**1.Break String into Dictionary Words??**

Given an English dictionary (implemented as a hashmap (word -> meaning)) and a string without spaces, output all possible combinations of valid English words that when combined, reproduce the input string.e.g. input: “programmerit”  
output: {{pro, gram, merit}, {program, merit}, {programmer, it}}  
output:

Solution:**Recursive implementation:**  
The idea is simple, we consider each prefix and search it in dictionary. If the prefix is present in dictionary, we recur for rest of the string (or suffix).

|  |
| --- |
| public static HashMap<String,String> getDictionary() |
| { |

|  |
| --- |
| HashMap<String,String> hm=new HashMap<String,String>(); |
| hm.put("a","a"); |

|  |
| --- |
| hm.put("it","it"); |
| hm.put("am","am"); |

|  |
| --- |
| hm.put("ram","ram"); |
| hm.put("pro","pro"); |

|  |
| --- |
| hm.put("grammer","grammer"); |
| hm.put("program","program"); |

|  |
| --- |
| hm.put("programmer","programmer"); |
| hm.put("me","me"); |

|  |
| --- |
| hm.put("merit","merit"); |
| return hm; |

|  |
| --- |
| } |
|  |

|  |
| --- |
| public static void findWords(String s, int length, String out) |
| { |

|  |
| --- |
| HashMap<String,String>hm = getDictionary(); |
| for (int i=1; i <= length; i++) |

|  |
| --- |
| { |
| String subwords = s.substring(0, i); |

|  |
| --- |
| if (hm.containsKey(subwords)) |
| { |

|  |
| --- |
| if (i == length) |
| { |

|  |
| --- |
| out = out + subwords; |
| System.out.println(out); |

|  |
| --- |
| return; |
| } |

|  |
| --- |
| findWords(s.substring(i,length), length-i, out+subwords+" "); |
| } |

|  |
| --- |
| } |
| } |

|  |
| --- |
|  |
| public static void main (String[] args) throws java.lang.Exception |

|  |
| --- |
| { |
| String s = "programmerit"; |

|  |
| --- |
| findWords(s,s.length(),""); |
| } |

**2. Implement Stack and Queue using Linked List in Java??**

Ans: The implementation of a linked list is pretty simple in Java. Each node has a value and a link to next node.

|  |
| --- |
| class Node { |
| int val; |

|  |
| --- |
| Node next; |
|  |

|  |
| --- |
| Node(int x) { |
| val = x; |

|  |
| --- |
| next = null; |
| } |

|  |
| --- |
| } |

Two popular applications of linked list are stack and queue.

**Stack:** What is stack? Stack is a linear data structure which implements data on last in first out criteria. Here is a java program to implement stack using linked list.

|  |
| --- |
| class Stack{ |
| Node top; |

|  |
| --- |
|  |
| public Node peek(){ |

|  |
| --- |
| if(top != null){ |
| return top; |

|  |
| --- |
| } |
|  |

|  |
| --- |
| return null; |
| } |

|  |
| --- |
|  |
| public Node pop(){ |

|  |
| --- |
| if(top == null){ |
| return null; |

|  |
| --- |
| }else{ |
| Node temp = new Node(top.val); |

|  |
| --- |
| top = top.next; |
| return temp; |

|  |
| --- |
| } |
| } |

|  |
| --- |
|  |
| public void push(Node n){ |

|  |
| --- |
| if(n != null){ |
| n.next = top; |

|  |
| --- |
| top = n; |
| } |

|  |
| --- |
| } |
| } |

**Queue**: Queue is a data structure, that uses First in First out(FIFO) principle. Queue can be implemented by stack, array and linked list. Here is java program to implement queue using linked list.

|  |
| --- |
| class Queue{ |
| Node first, last; |

|  |
| --- |
|  |
| public void enqueue(Node n){ |

|  |
| --- |
| if(first == null){ |
| first = n; |

|  |
| --- |
| last = first; |
| }else{ |

|  |
| --- |
| last.next = n; |
| last = n; |

|  |
| --- |
| } |
| } |

|  |
| --- |
|  |
| public Node dequeue(){ |

|  |
| --- |
| if(first == null){ |
| return null; |

|  |
| --- |
| }else{ |
| Node temp = new Node(first.val); |

|  |
| --- |
| first = first.next; |
| return temp; |

|  |
| --- |
| } |
| } |

|  |
| --- |
| } |

**3. Check for Balanced Parentheses in an Expression??**

Ans: Given a string containing just the characters ‘(‘, ‘)’, ‘{‘, ‘}’, ‘[' and ']‘, determine if the input string is valid.

