Q&A 7 Oct 10-12noon

Midterm paper discussion

- Q1.

```
int[] arr = ...; // N numbers here
ArrayList<Integer> list = new ArrayList<>();

// missing code, see (a) - (d)

for (int idx = 0; idx < list.size(); idx++)
        arr[idx] = list.get(idx); // copy back from AL to array
```

The missing code snippet uses either insertion sort or **improved** bubble sort, but modified to work with **ArrayList** instead of an array. This description should be intuitive enough, but for your convenience:

```
void insertionSort(ArrayList<Integer> a){
        for (int i = 1; i < a.size(); i++) {
                int next = a.get(i);
                int j;
                for (j = i - 1; j >= 0 && a.get(j) > next; j--)
                        a.set(j+1, a.get(j));
                a.set(j+1, next);
        }
}
```

```
void bubbleSort(ArrayList<Integer> a) {
        for (int i = 1; i < a.size(); i++) {
                boolean is_sorted = true;
                for (int j = 0; j < a.size() - i; j++) { // left to right pass
                        if (a.get(j) > a.get(j+1)) {
                                swap(a, j, j+1);
                                is_sorted = false;
                        }
                }
                if (is_sorted)
                        return;
        }
}
```

What is the time complexity of each of these cases? Fill in the big-O time complexity (need tightest bound) of each case **without justification**:

a)

```
for (int idx = 0; idx < arr.length; idx++) {
        list.add(0, arr[idx]); // add to front of list
        insertionSort(list);
}
```

O(N^2)

b)

```
for (int idx = 0; idx < arr.length; idx++) {
        list.add(arr[idx]); // add to back of list
        insertionSort(list);
}
```

O(N^2)

c)

```
for (int idx = 0; idx < arr.length; idx++) {
        list.add(0, arr[idx]); // add to front of list
        bubbleSort(list); // improved bubble sort
}
```

O(N^2)

d)

```
for (int idx = 0; idx < arr.length; idx++) {
        list.add(arr[idx]); // add to back of list
        bubbleSort(list); // improved bubble sort
}
```

O(N^3)

Question 2

In cryptography, it is common to compute:

(**a ^ b**) modulo **M** i.e. the remainder from ($\mathbf{a^b}$ / **M**)

where **a**, **b**, and **M** are positive integers and **M** > 1

One efficient (and correct) way to calculate this result is to calculate its "square root" recursively:

```
int modExp(int a, int b, int M){ // rec. calls underlined for convenience
        if (b == 0)
                return 1;
        if (b % 2 == 0) {
                int sqrt = modExp(a, b / 2, M);
                return (sqrt * sqrt) % M; // Statement S1
        }
        int less = modExp(a, b - 1, M);
        return (a * less) % M; // Statement S2
}
```

Currently, statements S1 and S2 run in O(1) time. However, Uncle Grandpa made mistakes while implementing the function =O... Help Uncle Grandpa compute the big-O time complexity (need tightest bound) of modExp(a, b, M) in terms of a, b and/or M. Remember to use the correct variable(s). Do NOT provide justification:

a) The mistake is that statements **S1** and **S2** are replaced by statements that each run in O(**a**) time

O(a log b)

b) The mistake is that statements **S1** and **S2** are replaced by statements that each run in O(log b) time

O(log **b log b)**

c) The mistake is that statements **S1** and **S2** are replaced by statements that each run in O(**b** log **b**) time

O(b log b)

- The keys here are:

- when analyzing recursive functions, eliminate variables that don't affect the num of ops

- variables may shrink as they recurse (b has different meaning in each call)

- just because the argument of a log divides by 2 each time, doesn't mean you have a GP

Q2b

call(prob size)	**operations**
rec(b)	log b == (log b)
rec(b/2)	log (b/2) == (log b) - 1
rec(b/4)	log (b/4) == (log b) - 2
rec(b/8)	log (b/8) == (log b) - 3
...	
rec(16)	log 16 == 4
rec(8)	log 8 == 3
rec(4)	log 4 == 2
rec(2)	log 2 == 1
rec(1)	1 (can't be log1 which is 0)
rec(0)	1

Total ops = 2 + 1 + 2 + 3 + 4 + ... + ((log b) - 1) + log b = (log b)((log b) + 1) / 2 = O((log b)^2)

(Or, AP with largest term O(log b), and sum of AP with constant common difference is always O((largest term)^ 2))

- Even if b is odd (worst case is when b = 2^k - 1 for some integer k >= 2),

you will alternate between even case and odd case of almost the same prob sizes, so it's still same time compl

Q2c

call(prob size)	**operations**
rec(b)	b log b == b log b
rec(b/2)	b/2 log (b/2) < b/2 log b
rec(b/4)	b/4 log (b/4) < b/4 log b
rec(b/8)	b/8 log (b/8) < b/8 log b
...	
rec(16)	16 log 16 < 16 log b
rec(8)	8 log 8 < 8 log b
rec(4)	4 log 4 < 4 log b
rec(2)	2 log 2 < 2 log b
rec(1)	1 (can't be 1 log1 which is 0) < log b
rec(0)	1

