TASK

Algorithms - Sorting and Hashing

Visit our website

Introduction

Welcome to the Algorithms - Sorting and Hashing Task!

In 2007 the former CEO of Google, Eric Schmidt, ask 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 your mentor 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 your mentor. You can also schedule a call or get support via email.

Your mentor is 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!

Sorting

Introduction to Sorting

Sorting is an extremely important concept in computer science that introduces you to many key topics. Some of the reasons to study sorting algorithms are:

- They explain many creative approaches to problem solving, which can be used to solve other problems.
- They allow you to practice fundamental programming techniques such as using selection statements, loops, methods, and arrays.
- They are good examples to demonstrate the performance of algorithms.

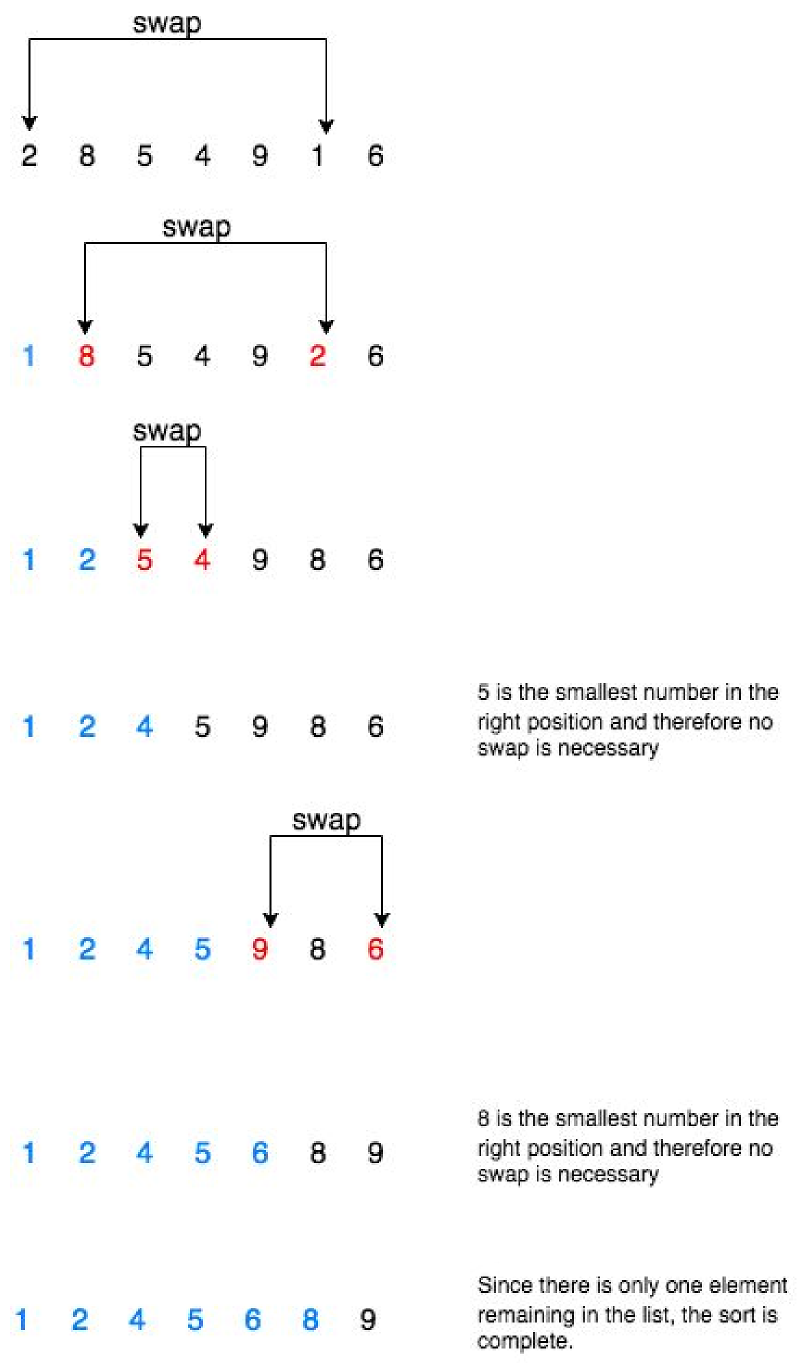
You are able to sort many types of data such as, integers, doubles, characters and objects, but for simplicity in this task we will only be sorting integers and storing them in an array in ascending order.

The algorithms for sorting that we will be looking at in this task are:

- Selection sort
- Insertion sort
- Bubble sort
- Merge sort
- Quick sort

Selection Sort

Selection sort enables you to sort a list in ascending order. With selection sort you first find the smallest number in the list and swap it with the first element. You then find the smallest number in the remaining list of unsorted elements swap it with the second element. This goes on until only one number is left. The diagram below demonstrates how selection sort is used to sort the list [2, 8, 5, 4, 9, 1, 6].


swap
2 8 5 4 9 1 6
swap
1 8 5 4 9 2 6
swap
1 2 5 4 9 8 6
1 2 4 5 9 8 6
5 is the smallest number in the right position and therefore no swap is necessary
swap
1 2 4 5 9 8 6
1 2 4 5 6 8 9
8 is the smallest number in the right position and therefore no swap is necessary
1 2 4 5 6 8 9
Since there is only one element remaining in the list, the sort is complete.

