IAE ACADEMY INSTITUTE FOR ENGINEERS

GATE CS · DA

IAE Academy free study material · Updated 27 October 2026

Data Structures and Algorithms Formula Sheet

INSIDE

- 75 formulas in 7 topics
- When to use each one
- Taken from the free topic notes on iaeacademy.in

Online, one page per topic: **iaeacademy.in/study/data-structures-algorithms/formula-sheet**

FREE COUNSELLING

Preparing for GATE CS or DA? Talk to IAE Academy.

Or call **+91 84204 36524**

Every day, 10:30 am to 7:30 pm

Chat on WhatsApp

Free. Not for sale. Pass it on.

Arrays, stacks, queues and linked lists

1-D ARRAY ELEMENT ADDRESS

$$\text{Addr}(A[i]) = \text{Base} + (i - L) \times w$$

w = size of one element in bytes; L = lower bound of the index (L = 0 for a C array, giving Base + i*w); Base = address of the first element A[L].

2-D ARRAY ADDRESS (ROW-MAJOR, 0-INDEXED)

$$\text{Addr}(A[i][j]) = \text{Base} + (i \times \text{cols} + j) \times w$$

cols = number of columns. C stores 2-D arrays row-major, and these notes assume row-major unless a question says column-major. For column-major, whole columns are stored one after another: Base + (j*rows + i)*w, where rows = number of rows.

2-D ARRAY ADDRESS WITH ARBITRARY LOWER BOUNDS (ROW-MAJOR)

$$\text{Addr}(A[i][j]) = \text{Base} + \big((i - L_1) \times n_c + (j - L_2)\big) \times w$$

Array declared as A[L1..U1][L2..U2]; n_c = U2 - L2 + 1 is the number of columns; Base = address of A[L1][L2]. Column-major: Base + ((j - L2)*n_r + (i - L1))*w with n_r = U1 - L1 + 1.

LOWER-TRIANGULAR MATRIX STORED PACKED (ROW-MAJOR)

$$\text{index}(i, j) = \frac{i(i+1)}{2} + j$$

Holds for a 0-indexed n x n lower-triangular matrix (only entries with i >= j are stored) packed row by row into a 1-D array. Rows 0 to i-1 contribute 1 + 2 + ... + i = i(i+1)/2 entries. Address = Base + index*w. The packed array holds n(n+1)/2 elements.

NUMBER OF DISTINCT VALID STACK-PERMUTATION OUTPUTS OF N ELEMENTS

$$C_n = \frac{(2n)!}{(n+1)!\,n!} = \frac{1}{n+1}\binom{2n}{n}$$

n distinct elements pushed in a fixed order (1, 2, ..., n) through one stack, with pushes and pops interleaved freely. C_n (the nth Catalan number) counts the distinct output sequences: C_1 = 1, C_2 = 2, C_3 = 5, C_4 = 14, C_5 = 42.

CIRCULAR QUEUE INDEX WRAPAROUND

$$\text{rear} = (\text{rear} + 1) \bmod \text{SIZE}$$

SIZE = capacity of the underlying array. Every dequeue advances front by the same rule: front = (front + 1) mod SIZE.

FLOYD CYCLE-DETECTION MEETING PROPERTY

$$d_{\text{meet}\to\text{start}} = a \bmod c$$

a = links from the head to the cycle's start node; c = number of nodes in the cycle; d = the SMALLEST number of links followed forward from the first meeting node to the cycle start (0 to c-1). Holds when both pointers start at the head and slow moves 1 and fast moves 2 links per iteration. d equals a itself only when a < c. In every case, advancing a links from the meeting node reaches the cycle start, which is why the reset-to-head method below works.

Trees, binary search trees and binary heaps

MAXIMUM NODES AT A GIVEN LEVEL

$$\text{max nodes at level } i = 2^i$$

Root at level 0. An upper bound for any binary tree, reached exactly when level i is completely filled.

MAXIMUM NODES IN A TREE OF GIVEN HEIGHT

$$\max \text{ nodes for height } h = 2^{h+1} - 1$$

Reached exactly by a *perfect* binary tree of height h (every level completely filled).