The brackets must close in the correct order, “()” and “()[]{}” are all valid but “(]” and “([)]” are not.

Algorithm:  
1) Declare a character stack S.  
2) Now traverse the expression string exp.  
a) If the current character is a starting bracket (‘(‘ or ‘{‘ or ‘[‘) then push it to stack.  
b) If the current character is a closing bracket (‘)’ or ‘}’ or ‘]’) then pop from stack and if the popped character is the matching starting bracket then fine else parenthesis are not balanced.  
3) After complete traversal, if there is some starting bracket left in stack then “not balanced”

Implementation:

|  |
| --- |
| public static boolean isValid(String s) { |
| char[] charArray = s.toCharArray(); |

|  |
| --- |
|  |
| HashMap<Character, Character> map = new HashMap<Character, Character>(); |

|  |
| --- |
| map.put('(', ')'); |
| map.put('[', ']'); |

|  |
| --- |
| map.put('{', '}'); |
|  |

|  |
| --- |
| Stack<Character> stack = new Stack<Character>(); |
|  |

|  |
| --- |
| for (Character c : charArray) { |
| if (map.keySet().contains(c)) { |

|  |
| --- |
| stack.push(c); |
| } else if (map.values().contains(c)) { |

|  |
| --- |
| if (!stack.isEmpty() && map.get(stack.peek()) == c) { |
| stack.pop(); |

|  |
| --- |
| } else { |
| return false; |

|  |
| --- |
| } |
| } |

|  |
| --- |
| } |
| return stack.isEmpty(); |

|  |
| --- |
| } |

Time Complexity: O(n)  
Auxiliary Space: O(n) for stack.

**4. Min Number of Platforms Required for a Railway Station??**

Ans: At a bus-station, you have time-table for buses arrival and departure. You need to find the minimum number of platforms so that all the buses can be placed as per their schedule.

For example consider the above example.

arr[] = {9:00, 9:40, 9:50, 11:00, 15:00, 18:00}  
dep[] = {9:10, 12:00, 11:20, 11:30, 19:00, 20:00}

All events sorted by time.

Total platforms at any time can be obtained by subtracting total departures from total arrivals by that time.

|  |
| --- |
| Time Event Type Platforms |
| 9:00 A 1 |

|  |
| --- |
| 9:10 D 0 |
| 9:40 A 1 |

|  |
| --- |
| 9:50 A 2 |
| 11:00 A 3 |

|  |
| --- |
| 11:20 D 2 |
| 11:30 D 1 |

|  |
| --- |
| 12:00 D 0 |
| 15:00 A 1 |

|  |
| --- |
| 18:00 A 2 |
| 19:00 D 1 |

|  |
| --- |
| 20:00 D 0 |

Minimum Platforms needed on railway station = Maximum platforms needed at any time = 3

|  |
| --- |
| int findPlatform(int arr[], int dep[], int n) |
| { |

|  |
| --- |
| int plat\_needed = 1, result = 1; |
| int i = 1, j = 0; |

|  |
| --- |
|  |
| // Similar to merge in merge sort to process all events |

|  |
| --- |
| while (i < n && j < n) |
| { |

|  |
| --- |
| if (arr[i] < dep[j]) |
| { |

|  |
| --- |
| plat\_needed++; |
| i++; |

|  |
| --- |
| if (plat\_needed > result) // Update result if needed |
| result = plat\_needed; |

|  |
| --- |
| } |
| else // Else decrements count of platforms needed |

|  |
| --- |
| { |
| plat\_needed--; |

|  |
| --- |
| j++; |
| } |

|  |
| --- |
| } |
|  |

|  |
| --- |
| return result; |
| } |

Algorithmic : Dynamic Programming

Time Complexity: O(nLogn), assuming that a O(nLogn) sorting algorithm for sorting arr[] and dep[].