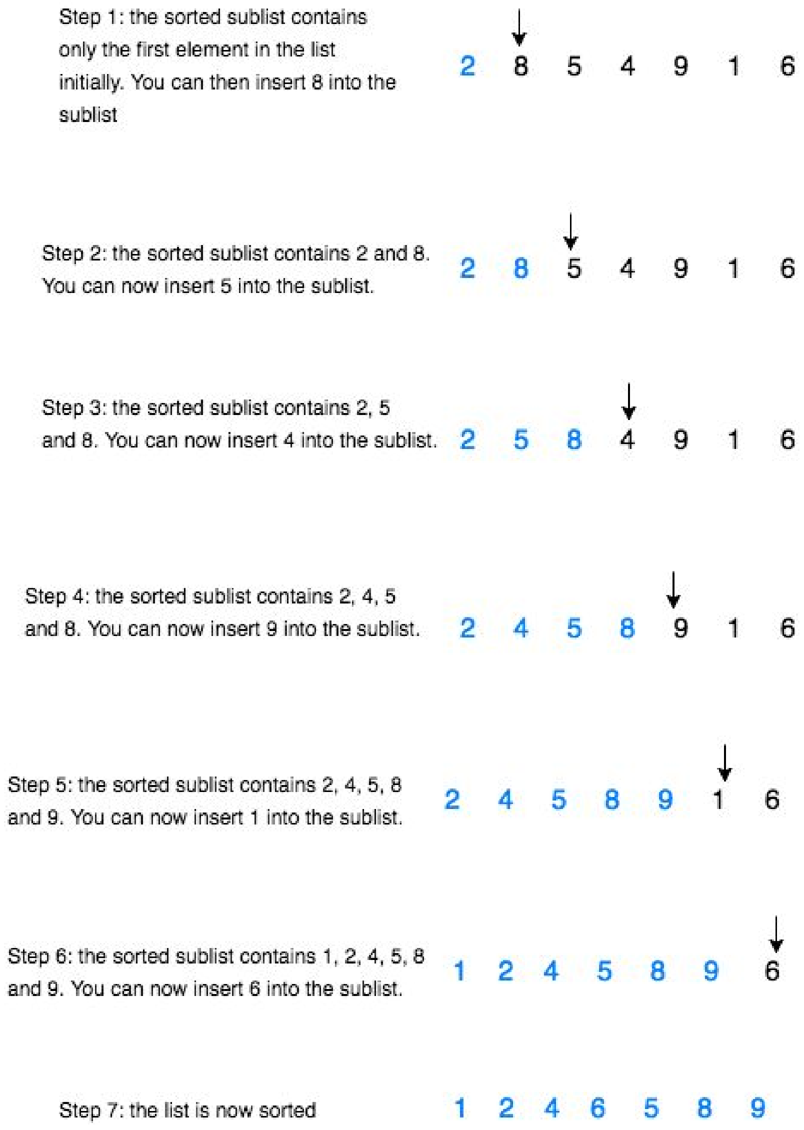
As you can see from the diagram above, selection sort repeatedly selects the smallest number in the unsorted portion of the list and swaps it with the first number. Now that we know how selection sort works, we can now implement it in Java. The program is shown below:

```
public class SelectionSort {

    public static void selectionSort(int[] list) {
        for (int i = 0; i < list.length - 1; i++) {
            // Find the minimum number  in the list
            double currentMin = list[i];
            int currentMinIndex = i;

            for(intj=i+1;j<list.length;j++){
                if (currentMin > list[j]) {
                    currentMin = list[j];
                    currentMinIndex = j;
                }
            }

            // Swap the minimum number with the next unsorted number
            if (currentMinIndex != i) {
                list[currentMinIndex] = list[i];
                list[i] = currentMin;
            }
        }
    }
}
```

The selectionSort(int[] list) method sorts any array of integer elements. The method uses nested for loops. The outer loop is used to find the smallest element in the list and swap it with list[i].

Insertion Sort

Insertion sort enables you to sort a list in ascending order by repeatedly inserting a new element into a sorted sublist until the whole list is sorted. The diagram below shows you how to sort the list [2, 8, 5, 4, 9, 1, 6] using insertion sort.



Step 1: the sorted sublist contains only the first element in the list initially. You can then insert 8 into the sublist

Step 2: the sorted sublist contains 2 and 8. You can now insert 5 into the sublist.

Step 3: the sorted sublist contains 2, 5 and 8. You can now insert 4 into the sublist.

Step 4: the sorted sublist contains 2, 4, 5 and 8. You can now insert 9 into the sublist.

Step 5: the sorted sublist contains 2, 4, 5, 8 and 9. You can now insert 1 into the sublist.

Step 6: the sorted sublist contains 1, 2, 4, 5, 8 and 9. You can now insert 6 into the sublist.