MINIMUM NODES IN A TREE OF GIVEN HEIGHT

$$\min \text{ nodes for height } h = h + 1$$

Reached by any tree with exactly one node per level: left-skewed, right-skewed or zig-zag.

HEIGHT RANGE FOR N NODES

$$\lfloor \log_2 n \rfloor \le h \le n - 1$$

The lower bound follows from $n \le 2^{h+1} - 1$. A *complete* binary tree with n nodes, and so every binary heap, has height exactly $\lfloor \log_2 n \rfloor$.

LEAVES VERSUS NODES WITH TWO CHILDREN (ANY BINARY TREE)

$$n_0 = n_2 + 1$$

Holds for **every** non-empty binary tree, whatever the number n_1 of one-child nodes. Here n_0 = leaves and n_2 = nodes with two children. Proof: count edges two ways. With $n = n_0 + n_1 + n_2$ nodes there are $n - 1$ edges, and each edge hangs below exactly one parent, so $n - 1 = n_1 + 2n_2$. Subtracting gives $n_0 = n_2 + 1$.

FULL (STRICT) BINARY TREE: LEAVES, INTERNAL NODES, TOTAL

$$L = I + 1, \qquad n = 2I + 1$$

In a *full* binary tree every node has 0 or 2 children, so $n_1 = 0$. With I internal nodes it has $L = I + 1$ leaves and $n = 2I + 1$ nodes in all (always odd). More generally, a full k-ary tree (every node has 0 or k children) with I internal nodes has $L = (k - 1)I + 1$ leaves.

BST HEIGHT FOR N KEYS

$$\lfloor \log_2 n \rfloor \le h \le n - 1$$

The lower bound holds when the tree is complete. The upper bound is a chain, and every operation then costs $\Theta(n)$ in the worst case. Sorted or reverse-sorted input gives a chain, but so do other orders, such as 1, 5, 2, 4, 3. Of the $n!$ insertion orders of n distinct keys, exactly 2^{n-1} give height $n - 1$. When all insertion orders are equally likely, the expected height is $\Theta(\log n)$.

BALANCE FACTOR

$$\mathrm{BF}(\text{node}) = \text{height}(\text{left subtree}) - \text{height}(\text{right subtree})$$

An AVL tree needs $\mathrm{BF} \in \{-1, 0, 1\}$ at every node, with the height of an empty subtree taken as -1. A node whose BF becomes $+2$ or -2 after an update must be rebalanced by rotation.

MINIMUM NODES IN AN AVL TREE OF HEIGHT H

$$N(h) = N(h-1) + N(h-2) + 1, \quad N(0) = 1,\ N(-1) = 0$$

The sparsest AVL tree of height h is a root whose two subtrees are sparsest AVL trees of heights $h - 1$ and $h - 2$. This gives $N(0), N(1), \cdots = 1, 2, 4, 7, 12, 20, 33, \ldots$ and in closed form $N(h) = F_{h+3} - 1$, where $F_1 = F_2 = 1$ are the Fibonacci numbers. Consequently an AVL tree with n nodes has height $h < 1.4405 \log_2(n + 2) - 0.3277$, which is $O(\log n)$. The maximum number of nodes of height h is $2^{h+1} - 1$, as for any binary tree.

HEIGHT, LEAVES AND INTERNAL NODES OF AN N-ELEMENT HEAP

$$h = \lfloor \log_2 n \rfloor, \quad \text{leaves} = \left\lceil \frac{n}{2} \right\rceil, \quad \text{internal} = \left\lfloor \frac{n}{2} \right\rfloor$$

In the 1-indexed layout the leaves are at indices $\lfloor n/2 \rfloor + 1$ to n; in the 0-indexed layout at $\lfloor n/2 \rfloor$ to $n - 1$. In a max-heap with distinct keys the minimum is always at a leaf, since an internal node has a smaller child, and it can be at any leaf. Finding it therefore means scanning the $\lceil n/2 \rceil$ leaves, which is $\Theta(n)$.

