Hyperiondev

TASK

Algorithms I — Sorting and Hashing

Visit our website

Introduction

WELCOME TO THE ALGORITHMS I — SORTING AND HASHING TASK!

In 2007 the former CEO of Google, Eric Schmidt, asked the then presidential candidate Barack Obama what the most efficient way to sort a million 32-bit integers was (**https://www.youtube.com/watch?v=k4RRi_ntQc8**). Obama answered that the bubble sort would not be the best option. Was he correct? By the end of this task you might be able to answer that question!

Get in touch

Connect for support

Remember that with our courses, you're not alone! You can contact an expert code reviewer to get support on any aspect of your course.

The best way to get help is to login to **www.hyperiondev.com/portal** to start a chat with a code reviewer. You can also schedule a call or get support via email.

Our expert code reviewers are happy to offer you support that is tailored to your individual career or education needs. Do not hesitate to ask a question or for additional support!

ALGORITHMS

As you know, an algorithm is a clear, defined way of solving a problem that can be repeated. It is a set of instructions that can be followed to solve a problem. As a Software Engineer, every time you write code, you are creating an algorithm — a set of instructions that solves a problem.

There are a number of well-known algorithms that have been developed and extensively tested to solve common problems. In this task, you will be learning about algorithms used to sort and to search through a data structure that contains a collection of data (e.g. a List). In other words, in the previous Java Collections Framework tasks, you learned what type of **data structures** can be used to store collections of data. In this task, you are going to learn about **algorithms** that manipulate the data in those data structures.

There are two main benefits that you should derive from this task:

1. You will get to analyse and learn about algorithms that have been developed and tested by other experienced Software Engineers. This will help you to learn by example how to write good algorithms.
2. Since these algorithms have already been implemented by others, you will see how to use the algorithms without writing them from scratch, i.e. you don't have to reinvent the wheel.

Let's consider algorithms for sorting data.

SORTING ALGORITHMS

If you want to sort a set of numbers into ascending order, how do you normally go about it? The most common way people sort things is to scan through the set, find the smallest number, and put it first. Then find the next smallest number and put it second. This will continue until you have gone through all the numbers.

This is an effective way of sorting, but for a program, it can be time and memory consuming. In this section we will look at three popular sorting methods (and equally popular topics for interview questions!): Bubble Sort, Quick Sort and Merge Sort (for context, Java Collections Framework uses a modified Merge Sort method). If you want to have a look at visual representations of any of the sorting methods above, have a look **here**.

Bubble Sort

Bubble Sort got its name due to the nature of items 'bubbling up' to the top. The process of Bubble Sort is to compare an item (a) with the item next to it (b). If *a* is bigger than *b*, they swap places. This process continues until all the items in the list have been compared, then the process starts again. Let's look at the example below:

8	3	1	4	7	**First iteration:** Five numbers in random order.
3	**8**	1	4	7	3 < 8 so 3 and 8 swap
3	1	**8**	4	7	1 < 8, so 1 and 8 swap
3	1	4	**8**	7	4 < 8, so 4 and 8 swap
3	1	4	7	**8**	7 < 8, so 7 and 8 swap (do you see how 8 has bubbled to the top?)
3	1	4	7	8	**Next iteration:**
1	**3**	4	7	8	1 < 3, so 1 and 3 swap. 4 > 3 so they stay in place and the iteration ends

Bubble Sort is $O(n^2)$ and is written below (Adamchick, 2009):

```
void bubbleSort(int ar[]) {
    for (int i = (ar.length - 1); i >= 0; i--) {
        for (int j = 1; j <= i; j++) {
            if (ar[j-1] > ar[j]) {
                int temp = ar[j-1];
                ar[j-1] = ar[j];
                ar[j] = temp;
            }
        }
    }
}
```

Practise this concept with some number sequences of your own. It can be a tricky concept to start but becomes quite simple as you become more familiar with it.

