**Row-major and column-major order**

**1-D array**

Address of A[I] = B + W \* (I – LB)

I = address to be found,

B = Base address,

W = array(in byte)

LB = Lower Bound of subscript(If not specified assume zero).

Example:

A[1300…………1900] as 1020 and the size of each element is 2 bytes in the memory, find the address of A[1700]?

B = 1020

LB = 1300

W = 2 Byte

I = 1700

Address of A[I] = B + W \* (I – LB)

Address of A[1700] = 1020 + 2 \* (1700 – 1300)

= 1020 + 2 \* (400)

= 1020 + 800

Address of A[1700] = 1820

**2-D array**

**Row-major order**

Address of A[I][J] = B + W \* ((I – LR) \* N + (J – LC))

I = Row address to be found,

J = Column address to be found,

B = Base address

W = array(in byte)

LR = Lower Limit of row (If not given assume it as zero),

LC = Lower Limit of column (If not given assume it as zero),

N = Number of column given in the matrix.

Example:

Given an array, arr[1………10][2………15] with base value 100 and the size of each element is 1 Byte in memory. Find the address of arr[8][6] with the help of row-major order?

B = 100

W = 1 Bytes

I = 8

J = 6

LR = 1

Number of column given in the matrix N = Upper Bound – Lower Bound + 1

= 15 – 1 + 2

= 16

Address of A[I][J] = B + W \* ((I – LR) \* N + (J – LC))

Address of A[8][6] = 100 + 1 \* ((8 – 1) \* 16 + (6 – 2))

Address of A[I][J] = 216

**column-major order**

B = 100

W = 1 Bytes

I = 8

J = 6

LR = 1

Number of column given in the matrix M = Upper Bound – Lower Bound + 1

= 10 – 1 + 1

= 10

Address of A[I][J] = B + W \* ((J – LC) \* M + (I – LR))

Address of A[8][6] = 100 + 1 \* ((6 – 2) \* 10 + (8 – 1))

Address of A[I][J] = 147

**3D-ARRAY**

**Row Major Order**

Address of [i][j][k] = B + W \* {[(I – LR) \* N] + [(J – LC)]\* R + [K – LB]}

I = Row address to be found,

J = Column address to be found,

K = Block address to be found,

B = Base address

W = Storage size of array(in byte),

LR = Lower Limit of row (If not given assume it as zero

LC = Lower Limit of column (If not given assume it as zero),

LB = Lower Limit of blocks

N = Number of column

R = Number of blocks

Example:

Given an array arr[1:9, -4:1, 5:10] with base value 400 and size of each element is 2 Bytes in memory find the address of element arr[5][-1][8] with the help of row-major order?

I= 5

J = -1

K = 8

B = 400

W = 2

LR = 1

LC = -4

LB = 5

Number of column given in the matrix N = Upper Bound – Lower Bound + 1

= 1 – (-4) + 1

= 6

Number of blocks given in the matrix R = Upper Bound – Lower Bound + 1

= 10 – 5 + 1

= 6

Address of[i][j][k] = B + W \* {[(I – LR) \* N] + [(J – LC)] \* R + [K – LB]}

Address of[i][j][k] = 400 + 2 \* {[(5 – 1) \* 6] + [(-1 + 4)]} \* 6 + [8 – 5]

= 400 + 2 \* ((4 \* 6 + 3) \* 6 + 3)

= 400 + 2 \* (165)

= 730

**Column Major Order:**

Address of[i][j][k] = B + W \* {[(I – LR)] + [(J – LC) \* M]\* R + [K – LB]}

I = Row address to be found,

J = Column address to be found

K = Block address to be found

B = Base address

W = Storage array(in byte),

LR = Lower Limit of Row (If not given assume it as zero),

LC = Lower Limit of column (If not given assume it as zero),

LB = Lower Limit of blocks

M = Number of rows

R = Number of blocks

Example:

Given an array arr[1:8, -5:5, -10:5] with base value 400 and size of each element is 4 Bytes in memory find the address of element arr[3][3][3] with the help of column-major order?

Given:

I = 3

J = 3

K = 3

B = 400

W = 4

LR = 1

LC = -5

LB = -10

Number of rows given in the matrix M = Upper Bound – Lower Bound + 1

= 8 – 1 + 1

= 8

Number of blocks given in the matrix R = Upper Bound – Lower Bound + 1

= 5 + 10 + 1

= 16

Address of[i][j][k] = B + W \* {[(I – LR)] + [(J – LC) \* M]\* R + [K – LB]}

Address of[3][3][3] = 400 + 4 \* {[(3 – 1)] + [3 + 5] \* 8]} \* 16 + [3 + 10]

= 400 + 4 \* ((2 + 64) \* 16 + 13)

= 400 + 4 \* (1069)

= 400 + 4276

= 4676

Big-O Notation :

A measure of how well a computer algorithm scales as the amount of data involved increases. It tell that how long an algorithm takes to run now it's depending on the size of inputs. It depends on how much input you are passing.

So, now question arises that what actually Big O notation measures,

There are three different cases :

1. Best Case

2. Average Case

3. Worst Case

Big O notation specifically describes the worst case scenario of your algorithm how bad your algorithm can truly run as the input gets arbitrarily large.