BUILD-HEAP RUNNING TIME

$$\Theta(n)$$

Bottom-up build-heap calls sift-down (heapify) on every internal node, from the last one back to the root. An n-element heap has at most $\lceil n/2^{h+1} \rceil$ nodes at height h, and sifting one down costs $O(h)$. The total is $O\left(n \sum_{h \geq 0} h/2^h\right) = O(n)$, because $\sum_{h \geq 0} h/2^h = 2$. It is linear, NOT $\Theta(n \log n)$. Building by n successive inserts instead costs $O(n \log n)$ in the worst case.

HEAP SORT

$$\Theta(n \log n) \text{ time (worst case)}, \quad O(1) \text{ auxiliary space}$$

Build a max-heap in $\Theta(n)$. Then $n - 1$ times: swap the root with the last element of the heap, shrink the heap by 1 and sift down the new root, each costing $O(\log n)$. It sorts in place and is not stable. $O(1)$ auxiliary space assumes an iterative sift-down; the recursive `maxHeapify` below uses $O(\log n)$ stack space.

CATALAN NUMBER

$$C_n = \frac{1}{n+1}\binom{2n}{n} = \frac{(2n)!}{(n+1)!\,n!}, \qquad C_0 = 1,\ C_n = \sum_{i=0}^{n-1} C_i\, C_{n-1-i}$$

The recurrence chooses the root's left subtree size i; the right subtree then has $n - 1 - i$ nodes. The first values are $C_0, \ldots, C_6 = 1, 1, 2, 5, 14, 42, 132$. C_n is the number of binary tree shapes with n nodes and the number of BSTs on n distinct keys. It also counts the ways to fully parenthesise a product of $n + 1$ factors, and the balanced bracket strings of length $2n$. If n distinct labels may be placed freely in the nodes (no BST rule), the number of labelled binary trees is $n!\, C_n$.

Asymptotic complexity and recurrences

BIG-O (UPPER BOUND)

$$f(n) = O(g(n)) \iff \exists\, c > 0,\ n_0 > 0 \text{ such that } 0 \leq f(n) \leq c \cdot g(n)\ \ \forall n \geq n_0$$

f grows no faster than g, up to a constant factor.

BIG-OMEGA (LOWER BOUND)

$$f(n) = \Omega(g(n)) \iff \exists\, c > 0,\ n_0 > 0 \text{ such that } 0 \leq c \cdot g(n) \leq f(n)\ \ \forall n \geq n_0$$

f grows at least as fast as g, up to a constant factor.

BIG-THETA (TIGHT BOUND)

$$f(n) = \Theta(g(n)) \iff \exists\, c_1, c_2 > 0,\ n_0 > 0 \text{ such that } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)\ \ \forall n \geq n_0$$

f and g grow at the same rate. Equivalently, $f = O(g)$ and $f = \Omega(g)$ both hold.

CHANGE OF BASE FOR LOGARITHMS

$$\log_a n = \frac{\log_b n}{\log_b a} \quad (a, b > 1 \text{ constants})$$

The two logarithms differ only by the constant factor $1/\log_b a$, which is why the base is irrelevant inside O, Ω and Θ.

LOG VS POLYNOMIAL VS EXPONENTIAL VS FACTORIAL

$$\lim_{n\to\infty} \frac{(\log n)^k}{n^\epsilon} = 0, \qquad \lim_{n\to\infty} \frac{n^k}{c^n} = 0, \qquad \lim_{n\to\infty} \frac{c^n}{n!} = 0 \qquad (k \text{ any constant},\ \epsilon > 0,\ c > 1)$$

These three limits give the dominance rules:

- Any positive power of n beats any power of $\log n$.
- Any exponential with base greater than 1 beats any fixed power of n.
- $n!$ beats any exponential with a constant base.