Total ops < 1 + (log b)(1 + 2 + 4 + 8 + 16 + ... + b/8 + b/4 + b/2 + b) = O(b log b)

(2nd bracket is a GP with largest term O(b), and sum of GP with constant common ratio is always O(largest term))

How do we know that O(b log b) is tight, i.e. our bound is not too high up?

b log b < Total ops because the 1st call is already O(b log b)

So the total ops can't be of a time complexity lower than O(b log b)

- Question 3 [9 marks, 3 x 3]
- It is now the year 2040 and robots run the library. Books are arranged in very long piles vertically, since robots can move much faster and vertically higher than humans =X ... Next to some piles of books, there are empty spaces, and/or robot-controlled dropships that one at a time loads books from the top but drops books out from the bottom
- A pile of books can be modeled as a stack, with the book nearest the ceiling being at the top of the stack An empty space can be modeled as a stack, books are (of course) added to the top A dropship can be modeled as a queue, with the tail being the top where books are added, and the front of the queue being the bottom of the ship where the books are dropped from
- Robots can only transfer one book at a time, but are given commands in the form of:

  <from><to>(<num>)

  As an example:

  AB(n/2)

  indicates that the robot will transfer (n/2) books, one at a time, from location A to location B

- Example
- Say pile P has n books, you want to flip P EXCEPT for the bottom b books which should remain untouched, and you are given 2 Stacks A and B. One of the 2 possible programs (without more movements than necessary) would be:

  PA(n-b)AB(n-b)BP(n-b)

- You may notice, the program to give the robot MUST:

  - be a series of commands WITHOUT any separator in between commands

  - have NO space(s) both within and outside the parentheses

- You will definitely get marks deducted, or even get 0, if you do not follow this format STRICTLY

- Your Tasks
- A pile P has n books. Move the top c (c ≤ n)books to the bottom of P such that the relative positions of the books within the top section is still the same, and the relative positions of books within the bottom sections is still the same
- E.g. n = 7, c = 3, T U V W X Y Z are the 7 books in P
- P before: Ground [T, U, V, W, X, Y, Z] Ceiling
- P after: Ground [X, Y, Z, T, U, V, W] Ceiling

- There are 3 different independent scenarios. In each scenario, write a program for the robot to run (without performing more movements than necessary) using the format shown above, without comments/justification:

a) You have 2 empty spaces A and B, but NO dropship

Program: PA(c)PB(n-c)AP(c)BP(n-c)

b) You have only ONE empty space A and ONE dropship Q

Program: PA(c)PQ(n-c)AP(c)QA(n-c)AP(n-c) or a few other possible ans

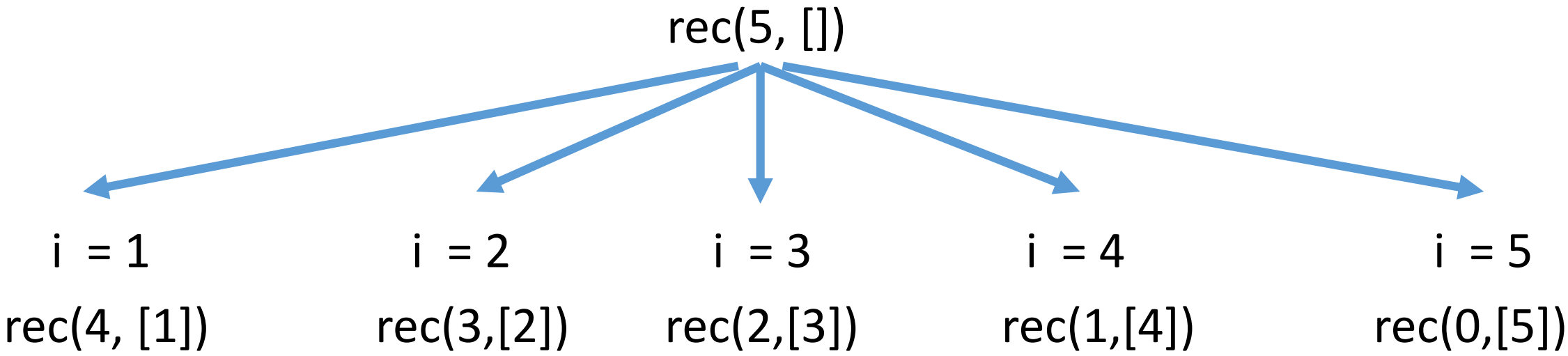
c) You have only ONE dropship Q but NO empty space