Step 7: the list is now sorted

From the diagram above, you can see how insertion sort repeatedly inserts a new element into a sorted sublist. We shall now implement insertion sort in Java.

```java
public class InsertionSort {

    public static void insertionSort(int[] list) {
        for (int i = 1; i < list.length; i++) {
            // Insert list[i] into a sorted sublist list[0..i-1] so that
            // list[0..i] is sorted.
            double currentElement = list[i];
            int k;
            for (k = i - 1; k >= 0 && list[k] > currentElement; k--) {
                list[k + 1] = list[k];
            }
            // Insert the current element into list[k + 1]
            list[k + 1] = currentElement;
        }
    }
}
```

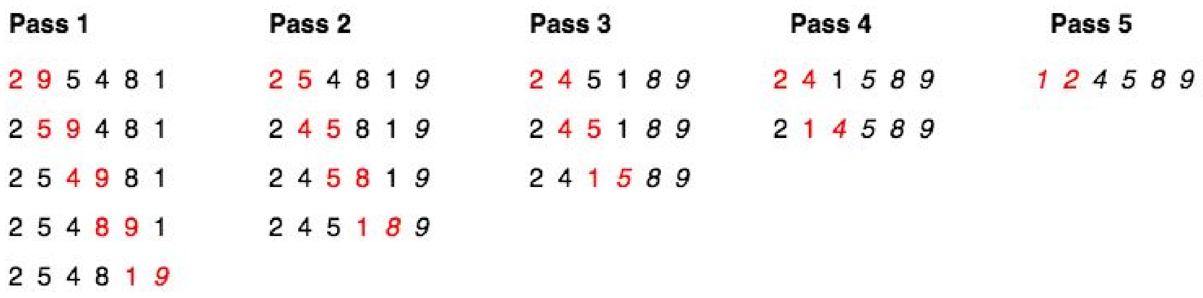
The insertionSort(int[] list) method sorts any array of integer elements. Like selection sort, the method uses nested for loops. The outer loop is used to obtain a sorted sublist, which ranges from list[0] to list[i]. The inner loop inserts list[i] into the sublist from list[0] to list[i-1].

Bubble Sort

The bubble sort algorithm sorts an array by making multiple passes through it. On each pass, successive neighboring pairs are compared and swapped if the elements are not in order. It is called bubble sort because smaller values tend to "bubble" to the top and large values sink to the bottom. The last element becomes the largest in the array after the first pass. After the second pass, the second-to-last element becomes the second largest in the array. This process is continued until all elements are sorted.

The diagram below shows the all the passes of bubble sort on an array of six elements (2 9 5 4 8 1).

Let's take a closer look at the first pass. Bubble sort first compares the elements in the first pair, 2 and 9. They are not swapped because they are already in order. Next, the elements in the second pair are compared, 9 and 5. They are swapped because 9 is greater than 5. Then, the elements in the third pair are compared, 9 and 4. They are swapped because 9 is greater than 4. The fourth pair is then compared, 9 and 8. They are also swapped. Finally the fifth pair are compared, 9 and 1. Once again 9 is swapped with 1. In the diagram the pairs being compared are shown in red and the numbers that have been sorted are italicized.



Notice that after the first pass, the largest number, 9, is placed last in the array. In the second pass, like the first pass, you compare each pair of numbers sequentially, however you do not need to compare the last pair because the last element in the array is the largest already. Likewise, in the third pass you don't need to compare and order the last two elements because they are already in order. For the kth pass therefore, you don't need to compare and order the last k - 1 elements, because they are ordered already.

The algorithm for bubble sort is shown below:

```
boolean needNextPass = true;

for (int k = 1; k < list.length && needNextPass; k++) {
    // Array may be sorted and next pass not needed
    needNextPass = false;

    // Perform the kth pass
    for(int i=0;i<list.length-k;i++){
        if (list[i] > list[i + 1]) {
            swap list[i] with list[i + 1];
            needNextPass = true; // Next pass still needed
        }
    }
}
```

Note that if no swap takes place in a given pass, you do not need to perform the next pass, because the elements are already sorted.

The bubble sort algorithm is implemented in the following java program:

```
public class BubbleSort {

    public static void bubbleSort(int[] list) {
        boolean needNextPass = true;

        for (int k = 1; k < list.length && needNextPass; k++) {
            // Array may be sorted and next pass not needed
            needNextPass = false;
            for (int i = 0; i < list.length - k; i++) {
                if (list[i] > list[i + 1]) {
                    // Swap list[i] with list[i + 1]
                    int temp = list[i];
                    list[i] = list[i + 1];
                    list[i + 1] = temp;

                    needNextPass = true; // Next pass still needed
                }
            }
        }
    }
}
```

The best case is that the bubble sort algorithm only needs a single pass to sort the array and the worst case is that it will require n - 1 passes. The first pass makes n - 1 comparisons; the second pass makes n - 2 comparisons, and so on until the last pass which only makes 1 comparison. The total number of comparisons are therefore:

(n - 1) + (n - 2) + . . . + 2 + 1

= (n -1)n / 2

= n2 / 2 - n / 2

= O(n2)

The Big O notation for bubble sort is therefore O(n2).