**5. Largest subarray with equal number of 0s and 1s??**

Ans: Problem: Given an binary array containing only 0s and 1s, find the largest subarray which contain equal no of 0s and 1s.

Examples:

Input: arr[] = {1, 0, 1, 1, 1, 0, 0,0,1}  
Output: 1 to 8 (Starting and Ending indexes of output sub array)

**Implementation:**  
In this method, use two nested loops. The outer loop picks a starting point i. The inner loop considers all subarrays starting from i. If size of a subarray is greater than maximum size so far, then update the maximum size.  
In this implementation, 0s are considered as -1 and sum of all values from i to j is calculated. If sum becomes 0, then size of this subarray is compared with largest size so far. Complexity of this method is O(n^2).

|  |
| --- |
| int findLargestSubArray(int arr[], int n) |
| { |

|  |
| --- |
| int sum = 0; |
| int maxsize = -1, fromIndex; |

|  |
| --- |
|  |
| // Pick a starting point as i |

|  |
| --- |
| for (int i = 0; i < n-1; i++) |
| { |

|  |
| --- |
| if(arr[j] == 0) |
| sum = -1; |

|  |
| --- |
| else |
| sum = 1; |

|  |
| --- |
|  |
| // Consider all subarrays |

|  |
| --- |
| for (int j = i+1; j < n; j++) |
| { |

|  |
| --- |
| if(arr[j] == 0) |
| sum += -1; |

|  |
| --- |
| else |
| sum += 1; |

|  |
| --- |
|  |
| /\* If this is a 0 sum subarray, then compare it with |

|  |
| --- |
| maximum size subarray calculated so far \*/ |
| if(sum == 0 && maxsize < j-i+1) |

|  |
| --- |
| { |
| maxsize = j - i + 1; |

|  |
| --- |
| fromIndex = i; |
| } |

|  |
| --- |
| } |
| } |

|  |
| --- |
|  |
| return maxsize; |

|  |
| --- |
| } |

Time Complexity: O(n^2)  
Auxiliary Space: O(1)

**Question 7.If two English words have the same characters and the occurrence number of each character??**

is also identical respectively, they are anagrams. The only difference between a pair of anagrams is the order of

characters. For example, “silent” and “listen”, “evil” and “live” are two pairs of anagrams.

Please implement a function to verify whether two words are a pair of anagrams.

Two strings of a pair of anagrams have the same set of characters.

The only difference in them is the

order of characters. Therefore, they will become the same string if they are sorted. For example, both

“silent” and “listen” become “eilnst” after they are sorted. It costs O(*n*log*n*) time to sort a string with *n* characters. Let’s explore more efficient solutions.

A data container is used to store the occurrence number of each character. Each record in the container is composed of a character and its occurrence number. To solve this problem, it scans all characters in a string one by one. It checks the existence of the scanned character in the container. If the character already exists, its occurrence number increases; otherwise, a new record about the scanned character is inserted, with occurrence number as 1. There are two requirements of the data container: (1)

each record maps a character to an integer number, and (2) each record can be accessed and updatedefficiently. A hash table fulfills these requirements.

In order to solve the preceding problems in C/C++, a hash table is implemented as an array in which the index of an element is the key and the element is the value. If the problem is solved in Java, the

solution might be simpler because there is a type **HashMap** for hash tables. The code in

based on **HashMap** in Java.