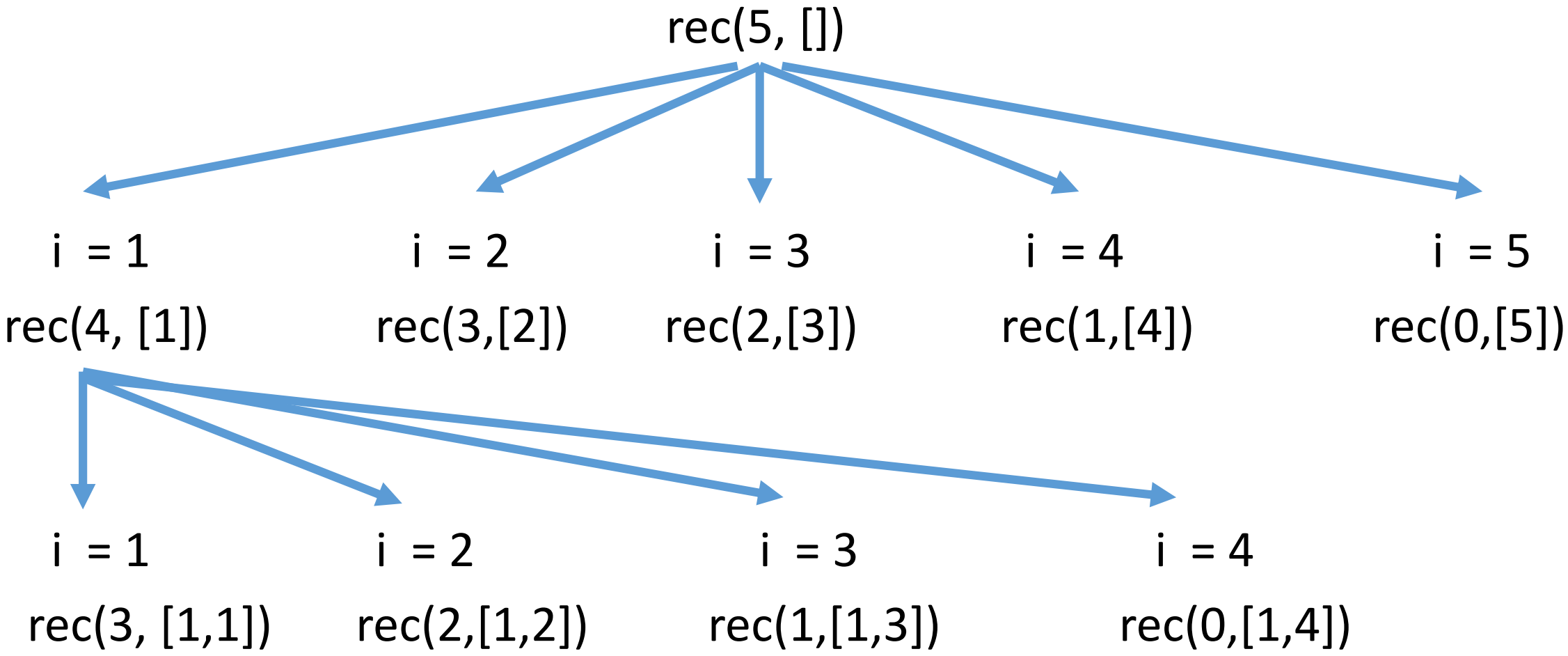
Program: PQ(c)QP(c)PQ(n)QP(n)PQ(n-c)QP(n-c)

