and put into the index 0 of the result list. The ***front*** will be incremented to point at index 5 and ***ind*** will be decremented to point at index -1. When ***ind*** is lower than 0, the while loop ends at here with 10th time of cycles. As such, the result list is then completed.



**2.2 Pseudocode of Big-O (*n*)**

*n* ← input size

*ar* ← list with random integer between -*n* to *n*

*ar* ← sorting into ascending order

function bign(*ar*)

*front* ← 0

*back* ← size of list - 1

*ind* ← size of list - 1

**while** *ind* ≤ 0 **do**

**if** absolute of *ar*[*front*] ≥ *ar*[*back*] **then** #comparing front and behind part elements

*ans*[*ind*] ← *ar*[*front*]*\** *ar*[*front*] #square element then add into result list

*front* ← *front* + 1

**else if** absolute of *ar*[*front*] < *ar*[*back*] **then** #comparing front and behind part elements

*ans*[*ind*] ← *ar*[*back*]\* *ar*[*back*] #square element then add into result list

*back* ← *back* - 1

*ind* ← *ind* - 1

**print** *ans #*print result list

**2.3 General Idea of Big-O (*n*)**

A simplified Big-O (n) solution is done by :-

1. Comparison between front part of element and behind part of element in the list.
2. The greater value element will be squared and inserted into the result list.

If both elements have the equal values, the front part element will still be chosen instead.

1. Repeat step 1 and 2 until the all element are squared and added into the result list.

Comparison

Square & Insertion

Repeat

**2.4 Algorithm of Big-O (*)***

This algorithm is also named as Selection Sort and it requires 1 variable and 1 list for output as stated below:

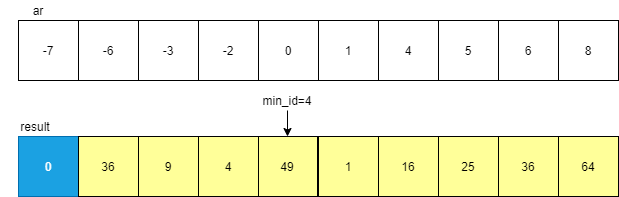
* *min\_id*, a variable that holds the index of the smallest value element in the unsorted sub list
* *result*, an output list with same input size or length as input list

Steps:

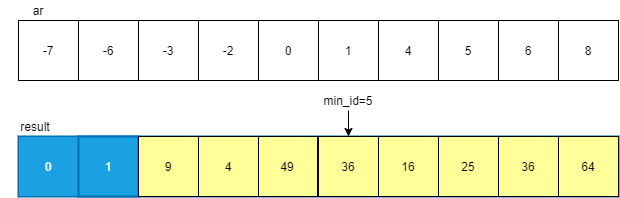
1. Before entering the first loop, every element in the original list is squared and put into the ***result*** list. Blue boxes of the result list will be represented as sorted sub list while yellow boxes of the result list are unsorted sub list.



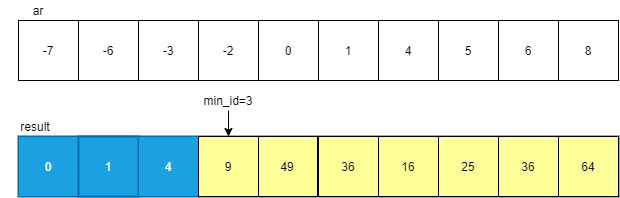
1. In the first loop, ***min\_id*** is set to be at index 0 initially which has the value of 49. Then, ***min\_id*** is set to be at index 4 since 0 is the smallest element in the unsorted sub-list. At the end of the loop, the elements in index 0 and index 4 are swapped. Noted that left side sorted sub list has 1 element now and unsorted sub list has 1 lesser element.



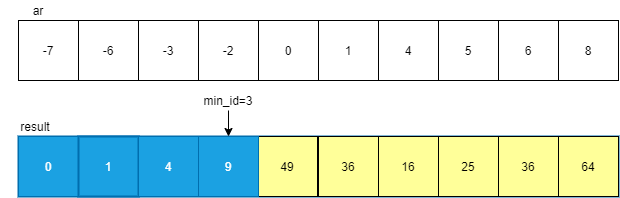
1. In the second loop, ***min\_id*** is set to be at index 1 initially. Then, ***min\_id*** is set to be at index 5 since 1 is the smallest element in the unsorted sub-list. At the end of the loop, the element in index 1 and index 5 are swapped.