Quick Sort

Quick Sort, as the name suggests, is a fast version of sorting. We start with a *pivot* point (usually the first element, but can be any element). We then look at the next

element. If it is bigger than the pivot point, it stays on the right, if it is smaller than the pivot point, it moves to the left of it. Now we have divided the array into less-than-pivot, pivot and greater-than-pivot. These sublists each get a new pivot point and the same iteration applies. This 'divide and conquer' process continues until each element has become a pivot point. See the example below:

4	3	2	8	7	5	1	**Pivot point:** 4
(**3**	2	1)	4	(**8**	7	5)	Divided list with new pivot points for each (3 and 8)
(**2**	1)	3	4	(**7**	5)	8	Divided list with new pivot points each (2 and 7)
(**1**)	2	3	4	(**5**)	7	8	Divided list with new pivot points each (1 and 5)
1	2	3	4	5	7	8	Nothing changes with above pivot points so the list is sorted

Quick Sort is O(log n) and is written below (Dalal, 2004):

```
public void quicksort(int[] list, int low, int high) {
    if (low < high) {
        int mid = partition(list,low,high);
        quicksort(list, low, mid-1);
        quicksort(list, mid+1, high);
    }
}
```

This is the partition method that is used to select the pivot point and move the elements around:

```
public int partition(int[] list, int low, int high) {

    // The pivot point is the first item in the subarray
    int pivot = list[low];

    // Loop through the array. Move items up or down the array so that they
    // are in the proper spot with regards to the pivot point.
    do {
        // can we find a number smaller than the pivot point? keep
        // moving the ''high'' marker down the array until we find
        // this, or until high=low
        while (low < high && list[high] >= pivot) {
            high--;
        }
```

```
        if (low < high) {
            // found a smaller number; swap it into position
             swap(list, low, high);
            // now look for a number larger than the pivot point
            while (low < high && list[low] <= pivot) {
                low++;
            }

            if (low < high) {
                // found one! move it into position
                swap(list, high, low);
            }
        }
    } while (low < high);

    // move the pivot back into the array and return its index
    list[low] = pivot;
    return low;
}

public void swap(int[] list, int firstIndex, int secondIndex) {
             int temp = list[firstIndex];
             list[firstIndex] = list[secondIndex];
             list[secondIndex] = temp;
      }
```

Note how the partition method is actually doing most of the heavy lifting, while the quicksort method just calls the partition method. This is a bit confusing at first, but keep practising to help you get the hang of the concept.

Merge Sort

Like Quick Sort, Merge Sort is also a 'divide and conquer' strategy. In this strategy, we break apart the list to then put it back together in order. To start, we break up the list into two and keep dividing until each element is on its own. Next, we start combining them two at a time and sorting as we go. Have a look at the example below. We start by dividing up the elements:

4	3	2	8	7	5	1	**List of numbers**
(4	3	2	8)	(7	5	1)	The list is divided into 2
(4	3)	(2	8)	(7)	(5	1)	Divided lists are divided again

(4)	(3)	(2)	(8)	(7)	(5)	(1)	Divided until each element is on its own

Next, we start to pair the elements back together, sorting as we go.

First merge

(4)	(3)	(2)	(8)	(7)	(5)	(1)	**Individual elements**
(4	3)	(2	8)	(7	5)	(1)	Elements are paired and compared
(3	4)	(2	8)	(5	7)	(1)	Sorted paired lists

Now we combine again. We sort as we combine by comparing the items:

Second merge part 1

(3	4)	(2	8)	
2				1. 3 vs. 2 — 3 > 2 so 2 becomes index 0
2	**3**			2. 3 vs 8 — 3 < 8 so 3 becomes index 1
2	3	**4**		3. 4 vs. 8 — 4 < 8 so 4 becomes index 2
2	3	4	**8**	4. 8 is leftover so becomes index 3

Second merge part 2

(5	7)	(1)	
1			5. 5 vs. 1 — 5 > 1 so 1 becomes index 0
1	**5**		6. 5 vs. 7 — 5 < 7 so 5 becomes index 1
1	5	**7**	7. 7 is leftover so becomes index 2

Finally, we sort and merge the last two lists:

Third merge

(2	3	4	8)	(1	5	7)	
1							1. 2 vs. 1 — 1 < 2 so 1 becomes index 0

1	**2**						2. 2 vs. 5 — 2 < 5 so 2 becomes index 1
1	2	**3**					3. 3 vs. 5 — 3 < 5 so 3 becomes index 2
1	2	3	**4**				4. 4 vs. 5 — 4 < 5 so 4 becomes index 3
1	2	3	4	**5**			5. 8 vs. 5 — 8 > 5 so 5 becomes index 4
1	2	3	4	5	**7**		6. 8 vs. 7 — 8 > 7 so 7 becomes index 5
1	2	3	4	5	7	**8**	7. 8 is leftover so becomes index 6

Merge Sort is O(n logn) and is written below (Dalal, 2004):

```
public void mergeSort(int[] list, int low, int high) {
    if (low < high) {
        // find the midpoint of the current array
        int mid = (low + high)/2;
        // sort the first half
        mergeSort(list, low, mid);
        // sort the second half mergeSort(list,mid+1,high);
        // merge the halves
        merge(list, low, high);
    }
}
```

The above method finds the midpoint of the array and recursively divides it until all the elements are the only ones in their own arrays. Once this is done the following method is called recursively to put them back together in order (Dalal, 2004):.

```
public void merge(int[] list, int low, int high) {

    // temporary array stores the ''merge'' array within the method.
     int[] temp = new int[list.length];

     // Set the midpoint and the end points for each of the subarrays
    int mid = (low + high)/2;
    int index1 = 0;
    int index2 = low;
    int index3 = mid + 1;

    // Go through the two subarrays and compare, item by item,
    // placing the items in the proper order in the new array
    while (index2 <= mid && index3 <= high) {
        if (list[index2] < list[index3]) {
```

```
        temp[index1] = list[index2];
        index1++; index2++;
        }
        else {
        temp[index1] = list[index3];
        index1++; index3++;
        }
    }

    // if there are any items left over in the first subarray, add them to
    // the new array
    while (index2 <= mid) {
    temp[index1] = list[index2];
    index1++;
    index2++;
    }

    // if there are any items left over in the second subarray, add them
    // to the new array
    while (index3 <= high) {
    temp[index1] = list[index3];
    index1++;
    index3++;
    }

    // load temp array's contents back into original array
    for (int i=low, j=0; i<=high; i++, j++) {
    list[i] = temp[j];
    }
}
```

As mentioned earlier, Merge Sort is used in the Collections Framework as `Collections.sort()`. For example:

```
import java.util.*;

public class Sort {
    public static void main(String[] args) {

        List<Integer> myList = new ArrayList<>();
        myList.add(10);
        myList.add(2);
        myList.add(23);
        myList.add(14);

        System.out.println("List: " + myList);
```

```
        Collections.sort(myList);
        System.out.println("Sorted List: " + myList);
    }
}
```

HASHING

What is Hashing?

Before we look at what hashing is, we need to understand what a map is. A map is a data structure that is implemented using hashing. It is a container object that stores entries, each of which contains two parts: a key and a value. The key is used to search for the corresponding value. These terms may sound familiar to you. In fact, another name of a map is a dictionary (Remember dictionaries from Level 1?). A map is also called a hash table or an associative array.

Hashing is a very efficient way to search for an element. Hashing can be used to implement a map to search, insert and delete an element in O(1) time. If you know what the index of an element is, you can retrieve that element in O(1) time. That means we are able to store the values in an array and use the key as the index to find the value. That is, of course, if you can map a key to an index.

The array that stores the values is called a hash table. The function or algorithm that maps a key to an index in the hash table is called a hash function. Hashing is a technique that retrieves the value using the index obtained from the key without performing a search.

As an example to help you understand how hashing works, suppose that we want to map a list of string keys to string values. In this example, we are going to map a list of countries to their capital cities. Let's say that the data in the table below are stored in a map.

Key	Value
England	London
Australia	Canberra
France	Paris
Spain	Madrid

Now, let's suppose that our hash function simply determines the length of the string. We have two arrays: one to store the keys and another to store the values. Our hash function converts a key to an integer value called a hash code. To put an item into the hash table, we need to compute the hash code, which in this case is to count the number of characters and put the key and value in the arrays at the corresponding index.

