1. $\underline{k}$

1 $\quad T(n) = 2T(n/3) + n \qquad\qquad T(1) = 1$

$T(n/3) = 2T(n/9) + n/3$

2 $\quad T(n) = 2(2T(n/9) + n/3) + n$

$T(n/9) = 2T(n/27) + n/9$

3 $\quad T(n) = 2(2(2T(n/27) + n/9) + n/3) + n$

$$T(k) = 2(2(2T(n/3^k) + n/3^{k-1}) + n/3^{k-2}) + n/3^{k-k}$$

$$= 2^k T(n/3^k) + 2^{k-1}\, n/3^{k-1} + 2^{k-2}\, n/3^{k-2} + 2^{k-k}\, n/3^{k-k}$$

$$= 2^k T(n/3^k) + (2/3)^{k-1} n + \ldots\ldots + (2/3)^0 n \qquad [n = 3^k]$$

$$= 2^k T(3^k/3^k) + (2/3)^{k-1} \times 3^k + \ldots\ldots + (2/3)^0 \times 3^k$$

$$= 2^k T(1) + (2/3)^{k-1} \times 3^k$$

$k = \log_3 n$

$$T(n) = 2^{\log_3 n} + (2/3)^{\log_3 n - 1} \times 3^{\log_3 n}$$

$$\Rightarrow T(n) = 2^{\log_3 n} + \cancel{n} \times \frac{2^{\log_3 n}}{\cancel{n}} \times 3/2$$

$\therefore$ Ans $\Rightarrow T(n) = 2^{\log_3 n} + 3/2(2 \log_3 n)$

Scanned by CamScanner

2\. k

1 $T(n) = T(n-1) + \log n$, $T(1) = 1$

$T(n-1) = T(n-2) + \log(n-1)$

$[n = 2^k]$

2 $T(n) = T(n-2) + \log(n-1) + \log n$

$T(n-2) = T(n-4) + \log(n-3) + \log(n-2)$

3 $T(n) = T(n-4) + \log(n-3) + \log(n-2) + \log n$

$T(k) = T(n-2^{k-1}) + \log(n-(2^{k-1}-1)) + \log(n-(2^{k-1}-2)) + \log(n-(2^{k-1}-k))$

$= T(2^k - 2^{k-1}) + \log(2^k - 2^{k-1}+1) + \log(2^k - 2^{k-1}+2) + \log(2^k - 2^k + k)$

$= T(2^k(1-\frac{1}{2})) + \log(2^k(1-\frac{1}{2}) + \frac{n(n+1)}{2})$

$= T(2^{k-1}) + \log(2^{k-1} + \frac{2^k(2^k+1)}{2})$

$= T(2^{k-1}) + \log(2^{k-1} + 2^{k-1}(2^k+1))$

$\Rightarrow T(2^{k-1}) + \log(2^{k-1}(2^k+2))$

$\therefore$ Ans: $T(n) = T(n-1) + \log((n-1)(n+2))$

3. S =

0	1	2	3	4	5	6	7
85	24	63	45	17	31	96	50

Pivot: 17

[ ] [17] [85 | 24 | 63 | 45 | 31 | 96 | 50]

Pivot: 45

[17 | 24 | 31] [45] [85 | 63 | 96 | 50]

Pivot: 85

[17 | 24 | 31 | 45 | 63 | 50] [85] [96]

Pivot: 63

[17 | 24 | 31 | 45 | 50] [63] [85 | 96]

Pivot: 50

[17 | 24 | 31 | 45] [50] [63 | 85 | 96]

17	24	31	45	50	63	85	96
0	1	2	3	4	5	6	7

```
4. public void Mergesort (int LowI, int highI){
      if(lowI<highI){
            int middleI = lowI+((highI-lowI)/2);
            Mergesort(lowI, middleI);
            MergeSort(middleI+1, highI);
            mergeEverything(lowI, middleI, highI);
      }
}

      int count = 0;

public void mergeEverything(int LowI, int middleI, int highI){
      for(int i = lowI; i< highI; i++){
            tempA[i] = A[i];
      }
      int i = lowI;
      int j = middleI + 1;
      int k = lowI;
      while (i≤ middleI && j < highI){
            if(tempA[i] ≤ tempA[j]){
                  A[k] = tempA[i];
                  i++;
            }else{
                  A[k] = tempA[j];
                  j++;
                  count ++;
            }
            k++;
      }
      while (i≤middleI){
            A[k] = tempA[i];
            k ++;
            i ++;
      }
}
System.out.println("# of inversions ="+ count);
```

5. (1) $f(N) = N + c$

- Initial size = 0
- No. of operations on the array depends on size of c. eg:- If c = 1, then...
  0+1 = 1; 1+1 = 2; 2+1 = 3; 3+1 = 4...
  However if c = 1000, then.
  0+1000 = 1,000; 1000+1000 = 2000....
- Requires ($f(N)$ + N + 1+c) units of time to add a new value, once array is full.
- Extending from previous point, the number of times required to copy over the array to the bigger array depends on "C". eg:- if c = 1, the older array has to be copied over n times (considering n values have to be put in). On the other hand, it can be copied over only once in n values need to be put in, & "C" = n
- Has the possibility of wasting a lot of memory if c is too large at the cost of saving the number of times the array needs to grow. eg:-

$f(N) = \underbrace{N}_{0} + \underbrace{C}_{10,000}$ # values = 5,000

10,000 - 5,000 = 5,000 spaces empty.

(2) $f(N) = 2N$

- Initial size = 1
- Will always double in size when it grows.
  eg:- $1 \rightarrow 2 \rightarrow 4 \rightarrow 8 \rightarrow 16 \rightarrow 32 \rightarrow 2^N$
- Requires ($2f(N)$ + N + 1) units of time to add a new value, once array is full.
- Extending from previous point, the number of times the array grows and has the old array copied over depends entirely on the # of values need to be input.
  eg:- Array is currently at 2 spaces free, & requires 8 new values to be input. It will grow once at the rate of 2^N, and 6 spaces will be occupied, leaving $2^N - 6$ spaces free.
- Also has a possibility of wasting space but comparitively lesser. The more # of times it grows, the higher the probability of wasting space.

eg:- $f(N) = 2\underbrace{(N)}_{2^{12}}$ # values = 5,000

$2 \times 2^{12} = 2^{13}$

$2^{13} - 5,000 = 3,192$ spaces empty.
Grows 13 times but wastes only 3,192 spaces.

· Growth rate of array cannot be predicted unless "C" is known.	· The rate at which this array grows is more predictable.
· Runtime is still O(N)	· Runtime still O(N)
· Design and/or implemen-tation is stable how-ever an inexcerienced programmer might end up adding the value of "c" to the last free cell rather than adding c spaces to the larger array.	· Design and/or implemen-tation is more stable. Although an inexperienced programmer might end up multiplying each value in each cell by 2 insted of doubling the size.
· Slightly better to eliminate unneccessarily copying values over and over between arrays. Wastes more space.	· Slightly better to eliminate unneccessary wastage of space.

Personally, I believe (1) $f(N) = N + c$ is a better strategy to increase the size of the array in stacks. With careful planning and good predict-ions, this could be ~~implented~~ implemented, to not waste too much space while not having to copy the values to many times.