Data StructuQre & Algorithm Past Year 2017 (September)

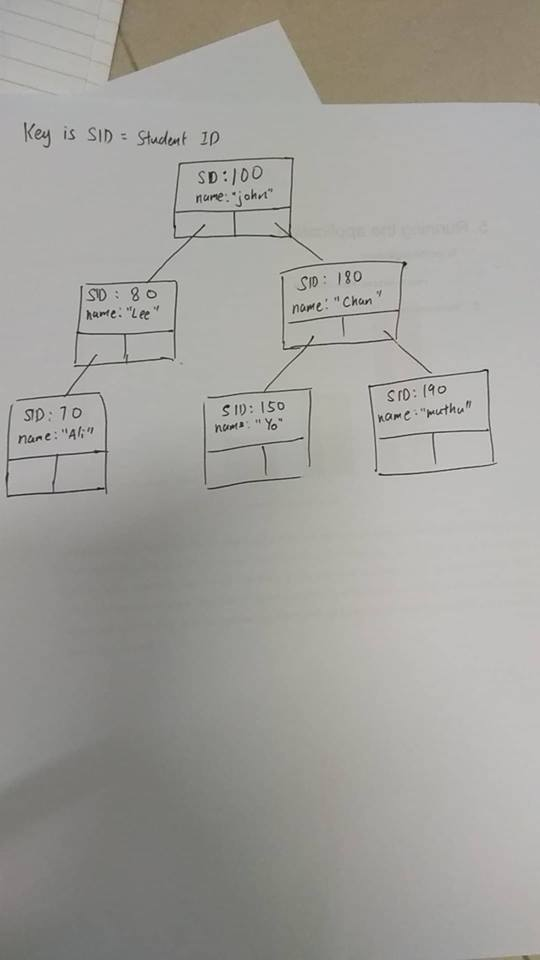
1(a)(i)

Because Big-O notation analysis focus on **growth rate**. When the input is very very large, the constant factor and non-dominating term become insignificant, thus they can be ignored so that the comparison of analysis of different algorithm can be simplified.

1(a)(ii)

[123] → [5 \* n] → [n \* log(n)] → [2n2 + 5 \*log(n)] → [4 \* n3 + n] → [7n4] → [6 \* n5]

1(b)



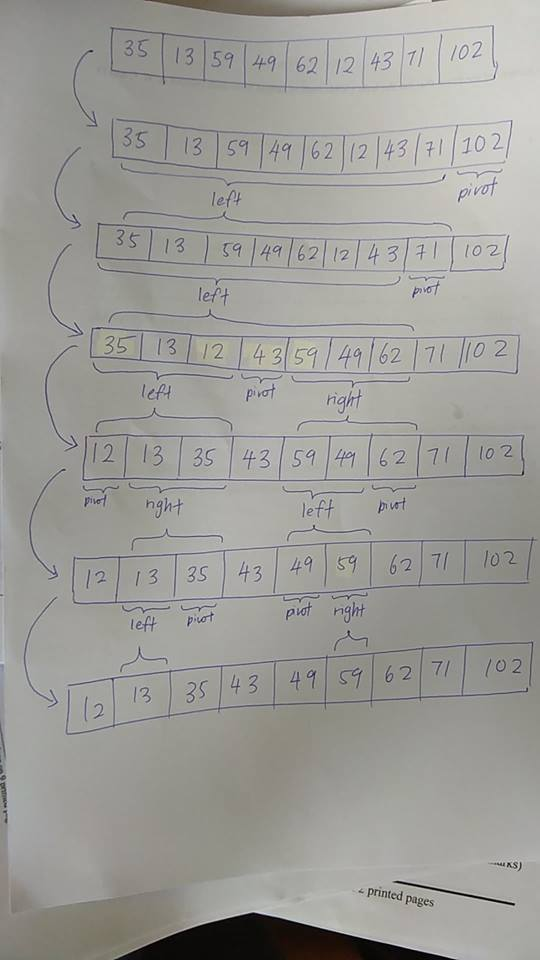
1(c) (i)

| **BASIS FOR COMPARISON** | **QUICK SORT** | **MERGE SORT** |
| --- | --- | --- |
| Partitioning of the elements in the array | The splitting of a list of elements is not necessarily divided into half. | Array is always divided into half (n/2). |
| Worst case complexity | O(n2) | O(n log n) |
| Works well on | Smaller array | Operates fine in any type of array. |
| Speed | Faster than other sorting algorithms for small data set. | Consistent speed in all type of data sets. |
| Additional storage space requirement | Less | More |
| Efficiency | Inefficient for larger arrays. | More efficient. |
| Sorting method | Internal | External |

1(c)(ii)

Initial array = [35, 13, 59, 49, 62, 12, 43, 71, 102]

Note: I will take the last element as pivot.



