0-cheatsheet-and-notes

Contents

General notes

1. If the task seems completely illogical, do not read the problem and take it at face value, they are probably misusing the terms to confuse you.
2. When solving and issue, start off by writing down the issue, on paper, then ask questions and details, write the answers too.
3. When the task or problem is related to the topics below, write down everything you know about the topic, i.e linked lists, stacks, queues, trees, graphs, arrays, strings, recursion, sorting etc.
4. Write multiple implementations starting off with very rough draft, of simple pseudo code, go through it, test it, check if it makes sense, then refine it, into actual code, over multiple drafts.
5. Do not write any boilerplate code, just roughly draft it over, or even write it down in plain text, i.e create class with the following fields, instead of actually writing down the class
6. Come up with the base case, or a very simple small test case which you can use to go through the algorithm to do rough validation, then come up with a more convoluted bigger test case to cover the edge cases

Big notation

1. Big 0 is NOT about absolute values, of time or/and space, it is about how the algorithm is scaling, with respect to some input

```
// scale linearly with N, the complexity is just bigO(n)
for (int i = 0; i < N; i++) {
    doSomeWork();
}

// scale linearly with N, multiplier is irrelevant, the complexity is
   still bigO(n)
for (int i = 0; i < 100000 * N; i++) {
    doSomeWork();
}

// scale with N^2, quadratic complexity, therefore the complexity is bigO
   (n^2)
for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) {
        doSomeWork();
    }
}
```

2. Establish what the N is, is it the count of elements in an array, is it the number of function calls, is it the depth of a tree, how is it co-related to the work being done in this algorithm. The unit of N, is very important.

```
// n is the size of the input, the complexity is then bigO(n)
for (int i = 0; i < array.size; i++) {
    doSomeWork();
}

// n is the input, but the complexity is not bigO(n), because for
// each n, we make two calls to the function, the complexity is
// bigO(2^n), where 2 is the branching factor and n is the depth to
// which we go, for n = 2, the depth of the recursive calls will be 2,
// for n = 4, the depth of the recursive calls would be 4

// O(2^N) are often recursive algorithms that solve a problem of size N
// by recursively solving two smaller problems of size N-1.
int f(int n) {
    if(n <= 1) {
        return 1;
    }
    return f(n - 1) + f(n - 1);
}
```

3. Drop non-dominant terms, constants or other inputs, if the complexity ends up being, know your inputs, if there are multiple, do they have any relation

   - O(N^2 + N) = O(N^2) - quadratic, (N is not the dominant)
   - O(N^3 + log2(N)) = O(N^3) - cubic (log is non-dominant)

- O(N^2 + 10*N) = O(N^2) - quadratic (linear 10N is non-dominant)
- O(2^N + 1000*N) = O(2^N) - quadratic (linear 1000N is non-dominant)
- O(2^N + B) = assume B is much smaller than N, then - O(2^N)

4. Amortized complexity, occurs when there is a very small chance the algorithm would perform outside the tight boundaries i.e inserting an element into a static array, at some point we would fill it up and have to resize it which is bigO(n) time, but for the most part inserting in static array is bigO(1). The amortized complexity is bigO(1).

5. Log of base two is what we usually use in most algorithms, remember that, log2(n) = k <==> 2^k = n. The result of the log is the power on which we have to raise the number 2 to get n as a result. Log runtimes come up most in binary search and in quick sort

6. Know your scales, in ascending order in relation to N, we have the 6 most used ones,

   - constant - bigO(1) (linked list add, array add, hash-map get)
   - log - bigO(log(N)) (binary search)
   - linear - bigO(N) (array find)
   - n * log(n) - bigO(N * log(N)) (quicksort)
   - polynomial N^M - bigO(N^M) (print 2d array)
   - expnential 2^N - bigO(2^N) (f(n-1) + f(n-1))

Best Conceivable Time (BCR)