The table below shows how a hash function obtains an index from a key and uses the index to retrieve the value for the key. Spain, for example, has a length (hash code) of 5, so it is stored in position 5 in the keys array and Madrid is stored in position 5 of the values array. In the end, we should end up with the following:

Position	Keys array	Values array
1		
2		
3		
4		
5	Spain	Madrid
6	France	Paris
7	England	London
8		
9	Australia	Canberra

Notice that our arrays must be able to accommodate the longest string, which in this case is Australia. Some space is inevitably wasted because there are no keys that are less than 5 letters, for example, nor is there an 8-letter key. This is not a disaster, however, you can easily see that this method would not work for storing arbitrary strings. Imagine if one of your string keys were 10 000 characters long but the rest were under 100 characters long. The majority of the space in the array would go to waste. Also, what would happen if you had more than one key with the same hash code? For example Egypt and Spain (both have a length of 5).

Ideally, we would like to design a function that maps each key to a different index in the hash table. These functions are known as perfect hash functions. Perfect hash functions are difficult to find, however. A collision occurs when two or more keys are mapped to the same hash value (for example, Egypt and Spain). There are

ways to deal with collisions, however, it is better to avoid them in the first place. You should aim to design fast and easy-to-compute hash functions that minimise collisions.

Hash Functions

The `hashCode` method, which returns an integer hash code, is found in the `Object` class. This method returns the memory address for the object by default. The `hashCode` method has different implementations in different classes. For example, the following code:

```
Integer object1 = new Integer(1234);
String object2 = new String("1234");
System.out.println("hashCode for an integer 1234 is " + object1.hashCode());
System.out.println("hashCode for a string 1234 is " + object2.hashCode());
```

Will print out:

```
hashCode for an integer 1234 is 1234
hashCode for a string 1234 is 1509442
```

You should override the default hash functions to provide a one that better handles your data.

Hash Codes for Primitive Data Types

If your keys are of type `byte`, `short`, `int`, or `char`, you can simply cast them into `int`. Two different keys of any one of these types will therefore have different hash codes.

If your key is of type `float` you can use `Float.floatToIntBits(key)` as the hash code. **floatToIntBits(float f)** returns an `int` value whose bit representation is the same as the bit representation for the floating number `f`. Two different keys of type float will therefore also have different hash codes.

If your key is of the type `long`, you cannot simply cast it to `int`. This is because all keys that differ in only the first 32 bits will have the same hash code. You, therefore, need to divide the 64 bits into two halves and perform the *exclusive-or* operation (^) to combine the two halves in order to take the first 32 bits into account. This is called folding. The hash code for a long keyword is:

```
int hashCode = (int)(key ^ (key >> 32));
```

You might be a little unfamiliar with some of the symbols in the hash code above. Let's take a closer look. `>>` is known as the right shift operator. It shifts the bits 32 positions to the right. `^` is the *exclusive-or* operator. It takes two-bit patterns of equal length and performs the **exclusive-or** operation on each pair of corresponding bits. For example, `1010110 ^ 0110111` will give you `1100001`.
For a key of type `double`, you need to first convert it to a `long` value using `Double.doubleToLongBits` and then perform a folding like with any long value. The hash code for a double keyword is:

```
long bits = Double.doubleToLongBits(key);
int hashCode = (int)(bits ^ (bits >> 32));
```

Hash Codes for Strings

Most often, keys are `string` values. Therefore, it is important that you design a good hash function for them. One approach is simply to add the Unicode of all the characters in the string together as the hash code. This can work if no two keys contain the same letters, however, it will produce collisions if they do contain the same letters. An example of this is the words *coin* and *icon*. Both these words contain exactly the same letters.

A better hash function is one that generates a hash code that takes the position of each letter into account. This hash code is:

$$s_0 * b_{(n-1)} + s_1 * b_{(n-2)} + \ldots + s_{n-1}$$

where s_i is `s.charAt(i)`. This is called a polynomial hash code as it is a polynomial for some positive value `b`. You can use **Horner's rule** to calculate the hash code efficiently:

$$(\ldots((s_0 * b + s_1)b + s_2)b + \ldots + s_n - 2)b + s_{n-1}$$

An appropriate value for `b` should be chosen to minimise the number of collisions. According to experiments that were conducted, good choices for `b` are 31, 33, 37, 39, and 41. The hashCode is overridden using the hash code above with `b` being 31 in the **String class**.