2(a)

* The base case is not defined, or is defined wrongly
* The result is diverging through each recursion, instead of converging to a single result

2(b)

public static int parseBinary(String binaryStr) {

// Assume that the input is correct

int value = Character.getNumericValue(binaryStr.charAt(0)));

value = value \* (int)Math.pow(2, binaryStr.length() - 1);

if(binaryStr.length() == 1) {

return value;

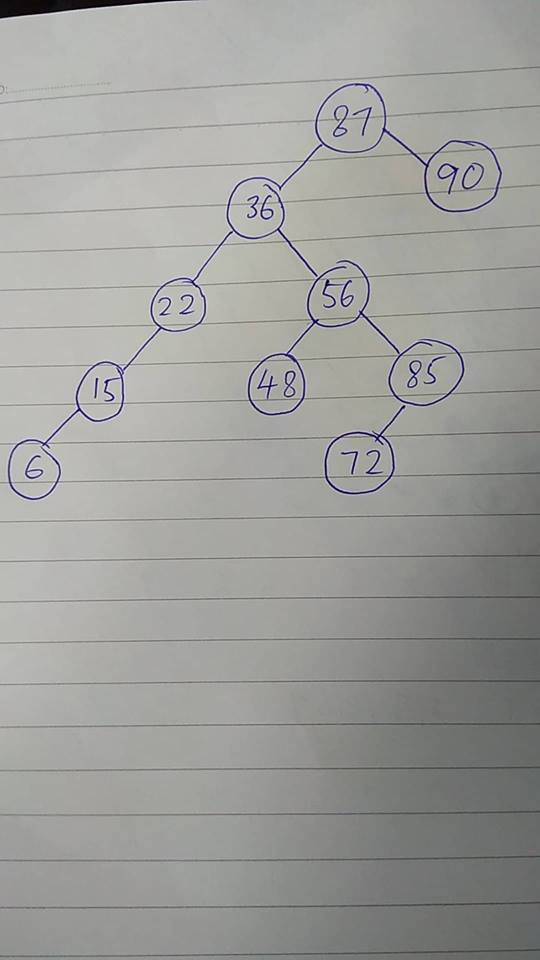
} else {

return value + parseBinary(binaryStr.substring(1));

}

}

2(c)(i)



2(c)(ii)

public static long getSize(File file) {

if(file.isDirectory()) {

long result = 0;

List<File> files = file.listFiles();

for(File f : files) {

result += getSize(f);

}

return result;

} else if (file.isFile()) {

return file.length();

} else {

throw new Error(“The file is not a dir nor a file”);

}

}

3(a)

Any data structure that contains reference to itself. For example, LinkedList, BinaryTree and Nodes etc. When we want to iterate through the elements of such data structures, we need to call their method recursively. One example is the File data structure in Question 2(c)(ii).

3(b)(i) [ A, D, E, G ]

3(b)(ii) [ D, E, F, G ]

3(c)

public static void main(String args[]) {

Stack<Integer> primeNumbers = new Stack<Integer>();

for(int i = 0; i < 50; i ++)

if(isPrime(i))

primeNumbers.push(i);

while(primeNumbesr.empty())

System.out.println(primeNumbers.pop());

}

public static boolean isPrime(int x) {

for(int i = 2; i < x; i++) {

if(x % i == 0) {

return false;

}

}

return true;

}

4(a) Graph. Because by using graph we can use algorithm such as Minimum Spanning Tree or Shortest Distance algorithm to minimize the cost of laying telephone networks.

4(b) Stack. Store a stack of user actions. When user do something, push the action onto the stack. When, the user click undo, pop the action and call undo of the action.