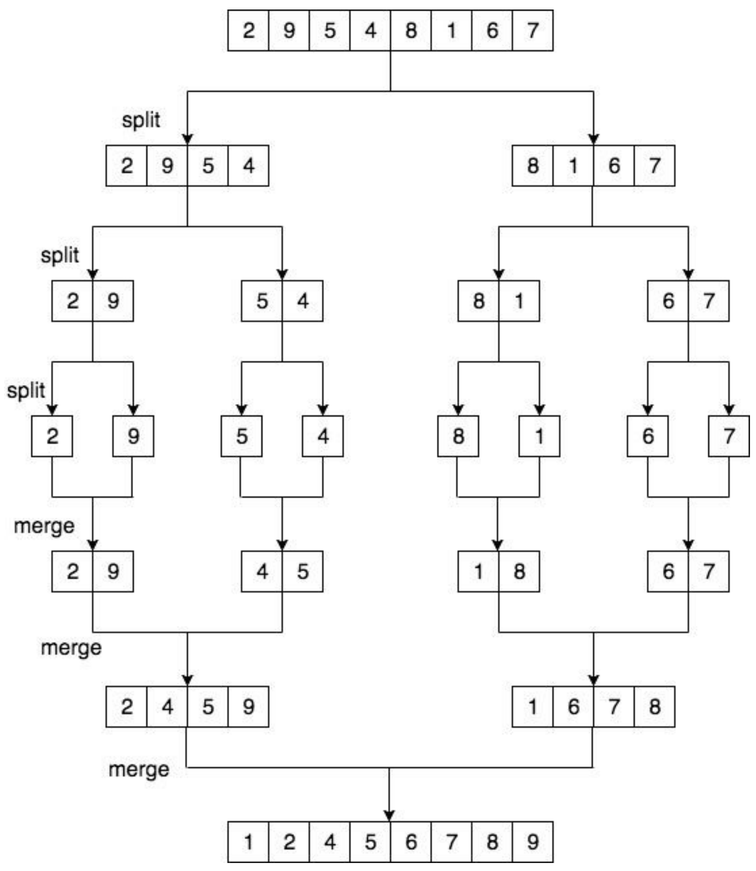
Merge Sort

Merge sort sorts an array by dividing the array into two halves and applying a merge sort on each half recursively. After the two halves are sorted, they are merged back together.

The algorithm for merge sort is shown below:

```
public static void mergeSort(int[] list) {
    if (list.length > 1) {
        // Sort the first half of the list
        mergeSort(list[0 ... list.length / 2]);
        // Sort the second half of the list
        mergeSort(list[list.length / 2 + 1 ... list.length]);
        //merge the two halves together
        merge list[0 ... list.length / 2] with
           list[list.length / 2 + 1 ... list.length];
     }
}
```

The diagram below shows how merge sort is used to sort an array of eight elements:



In the diagram above we sort an array with eight elements (2, 9, 5, 4, 8, 1, 6, 7). This array is then split into two halves, (2, 9, 5, 4) and (8, 1, 6, 7). Merge sort is then applied to each of these new subarrays recursively. This splits (2, 9, 5, 4) into (2, 9) and (5, 4) and (8, 1, 6, 7) into (8, 1) and (6, 7). Each of the new subarrays are then split again by applying merge sort until the subarray contains a single element. Array (2, 9) is split into (2) and (9), (5,4) is split into (5) and (4), (8,1) is split into (8) and (1) and (6, 7) is split into (6) and (7). Since all the subarrays contain only one element, they cannot be split further. We now merge (2) and (9) into a new sorted array (2, 9), (5) and (4) into the new sorted array (4, 5), and so on. We can then merge (2, 9) with (4, 5) into a new sorted array (2, 4, 5, 9) and then finally merge (2, 4, 5, 9) with (1, 6, 7, 8) into a new sorted array (1, 2, 4, 5, 6, 7, 8, 9).

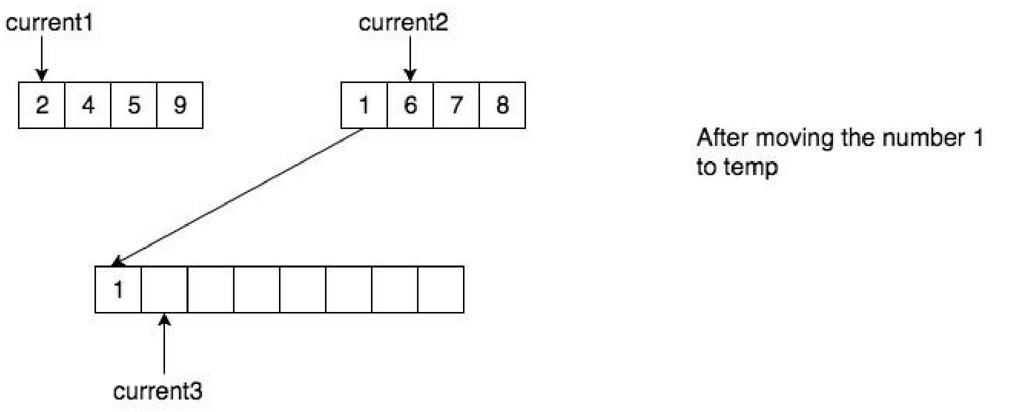
As you can see, the recursive call to MergeSort will continue to divide the array until each subarray contains only a single element. These single element subarrays are then merged back into a larger sorted subarray until there is just one sorted array.