Compressing Hash Codes

It is possible for the hash code for a key to be an extremely large integer that is out of the range for the hash-table index. In this case, you need to scale it down to fit within the range of the index. Let us assume that the index of a hash table is between `0` and `n - 1` (there are `n` different indices). The most common way to ensure an integer is between `0` and `n - 1` is to use the following:

```
h(hashCode) = hashCode % n
```

`n` should be a prime number that is greater than 2 to make sure that the indices are spread evenly.

Handling Collisions

As explained previously, when two keys are mapped to the same index in a hash table a collision occurs. There are two ways of handling collisions:

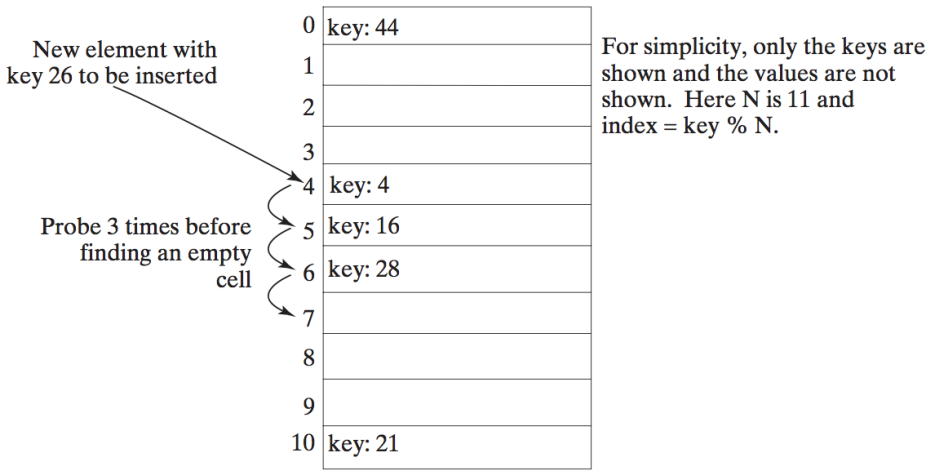
- Open addressing
- Separate chaining

Open Addressing

Open addressing handles collisions by finding an open location in the hash table if a collision occurs. There are several variations of open addressing, namely: **linear probing**, **quadratic probing**, and **double hashing**. In this task, we will look at linear probing.

Linear probing finds the next available location when a collision occurs sequentially. If a collision occurs at `hashTable[key % n]`, for example, `hashTable[(key + 1) % n]` is checked to see if it is available and if not `hashTable[(key + 2) % n]` is checked and so on, until an available location is found. When probing reaches the end of the table, it goes back to the beginning of the table. The hash table is treated as if it were circular.

For example, take a look at the diagram below. Let's say we would like to store an element with the key 26 into a hash table that has an index between 0 and 10. Therefore, `n = 11` and the index is `key % n`. The index for the key 26 will therefore be `26 % 11 = 4`, however, as you can see there is another element stored there with a key of 4. The hash table is probed three times before an empty cell is found.