*Java Code for Anagrams*

**boolean areAnagrams(String str1, String str2) {**

**if(str1.length() != str2.length())**

**return false;**

**Map<Character, Integer> times = new HashMap<Character, Integer>();**

**for(int i = 0; i < str1.length(); ++i) {**

**Character ch = str1.charAt(i);**

**if(times.containsKey(ch))**

**times.put(ch, times.get(ch) + 1);**

**else**

**times.put(ch, 1);**

**}**

**for(int i = 0; i < str2.length(); ++i) {**

**Character ch = str2.charAt(i);**

**if(!times.containsKey(ch))**

**return false;**

**if(times.get(ch) == 0)**

**return false;**

**times.put(ch, times.get(ch) - 1);**

**}**

**return true;**

**}**

In this code, it increases the occurrence numbers when it scans the string **str1** and decreases the

occurrence numbers when it scans the string **str2**. If two strings compose a pair of anagrams, all

occurrence numbers in the hash map **times** should be 0 eventually.

It scans both strings **str1** and **str2** once. When a character is scanned, it accesses a record in the

hash table and updates it. Both operations cost O(1) time. Therefore, it costs O(*n*) time if the length of

strings is *n*. It allocates some auxiliary space to accommodate the hash map. If strings only contain ASCII

characters, there are 256 characters at most, so there are 256 records in the hash map at most. Therefore,

the space efficiency is O(1).

**Question 8. If an element at the left side is greater than another element at the right side, they form a**

***reversed pair* in an array. How do you get a count of reversed pairs?**

**For example, there are five reversed pairs in the array {7, 5, 6, 4}, which are (7, 5), (7, 6), (7, 4), (5, 4), and (6, 4).**

**Ans:** The brute-force solution is to find the number of reversed pairs while scanning the array. A scanned

number is compared with all numbers behind it. If the number behind is less than the currently visited

number, a reversed pair is found. Since it compares a number with O(*n*) numbers in an array with size *n*,

the time complexity is O(*n*2). Let’s try to improve the performance.

Since it costs too much time to compare a number with all numbers on its right side, it may improve

efficiency if every two adjacent numbers are compared. Let’s take the array {7, 5, 6, 4} as an example, as

shown in Figure 7-2.

The new solution splits the whole array into two sub-arrays of size 2 (Figure 7-2(a)) and continues to

split the sub-arrays till the size of sub-arrays is 1 (Figure 7-2(b)). Then it merges the adjacent sub-arrays

and gets number of reversed pairs in them. The number in the sub-array {5} is less than the number in

the sub-array {7}, so these two numbers compose a reversed pair. Similarly, numbers in the sub-arrays

{4} and {6} also compose a reversed pair. After the number of reversed pairs is found between every two

adjacent sub-arrays with size 1, they are merged to form a set of sub-arrays with size 2

Sub-arrays are sorted when they are merged in order to avoid counting the reversed pairs again inside

the merged arrays.

*The process to get the number of reversed pairs in the array {7, 5, 6, 4}. (a) Split the array with*

*size 4 into two arrays with size 2. (b) Split each array with size 2 into two arrays with size 1. (c) Merge every two adjacent arrays with size 1, sort them, and get the number of reversed pairs. (d) Merge arrays with size*

*2, merge them, and get the number of reversed pairs.*

Let’s continue to count the number of reversed pairs between sub-arrays with size 2

illustrates the detailed process .Two pointers (*P*1 and *P*2) are initialized to the end of two sub-arrays. If the number referenced by *P*1

is greater than the number referenced by *P*2, there are some reversed pairs in such cases. As shown in

Numbers in the second sub-array. There are no reversed pairs if the number referenced by *P*1 is less than

or equal to the number referenced by *P*2 (Figure 7-3(b)). Another pointer *P*3 is initialized to the end of the

Merged array. The greater number of the two referenced by *P*1 and *P*2 is copied to the location referenced

by *P*3 in order to keep numbers in the merged array sorted. It moves these three pointers backward and

Continues to compare, count, and copy until one sub-array is empty. Then the remaining numbers in

the other sub-array are copied to the merged array.