1. Some problems, which seem to be n^2 can be actually linear. Best Conceivable run-time tells us the lowest boundary of complexity a given algorithm can take, where no faster algorithm can be conceived. For example take a look at if the problem to be solved is restricted in special ways

```
// given the two sorted arrays, find all common elements within them
// (13, 27, 35, 40, 49, 55, 59), (17, 35, 39, 40, 55, 58, 60)

// most optimal solution since both arrays have the same size and they
// are already sorted is to go through them at the same time,
// instead of trying to go with the n^2 solution or saving in
// hash-map, and then looking in the other array, below is O(N)
int firstIterator = 0;
int secondInterator = 0;
List<String> results = new ArrayList<>();
while (firstIterator < first.size() && secondInterator < second.size()) {
    Integer firstElement = first.get(firstIterator);
    Integer secondElement = second.get(secondInterator);

    if (firstElement.equals(secondElement)) {
        // if both pointers point to the same integer value element, then
            move
        // both forward and remember the value which was matching
        firstIterator++;
        secondInterator++;
        matching.add(firstElement);
    } else if (firstElement < secondElement) {
        // the first element is smaller than the second therefore we move
            the pointer
        // of the first array forward, only
```

```
            firstIterator++;
        } else {
            // the second element is smaller than the first therefore we move
                the pointer
            // of the second array forward, only
            secondInterator++;
        }
    }
    return results.size();
```

2. Leverage additional data structures or algorithms, i.e hash tables, quick sort, sets or trees etc. This way the time complexity will drop, be careful with the space complexity, try to find a balance, do not blow the one for the other

```
    // pairs from (0, 1, 2, 3, 4, 7, 10, 11, 12, 13, 15), that sum to 13
    // first we sort the array of unique numbers, this would allow us to
    // do a quick binary search over the entire array
    input = input.stream().sorted().collect(Collectors.toList());
    Object[] entries = input.toArray();

    List<String> results = new ArrayList<>();
    for (Integer entry : input) {
        // we know what value we are looking for, since we know the sum = 13
        // therefore x + current = sum, or x = sum - current, we can lookup
        // the value of x, if it is truly present in the array
        Integer lookup = sum - entry;
        if (Arrays.binarySearch(entries, lookup) != -1) {
            // the value of the lookup was present in the array, add it
            sums.add(String.format("%d + %d = %d", entry, lookup, sum));
        }
    }
    return results.size();
```

Arrays and Strings

1. Static linear arrays are of fixed size, where inserting and element is constant time, until the array is full, after it is full insertion is not possible, unless array is re-sized, see amortized complexity.

2. When constructing strings, keep in mind that most languages offer immutable string objects, therefore use an alternative which would allow you to append, cut and prepend such as `StringBuilder`

```
    // { "one", "two", "three", "four", "five", "six", "seven", "eight", "
       nine" }
    // On each new word, a copy of the previous result would be made, and the
        new
    // word appended to it, making this complexity O(n^2), where n is the
    // number words to append from the words list. Use String Builder instead
    String[] words = { ... };
    String result = ""
    for (String word : words) {
       result = result + word
    }
    return result;
```

3. Permutations of a string, very often required algorithm question, how to generate all Permutations of a string or how to check if a longer string contains a permutation of another shorter string within it.

```
    // take the input shown below, and find all permutations of it
    // given "abcd" - it has total of 4 * 3 * 2 * 1 = 4! = 24 perms

    if (input.length() == 1) {
        // a string input with length 1, has no permutations, or
        // think of it like it has only one permutation, itself
        return Arrays.asList(input);
    }
    // pull the last element from the input, and remember it
    String suffix = input.substring(input.length() - 1);

    // new input string, is the original with the tail cut off
    String cut = input.substring(0, input.length() - 1);

    // generate list of permutations for the new cut sub string
    List<String> permutations = permutateStringHelper(cut);

    // we hold the actual final result here
    List<String> result = new ArrayList<>();

    // for each permutation of the cut string, put/stick last element of
    // the original at both ends first, and then in between the string
    // too. thus the new string result will contain the last character
    // of the original in each position
    for (String perm : permutations) {
        // add at both ends of the permutated string
        result.add(perm + suffix);
        result.add(suffix + perm);

        // add it, in between the permutated strings
        for (int i = 0; i < perm.length() - 1; i++) {
            String head = perm.substring(0, i + 1);
            String tail = perm.substring(i + 1, perm.length());
            result.add(head + suffix + tail);
        }
    }
    return result;
```

4. Removing and inserting elements to an array. Could be done in the following ways

   - removing by shifting elements to the left - this is done by traversing the array from the position of removal, to the right, assigning the next element to the current element and moving until the end of the array, once at the end, the final two elements would be duplicated, therefore we can reduce the size of the array by 1

```
    // start from the end of the array, move backwards,
    for (int i = n - 1; i > k; i--) {
        a[i] = a[i - 1];
    }
    a[k] = 0
```

- inserting an empty space - by shifting elements to the right, similar to the approach above, however we start from the end of the array, moving elements to the right, making space for new elements instead, this assumes the array already has sufficient size to hold the new element or elements.