STIRLING'S APPROXIMATION AND LOG(N!)

$$n! = \Theta\left(\sqrt{2\pi n}\left(\frac{n}{e}\right)^n\right), \qquad \log(n!) = \Theta(n \log n)$$

Consequences:

- $2^n = o(n!)$ and $n! = o(n^n)$.
- The elementary bounds $(n/2)^{n/2} \le n! \le n^n$ are enough on their own to give $\log(n!) = \Theta(n \log n)$.
- This result is behind the $\Omega(n \log n)$ lower bound on the number of comparisons for comparison-based sorting: a decision tree with $n!$ leaves has height at least $\log(n!)$.

EXPONENT-LOG SWAP IDENTITY

$$a^{\log_b n} = n^{\log_b a} \quad (a > 0,\ b > 1)$$

Take $\log_b$ of both sides: each becomes $\log_b n \cdot \log_b a$. Examples: $2^{\log_2 n} = n$, $4^{\log_2 n} = n^2$, $2^{\log_4 n} = \sqrt{n}$, $3^{\log_2 n} = n^{\log_2 3}$.

ARITHMETIC SERIES

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

This is the count when the inner loop runs exactly i times as i goes from 1 to n. If the inner loop runs $n - i$ times, the exact count changes to $n(n-1)/2$, and $ki + c$ times gives $k\,n(n+1)/2 + cn$. Both are still $\Theta(n^2)$ when $k > 0$.

SUMS OF POWERS

$$\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6} = \Theta(n^3), \qquad \sum_{i=1}^{n} i^k = \Theta(n^{k+1}) \quad (k \ge 0 \text{ constant})$$

This applies to triple loops such as i from 1 to n, j from 1 to i, k from 1 to j, and to inner loops that run i^2 times.

GEOMETRIC SERIES

$$\sum_{k=0}^{m} r^k = \frac{r^{m+1} - 1}{r - 1} \quad (r \ne 1), \qquad \sum_{k=0}^{\log_2 n} 2^k = 2n - 1 \quad (n \text{ a power of } 2)$$

When $r > 1$, the sum is $\Theta(\text{its largest term})$. When $0 < r < 1$, it is $\Theta(1)$.

HARMONIC SERIES AND LOG(N!)

$$\sum_{i=1}^{n} \frac{1}{i} = \ln n + O(1) = \Theta(\log n), \qquad \sum_{i=1}^{n} \log i = \log(n!) = \Theta(n \log n)$$

The harmonic sum appears when an inner loop steps by the outer index (inner count about n/i). The $\log(n!)$ sum appears when an inner loop runs $\log i$ times.

MULTIPLICATIVE (LOGARITHMIC) LOOP

$$i = 1, c, c^2, \ldots \le n \ \text{ gives } \ \lfloor \log_c n \rfloor + 1 = \Theta(\log n) \text{ iterations} \quad (\text{integer } c \ge 2,\ n \ge 1)$$

This is the exact count for `for (i = 1; i <= n; i *= c)`. For example, with $c = 2$ and $n = 1024$ the loop runs $10 + 1 = 11$ times, not 10.

MASTER THEOREM: GENERAL FORM

$$T(n) = a\,T\left(\frac{n}{b}\right) + f(n), \qquad a \ge 1,\ b > 1$$

a is the number of subproblems, b is the factor by which the size shrinks, and $f(n)$ is the work done outside the recursive calls (dividing plus combining). Compare $f(n)$ with $n^{\log_b a}$, which is the number of leaves of the recursion tree.

MASTER THEOREM FOR POLYNOMIAL F(N)

$$T(n) = a\,T\left(\frac{n}{b}\right) + \Theta(n^d) \implies T(n) = \begin{cases} \Theta(n^{\log_b a}) & d < \log_b a \\ \Theta(n^d \log n) & d = \log_b a \\ \Theta(n^d) & d > \log_b a \end{cases}$$

This is the special case $f(n) = \Theta(n^d)$ with $d \geq 0$ a constant. It is the quickest form to apply when f has no log factor.