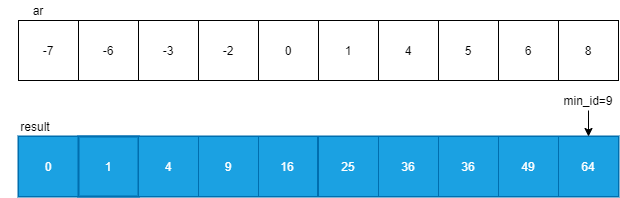
1. In the third loop, ***min\_id*** is set to be at index 2 initially. Then, ***min\_id*** is set to be at index 3 since 2 is the smallest element in the unsorted sub-list. At the end of the loop, the element in index 2 and index 3 are swapped.



1. In the fourth loop, ***min\_id*** is set to be at index 3 initially. Then, ***min\_id*** remain unchanged since 9 is the smallest element in the unsorted sub-list. At the end of the loop, the element in index 3 and index 3 are swapped.



1. Similar operations are continued until the whole list has blue boxes which indicate that all elements are sorted as shown below.



**2.5 Pseudocode of Big-O ()**

*n* ← input size

*ar* ← list with random integer between -*n* to *n*

*ar* ← sorting into ascending order

function bignsquare(*ar*)

**for** *i* ← 0 **to** *n*-1 **do**

*ans[i]* ← *ar[i]*\* *ar[i] #*square all element of list

**for** *i* ← 0 **to** *n*-1 **do**

*min\_id* ← *i*

**for** *j* ← *i* + 1 **to** *n*-1 **do**

**if** *ans[i] < ans[min\_id]* **then** #comparison between current lowest integer

*min\_id* ← *j* with current iterating element

*temp* ← *ans[i] #*swapping process

*ans[i]* ← *ans[min\_id]*

*ans[min\_id]* ← *temp*

**print** *ans #*print result list

Square & Insertion

**2.6 General Idea of Big-O ()**

This Big-O () solution is also named as Selection Sort and can be simplified by: -

1. Squares all element of list then only add into result list.
2. Find smallest integer and swapped position into the left side of result list.

Sorting

1. Repeat step 2 until no more element at the right side of result list.

**3.0 Results and Discussion**

In the both experiments, our input size is limited to *n* = 20 000 at maximum due to the facts that it will takes a much longer time to run the algorithm of Big-O () when *n* is even bigger ( >20 000).

**3.1 Screenshot**

**3.1.1 Experiment 1**

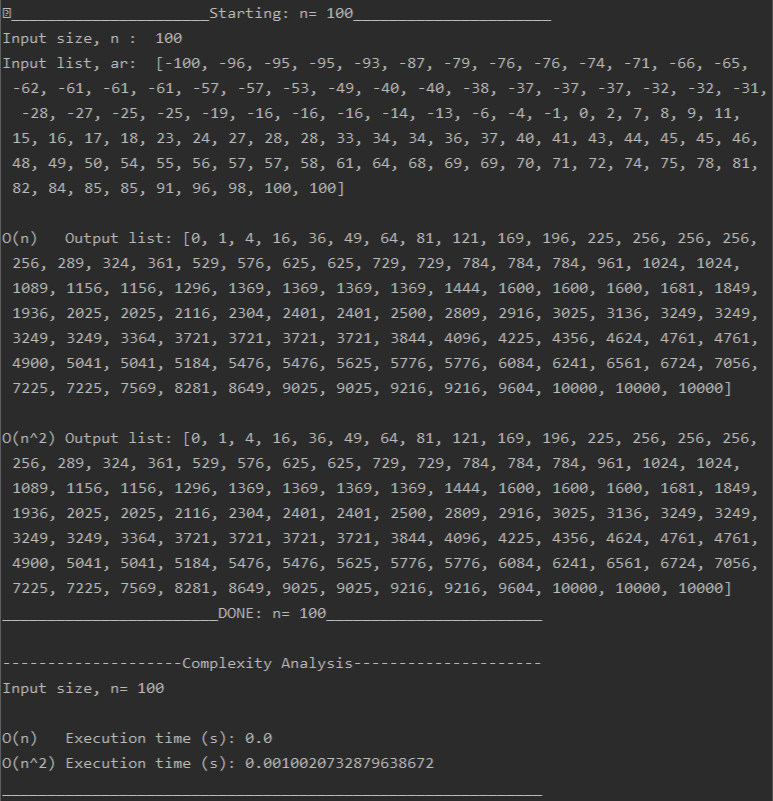


Figure 1.0: Sample output of the time execution(s) required for both algorithm of Big-O (*n*) and Big-O ()

when *n* =100 and the list is in the random order.