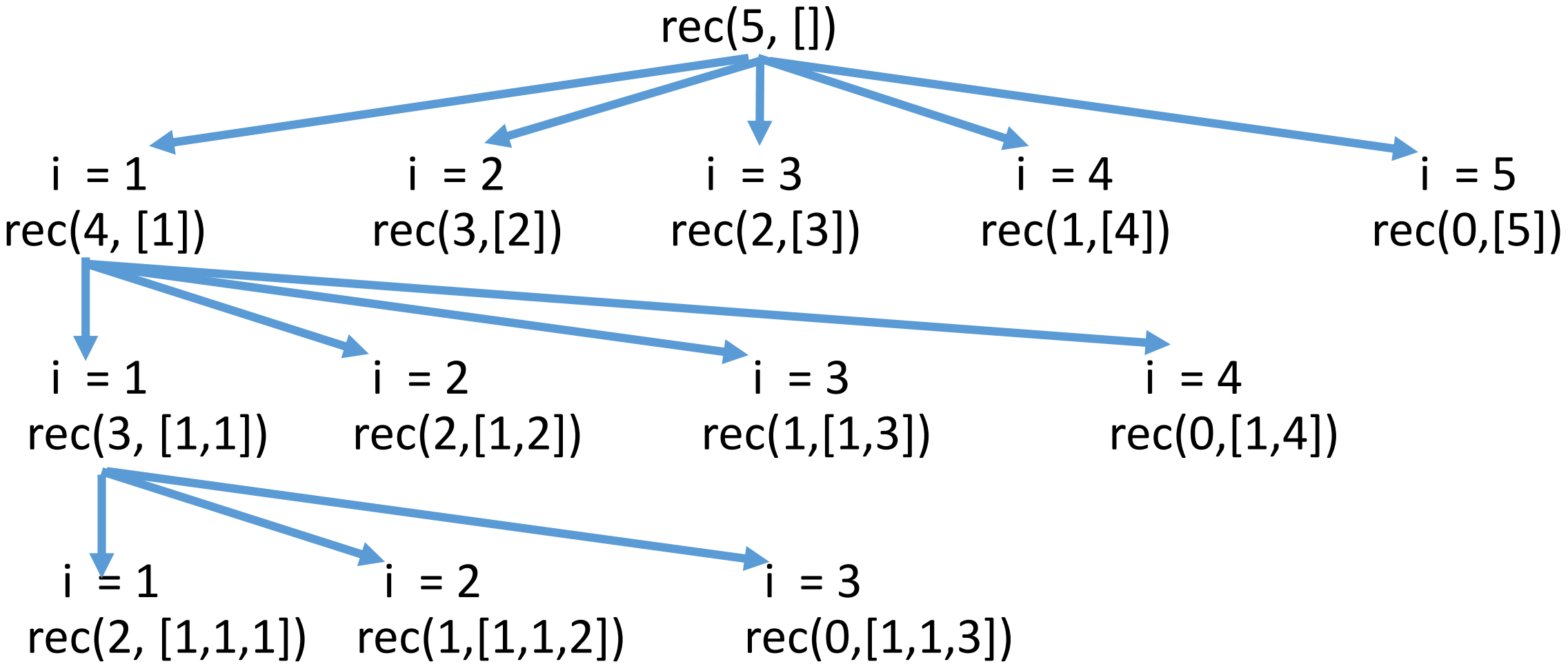
Question 4

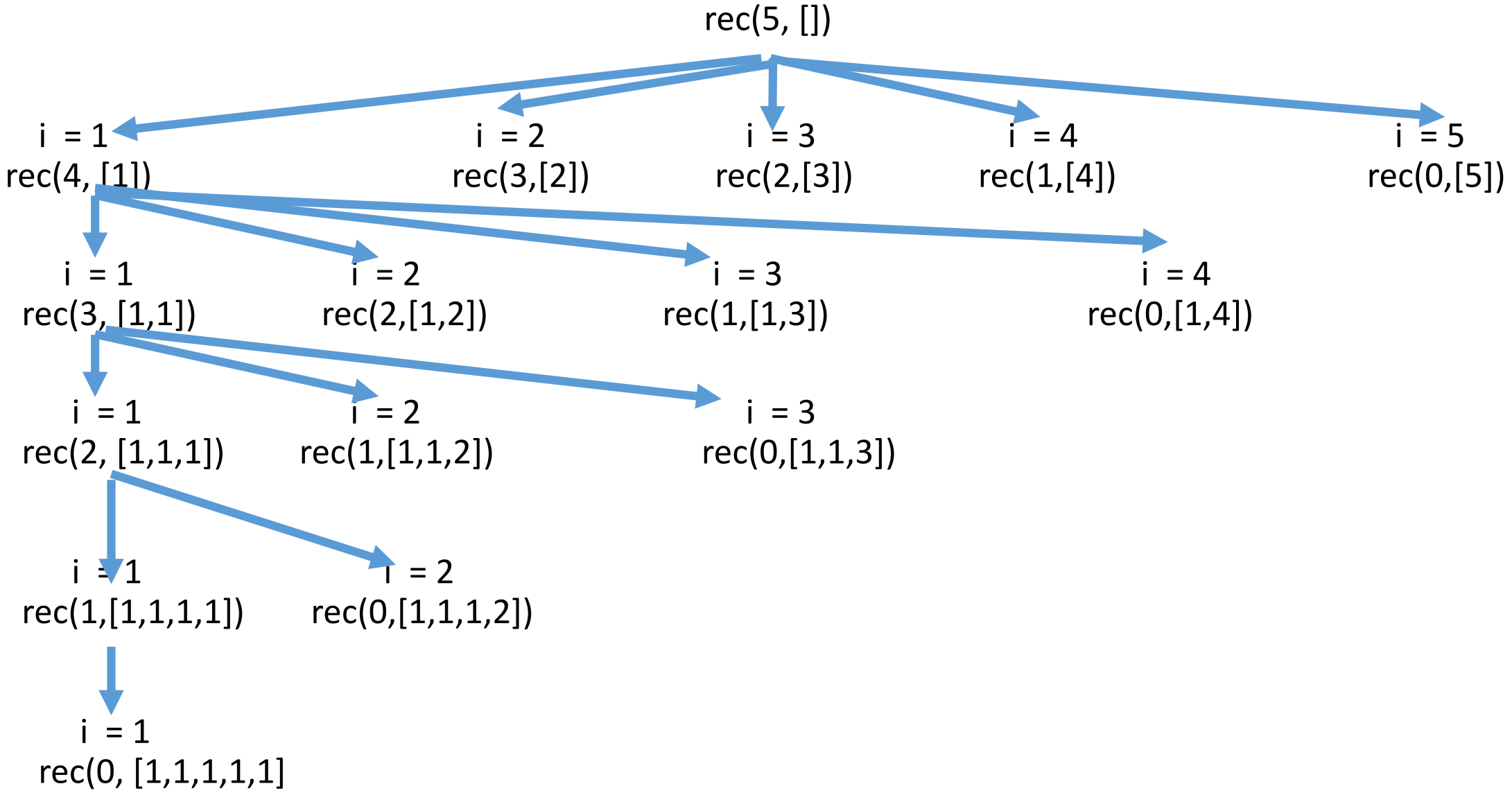
Examine the code snippet below:

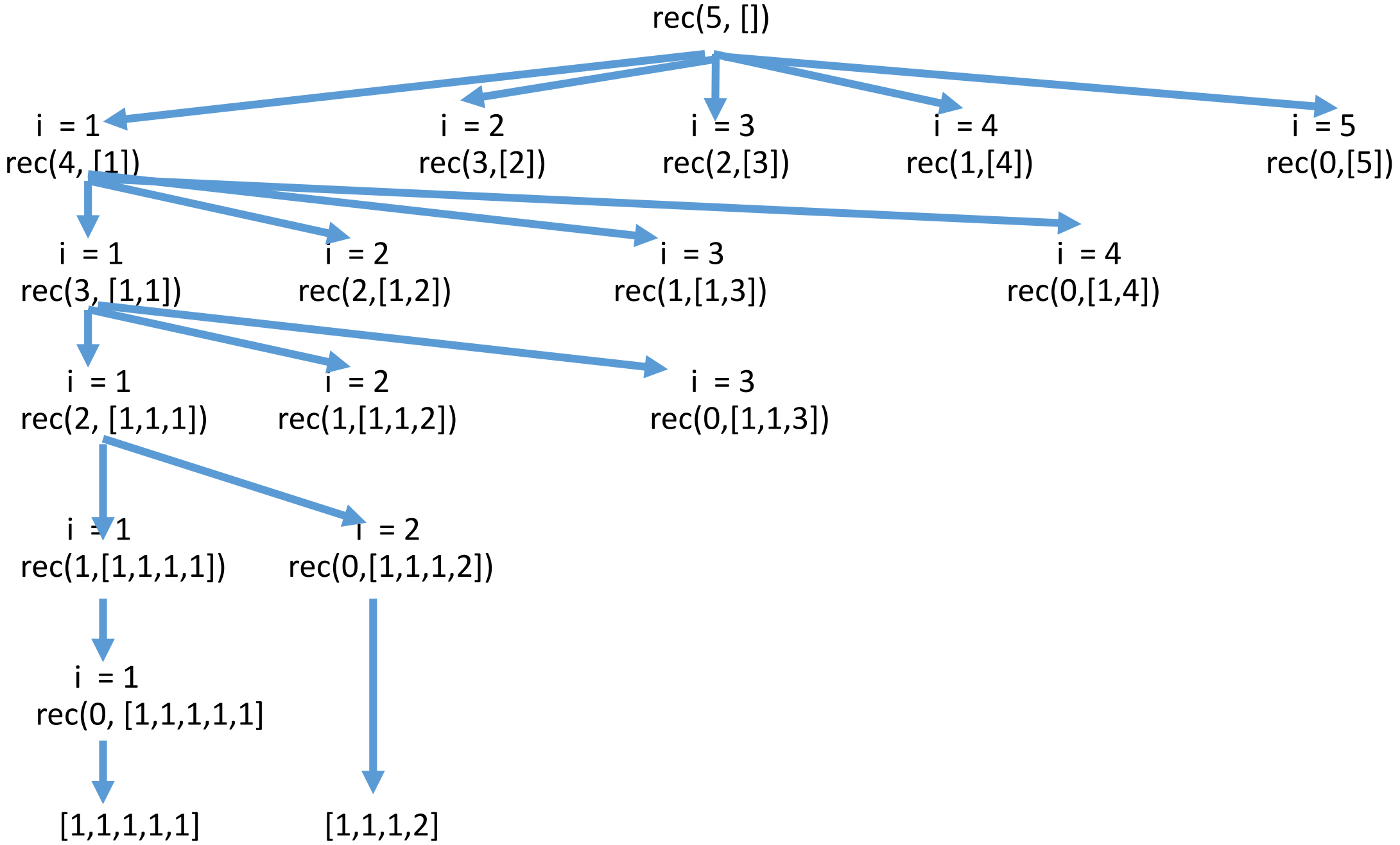
```
private static void rec(int x, ArrayList<Integer> a) {
    if (x < 0) return;
    if (x == 0) {
        System.out.print(a);
        return;
    }
    for (int i = 1; i <= x; i++) {
        ArrayList<Integer> copy = new ArrayList<>(a);
        copy.add(i);
        rec(x - i, copy);
     }
}
public static void rec(int x) { // pre-cond : x > 0
     rec(x, new ArrayList<Integer>());
}
```