For example,

Stack<Action> history = new Stack<Action>();

while(true) {

Action lastAction = getUserAction(); // wait for user to do something

if (lastAction.name != “undo”) {

history.push(lastAction);

lastAction.execute();

} else {

Action toBeUndone = history.pop();

toBeUndone.undo();

}

}

4(b)

|  |  |  |
| --- | --- | --- |
| **Criteria** | **ArrayList** | **LinkedList** |
| Inserting/deleting new element at a specific index | Not efficient, because need to shift elements.  Efficiency = O(n) | Efficient, because only two node references need to be updated.  Efficiency = O(1) |
| Retrieving element at a specific index | Efficient, because no need to traverse through elements.  Efficiency = O(1) | Not efficient, because need to traverse through elements to get the specified element.  Efficiency = O(n) |

4(c)

public static void main(String[] args) {

LinkedList<String> names = new LinkedList<String>(new String[] {

“George”, …

“Ryan”

});

List duplicates = findDuplicate(names);

if(duplicates.length() > 0) {

System.out.println(“The duplicates are : “ + duplicates);

} else {

System.out.println(“There are no duplicates”);

}

}

// Finding duplicates using memoization

// See https://en.wikipedia.org/wiki/Memoization

**public** **static** List findDuplicate(Collection c) {

HashMap<Object, Boolean> memo = **new** HashMap<Object, Boolean>();

List duplicates = **new** ArrayList();

**for**(Object o : c) {

**if**(memo.getOrDefault(o, **false**)) {

duplicates.add(o);

} **else** {

memo.put(o, **true**);

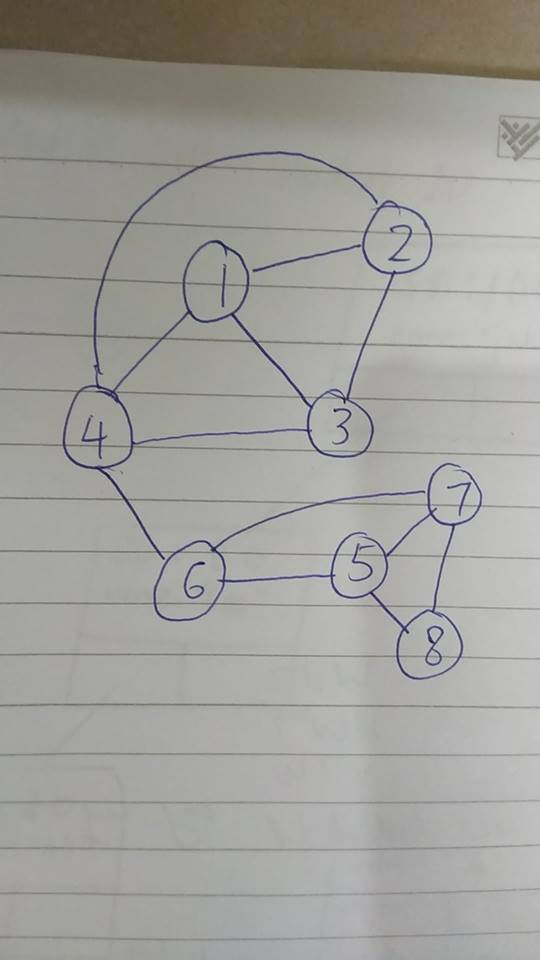
}

}

**return** duplicates;

}

5(a)(i)



5(a)(ii)

How to do depth first traversal? Use stack.

// this is just pseudocode

let result = [];

while(stack.notEmpty()) {

let item = stack.pop()

stack.pushItems(item.adjacents())

result.add(item);

}

Assume that we start from Vertex-1.

|  |  |  |
| --- | --- | --- |
| Step | Stack Content | Popped Item |
| 1 | [1] | - |
| 2 | [2, 3, 4] | 1 |
| 3 | [2, 3, 6] | 4 |
| 4 | [2, 3, 5, 7] | 6 |
| 5 | [2, 3, 5, 8] | 7 |
| 6 | [2, 3, 5] | 8 |
| 7 | [2, 3] | 5 |
| 8 | [2] | 3 |
| 9 | [] | 2 |

Answer = 1 → 4 → 6 → 7 → 8 → 5 → 3 → 2

5(a)(iii)

How to do breath-first traversal? Use queue.

// this is just pseudocode

let result = [];

while(queue.notEmpty()) {

let item = queue.dequeue()

queue.enqueueItems(item.adjacents())

result.add(item);

}

|  |  |  |
| --- | --- | --- |
| Step | Queue Content | Dequeued Item |
| 1 | [1] | - |
| 2 | [2, 3, 4] | 1 |
| 3 | [3, 4] | 2 |
| 4 | [4] | 3 |
| 5 | [6] | 4 |
| 6 | [5, 7] | 6 |
| 7 | [7, 8] | 5 |
| 8 | [8] | 7 |
| 9 | [] | 8 |

Answer = 1 → 2 → 3 → 4 → 6 → 5 → 7 → 8