**3.1.2 Experiment 2**

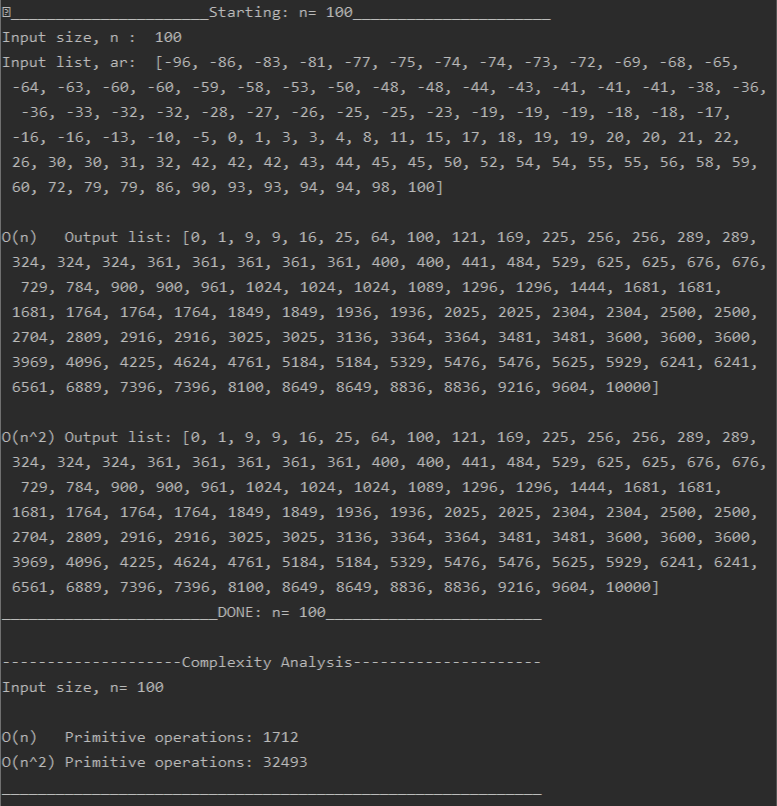


Figure 2.0: Sample output of the number of primitive operations required for both algorithm of Big-O (*n*) and Big-O () when *n* =100 and the list is in a random order.

**3.2 Graph**

All the tests in this assignment are ran on 2 different devices. One with CPU model of Intel i5-8300h and 8GB of RAM, and another with CPU model of Intel i7-8750h and 12GB of RAM. The first devices will be named as Device A whereas the another one will be named as Device B in the later discussion.

**3.2.1 Experiment 1**

**3.2.1.1 Algorithm of Big-O (*n*)**

Figure 3.0: Plotted graph for execution time (in seconds) to run the Big-O (*n*) algorithm

for different value of *n* and type of case on Device A.

Figure 4.0: Plotted graph for execution time (in seconds) to run the Big-O (*n*) algorithm

for different value of *n* and type of case on Device B.

Based on Figure 3.0 and Figure 4.0, we can see that at the most value of *n* and in worst case, Device B have a shorter execution time than Device A. For example, when *n* =20000, Device A required 0.1522s to run the whole program meanwhile Device B required only 0.02297s. This is due to the reason that Device B have a better CPU and more available physical memory to perform faster arithmetic operations and memory allocation for the program execution.

In term of the runtime required for random, best and worst case, we can see that worst case required the longest time among all. Meanwhile, best case required the shortest time and random case’s runtime is between best case and worst case on Device B. Device A shows that best case required longer time than random case. This might due to the external factor such as hardware and software capability or process running on the background of the system. Regardless, the difference is just by around 10ms which is insignificant to be considered in real time.

**3.2.1.2 Algorithm of Big-O (****)**

Figure 5.0: Plotted graph for execution time (in milliseconds) to run the Big-O () algorithm

for different value of *n* and type of case on Device A.

Figure 6.0: Plotted graph for execution time (in milliseconds) to run the Big-O () algorithm

for different value of *n* and type of case on Device B.

Based on Figure 5.0 and Figure 6.0, we can see that at the most value of *n* and in the worst case, Device B have a shorter execution time than Device A. For example, when *n* =20000, Device A required 26.55s to run the whole program meanwhile Device B required only 23.08s. This is due to the reason that Device B have a better CPU and more available physical memory to perform faster arithmetic operations and memory allocation for the program execution.

