

```

getTimeComplexity(int n){
    if(n<5){
        print("n less than 5"+n);
    }
    else{
        for(i=0;i<n;i++){
            print(i);
        }
    }
}

```

$n < 5$ Best case
 Time complexity = $O(1)$
 $n \geq 5$ Worst case
 Time complexity = $(n+1)1$
 $= n+1$
 $= n$

```

i = 1;
while(i < n){
    Statements;
    i = i * 2;
}

```

the loops i is incrementing by 2 each time.

i
 1
 2
 2^2
 $2^2 \cdot 2$
 $2^2 \cdot 2^2$
 \vdots
 $2^2 \cdot 2^2 \cdot 2^2 \dots k \text{ times}$

terminates
 $i \geq n$
 $2^k \geq n$ Since i is
 $\log 2^k \geq \log n$
 $k \geq \log n$
 $k = \log n$
 So it's $O(\log n)$

```

vector<int> func(vector<int> &gl, int k, vector<int> step)
{
    int curr, count;
    int n = step.size();
    vector<int> intermed;
    for(int i = 0; i < n; i++)
    {
        count = 1;
        for(int j = i + 1; j < n; j++)
        {
            if(step[i] == step[j])
            {
                curr = step[i];
                count++;
            }
            if(step[i] != step[j])
            {
                curr = step[i];
                intermed.push_back(count);
                intermed.push_back(curr);
                i = j;
                curr = step[j];
                count = 1;
            }
        }
        if(i == n-1)
        {
            intermed.push_back(count);
            intermed.push_back(curr);
        }
    }
    gl.insert(gl.end(), intermed.begin(), intermed.end());
    return intermed;
}

```

```

i = 0;
while(i < n){
    statements;
    i++;
}

```

$i = 0$ → 1
 $\text{while}(i < n)$ → $n+1$
 statements → 1
 $i++$ → 1
 Total time complexity
 $= (n+1)1$
 $= n+1$
 $= n$
 So its $O(n)$

```

void retry(int k, vector<int> &gl, vector<int> step) {
    try {
        gl.at(k);
    } catch (const out_of_range& oor) {
        step = func(gl, k, step);
        retry(k, gl, step);
    }
}

int kthDigit(int k)
{
    vector<int> step = {1,0};
    vector<int> gl = {0,1,0};
    retry(k, gl, step);
    return gl.at(k-1);
}

```

Build Solution

$$\begin{aligned}
 T(n) &= 2T\left(\frac{n}{2}\right) + 4n \\
 &= 2\left[2T\left(\frac{n}{2^2}\right) + 4\frac{n}{2}\right] + 4n \\
 &= 2^2\left[2T\left(\frac{n}{2^3}\right) + 4\frac{n}{2^2}\right] + 4n + 4n \\
 &= 2^3T\left(\frac{n}{2^3}\right) + 4n + 4n + 4n \\
 &= 2^3\left[2T\left(\frac{n}{2^4}\right) + 4\frac{n}{2^3}\right] + 4n + 4n + 4n \\
 &= 2^4T\left(\frac{n}{2^4}\right) + 4n + 4n + 4n + 4n \\
 &= 2^i T\left(\frac{n}{2^i}\right) + i(4n)
 \end{aligned}$$

$\frac{n}{2^i} = 1 \Rightarrow n = 2^i \Rightarrow \log_2 n = i$

Expand Scratch

$$\begin{aligned}
 T\left(\frac{n}{2}\right) &= 2T\left(\frac{n}{2^2}\right) + 4\left(\frac{n}{2}\right) \\
 T\left(\frac{n}{2^2}\right) &= 2T\left(\frac{n}{2^3}\right) + 4\left(\frac{n}{2^2}\right) \\
 T\left(\frac{n}{2^3}\right) &= 2T\left(\frac{n}{2^4}\right) + 4\frac{n}{2^3}
 \end{aligned}$$