5 7 4 6

7

(a) (b) (c)

6 7 5 6 7

5 7 4 6 5 7 4 6

*P*1 *P*2

*P*3 *P*3 *P*3

*P*1 *P*2 *P*1 *P*2

*Merge sub-arrays). The last step, to copy the last remaining number 4 in the*

*second sub-array, is omitted. (a) There are two reversed pairs because the number pointed to by* P1 *isgreater than the number pointed to by* P2*, and* P2 *points to the second number and there are two numbers.*

*in the second sorted sub-array less than the number pointed to by* P1*. Copy the number pointed to by* P1 *tothe merged array and move* P1 *and* P3 *backward. (b) There are no reversed pairs because the numberpointed to by* P1 *is less than the number pointed to by* P2*. Copy the number pointed to by* P2 *to the merged array and move* P2 *and* P3 *backward. (c) There is a reversed pair because the number pointed to by* P1 *is*

*greater than the number pointed to by* P2*, and* P2 *points to the first number in the second sub-array. Copy the number pointed to by* P1 *to the merged array and move* P1 *and* P3 *backward.*

The process to count reversed pairs may be summarized as follows: It recursively splits an array into two sub-arrays. It counts reversed pairs inside a sub-array and then counts reversed pairs between two adjacent sub-arrays while merging them. Therefore, the solution can be implemented based on the

merge sort algorithm, as shown as below

*Java Code to Count Reversed Pairs*

**int countReversedPairs(int[] numbers) {**

**int[] buffer = new int[numbers.length];**

**return countReversedPairs(numbers, buffer, 0, numbers.length - 1);**

**}**

**int countReversedPairs(int[] numbers, int[] buffer, int start, int end) {**

**if(start >= end)**

**return 0;**

**int middle = start + (end - start) / 2;**

**int left = countReversedPairs(numbers, buffer, start, middle);**

**int right = countReversedPairs(numbers, buffer, middle + 1, end);**

**int between = merge(numbers, buffer, start, middle, end);**

**return left + right + between;**

**}**

**int merge(int[] numbers, int[] buffer, int start, int middle, int end) {**

**int i = middle; // the end of the first sub-array**

**int j = end; // the end of the second sub-array**

**int k = end; // the end of the merged array**

**int count = 0;**

**while(i >= start && j >= middle + 1) {**

**if(numbers[i] > numbers[j]) {**

**buffer[k--] = numbers[i--];**

**count += (j - middle);**

**}**

**else {**

**buffer[k--] = numbers[j--];**

**}**

**}**

**while(i >= start) {**

**buffer[k--] = numbers[i--];**

**}**

**while(j >= middle + 1) {**

**buffer[k--] = numbers[j--];**

**}**

**// copy elements from buffer[] to numbers[]**

**for(i = start; i <= end; ++i) {**

**numbers[i] = buffer[i];**

**}**

**return count;**

**}**

As we know, the time complexity for the merge sort algorithm is O(*n*log*n*), so it is better than the

brute-force solution that costs O(*n*2) time. The second solution allocates more memory with a buffer size

*n*, so there is a trade-off between time and space efficiency.

**Q9. Please implement a function to find how many times a number occurs in a sorted array.**

For instance, the output is 4 when the inputs are an array {1, 2, 3, 3, 3, 3, 4, 5} and the number 3 because 3 occurs 4 times in the given array.

**Ans:** In a sorted array, it is natural to utilize the binary search algorithm. In order to count the occurrences of the number 3 in the given sample array, we can find any number with value 3 using the binary search and then scan its two sides sequentially to get the first 3 and the last 3. The target number may occur O(*n*) times in an array with size *n*, so the time complexity is still O(*n*) and it is not better than the linear search. It looks like the binary search algorithm does not help much here. Let’s utilize the binary search algorithm thoroughly.

Assume the target value is *k*. Most of the time in the solution above is spent locating the first and last *k*. Let’s explore more efficient solutions to locate the first and last *k*.

A better solution always compares the number *m* in the middle with the target *k*. If *m* is greater than *k*, *k* can only appear in the first half of the array, and the second half can be ignored in the next round of search. Similarly, *k* can only appear in the second half when *m* is less than *k*, and the first half can be ignored in the next round of search.