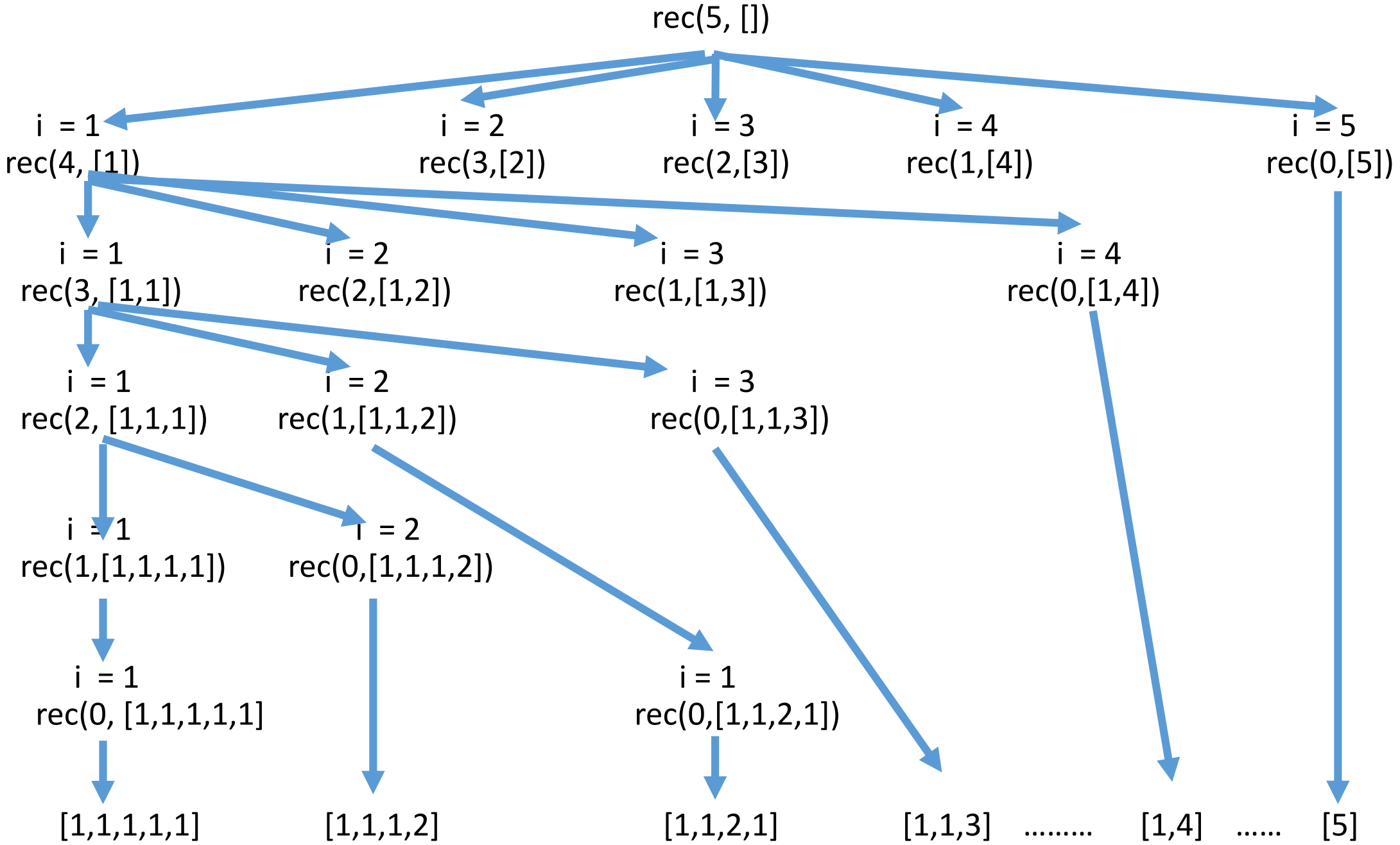

rec(5, [])
i = 1
rec(4, [1])
i = 2
rec(3,[2])
i = 3
rec(2,[3])
i = 4
rec(1,[4])
i = 5
rec(0,[5])


rec(5, [])
i = 1
rec(4, [1])
i = 2
rec(3,[2])
i = 3
rec(2,[3])
i = 4
rec(1,[4])
i = 5
rec(0,[5])
i = 1
rec(3, [1,1])
i = 2
rec(2,[1,2])
i = 3
rec(1,[1,3])
i = 4
rec(0,[1,4])


rec(5, [])
i = 1
rec(4, [1])
i = 2
rec(3,[2])
i = 3
rec(2,[3])
i = 4
rec(1,[4])
i = 5
rec(0,[5])
i = 1
rec(3, [1,1])
i = 2
rec(2,[1,2])
i = 3
rec(1,[1,3])
i = 4
rec(0,[1,4])
i = 1
rec(2, [1,1,1])
i = 2
rec(1,[1,1,2])
i = 3
rec(0,[1,1,3])


rec(5, [])
i = 1
rec(4, [1])
i = 2
rec(3,[2])
i = 3
rec(2,[3])
i = 4
rec(1,[4])
i = 5
rec(0,[5])
i = 1
rec(3, [1,1])
i = 2
rec(2,[1,2])
i = 3
rec(1,[1,3])
i = 4
rec(0,[1,4])
i = 1
rec(2, [1,1,1])
i = 2
rec(1,[1,1,2])
i = 3
rec(0,[1,1,3])
i = 1
rec(1,[1,1,1,1])
i = 2
rec(0,[1,1,1,2])
i = 1
rec(0, [1,1,1,1,1]


rec(5, [])
i = 1
rec(4, [1])
i = 2
rec(3,[2])
i = 3
rec(2,[3])
i = 4
rec(1,[4])
i = 5
rec(0,[5])
i = 1
rec(3, [1,1])
i = 2
rec(2,[1,2])
i = 3
rec(1,[1,3])
i = 4
rec(0,[1,4])
i = 1
rec(2, [1,1,1])
i = 2
rec(1,[1,1,2])
i = 3
rec(0,[1,1,3])
i = 1
rec(1,[1,1,1,1])
i = 2
rec(0,[1,1,1,2])
i = 1
rec(0, [1,1,1,1,1]
[1,1,1,1,1]
[1,1,1,2]


