Arun Agarwal Basic Algorithm Analysis

1. $O(n^2)$
2. $O(n^4)$
3. $O(n^2)$
4. $\frac{n(n+1)(2n+1)}{6} \rightarrow O(n^3)$
5. $O(1)$
6. $O(1)$
7. $O(n)$
8. $O(n)$
9. $O(n \log n)$
10. $O(n^3)$ → just getting an item from a specific index
11. get() is always a constant time $\boxed{O(1)}$ operation
12. remove() runs in linear $\boxed{O(n)}$ time because we have to iterate the entire array, in this case half of it to find the element qualifying for removal
13. The cost of a remove is O(n) because you have to shuffle the elements above/after that point "left" by one.

```
for(int i=n+1; i<arr.length; i++)
{
    arr[i-1]=arr[i] }
```

← only $\frac{n}{2}$ elements

14. Since you are simply adding the item to the end of the arraylist, the time complexity is constant (O(1)). However, if you were to add it at a certain index, it would take O(n) time.
15. Again, simply adding an element to the end of an arraylist is not an expensie operation, so its best and worst time complexity is $\boxed{O(1)}$. However, if you need to reallocate, you will need to go through the for-loop that moves the items from the original list to the new list, making the time complexity $\boxed{O(n)}$.
16. when we have number or data to enter or store. Because remaving and adding ~~too~~ are expensive operations.
17. The Redrata Path Problem, which some refer to as the Hamiltonian Path Problem is a problem in graph theory of determining whether there exists a path in an undirected or directed – in our case, undirected – graph that passes through each vertex exactly one. This problem is NP-complete, a problem that can be solved by a class of brute force algorithms and it can be used to simulcte any other problem with a similar algorithm. This corresponds to

the given puzzle because the map can be seen as an undirected graph with nodes referring to the stars and the paths branching out of the nodes being the possible roads to take from star to star. The goal for this graph is to travel to each and every vertex exactly once, following the definition of the Redrata Path Problem. A little addition to the Redrata Path problem for this case is that we have to follow a set start and end (but these can just be added as additional nodes in the graph).

(18) The number of different Hamiltonian cycles in a complete undirected graph of n vertices in $(n-1)!/2$

For example, if there are 10 verticos, the max paths ~~1st~~ 1 vertex could have is to the 9 other vertices $(n-1)$, the max paths the 2nd vertex could have ~~is to~~ without repeating is 8, and so on. This is the same as $(n-1)!$, and since it is an undirected graph, it would be $\frac{(n-1)!}{2}$ paths.

(19) The worst case time complexity is $O(\infty)$.

(20) The bogosort algorithm has no upper bound, so it can run infinitely. There is no gaurentee that a random shuffle will ever produce a sorted sequence.

(21) The average case runtime would be $O(n! \cdot n)$.

(22) There are $n!$ permutations, only one of which is correct (if there are distinct elements). You would then expect the right answer after about $O(n!)$ iterations. However, each shuffle/check operation (the for loop) is itself $O(n)$ run time. Therefore, it is $O(n \cdot n!)$ runtime overall.

(23) see other document

```
//2.1: Uniqueness:
public static <E> boolean unique(List<E> list)
{
        for (int i = 0; i < list.size() - 1; i++)  O(n)
        {
                for (int j= i + 1; j < list.size(); j++) O(n)
                {
                        if (list.get(i).equals(list.get(j)))
                        {
                                return false;  O(1)
                        }
                }
        }
        return true;  O(1)
}
```

Time Complexity:
$O(n) * O(n)$

$\boxed{O(n^2)}$

```
//2.2: All Multiples:
public static List<Integer> allMultiples(List<Integer> list, int integer)
{
        List<Integer> newList = new ArrayList<Integer>();  O(1)

        for (int i = 0; i < list.size(); i++)  O(n)
        {
                if (list.get(i) % integer == 0)
                {
                        newList.add(list.get(i));  O(1) (at the end of list)
                }
        }
        return newList;  O(1)
}
```

Time Complexity:
$\boxed{O(n)}$

```
//2.3: All Strings of Size:
public static List<String> allStringsOfSize(List<String> list, int length)
{
        List<String> newList = new ArrayList<String>();  O(1)

        for (int i = 0; i < list.size(); i++)  O(n)
        {
                if (list.get(i).length() == length)
                {
                        newList.add(list.get(i));  O(1)
                }
        }
        return newList;  O(1)
}
```

Time Complexity:
$\boxed{O(n)}$

```
//2.4: isPermutation:
//TO DO IN CLASS WEDNESDAY
public static <E> boolean isPermutation(List<E> list1, List<E> list2)
{
        if (list1.size() != list2.size())
        {
                return false;    O(1)
        }

        for (E item: list1) O(n)
        {
                int count1 = 0;  O(1)
                int count2 = 0;  O(1)

                //count the number of times item appears in A
                for (int i = 0; i < list1.size(); i++) O(n)
                {
                        E otherItem = list1.get(i);O(1)
                        if(item.equals(otherItem)) O(n)
                                count1++; O(1)
                }
                //count the number of times items appears in B
                for (int i = 0; i < list2.size(); i++) O(n)
                {
                        E otherItem = list2.get(i); O(1)
                        if(item.equals(otherItem)) O(n)
                                count2++; O(1)
                }

                if(count1 != count2)
                {
                        return false; O(1)
                }
        }
```

Time complexity:

O(n) * O(n)

$\boxed{O(n^2)}$

```
//2.5: String To List of Words:
public static List<String> stringToListOfWords(String words)
{
        String[] eachWord = words.split("\\s+"); O(n)

        List<String> newList = new ArrayList<String>(); O(1)

        //extra credit part to remove punctuation
        //also the part to get the items in the array into an arraylist
        for (int i = 0; i < eachWord.length; i++) O(n)
        {
                eachWord[i] = eachWord[i].replaceAll("[^a-zA-Z]", ""); O(1)

                if (!eachWord[i].equals("")) O(n)
                        newList.add(eachWord[i]); O(1)
}
        return newList; O(1)
}
```

Time complexity:

O(n) + O(n)

$\boxed{O(n)}$

```
//2.6: Remove All Instances:
public static <E> void removeAllInstances(List <E> list, E something)
{
                for (int i = 0; i < list.size(); i++)  O(n)
                {
                        if(list.get(i)==something)
                        {
                                list.remove(i);  O(n)
                                i--;   O(1)
                        }
                }
```

Time complexity:
$O(n^2)$