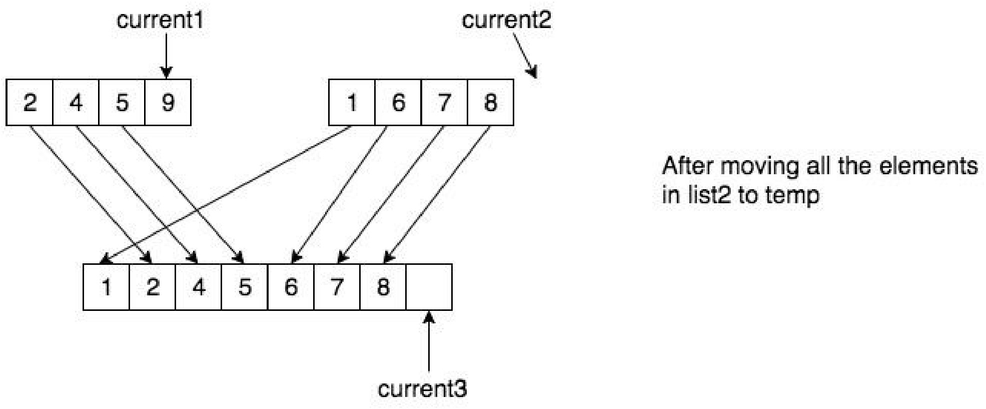
We can implement the merge sort algorithm as follows:

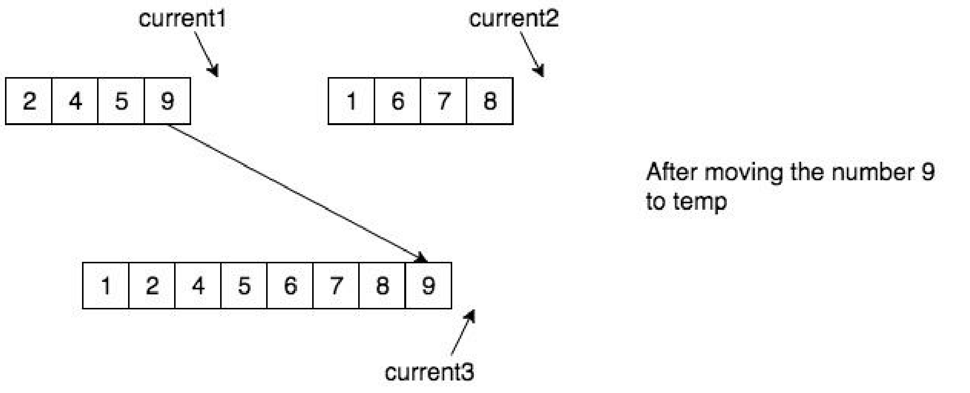
```
public class MergeSort {
    public static void mergeSort(int[] list) {
        if (list.length > 1) {
            // Sort the first half of the list
            int[] firstHalf = new int[list.length / 2];
            System.arraycopy(list, 0, firstHalf, 0, list.length / 2);
            mergeSort(firstHalf);

            // Sort the second half of the list
            int secondHalfLength = list.length - list.length / 2;
            int[] secondHalf = new int[secondHalfLength];

            System.arraycopy(list, list.length / 2, secondHalf, 0,
              secondHalfLength);
            mergeSort(secondHalf);

            // Merge the first half of the list with the second half of the list
            merge(firstHalf, secondHalf, list);
        }
    }

    // Merge two lists that have been sorted
    public static void merge(int[] list1, int[] list2, int[] temp) {
        int current1 = 0; // Current index in list1
        int current2 = 0; // Current index in list2
        int current3 = 0; // Current index in temp

        while (current1 < list1.length && current2 < list2.length) {
            if (list1[current1] < list2[current2])
                temp[current3++] = list1[current1++];
            else
                temp[current3++] = list2[current2++];
        }

       while (current1 < list1.length)
           temp[current3++] = list1[current1++];
       while (current2 < list2.length)
           temp[current3++] = list2[current2++];
    }
}
```

In the code above, the mergeSort method firstly creates a new array called firstHalf by making a copy of the first half of the list. mergeSort is then invoked recursively on firstHalf. Next a new array called secondHalf is created by making a copy of the second half of the list. mergeSort is then invoked recursively on secondHalf. After both firstHalf and secondHalf have been sorted they are then merged into one list using the merge method. The merge method merges the arrays list1 and list2, which are sorted into temp. current1 points to the current element to be considered in list1 and current2 points to the current element to be considered in list2. The merge method compares the current elements from list1 and list2 repeatedly and moves the smaller of the two to temp. If the smaller value is in list1, current1 is increased by one and likewise if the smaller value is in list2, current2 is increased by 1. In the end, all of the elements from either list1 or list2 should be moved to temp. If there are unmoved elements in list1, they are then copied to temp and, likewise, if there are unmoved elements in list2 they are copied to temp. list is now sorted.

The diagram below shows how the two arrays, list1 and list2, are merged.