rec(5, [])
i = 1
rec(4, [1])
i = 2
rec(3,[2])
i = 3
rec(2,[3])
i = 4
rec(1,[4])
i = 5
rec(0,[5])
i = 1
rec(3, [1,1])
i = 2
rec(2,[1,2])
i = 3
rec(1,[1,3])
i = 4
rec(0,[1,4])
i = 1
rec(2, [1,1,1])
i = 2
rec(1,[1,1,2])
i = 3
rec(0,[1,1,3])
i = 1
rec(1,[1,1,1,1])
i = 2
rec(0,[1,1,1,2])
i = 1
rec(0, [1,1,1,1,1]
i = 1
rec(0,[1,1,2,1])
[1,1,1,1,1]
[1,1,1,2]
[1,1,2,1]
[1,1,3]
.....
[1,4]
.....
[5]

Write the output that is printed after the call to [Edit:] rec(5) completes

(This is an INLINE question) Ans:

[1, 1, 1, 1, 1] without the newlines

[1, 1, 1, 2]

[1, 1, 2, 1]

[1, 1, 3]

[1, 2, 1, 1]

[1, 2, 2]

[1, 3, 1]

[1, 4]

[2, 1, 1, 1]

[2, 1, 2]

[2, 2, 1]

[2, 3]

[3, 1, 1]

[3, 2]

[4, 1]

[5]

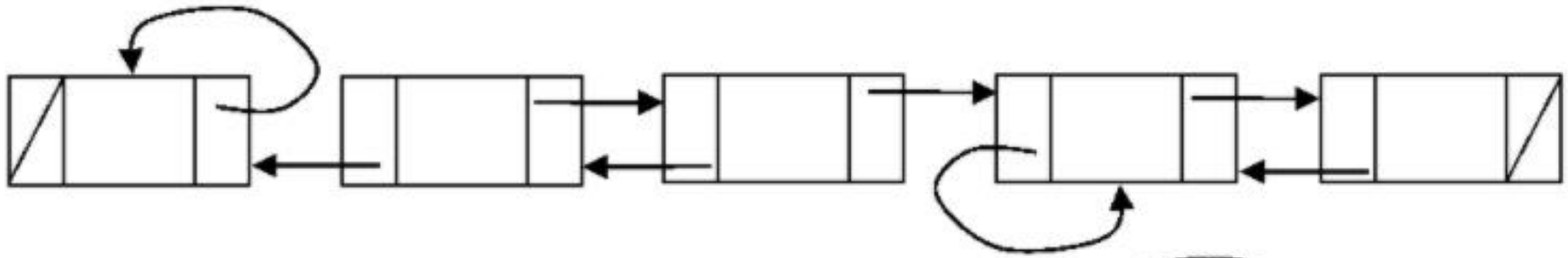
Question 5

Did you know that linked list can be used in files (and even in older file systems)? A large file need not be stored in one contiguous location on the disk. It can be stored as chunks of data (Nodes) that reference the next chunk. Consider a doubly-linked implementation that is NOT circular:

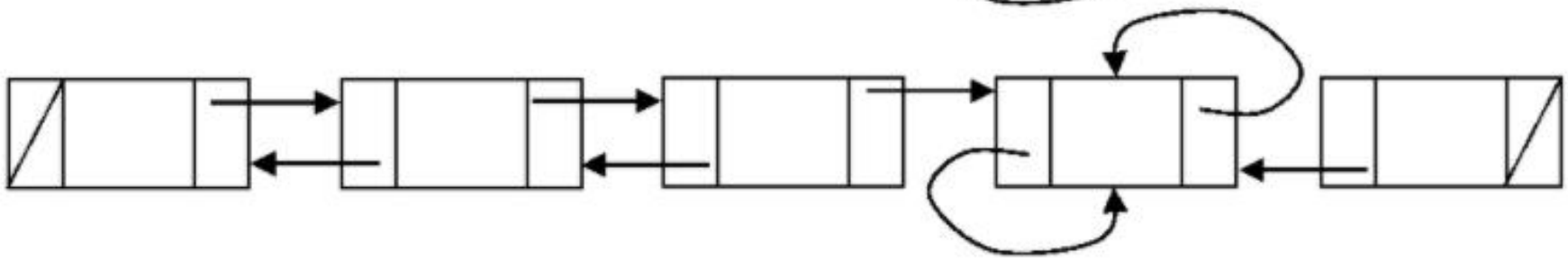
```
class Node {
    public Node next, prev;
    public YourFileData data;
    // omitted constructors -
   // doesn't concern you in this
   // question
   // accessors(getters),
   // mutators(setters)
}
```

```
class YourFile {
    private Node head, tail;
    private int numNodes;
// omitted some other usual methods to copy-construct,
// append, remove, clear, search which don't concern you
// in this question now some other methods which DO
// concern you in this question
    public int countForwardValidRefs() ...
    public int countBackwardValidRefs () ...
    public boolean canBeFixed() ... // for part (a)
    public void repair() ... // for part (b)
}
```