$$2^{\log_2 n} = n^{\log_2 2} = n$$

$$\begin{aligned}
 &\Rightarrow 2^{\log_2 n} T(1) + (\log_2 n)(4n) \\
 &= n(4) + 4n \log_2 n \\
 &= 4n + 4n \log_2 n = O(n \log n)
 \end{aligned}$$

```
int sumProductDigit(int a, int b)
{
    int ans = a*b;
    if(ans == 0)
    {
        return 0;
    }
    else if(ans % 9 == 0)
    {
        return 9;
    }
    else
    {
        return ans % 9;
    }
    // the digit sum of an integer = int % 9
}
```

```
int findKthSmallest(vector<int> A, vector<int> B, int k) {
    // Move elements from vector B to the end of vector A
    A.insert(A.end(), B.begin(), B.end());
    sort(A.begin(), A.end());
    return A[k-1]; // you will need to change this
}
```

```
void zigzagSort(vector<int> &nums) {
    bool flip = true;

    for (int i=0; i<nums.size()-2; i++)
    {
        if (flip) // allows the if else 1
        {
            if (nums[i] > nums[i+1])
                swap(nums[i], nums[i+1]);
        }
        else
        {
            if (nums[i] < nums[i+1])
                swap(nums[i], nums[i+1]);
        }
        flip = !flip; // flip the operation
    }
}
```

```
ListNode* insertionSortList(ListNode* head) {
    ListNode *start = head;

    ListNode *insert = start; // insertion ptr to insert data

    head = head->next;

    while(head != NULL){ // avoid seg fault
        insert = start;
        //head = head->next;
        while(insert != head){
            if(insert->val < head->val){
                int temp = head->val; // store head's data so it is not lost
                head->val = insert->val; // move the smaller int into location
                insert->val = temp; // move the larger int into location
            }
            else{
                insert = insert->next;
            }
        }
        head = head->next;
    }
    return start;
}
```

```
ListNode* findCycleStart(ListNode* head) {
    if (head == NULL || head->next == NULL) // return NULL;

    ListNode *slow = head;
    ListNode *fast = head;

    slow = slow->next;
    fast = fast->next->next; // initialize two pointers

    while (fast && fast->next) {
        if (slow == fast)
            break;
        slow = slow->next;
        fast = fast->next->next; // test for pointer
    }

    if (slow != fast){
        return NULL;
    }

    slow = head;
    while (slow != fast) {
        slow = slow->next;
        fast = fast->next; // if loop exists, pointer
    }

    return slow;
}
```

$$\begin{aligned}
 a &= b^{\log_b a}, \\
 \log_c(ab) &= \log_c a + \log_c b, \\
 \log_b a^n &= n \log_b a, \\
 \log_b a &= \frac{\log_c a}{\log_c b}, \\
 \log_b(1/a) &= -\log_b a, \\
 \log_b a &= \frac{1}{\log_a b}, \\
 a^{\log_b c} &= c^{\log_b a},
 \end{aligned}$$

Heapsort – $O(n \lg n)$

```

MAX-HEAPIFY(A, i)
1  l ← LEFT(i)
2  r ← RIGHT(i)
3  if l ≤ A.heap-size and A[l] > A[i]
4    largest ← l
5  else largest ← i
6  if r ≤ A.heap-size and A[r] > A[largest]
7    largest ← r
8  if largest ≠ i
9    exchange A[i] with A[largest]
10   MAX-HEAPIFY(A, largest)

```

PARIENT(i)

```

1  return ⌊i/2⌋

```

LEFT(i)

```

1  return 2i

```

RIGHT(i)

```

1  return 2i + 1

```

BUILD-MAX-HEAP(A)

```

1  A.heap-size ← A.length
2  for i ← ⌊A.length/2⌋ downto 1
3    MAX-HEAPIFY(A, i)

```

HEAPSORT(A)

```

1  BUILD-MAX-HEAP(A)
2  for i ← A.length downto 2
3    exchange A[1] with A[i]
4    A.heap-size ← A.heap-size - 1
5    MAX-HEAPIFY(A, 1)

```

$$T(n) = T(n-1) + T(0) + \Theta(n)$$

$$= T(n-1) + \Theta(n)$$

Quicksort – $O(n^2)$ but expected running time $O(n \lg n)$

QUICKSORT(A, p, r)

```

1  if p < r
2    q ← PARTITION(A, p, r)
3    QUICKSORT(A, p, q - 1)
4    QUICKSORT(A, q + 1, r)

```

PARTIT

```

1  x ← A[p]
2  i ← p
3  for j ← r downto p+1
4    if A[j] ≤ x
5      exchange A[i] with A[j]
6      i ← i + 1
7  exchange A[p] with A[i]
8  return i

```

$$T(n) = 2T(n/2) + \Theta(n)$$

RANDOMIZED-PARTITION(A, p, r)

```

1  i ← RANDOM(p, r)
2  exchange A[i] with A[p]
3  return PARTITION(A, p, r)

```

The new quicksort calls RANDOMIZED-PARTITION in place of PARTITION:

RANDOMIZED-QUICKSORT(A, p, r)

```

1  if p < r
2    q ← RANDOMIZED-PARTITION(A, p, r)
3    RANDOMIZED-QUICKSORT(A, p, q - 1)
4    RANDOMIZED-QUICKSORT(A, q + 1, r)