Let T(n) denote time required for sorting an array of n elements using a merge sort. Assume n is a power of 2. The merge sort algorithm splits the array into two subarrays, sorts the subarrays using the same algorithm recursively, and then merges the subarrays. Therefore, T(n) = T(n/2) + T(n/2) + mergetime

The first T(n/2) is the time for sorting the first half of the array and the second T(n/2) is the time for sorting the second half. It takes at most n - 1 comparisons to compare the elements from the two subarrays and n moves to move elements to the temporary array. Therefore, the total time to merge the two subarrays is 2n - 1. Therefore,

T(n) = T(n/2) + T(n/2) + 2n - 1 = O(nlog n)

The complexity of merge sort is O(nlog n) which is better than selection sort, insertion sort, and bubble sort, since the time complexity of these algorithms is O(n2).

Quick Sort

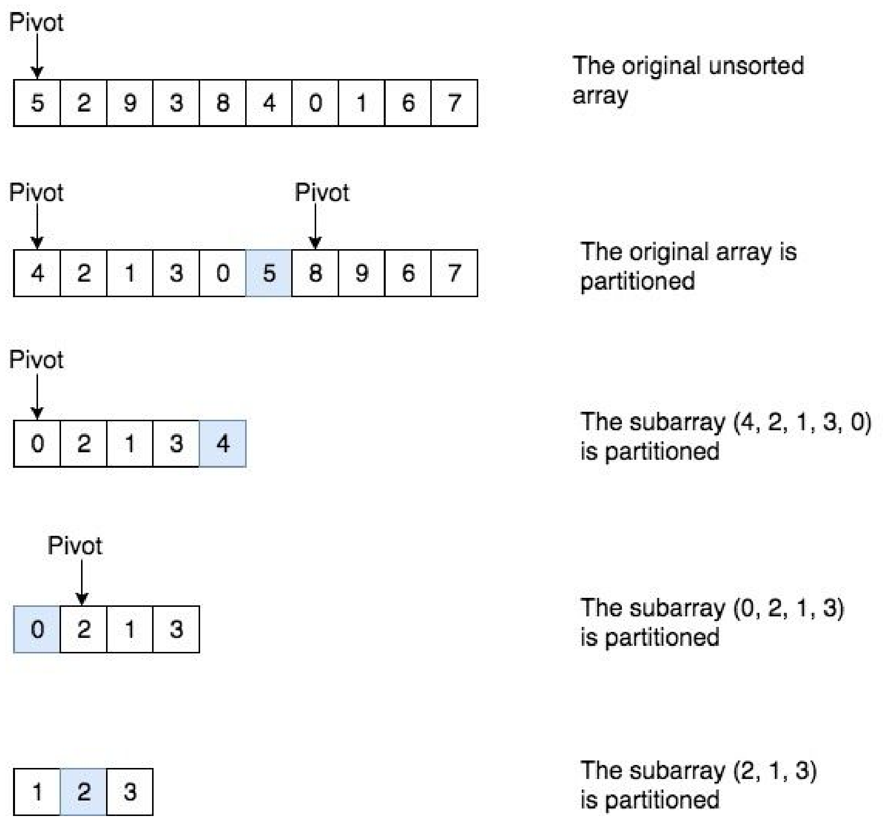
The quick sort algorithm sorts an array by selecting an element, called the pivot and dividing the array into two parts, so that all the elements in the first part are less than or equal to the pivot and all the elements in the second part are greater than the pivot. The quick sort algorithm is then recursively applied to the first part and then the second part.

The algorithm for quick sort, which was developed by C.A.R. Hoare in 1962, is shown below:

```
public static void quickSort(int[] list) {
    if (list.length > 1) {
        select a pivot;
        partition list into list1 and list2 such that
            all elements in list1 <= pivot and
            all elements in list2 > pivot;
        quickSort(list1);
        quickSort(list2);
    }
}
```

Each partition places the pivot in the right place and the choice of your pivot affects the algorithm’s performance. A pivot that divides the two parts of the array evenly should ideally be chosen, however in this task, we will assume the first element in the array is chosen for simplicity.

The diagram below shows how an array is sorted using quick sort:


Pivot
The original unsorted array
Pivot
Pivot
The original array is partitioned
Pivot
The subarray (4, 2, 1, 3, 0) is partitioned
Pivot
The subarray (0, 2, 1, 3) is partitioned
The subarray (2, 1, 3) is partitioned

In the above diagram the array (5, 2, 9, 3, 8, 4, 0, 1, 6, 7) is sorted using quick sort. Firstly, the first element, 5, is chosen as the pivot. The array is then partitioned into two parts and the highlighted pivot is placed in its correct position within the array. Quick sort is then applied on two partial arrays (4, 2, 1, 3, 0) and then (8, 9, 6, 7). The pivot 4 partitions (4, 2, 1, 3, 0) into one partial array (0, 2, 1, 3). Quick sort is then once again applied on (0, 2, 1, 3). The pivot 0 then partitions it into the subarray (2, 1, 3). Quick sort is applied on the subarray (2, 1, 3) and the pivot 2 is used to partition it into (1) and (3). Finally quick sort is applied in (1) and since the array only contains on element, no partition is needed.