Searching, sorting and hashing

LINEAR SEARCH: NUMBER OF KEY COMPARISONS

$$C_{\text{unsuccessful}}(n) = n, \qquad C_{\text{successful, avg}}(n) = \frac{1 + 2 + \cdots + n}{n} = \frac{n+1}{2}$$

The successful-search average assumes the key is present and equally likely to be at each of the n positions. Both the worst case and the average case are $O(n)$.

BINARY SEARCH COMPLEXITY

$$T(n) = T\left(\frac{n}{2}\right) + O(1) \;\Rightarrow\; T(n) = O(\log n), \qquad C_{\text{worst}}(n) = \lfloor \log_2 n \rfloor + 1$$

Convention: each probe of `a[mid]` counts as **one** (three-way) comparison. The worst case $\lfloor \log_2 n \rfloor + 1$ holds for both successful and unsuccessful search. If a question counts the `==` test and the `<` test as separate comparisons, recount under its rule. Valid only for a sorted array with random access.

FINDING THE MINIMUM AND MAXIMUM TOGETHER

$$C_{\text{min and max}}(n) = \left\lceil \frac{3n}{2} \right\rceil - 2 \quad (n \geq 2)$$

Method: take the elements in pairs. Compare the two elements of each pair with each other, then the smaller with the current minimum and the larger with the current maximum. That is 3 comparisons per pair instead of 4. Finding only the maximum (or only the minimum) takes exactly $n - 1$ comparisons. Both counts are the best possible for comparison-based algorithms.

LOWER BOUND FOR COMPARISON-BASED SORTING

$$C_{\text{worst}}(n) \geq \lceil \log_2(n!) \rceil = \Omega(n \log n)$$

Any sort that learns about the keys only by comparing pairs of them must make $\Omega(n \log n)$ comparisons in the worst case, and also on average. Proof idea: there are $n!$ possible orderings of n distinct keys. A binary decision tree that distinguishes them all has at least $n!$ leaves, so its height is at least $\lceil \log_2(n!) \rceil$, and $\log_2(n!) = \Theta(n \log n)$ by Stirling's approximation.

NUMBER OF INVERSIONS

$$0 \leq I \leq \frac{n(n-1)}{2}, \qquad E[I] = \frac{n(n-1)}{4}$$

The maximum occurs for reverse-sorted input. The expected value is for a uniformly random permutation of n distinct keys, since each of the $\binom{n}{2}$ pairs is inverted with probability $\frac{1}{2}$. Insertion sort runs in $\Theta(n + I)$ time, so it is fast on nearly sorted input.

BALANCED DIVIDE-AND-CONQUER RECURRENCE (MERGE SORT)

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n) \;\Rightarrow\; T(n) = \Theta(n \log n)$$

This applies whenever a problem of size n is split into 2 equal halves with linear combine cost. The recursion tree has $\log_2 n$ levels, each costing $\Theta(n)$ in total.

QUICKSORT RECURRENCES

$$\text{Worst (split } 0 : n-1\text{):} \qquad T(n) = T(n-1) + \Theta(n) \;\Rightarrow\; \Theta(n^2)$$

$$\text{Constant-fraction split } (0 < c < 1)\text{:} \quad T(n) = T(cn) + T((1-c)n) + \Theta(n) \;\Rightarrow\; \Theta(n \log n)$$

Even a lopsided but constant-fraction split, such as 1 : 9, gives $\Theta(n \log n)$. Only a split that leaves a constant number of elements on one side at every level gives $\Theta(n^2)$.

COMPARISONS TO MERGE TWO SORTED LISTS

$$\min(m, n) \le C_{\text{merge}}(m, n) \le m + n - 1$$

This is for the standard two-pointer merge of sorted lists of sizes m and n. The worst case $m + n - 1$ occurs when the lists interleave. The best case $\min(m, n)$ occurs when every element of the shorter list is smaller than every element of the longer one.

BINARY HEAP OPERATION COSTS (HEAP OF N ELEMENTS)

$$\text{build-heap} = O(n), \qquad \text{insert} = O(\log n), \qquad \text{extract-max} = O(\log n)$$

Bottom-up build-heap is $O(n)$, not $O(n \log n)$, because most nodes sit near the bottom and sift down only a short distance. Heap sort is build-heap followed by $n - 1$ extract-max operations: $O(n) + O(n \log n) = O(n \log n)$.