Linked lists

1. The most important properties of the linked list, generalized, given a node N a new element can be inserted after (for singly linked) or before and after for doubly linked lists. More specifically we can fast insert elements at the head or the tail.
2. Operations - creation is as simple as adding new nodes to the tail, deletion is simply detaching the node from other nodes which reference it and also fixing the links of these nodes to point to valid elements

```
    // construct list from (a, b, c, d, e, f, g, h, ...)

    // for each element in the input array of elements
    if (head == null) {
        head = new Node();
        head.value = entry;
        tail = head;
    } else {
        Node next = new Node();
        next.value = entry;
        tail.next = next;
        tail = next;
    }
```

```
    // initialization
    prev = head;
    curr = head;

    // move while curr != null
    if (curr.value == value) {
        if (prev != null) {
            // detach curr from prev
            prev.next = curr.next;
        }
        return curr;
    }
    prev = curr;
    curr = prev.next;
```

3. Runner approach - very important, very common, the runner approach uses two pointers into the list which advance at different pace, this can be used to inspect the list into the future, or re-arrange it while iterating over it

```
    // - `a1 - a2 - a3 - a4 - b1 - b2 - b3 - b4` - input
    // - `a1 - b1 - a2 - b2 - a3 - b3 - a4 - b4` - result
    Node slow = advance(head, 1)
    Node fast = advance(head, N)

    while(notAtTail) {
    }
```

4. Linked list loops - using the two pointers approach for fast and slow runner, or the hare and tortoise, a very important property is that no matter the number of steps the fast and slow pointer move forward, as long as the step is integer, both pointers will meet at some point at some node in the list

```
    Node slow = node;
    Node fast = node;

    // move both at different paces, if there is a loop in the list, they
       will
    // always meet at some point within this loop
    while (fast != null and fast.next != null) {
        slow = slow.next;
        fast = fast.next.next;

        // they met, there is a loop in the list, the same exact node, by
        // reference is contained in the list twice
        if (slow == fast) {
            break;
        }
    }
```

5. Recursive iteration - asked very often when working with linked lists, why is it useful? If we use a singly linked list, and we would like to iterate it forward, till the tail, and then backward, we can use recursion to do that.

```
     void recursive(Node node, int accum) {
        if(node == null) {
            return;
        }
        // pre-recursive call, will print the list in forward direction
        int curr = accum + 1
        println(node.value, curr);

        // this will drill down, till the tail, then start post recursive
        // unwinding of the function call stack, starting from the tail,
        recursive(node.next, curr);

        // post-recursive call, will print the list in reverse direction
        println(node.value, curr);
     }
```

Stack and Queue

1. Interface - be careful with the interface for both stack and queue, in java for example the peek and remove/pop would throw if used on empty stack, therefore you have to use isEmpty, to check first if you want to peek or pop

2. Iterations - the most common pattern for iterating over a stack or queue is to pop or remove elements until the stack or the queue are empty.

```
Stack<Integer> stack = new Stack<>();
while (!stack.isEmpty()) {
    stack.pop()
}

Queue<Integer> queue = new Queue<>();
while (!queue.isEmpty()) {
    queue.remove()
}
```

3. Ordering - remember that you can move all elements from one stack, to another, and the second stack would contain the same elements like the first one but in reverse order, this is often used to insert elements in the middle of a stack. That would also make it so that the second stack actually works as a queue, since the order is reversed and the top of the second stack contains the first element that was ever inserted. For the queue the rule is not the same, i.e removing from one queue and inserting into another, would effectively simply construct the same queue, since the removal is from the head of the queue and inserting is always at the end of the queue.
4. Recursion - recursive algorithms can be implemented in an iterative manner using a stack. This is useful when traversing graphs in depth, for example
5. Implementation - both usually are implemented using a linked list, doubly or singly, does not really matter. One could also use a static array as well, but the interface would not provide O(1), or rather the complexity will be O(1) but amortized