```

Counting Sort – $O(n)$

COUNTING-SORT(A, B, k)

```

1  let C[0..k] be a new array
2  for i ← 0 to k
3    C[i] ← 0
4  for j ← 1 to A.length
5    C[A[j]] ← C[A[j]] + 1
6  // C[i] now contains the number of elements equal to i.
7  for i ← 1 to k
8    C[i] ← C[i] + C[i - 1]
9  // C[i] now contains the number of elements less than or equal to i.
10 for j ← A.length downto 1
11   B[C[A[j]]] ← A[j]
12   C[A[j]] ← C[A[j]] - 1

```



(a)



(b)



(c)



(d)



(e)



(f)

Radix Sort – $O(d(n+k))$ if stable sort

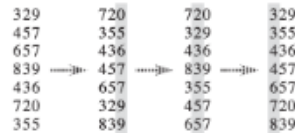
(Counting Sort) takes $O(n+k)$

RADIX-SORT(A, d)

```

1  for i ← 1 to d
2    use a stable sort to sort array A on digit i

```



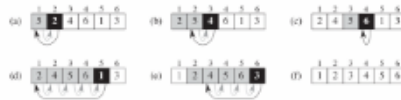
Insertion Sort $O(n^2)$

INSERTION-SORT(A)

```

1  for j ← 2 to A.length
2    key ← A[j]
3    // Insert A[j] into the sorted sequence A[1..j - 1].
4    i ← j - 1
5    while i > 0 and A[i] > key
6      A[i + 1] ← A[i]
7      i ← i - 1
8    A[i + 1] ← key

```



Sort $O(n \lg n)$

MERGE(A, p, q, r)

```

1  n1 ← q - p + 1
2  n2 ← r - q + 1
3  let L[1..n1 + 1] and R[1..n2 + 1] be new arrays
4  for i ← 1 to n1
5    L[i] ← A[p + i - 1]
6  for j ← 1 to n2
7    R[j] ← A[q + j]
8  L[n1 + 1] ← ∞
9  R[n2 + 1] ← ∞
10 i ← 1
11 j ← 1
12 for k ← p to r
13   if L[i] ≤ R[j]
14     A[k] ← L[i]
15     i ← i + 1
16   else A[k] ← R[j]
17     j ← j + 1

```

Merge

A ... 2 4 5 7

R ... 2 4 5 7

A ... 1 2 3 4 5 6 7

A ... 1 2 3 4 5 6 7

A ... 1 2 3 4 5 6 7

A ... 1 2 3 4 5 6 7

Algorithm 4 Selection Sort

```

1: for i ← 1 to n - 1 do
2:   min ← i
3:   for j ← i + 1 to n do
4:     // Find the index of the i-th smallest element
5:     if A[j] < A[min] then
6:       min ← j
7:   end if
8:   end for
9:   swap A[i] and A[min]
10: end for

```

(a) (b)

A ... 1 2 3 4 5 6 7

A ... 1 2 3 4 5 6 7

A ... 1 2 3 4 5 6 7

A ... 1 2 3 4 5 6 7

Merge sort -- $O(n \lg n)$

STACK-EMPTY(S)

```
1 if S.top == 0
2   return TRUE
3 else return FALSE
```

PUSH(S, x)

```
1 S.top = S.top + 1
2 S[S.top] = x
```

POP(S)

```
1 if STACK-EMPTY(S)
2   error "underflow"
3 else S.top = S.top - 1
4   return S[S.top + 1]
```

ENQUEUE(Q, x)

```
1 Q[Q.tail] = x
2 if Q.tail == Q.length
3   Q.tail = 1
4 else Q.tail = Q.tail + 1
```

DEQUEUE(Q)

```
1 x = Q[Q.head]
2 if Q.head == Q.length
3   Q.head = 1
4 else Q.head = Q.head + 1
5 return x
```

Arithmetic series

The summation

$$\sum_{k=1}^n k = 1 + 2 + \dots + n,$$

is an *arithmetic series* and has the value

$$\sum_{k=1}^n k = \frac{1}{2}n(n+1) = \Theta(n^2).$$

$$\sum_{k=0}^{\infty} kx^k = \frac{x}{(1-x)^2}$$

for $|x| < 1$.