In term of the runtime required for random, best and worst case, we can see that worst case required the longest time among all. Meanwhile, best case generally required the shortest time and random case’s runtime is between best case and worst case.

**3.2.1.3 Algorithm of Big-O (*n*) vs Big-O ()**

Figure 7.0: Plotted graph for average execution time (in seconds) to run the Big-O (*n*) and Big-O () algorithms

in the worst case at different value of *n* on Device A and Device B

To make the result seems fair, we had used the average time needed to run the program in worst case at different input size on both devices to plot the graph as shown in Figure 7.0.

Based on Figure 7.0, we can see that Big-O () required significantly longer time than Big-O (*n*) to execute the program when the input size is getting larger and larger. In term of runtime required, algorithm of Big-O (*n*) is significantly better than algorithm of Big-O (). The curve of Big-O (*n*) can be seen as linear while the curve of Big-O ( can be seen as increasing exponentially.

**3.2.1.4 Summary of Experiment 1**

Throughout Experiment 1, we conclude that having a device with greater specification such as CPU and RAM can greatly increase the process of executing program and thus reducing execution time needed to complete all the task. Therefore, having different results in this experiment between devices.

Besides that, Big-O () has greater growth rate. Big-O () curve growth increasing quadratically while Big-O (*n*) curve grow linearly. Big-O (*n*) algorithm provides better efficiency than Big-O (). Big-O (*n*) has lesser execution time while Big-O () algorithm takes much longer execution time on both different specification devices. In term of time complexity, we can see that Big-O (*n*) execution time grows nearly linearly but not exactly throughout all size of input. Similar result can also be seen on Big-O () except it grows nearly quadratically.

By calculating time execution, we can analyze that even with best case input, the execution time can be a bit longer than worst case or average case. Theoretically, best case should have the shortest execution time following by average case then worst case. However, in this experiment, the theory didn’t follow as expected and this could be due to external factor of the devices. Nevertheless, time execution can show or tell user about how much time required for the program to complete the task in real time but it might not be the ideal analysis for differentiating efficiency of algorithms.

**3.2.2 Experiment 2**

**3.2.2.1 Algorithm of Big-O (*n*)**

Figure 8.0: Plotted graph for number of primitive steps against input size, *n* for algorithm of Big-O (*n*) on both devices

Based on Figure 8.0, the results are the same on both devices since the number of primitive steps is not affected by the device’s specification. From the graph, we can see a linear pattern throughout all of the input size of *n*. We found out the linear pattern follows the function of 17*n* +12. Let *f* (*n*) and *g* (*n*) be two functions, *f* (*n*) = 17n + 12 and is order of *g* (*n*), where *g (n)* = 30*n*. *f* (*n*) g(*n*) when n ≥ 1 and thus *f* (*n*) is big-O of *g* (*n*).

**3.2.2.2 Algorithm of Big-O ()**

Figure 9.0: Plotted graph for number of primitive steps against input size, *n*

for algorithm of Big-O () for both cases on Device A.

Figure 10.0: Plotted graph for number of primitive steps against input size, *n* for algorithm of Big-O () for both cases on Device B.

Based on Figure 9.0 and Figure 10.0, the results are the same on both devices since the number of primitive steps is not affected by the device’s specification. From the graph, we can see a quadratic pattern throughout all of the input size of n. We found out the function of the graph is for best case and for worst case. Let *f* (*n*) and *g* (*n*) be two functions, *f* (*n*) = and is order of *g* (*n*), where *g* (*n*) = . *f* (*n*) ≤ *g* (*n*) when n ≥ 3 and thus *f* (*n*) is big-O of *g* (*n*).

The reason why there are best case and worst case for this algorithm is because of the following line in the algorithm:

**if** *ans[i] < ans[min\_id]* **then**

*min\_id* ← *i*

For the best case, the if statement body will not be executed throughout the whole program execution, due to the sequence of the list will always be in ascending order. Meanwhile for the worst case, the if statement body will be executed for times throughout the whole program execution.

**3.2.2.3** **Algorithm of Big-O (*n*) vs Big-O ()**

Figure 11.0: Plotted graph for average number of primitive steps against input size, *n* for algorithm of Big-O (n) and worst-case Big-O () on both devices.

To make the algorithm results seems more valid, we assumed the worst case for the algorithm of the Big-O ().