COUNTING SORT COMPLEXITY

$$T(n, k) = O(n + k), \qquad \text{extra space} = O(n + k)$$

n = number of elements; the keys are integers in $0..k-1$. The sort is linear when $k = O(n)$ and degrades when k is much larger than n. It is stable when the output is filled by scanning the input from right to left, as below. It is not in-place.

RADIX SORT COMPLEXITY

$$T(n, d, k) = O(d \cdot (n + k))$$

d = number of digits (passes) and k = the range of one digit ($k = 10$ for decimal digits, $k = 2$ for bits). Least-significant-digit radix sort applies a **stable** sort, usually counting sort, once per digit. The total is linear when d is a constant and $k = O(n)$. It is not in-place.

LOAD FACTOR

$$\alpha = \frac{n}{m}$$

n = number of keys stored, m = number of slots. With chaining, α can exceed 1. With open addressing every key needs its own slot, so $\alpha \le 1$. The open-addressing formulas below need $\alpha < 1$, i.e. at least one free slot.

EXPECTED PROBES: SEPARATE CHAINING (SIMPLE UNIFORM HASHING, NEW KEYS INSERTED AT THE HEAD OF THEIR CHAIN)

$$E[\text{unsuccessful}] = \alpha, \qquad E[\text{successful}] = 1 + \frac{\alpha}{2} - \frac{\alpha}{2n} \approx 1 + \frac{\alpha}{2}$$

These count the keys examined. An unsuccessful search walks the whole chain, whose expected length is α. A successful search examines the target plus the keys inserted into the same chain after it. Including the $O(1)$ hash computation, both searches take $\Theta(1 + \alpha)$ time. In the worst case, when all n keys hash to one slot, a search takes $\Theta(n)$.

EXPECTED PROBES: OPEN ADDRESSING UNDER UNIFORM HASHING (EVERY PROBE SEQUENCE EQUALLY LIKELY)

$$E[\text{unsuccessful}] \le \frac{1}{1-\alpha}, \qquad E[\text{successful}] \le \frac{1}{\alpha} \ln\left(\frac{1}{1-\alpha}\right)$$

These are upper bounds, valid for $\alpha < 1$. Uniform hashing is an idealisation that double hashing approximates well. Linear and quadratic probing do worse because of clustering.

EXPECTED PROBES: LINEAR PROBING (APPROXIMATIONS FOR LARGE TABLES)

$$E[\text{successful}] \approx \frac{1}{2}\left(1+\frac{1}{1-\alpha}\right), \qquad E[\text{unsuccessful}] \approx \frac{1}{2}\left(1+\frac{1}{(1-\alpha)^2}\right)$$

Valid for $\alpha < 1$. The squared term in the unsuccessful case shows how quickly primary clustering hurts as the table fills up.

Greedy, dynamic programming and divide-and-conquer

FRACTIONAL KNAPSACK

$$\max \sum_{i=1}^{n} v_i x_i \quad \text{subject to} \quad \sum_{i=1}^{n} w_i x_i \le W,\ 0 \le x_i \le 1$$

Applies only when any fraction of an item may be taken. Solved by sorting items on v_i/w_i in descending order and filling greedily until capacity is used up; only the last item taken can be partial. Time $O(n \log n)$.

EXPECTED (AVERAGE) CODE LENGTH

$$L = \sum_{i=1}^{n} p_i \, l_i$$

p_i is the probability of symbol i. If frequencies are given, divide each by the total frequency. l_i is its codeword length (the depth of its leaf). Huffman's construction minimises L over all prefix-free codes for the given distribution. When weights tie, different trees are possible, but they all have the same minimum L.

DIVIDE-AND-CONQUER RECURRENCE (GENERAL FORM)

$$T(n) = aT\left(\frac{n}{b}\right) + f(n), \qquad a \ge 1,\ b > 1$$

a = number of subproblems, n/b = size of each subproblem, f(n) = cost of the divide and combine steps outside the recursive calls.