For any sequence a_0, a_1, \dots, a_n ,

$$\sum_{k=1}^n (a_k - a_{k-1}) = a_n - a_0,$$

since each of the terms a_1, a_2, \dots, a_{n-1} is added in exactly once and subtracted out exactly once. We say that the sum *telescopes*. Similarly,

$$\sum_{k=0}^{n-1} (a_k - a_{k+1}) = a_0 - a_n.$$

As an example, let us prove that the geometric series $\sum_{k=0}^n 3^k$ is $O(3^n)$. More specifically, let us prove that $\sum_{k=0}^n 3^k \leq c3^n$ for some constant c . For the initial condition $n = 0$, we have $\sum_{k=0}^0 3^k = 1 \leq c \cdot 1$ as long as $c \geq 1$. Assuming that the bound holds for n , let us prove that it holds for $n + 1$. We have

$$\begin{aligned} \sum_{k=0}^{n+1} 3^k &= \sum_{k=0}^n 3^k + 3^{n+1} \\ &\leq c3^n + 3^{n+1} \quad (\text{by the inductive hypothesis}) \\ &= \left(\frac{1}{3} + \frac{1}{c}\right)c3^{n+1} \\ &\leq c3^{n+1} \end{aligned}$$

as long as $(1/3 + 1/c) \leq 1$ or, equivalently, $c \geq 3/2$. Thus, $\sum_{k=0}^n 3^k = O(3^n)$, as we wished to show.

$$\sum_{k=0}^{\infty} \frac{k^2}{2^k}.$$

we observe that the ratio of consecutive terms is

$$\frac{(k+1)^2/2^{k+1}}{k^2/2^k} = \frac{(k+1)^2}{2k^2} \leq \frac{8}{9}$$

if $k \geq 3$. Thus, the summation can be split into

$$\begin{aligned} \sum_{k=0}^{\infty} \frac{k^2}{2^k} &= \sum_{k=0}^2 \frac{k^2}{2^k} + \sum_{k=3}^{\infty} \frac{k^2}{2^k} \\ &\leq \sum_{k=0}^2 \frac{k^2}{2^k} + \frac{9}{8} \sum_{k=0}^{\infty} \left(\frac{8}{9}\right)^k \\ &= O(1), \end{aligned}$$

LIST-SEARCH(L, k)

```
1 x = L.head
2 while x != NIL and x.key != k
3   x = x.next
4 return x
```

LIST-DELETE'(L, x)

```
1 x.prev.next = x.next
2 x.next.prev = x.prev
```

LIST-DELETE(L, x)

```
1 if x.prev != NIL
2   x.prev.next = x.next
3 else L.head = x.next
4 if x.next != NIL
5   x.next.prev = x.prev
```

LIST-SEARCH'(L, k)

```
1 x = L.nil.next
2 while x != L.nil and x.key != k
3   x = x.next
4 return x
```

LIST-INSERT'(L, x)

```
1 x.next = L.nil.next
2 L.nil.next.prev = x
3 L.nil.next = x
4 x.prev = L.nil
```

$$\sum_{k=0}^n x^k = \frac{x^{n+1} - 1}{x - 1}, \quad \sum_{k=0}^{\infty} x^k = \frac{1}{1-x}.$$

For positive integers n , the n th harmonic number is

$$\begin{aligned} H_n &= 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \dots + \frac{1}{n} \\ &= \sum_{k=1}^n \frac{1}{k} \\ &= \ln n + O(1). \end{aligned} \quad (\text{A.9})$$

it holds for n , and we prove that it holds for $n + 1$

$$\begin{aligned} \sum_{k=1}^{n+1} \frac{1}{k} &= \sum_{k=1}^n \frac{1}{k} + \frac{1}{n+1} \\ &= \frac{1}{2}n(n+1) + (n+1) \\ &= \frac{1}{2}(n+1)(n+2). \end{aligned}$$

$$\frac{(k+2)/3^{k+2}}{(k+1)/3^{k+1}} = \frac{1}{3} \cdot \frac{k+2}{k+1} \leq \frac{2}{3}$$

for all $k \geq 0$. Thus, we have

$$\begin{aligned} \sum_{k=1}^{\infty} \frac{k}{3^k} &= \sum_{k=0}^{\infty} \frac{k+1}{3^{k+1}} \\ &\leq \frac{1}{3} \cdot \frac{1}{1-2/3} \\ &= 1. \end{aligned}$$

Approximation by integrals

When a summation has the form $\sum_{k=m}^n f(k)$, where $f(k)$ is a monotonically increasing function, we can approximate it by integrals:

$$\int_m^n f(x) dx \leq \sum_{k=m}^n f(k) \leq \int_{m-1}^{n+1} f(x) dx. \quad (\text{A.11})$$

When $f(k)$ is a monotonically decreasing function, we can use a similar method to provide the bounds

$$\int_m^{n+1} f(x) dx \leq \sum_{k=m}^n f(k) \leq \int_{m-1}^n f(x) dx. \quad (\text{A.12})$$