Based on Figure 11.0, we can see that the number of primitive steps involved in the algorithm of Big-O () is significantly greater than Big-O (n). In term of the primitive steps involved, the algorithm of Big-O () is worse than Big-O (*n*) in solving the same problem.

**3.2.2.4 Summary of Experiment 2**

Throughout Experiment 2, we conclude that having devices with different specification such as CPU and RAM does not affect in the experiment. The result in this experiment on both devices are the same.

Besides that, Big-O (*n*) curve grow linearly while Big-O () curve growth increasing quadratically and still has greater growth rate. Big-O (*n*) algorithm still provides better efficiency. This is due to Big-O (*n*) has lesser primitive operation counts while Big-O () algorithm has greater primitive operation counts on both different specification devices. By calculating primitive operations, we can analyze that Big-O (*n*) algorithm has the equal amount of primitive operations on average, best and worst case. Due to the good implementation of Big-O (*n*) algorithm, the primitive operation counts will always only depend on the input size, *n*, the sequence of the integers in input doesn’t affect it.

Meanwhile, Big-O () algorithm has different results on primitive operation counts of best case, average case and worst case. We can analyze that best case is the minimum value while worst case is the maximum value of primitive operation counts range, where average case will be within the range. As such, best case and worst case play much more important role in this. Due to nested for loop implementation with if comparison condition in Big-O () algorithm, best case and worst case will have different value of primitive operation counts. In such circumstances, worst case of both algorithms is the best choice for analysis. Nevertheless, Big-O (*n*) algorithm still has the upper hand and the better algorithm.

In term of time complexity, we can clearly see that Big-O (*n*) follows exactly a linear pattern for all size of input meanwhile Big-O () follows exactly a quadratic pattern for all size of input.

Conclusively, primitive operations count is the ideal analysis perspective in terms of efficiency of algorithms in different devices, the result will remain the same even with huge differences of specification devices. However, no matter how huge the differences of primitive operation counts between these algorithms and no matter how large the number will be, user will never know how much the algorithm will actually impact in real time.

**3.3 Experiment 1 vs Experiment 2**

In general, both Experiment 1 and Experiment proved that the algorithm of Big-O (*n*) is significantly better than Big-O () since it required shorter time of execution and fewer primitive operation steps are involved. We can also infer that the algorithm of Big-O () have a greater growth rate than Big-O (*n*).

In term of the algorithm efficiency analysis, Experiment 1 is less ideal to represent the growth rate of an algorithm since it is greatly affected by the hardware and software capability of the device used and external factor could affect the runtime of the program which hardly can be controlled. We can see that from the perspective that Device A and Device B have different execution time for the program although it is the same program. On the other hand, Experiment 2 can represent the growth rate of an algorithm effectively since it is not bound to the hardware or software capability of the device used. The result will be the same no matter which how different the specification of devices is used to run the same program. The result in Experiment 2 is much consistency and machine-independent and it can provide us insight about how much an algorithm efficiency is, compared to another algorithm.

In term of input cases analysis, Experiment 1 provide us little insight on how a different case of input will be related to an algorithm. The execution time results are not as accurate as expected from the theory of best case will have the shortest execution time following by average case and worst case. Due to external factor of the devices also related to the execution of program. Meanwhile, Experiment 2 did provide us how a well implemented algorithm could impact input cases. It is assumed that a complicated and much complex algorithm will have different results with different input cases, on the other hand, a well implemented algorithm could have consistent result with different input cases. This theory doesn’t always spot on, but the most important insight is this is reasoning why we should use asymptotic notation with input cases in algorithms. With asymptotic notation, we can determine the best, worst cases and most optimal of an algorithm is.

In a conclusion, Experiment 2 is the better choice in terms of algorithm analysis, while Experiment 1 could be used solely for visual effect due to huge calculation and numbers in Experiment 2 still doesn’t represent time but just efficiency.

**References**

1. Squares of a Sorted Array. (n.d.). Retrieved March 5, 2020, from <https://leetcode.com/problems/squares-of-a-sorted-array/description/>
2. Waxtap, ~. (2017, March 2). How to solve 'sortedSquaredArray' in CodeFights. Retrieved March 5, 2020, from <https://wxtp.wordpress.com/2017/03/02/how-to-solve-sortedsquaredarray-in-codefights/>
3. Data Structure and Algorithms Selection Sort. (n.d.). Retrieved March 11, 2020, from <https://www.tutorialspoint.com/data_structures_algorithms/selection_sort_algorithm.htm>