MERGE SORT RECURRENCE

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n) \to T(n) = \Theta(n \log n)$$

The same bound holds in the best, average and worst cases, because the split does not depend on the input order. Merging needs $O(n)$ extra space (standard merge sort is not in-place). Merge sort is stable.

QUICKSORT RECURRENCES

$$T_{\text{worst}}(n) = T(n-1) + \Theta(n) \to \Theta(n^2), \qquad T_{\text{best}}(n) = 2T\left(\frac{n}{2}\right) + \Theta(n) \to \Theta(n \log n)$$

The worst case happens when the pivot is always the smallest or largest element, giving (n−1, 0) splits. Example: an already sorted or reverse-sorted array with the first or last element as pivot. The best case is a perfectly balanced split. Any split in a fixed proportion (e.g. 1:9) still gives $O(n \log n)$. The average case (random pivot, or all input orders equally likely) is also $\Theta(n \log n)$. Partitioning needs no auxiliary array, but the recursion stack uses $O(\log n)$ space on average and $O(n)$ in the worst case. Quicksort is not stable.

BINARY SEARCH RECURRENCE

$$T(n) = T\left(\frac{n}{2}\right) + \Theta(1) \to T(n) = O(\log n)$$

Requires a sorted array (or a search space that is monotonic in the property being tested). The iterative version uses $O(1)$ extra space and the recursive one $O(\log n)$ stack. Inside $O(\cdot)$ the base of the logarithm does not matter; when an exact count is needed, use base 2.

MAX-SUBARRAY D&C RECURRENCE

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n) \to T(n) = \Theta(n \log n)$$

This is the divide-and-conquer bound. Kadane's algorithm solves the same problem in $\Theta(n)$ with a single linear scan (a DP-style running "best sum ending here"). That is asymptotically better, and optimal, since every element must be read.

KADANE'S RECURRENCE

$$\text{best_end_here}(i) = \max\big(A[i],\ \text{best_end_here}(i-1) + A[i]\big), \quad \text{best_end_here}(0) = A[0]$$

The answer is $\max_i \text{best_end_here}(i)$. It runs in $O(n)$ time with $O(1)$ extra space. With non-empty subarrays, an all-negative array returns its largest element.

LCS RECURRENCE

$$L(i,j) = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ L(i-1, j-1) + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j \\ \max\big(L(i-1,j),\ L(i,j-1)\big) & \text{if } i, j > 0 \text{ and } x_i \neq y_j \end{cases}$$

Indices are 1-based here: x_i is the i-th character of X and y_j the j-th of Y. The table is $(m+1) \times (n+1)$, and time and space are both $\Theta(mn)$. If only the length is needed, not the subsequence itself, two rows are enough, giving $O(\min(m,n))$ space.

EDIT DISTANCE RECURRENCE

$$D(i,j) = \begin{cases} i & \text{if } j = 0 \\ j & \text{if } i = 0 \\ D(i-1, j-1) & \text{if } x_i = y_j \\ 1 + \min\big(D(i-1,j),\ D(i,j-1),\ D(i-1,j-1)\big) & \text{if } x_i \neq y_j \end{cases}$$

In the last case the three terms are delete x_i, insert y_j and substitute. $D(i,0) = i$ means deleting all i characters of X's prefix; $D(0,j) = j$ means inserting all j characters. The table is $(m+1) \times (n+1)$, filled in $\Theta(mn)$ time. If only insertions and deletions are allowed, a match still copies $D(i-1,j-1)$, but a mismatch gives $1 + \min(D(i-1,j),\ D(i,j-1))$. That distance equals $m + n - 2 \cdot \text{LCS}(X,Y)$.

