**Time complexity**

* Merge sort faster than selection sort.
* Largest element in array- o(n)
* Bubble sort- o(n^2)
* Linear search worst case- o(n)- last element or not in array at all
* Insertion sort worst case - o(n^2); best case- o(n)
* Selection sort worst case and best case - o(n^2)
* In theoretical analysis, time: counting number of unit operations
* For merging two sorted arrays of size m and n into a sorted array of size m+n, we require operations – o(m+n)
* Factorial: o(n)
* Binary search- o(log n)
* Recurrence relation for merge sort: T(n)=2T(n/2)+o(n)
* What is the time complexity of merge sort: o(n\*log n)
* T(n) = 2T(n/2) + Logn ans:o(n)

**Space complexity**

* Factorial: o(n)
* Bubble sort: o(1) constant space
* Binary search: o(log n)
* Merge sort: o(n)
* Fibonacci: o(n)

for(int i = 0; i < n; i++){

for(; i < n; i++){

cout << i << endl;

}

}

Time complexity: o(n)

for(int i = 0; i < n; i++){

for(int j = 1 ; j < k; j++){

cout << (i + j ) << endl;

}

}

Time complexity: o(n)

int multiplyRec(int m, int n){

if(n == 1)

return m;

return m + multiplyRec(m, n - 1);

}

Time complexity: o(n)

int sumOfDigits(int n){

int sum;

if(n < 10){

return n;

}

sum = (n % 10) + sumOfDigits(n / 10);

return sum;

}

Time complexity: o(log n) to base 10

long fib(int n){

if(n == 0 || n == 1){

return n;

}

return fib(n - 1) + fib(n - 2);

}

Time complexity: o(2^n)

while(n){

j=n;

while(j>1){

j-=1;

}

n/=2;

}

Time complexity: o(n)

void fun2(int m, int n = 1)

{

if (n <= 0)

return;

if (n > m)

return;

fun2(m, 2\*n);

}

Time complexity: o(log m)

void sample(int n) {

for (int p = n; p > 0; p=p/2) {

for (int q = 0; q < p; q++) {

print(p, “ ”, q);

}

}

}

Time complexity: o(n): The innermost statement of function sample() is executed following times: n + n/2 + n/4 + ... 1, which is equal to n (calculated using sum of GP formula). So time complexity T(n) can be written as T(n) = O(n + n/2 + n/4 + ... 1) = O(n).

for(int i = 0; i < n ; i++){

int k = n;

while(k > 0){

k/=2;

}

}

Time complexity: o(n\*log n)

while(n > 0){

n = n / 4;

}

Time complexity: o(log n) base 4

int i, j, k=0;

for(i = n/2; i<=n; i++){

for(j=2; j<=n; j\*=2){

k = k + n/2;

}

}

Time complexity: o(n\*log n): j keeps doubling till it is less than or equal to n. Number of times, you can double a number till it is less than n would be log(n). So, j would run for O(log n) steps. i runs for n/2 steps. So, total steps ` = O (n/ 2 \* log (n)) = O(n logn)

int count = 0 ;

for(int i = N; i > 0; i/=2){

for(int j=0; j < i; j++){

count += 1;

}

}

Time complexity: o(n): In the first iteration, the j loop runs N times. ####In the second iteration, the j loop runs N / 2 times. ####In the ith iteration, the j loop runs N / 2^i times. ####So, the total number of runs of loop = N + N / 2 + N / 4 + … 1 = N \* (1 + 1/2 + 1/4 + 1/8 + … ) < 2 \* N