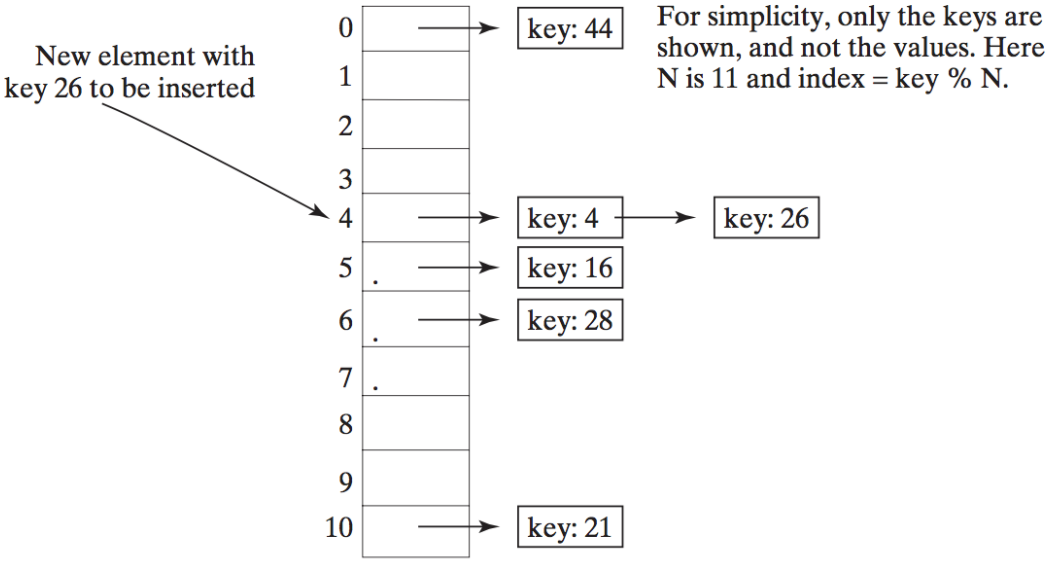
In order to search for an entry in the hash table, we need to obtain the index, in this case `k`, from the hash function for the key. We then check whether `hashtable[k % n]` contains the entry and if not check whether `hashTable[(k+1) % n]` contains the entry, and so on, until it is found or alternatively, an empty cell is reached.

To remove an entry from the hash table, you need to search for the entry that matches the key. A special marker is placed to denote that the entry is available if it is found. In the hash table, each cell has three possible states, namely: *occupied*, *marked* or *empty*. A *marked* cell is also available for insertion.

With linear probing groups of consecutive cells tend to be occupied. These groups are known as clusters. Each of these clusters is a probe sequence that you need to search when retrieving, adding, or removing an entry. Clusters may merge with each other to create even bigger clusters as they grow. This can slow down the search time further and is a major disadvantage of linear probing.

Separate Chaining

Rather than finding new locations, **separate chaining** places all entries with the same hash index in the same location. Each location uses a bucket to hold multiple entries. You can create a bucket using an `ArrayList` or `LinkedList`. For this task, we will be using the `LinkedList`. In the diagram below, you can see that each cell in the hash table is the reference to the head of a `LinkedList`. Starting from the head, elements are chained in the `LinkedList`.



Compulsory Task 1

Follow these steps:

- Create a Java file called **BubbleSort.java**
- Implement the Bubble sort algorithm on the following ArrayList:
  - [“right”, “subdued”, “trick”, “crayon”, “punishment”, “silk”, “describe”, “royal”, “prevent”, “slope”]

Compulsory Task 2

Follow these steps:

- Create a Java file called **QuickSort.java**
- In this task, the quick sort algorithm selects the first element in the list as the pivot. Revise it by selecting the median among the first, middle, and last elements in the list.

Compulsory Task 3

Answer the following question:

- Create a diagram that shows the hash table of size 11 after entries with the keys 34, 29, 53, 44, 120, 39, 45, and 40 are inserted, using separate chaining.

Optional Bonus Task

Answer the following question:

- Create a text file called **hashing.txt**. Inside, outline an algorithm that hashes a simple object with at least 2 attributes. For example, a fruit with a name and a colour.

Completed the task(s)?

Ask an expert to review your work!

Review work

Rate us

Share your thoughts

HyperionDev strives to provide internationally-excellent course content that helps you achieve your learning outcomes.

Think that the content of this task, or this course as a whole, can be improved or think we've done a good job?

Click here to share your thoughts anonymously.

References:

Adamchik, V. (2009). Sorting. Retrieved 25 February 2020, from https://www.cs.cmu.edu/~adamchik/15-121/lectures/Sorting%20Algorithms/sorting.html

Dalal, A. (2004). Searching and Sorting Algorithms. Retrieved 25 February 2020, from http://www.cs.carleton.edu/faculty/adalal/teaching/f04/117/notes/searchSort.pdf

University of Cape Town. (2014). Sorting, searching and algorithm analysis — Object-Oriented Programming in Python 1 documentation. Retrieved 25 February 2020, from https://python-textbok.readthedocs.io/en/1.0/Sorting_and_Searching_Algorithms.html