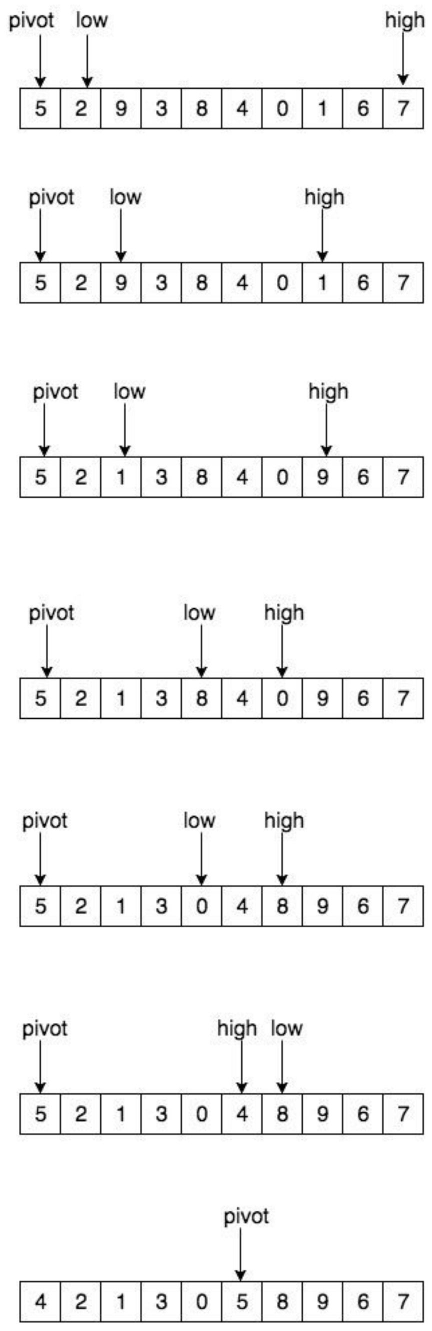
We can implement the quick sort algorithm as follows:

```
public class QuickSort {

    public static void quickSort(int[] list) {
        quickSort(list, 0, list.length - 1);
    }

    public static void quickSort(int[] list, int first, int last) {
        if (last > first) {
            int pivotIndex = partition(list, first, last);
            quickSort(list, first, pivotIndex - 1);
            quickSort(list, pivotIndex + 1, last);
        }
    }

    public static int partition(int[] list, int first, int last) {
        int pivot = list[first]; // Choose the first element as the pivot int
        int low = first + 1; // Index for forward search
        int high = last; // Index for backward search

        while (high > low) {
            // Search forward from left
            while (low <= high && list[low] <= pivot)
                low++;
            // Search backward from right
            while (low <= high && list[high] > pivot)
                high- -;
            // Swap two elements in the list
            if (high > low) {
                int temp = list[high];
                list[high] = list[low];
                list[low] = temp;
            }
        }

        while (high > first && list[high] >= pivot)
            high--;

        // Swap pivot with list[high]
        if (pivot > list[high]) {
            list[first] = list[high];
            list[high] = pivot;
            return high;
        }
        else {
            return first;
        }
    }
}
```

As you can see in the code above, there are two overloaded quickSort methods. The first method is used to sort an array while the second helper method is used to sort a partial array with a specified range.

The array list is partitioned by the partition method using the pivot. The first element in the partial array is chosen as the pivot. The variable low initially points to the second element in the partial array and the variable high points to the last element in the partial array. The method then searches forward in the array for the first element that is greater than the pivot starting from the left and then backward starting from the right for the first element in the array that less than or equal to the pivot. Next, it swaps these two elements and, using a while loop, repeats the same search and swap operations until all the elements are searched. The new index for the pivot that divides the partial array into two parts if the pivot has been moved, is returned by the method. The original index for the pivot is otherwise returned.

The diagram below shows how the array (5, 2, 9, 3, 8, 4, 0, 1, 6, 7) is partitioned. Firstly, the first element 5 is chosen as the pivot, the variable low contains the index that points to the element 2 and the variable high contains the index that points to the element 7. Low is then advanced forward to search for the first element (9) that is greater than the pivot and high is moved backward to search for the first element (1) that is less than or equal to the pivot. 9 and 1 are then swapped. The search then continues and low is moved to point to the element 8 and high is moved to point to the element 0. 8 and 0 are then swapped. low is continuously moved until it passes high. When all the elements have been examined the pivot is swapped with element 4 at the index high. When the method is finished, the index of the pivot is returned.



The pivot, low and high are initialized

Forward and backward search is preformed

9 and 1 are swapped

Search is continued

8 and 0 are swapped

High is less then low so the search is over

The pivot is now in the right place and the index of the pivot is returned

In order to partition an array of n elements it takes n comparisons and n moves in the worst case. The time required for the partition is therefore O(n). In the worst case, the pivot divides the array into one big subarray with the other array empty each time. The size of the subarray is one less than the one before it was divided. The algorithm therefore requires (n - 1) + (n - 2) + . . . + 2 + 1 = O(n2) time.