How do you continue searching when *m* equals *k*? If the number prior to the middle one is not *k*, the middle number is the first *k*. If the number before the middle number is also *k*, the first *k* should be in the first half of the array, and the second half can be ignored in the next round of search.

It is easy to implement code to find the first *k* based on recursion, as shown in Listing 8-1.

*Java Code to Get the First* k *in a Sorted Array*

**int get First(int[] numbers, int start, int end, int k) {**

**if(start > end)**

**return -1;**

**int middle = start + (end - start) / 2;**

**if(numbers[middle] == k) {**

**if((middle > 0 && numbers[middle - 1] != k)**

**|| (middle == 0))**

**return middle;**

**end = middle - 1;**

**}**

**else if(numbers[middle] > k){**

**end = middle - 1;**

**}**

**else {**

**start = middle + 1;**

**}**

**return getFirst(numbers, start, end, k);**

**}**

The method **getFirst** returns -1 if there is not a *k* in the array; otherwise, it returns the index of the first *k*. It finds the last *k* utilizing a similar process. If the middle number *m* is greater than *k*, *k* can occur only in the first half of the array. If *m* is less than *k*, *k* can occur only in the second half. When *m* equals *k*,it checks whether the number next to the middle one is also *k*. If it is not, the number in the middle is thelast *k*; otherwise, the last *k* occurs in the second half of the array.

The recursive code to get the last *k* is shown.

*Java Code to Get the Last* k *in a Sorted Array*

**int getLast(int[] numbers, int start, int end, int k) {**

**if(start > end)**

**return -1;**

**int middle = start + (end - start) / 2;**

**if(numbers[middle] == k) {**

**if((middle < numbers.length - 1 && numbers[middle + 1] != k)**

**|| (middle == numbers.length - 1))**

**return middle;**

**start = middle + 1;**

**}**

**else if(numbers[middle] > k){**

**end = middle - 1;**

**}**

**else {**

**start = middle + 1;**

**}**

**return getLast(numbers, start, end, k);**

**}**

Similar to **getFirst**, the method **getLast** returns -1 when there is not a *k* in the array; otherwise, it

returns the index of the last *k*.

When the first *k* and last *k* are found, it gets the occurrence numbers of *k* based on their indexes, as

**Q10. Given an increasingly sorted array and a number *s*, is there a pair of two numbers in the** **array whose sum is *s*?**

For example, if the inputs are an array {1, 2, 4, 7, 11, 15} and the number 15, there are two numbers, 4 and 11,whose sum is 15.??

**Ans:** Let’s first have a try selecting two numbers (*n*1 and *n*2) from the input array. If their sum equals to *s*, it islucky because two required numbers have been found. If the sum is less than *s*, it replaces *n*1 with its nextnumber because the array is increasingly sorted and the next number of *n*1 should be greater than *n*1. If

the sum is greater than *s*, the number *n*2 can be replaced with its preceding number in the sorted array,which should be less than *n*2.

Take the array {1, 2, 4, 7, 11, 15} and the number 15 as an example. At the first step, *n*1 is the first number (also the least one) 1 and *n*2 is the last number (also the greatest one) 15. It moves *n*2 backward to the number 11 because their sum 16 is greater than 15.

At the second step, the two numbers are 1 and 11, and their sum, 12, is less than 15. Therefore, it moves the *n*1 forward and *n*1 becomes 2.

The two numbers are 2 and 11 at the third step. Since their sum, 13, is less than 15, it moves *n*1

forward again.Now the two numbers are 4 and 11 and their sum is 15, which is the expected sum.

*The Process to Find a Pair of Numbers Whose Sum Is 15 out of the Array {1, 2, 4, 7, 11, 15}*

**Step *n*1 *n*2 *n*1+ *n*2 Comparing *n*1+ *n*2 with *s* Operation**

1 1 15 16 Greater Select the preceding number of *n*2

2 1 11 12 Less Select the next number of *n*1

3 2 11 13 Less Select the next number of *n*1

4 4 11 15 Equal -

It is not difficult to write code with the detailed analysis above, as shown in Listing 8-8.