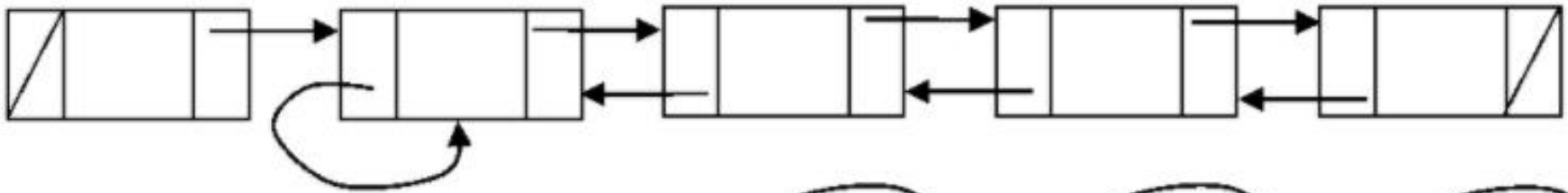
- For convenience, the Node attributes have been made public, so that you may either use getters/setters in your code, or access the attributes directly (without penalty)
- In this fictitious scenario, due to a previous bug while editing the file, the prev and/or next reference(s) of 0, 1 or more Nodes were corrupted, in that they point back to the same Node. Oh no! The head, tail, and numNodes attributes still store the correct number of nodes though



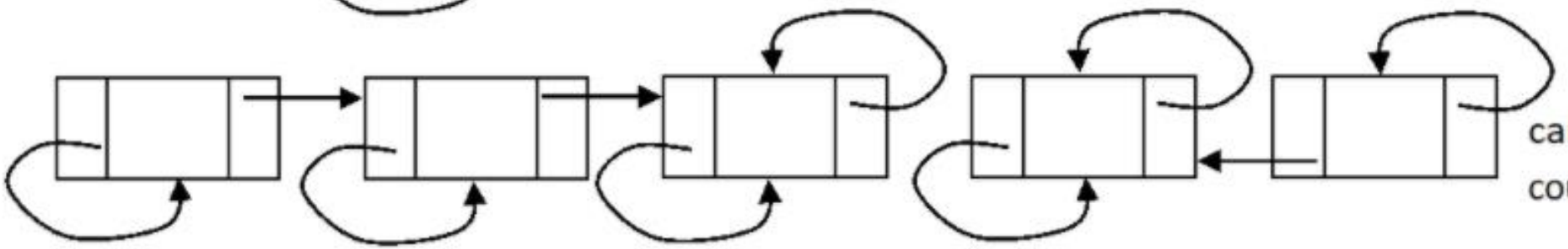
canBeFixed() -> false



canBeFixed() -> true
countForwardValidRefs() -> 3



canBeFixed() -> true
countForwardValidRefs() -> 5
countBackwardValidRefs() -> 3



canBeFixed() -> true
countForwardValidRefs() -> 2

Fortunately, Yuan Bin has written 2 methods, one for each direction. For a given direction, the method helps count the number of consecutive nodes that have good links in that direction. You do NOT need to understand the internal working of each method, but here it is:

```
public int countForwardValidRefs() {
   int valid = 0;
   for (Node curr = head; valid < numNodes && curr.next != curr;
                                                valid++, curr = curr.next);
   return valid;
}

public int countBackwardValidRefs(){
    int valid = 0;
    for (Node curr = tail; valid < numNodes && curr.prev != curr;
                                                valid++, curr = curr.prev);
    return valid;
}
```

a) Write Java code to complete the canBeFixed() method which returns whether the linked list can be repaired, such that the prev and next pointers are all restored to how a linked list should look like. If there is no error, canBeFixed() should return true
[Line limit 24]

```
public boolean canBeFixed() {
    return countForwardValidRefs() + countBackwardValidRefs() >= numNodes – 2;
}
```

b) Write Java code to complete the repair() method which repairs the linked list, if it can be fixed. For bonus credit (+3 marks), this should be done in O(N) time. [Line limit 40]

```
public void repair()  {
    if (numNodes == 0 || !canBeFixed()) return; // can't fix
    head.prev = null;
    tail.next = null; // ends handled differently
    int forwardTimes = Math.min(countForwardValidRefs(), numNodes - 1),
        backwardTimes = Math.min(countBackwardValidRefs(), numNodes - 1);
    Node forwardNode = head, backwardNode = tail;
```

```
for (int count = 0; count < forwardTimes; count++) {
        forwardNode.next.prev = forwardNode;
        forwardNode = forwardNode.next; // going fwds, fix backward links
}
for (int count = 0; count < backwardTimes; count++) {
        backwardNode.prev.next = backwardNode;
        backwardNode = backwardNode.prev; // going backwards, fix fwd links
}
if (forwardTimes + backwardTimes == numNodes - 2) { // special case
        forwardNode.next = backwardNode; // reattach 2 parts
        backwardNode.prev = forwardNode;
}
```