Big O Complexity : O(1) < O(n) < O (n log n) < O(n2 ) < O (2n) < O (n!)

|  |  |  |
| --- | --- | --- |
| **Expression** | **Dominant term (s)** | **O (...)** |
| 5 + 0.001 n3 + 0.025 n | 0.001 n3 | O ( n3 ) |
| 500n + 100 n1.5 + 50 n log10n | 100 n1.5 | O( n1.5 ) |
| 0.3n + 5 n1.5 + 2.5  n1.75 | 2.5  n1.75 | O ( n1.75  ) |
| n2log2n + n ( log2n )2 | n2log2n | O(n2log n) |
| n log3 n + n log2n | n log3 n , n log2 n  here both terms are dominating | O(n log n) |
| 3 log8n + log2 log2 log2 n | 3 log8n | O( log n) |
| 100n + 0.01 n2 | 0.01 n2 | O( n2 ) |
| 0.01 n + 100 n2 | 100 n2 | O( n2 ) |
| 2n + n0.5+ 0.5 n1.25 | 0.5 n1.25 | O ( n1.25  ) |
| 0.01 n log2 n + n ( log2n)2 | n ( log2n)2 | O ( n ( logn)2 ) |
| 100n log3 n + n3 + 100n | n3 | O ( n3 ) |
| 0.03 log4 n + log2 log2 n | 0.03 log4 n | O( log n) |

Table for calculating Big O for their respective Expression :

Big O is calculated on the basis of the dominating term, because the largest term is responsible for the Worst Case Scenario.

Why is the need to convert an infix notation into postfix notation? Convert the following infix notation into postfix notation using stack. Show all the steps. (Consider operator ‘^’ as power operator)  
  
Infix expressions are readable and solvable by humans.

* + We can easily distinguish the order of operators, and also can use the parenthesis to solve that part first during solving mathematical expressions.
  + The computer cannot differentiate the operators and parenthesis easily, that's why postfix conversion is needed.

| **Sr. no.** | **Expression** | **Stack** | **Postfix** |
| --- | --- | --- | --- |
| 0 |  | ( |  |
| 1 | A | ( | A |
| 2 | + | ( + | A |
| 3 | B | ( + ( | AB |
| 4 | – | ( + ( | AB– |
| 5 | C | ( + ( ( | AB–C |
| 6 | ^ | ( + ( ( ^ | AB–C |
| 7 | D | ( + ( ( ^ ( | AB–CD |
| 8 | – | ( + ( ( ^ ( | AB–CD– |
| 9 | E | ( + ( ( ^ ( | AB–CD–E |
| 10 | ) | ( + ( ( ^ | AB–CD–E |
| 11 | \* | ( + ( ( \* | AB–CD–E^ |
| 12 | F | ( + ( ( \* | AB–CD–E^F |
| 13 | ) | ( + ( | AB–CD–E^F\* |
| 14 | / | ( + ( / | AB–CD–E^F\* |
| 15 | G | ( + ( / | AB–CD–E^F\*G |
| 16 | ) | ( + | AB–CD–E^F\*G/ |
| 17 | ) |  | AB–CD–E^F\*G/+ |

2. Covert the given expression into its equivalent postfix expression using stack. Show the contents of stack for each step.(A+B)\*(C^(D–E)+F)–G.

| **Sr. no.** | **Expression** | **Stack** | **Postfix** |
| --- | --- | --- | --- |
| 0 |  | ( |  |
| 1 | A | ( ( | A |
| 2 | + | ( ( + | A |
| 3 | B | ( ( + | AB |
| 4 | ) | ( | AB+ |
| 5 | \* | ( \* | AB+ |
| 6 | C | ( \* ( | AB+C |
| 7 | ^ | ( \* ( ^ | AB+C |
| 8 | D | ( \* ( ^ ( | AB+CD |
| 9 | – | ( \* ( ^ ( | AB+CD– |
| 10 | E | ( \* ( ^ ( | AB+CD–E |
| 11 | ) | ( \* ( ^ | AB+CD–E |
| 12 | + | ( \* ( + | AB+CD–E^ |
| 13 | F | ( \* ( + | AB+CD–E^F |
| 14 | ) | ( \* | AB+CD–E^F+ |
| 15 | – | ( \* | AB+CD–E^F+– |
| 16 | G | ( \* | AB+CD–E^F+–G |
| 17 | ) |  | AB+CD–E^F+–G\* |

3. Convert the following infix expression into postfix expression A + B \*( C ^ D \* E ) –F

| **Sr. no.** | **Expression** | **Stack** | **Postfix** |
| --- | --- | --- | --- |
| 0 |  | ( |  |
| 1 | A | ( | A |
| 2 | + | ( + | A |
| 3 | B | ( + | AB |
| 4 | \* | ( + \* | AB |
| 5 |  | ( + \* ( | AB |
| 6 | C | ( + \* ( | AB C |
| 7 | ^ | ( + \* ( ^ | AB C |
| 8 | D | ( + \* ( ^ | AB CD |
| 9 | \* | ( + \* ( \* | AB CD^ |
| 10 | E | ( + \* ( \* | AB CD^E |
| 11 | ) | ( + \* | AB CD^E\* |
| 12 | – | ( + \* | AB CD^E\*– |
| 13 | F | ( + \* | AB CD^E\*–F |
| 14 | ) |  | AB CD^E\*–F\*+ |