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. An example of a map is a dictionary, where the words are keys and their definitions are the values. In fact another name of a map is actually dictionary. 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. So 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 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 is no keys that are less than 5 letters for example. Neither is there a 8 letter key. This is not a big disaster in this case, however, you can easily see that this method would not work for storing arbitrary strings. Imagine if one of your string keys was 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 have more than one key with the same hash code? For example Egypt and Spain (they 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 just avoid them in the first place. You should aim to design fast and easy to compute hash functions that minimises 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 implementation 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 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 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. It is therefore important that you design a good hash function for them. On 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:
s0 * b(n - 1) + s1 * b(n -2) + ... + sn-1

where, si 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:

(. . . ((s0 * b + s1)b + s2)b + . . . + sn - 2)b + sn - 1

An appropriate value for b should be chosen to minimise the amount 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. You therefore 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 which 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:

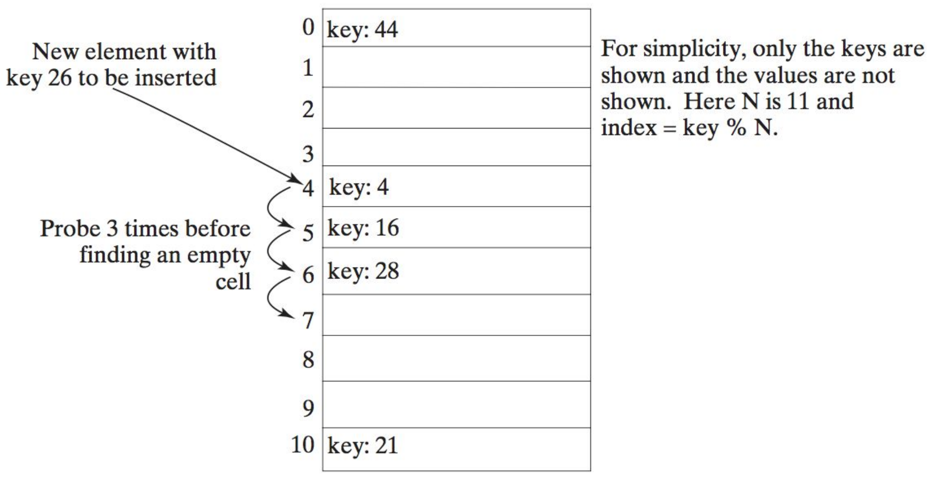
- open addressing
- separate chaining

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 is 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.


New element with key 26 to be inserted
Probe 3 times before finding an empty cell
0
1
2
3
4
5
6
7
8
9
10
key: 44
key: 4
key: 16
key: 28
key: 21
For simplicity, only the keys are shown and the values are not shown. Here N is 11 and index = key % N.

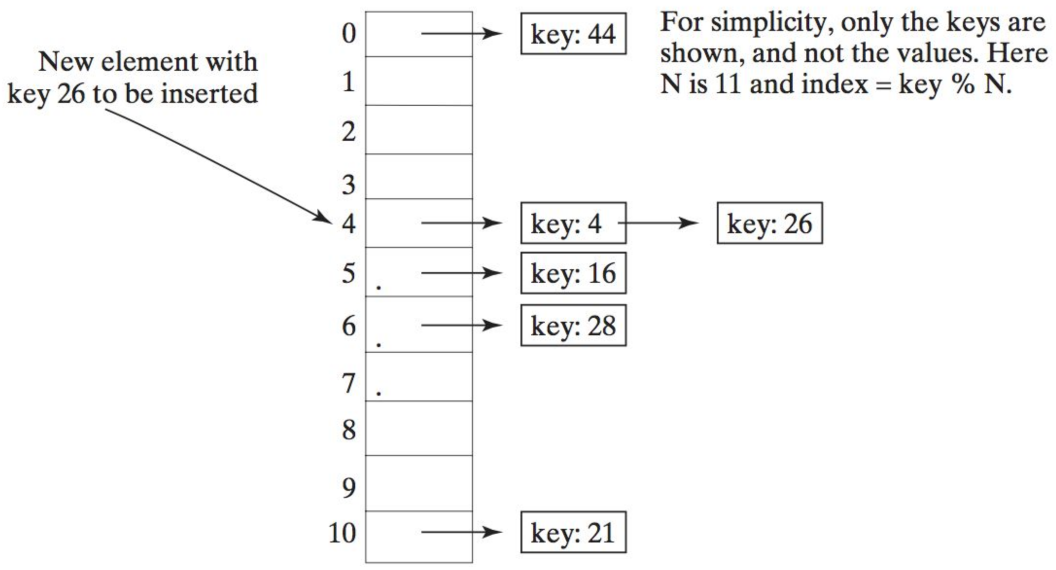
In order to search for an entry in the hash table, we need to obtain the index, in this case k, from the hash function for the key. Then check whether hashtable[k % n] contains the entry and if not check whether hashTable[(k+1) % n] contains the entry, and so on, until it is found or alternatively, an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


New element with key 26 to be inserted
0
1
2
3
4
5
6
7
8
9
10
key: 44
key: 4
key: 26
key: 16
key: 28
key: 21
For simplicity, only the keys are shown, and not the values. Here N is 11 and index = key % N.

Compulsory Task

Answer the following questions:

- 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.
- Write the algorithm for searching for entries using linear probing.
- Write the algorithm for removing entries using linear probing.
- Create a diagram similar to the one above 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.

Rate us

Share your thoughts

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

Think 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.