*Java Code to Get a Pair with a Sum*

**boolean hasPairWithSum(int numbers[], int sum) {**

**boolean found = false;**

**int ahead = numbers.length - 1;**

**int behind = 0;**

**while(ahead > behind) {**

**int curSum = numbers[ahead] + numbers[behind];**

**if(curSum == sum) {**

**found = true;**

**break;**

**}**

**else if(curSum > sum)**

**ahead --;**

**else**

**behind ++;**

**}**

**return found;**

**}**

In this code, **ahead** is the index of *n*2, and **behind** is the index of *n*1. The time complexity is O(*n*) for an

array with *n* elements because it only scans the input array once.

**Question 11. Given an array, please check whether it contains three numbers whose sum equals 0.**

This problem is also required to find some numbers with a given array and sum, so it is similar to theprevious problem. We may get some hints from the solution above.

The solution above is based on an increasingly sorted array, so first, the input array is sorted

increasingly, too. Second, the sorted array is scanned from beginning to end. When the *ith* number with value *ai* is scanned, we try to find a pair of numbers whose sum is -*ai* in the array excluding the *ith* number.

Let’s modify the method **hasPairWithSum**

*Java Code to Get a Pair with a Sum Excluding a Number*

**boolean hasPairWithSum(int numbers[], int sum, int excludeIndex) {**

**boolean found = false;**

**int ahead = numbers.length - 1;**

**int behind = 0;**

**while(ahead > behind) {**

**if(ahead == excludeIndex)**

**ahead--;**

**if(behind == excludeIndex)**

**behind++;**

**int curSum = numbers[ahead] + numbers[behind];**

**if(curSum == sum) {**

**found = true;**

**break;**

**}**

**else if(curSum > sum)**

**ahead --;**

**else**

**behind ++;**

**return found;**

**}**

It checks whether there are two numbers whose sum is **sum** in **numbers** excluding the number with

index **excludeIndex**. It then checks whether there are three numbers in an array with sum 0 with the

method in Listing 8-10.

*Listing 8-10. Java Code to Get Three Numbers with Sum 0*

**boolean hasTripleWithSum0(int numbers[]) {**

**boolean found = false;**

**if(numbers.length < 3)**

**return found;**

**Arrays.sort(numbers);**

**for(int i = 0; i < numbers.length; ++i) {**

**int sum = -numbers[i];**

**found = hasPairWithSum(numbers, sum, i);**

**if(found)**

**break;**

**}**

**return found;**

**}**

It contains two steps in the function **hasTripleWithSum0**. It costs O(*n*log*n*)time to sort *n* numbers inits first step. At the second step, it costs O(*n*) time for each number to call **hasPairWithSum**, so it costsO(*n*2) time in the **for** loop. Therefore, its overall time complexity is O(*n*2).

**Question 12 .Given an array, please check whether it contains a subset of numbers (with one number atleast) whose sum equals 0.??**

Ans:

A subset of an array is a combination of numbers in the array. We have discussed two different algorithms to get combinations of characters in a string in the section *Permutations and Combinations*.It is a similar process to get combinations of an array. Listing 8-11 shows the sample code based on bit

operations to get subsets of an array.

*Listing 8-11. Java Code to Get a Subset with Sum 0*

**boolean hasSubsetWithSum0(int numbers[]) {**

**BitSet bits = new BitSet(numbers.length);**

**while(increment(bits, numbers.length)) {**

**int sum = 0;**

**boolean oneBitAtLeast = false;**

**for(int i = 0; i < numbers.length; ++i) {**

**if(bits.get(i)) {**

**if(!oneBitAtLeast)**

**oneBitAtLeast = true;**

**sum += numbers[i];**

**}**

**}**

**if(oneBitAtLeast && sum == 0)**

**return true;**

**}**

**return false;**

**}**

The method **increment** is the same as the method in the section *Permutations and Combinations* to

get combinations of a string.