0/1 KNAPSACK RECURRENCE

$$K(i,c) = \begin{cases} 0 & \text{if } i = 0 \\ K(i-1,c) & \text{if } w_i > c \\ \max\big(K(i-1,c),\ v_i + K(i-1, c-w_i)\big) & \text{if } w_i \leq c \end{cases}$$

$K(i,c)$ = best value using the first i items with capacity c. The table is $(n+1) \times (W+1)$, filled in $\Theta(nW)$ time. This is **pseudo-polynomial**: polynomial in the numeric value of W, but exponential in the number of bits needed to write W. The DP is therefore efficient for small integer W, while 0/1 knapsack in general is NP-hard. Space can be reduced to $O(W)$ with one array, if c runs from W down to w_i for each item.

SUBSET SUM RECURRENCE

$$S(i,t) = S(i-1,t) \ \vee\ \big(t \geq a_i \ \wedge\ S(i-1,\ t-a_i)\big)$$

$S(i,t)$ is true if and only if some subset of the first i numbers sums exactly to t. Base cases: $S(0,0) = \text{true}$ and $S(0,t) = \text{false}$ for $t > 0$. The table has $(n+1) \times (S+1)$ Boolean entries, filled in $O(nS)$ time, which is also pseudo-polynomial.

MATRIX CHAIN MULTIPLICATION RECURRENCE

$$m(i,j) = \min_{i \leq k < j} \big(m(i,k) + m(k+1,j) + p_{i-1}\, p_k\, p_j\big), \qquad m(i,i) = 0$$

$m(i,j)$ is the minimum cost of multiplying $A_i A_{i+1} \cdots A_j$; a single matrix costs nothing. Fill by increasing chain length $l = j - i + 1$ from 2 up to n. The answer is $m(1,n)$, the last cell filled. Time $\Theta(n^3)$ (indices i, j and split k), space $\Theta(n^2)$.

Graphs: traversals, minimum spanning trees and shortest paths

SPACE COST OF THE TWO REPRESENTATIONS

$$\text{Adjacency matrix: } O(V^2) \text{ space}, \qquad \text{Adjacency list: } O(V+E) \text{ space}$$

Use the matrix when the graph is dense (E close to V^2) or constant-time edge checks are needed; use the list when the graph is sparse (E much smaller than V^2). An undirected graph's adjacency lists hold 2E entries in total (each edge appears in both endpoints' lists); a directed graph's hold E.

HANDSHAKE LEMMA AND EDGE BOUNDS (SIMPLE GRAPHS)

$$\sum_{v \in V} \deg(v) = 2E, \qquad E \le \frac{V(V-1)}{2} \text{ (undirected)}, \qquad E \le V(V-1) \text{ (directed)}$$

The degree sum counts every undirected edge once from each end. Consequence: every undirected graph has an even number of odd-degree vertices. In a directed graph, the sum of in-degrees = the sum of out-degrees = E.

PARENTHESIS THEOREM (FOR TWO VERTICES WITH D[U] < D[V])

$$\text{either } \; d[u] < d[v] < f[v] < f[u] \quad \text{or} \quad d[u] < f[u] < d[v] < f[v]$$

Exactly one of the two holds. The first means v is a descendant of u in the DFS forest (the intervals are nested). The second means neither is a descendant of the other (the intervals are disjoint). Partial overlap, d[u] < d[v] < f[u] < f[v], is impossible.

NUMBER OF EDGES IN A SPANNING TREE OR FOREST

$$|E_{\text{MST}}| = V - 1, \qquad |E_{\text{spanning forest}}| = V - k$$

The first holds for any spanning tree of a connected graph on V vertices, whatever the weights. The second is for a graph with k connected components, where Kruskal run to completion produces a minimum spanning forest.

NUMBER OF LABELLED SPANNING TREES OF THE COMPLETE GRAPH K_N

$$n^{n-2}$$

For example, K_4 has 4^2 = 16 spanning trees. This counts spanning trees, not MSTs; with distinct weights, exactly one of them is the MST.

DIJKSTRA'S TIME COMPLEXITY

$$O((V+E)\log V) \text{ (binary heap)}, \qquad O(V^2) \text{ (array scan)}, \qquad O(E + V\log V) \text{ (Fibonacci heap)}$$

Correctness is guaranteed only when all edge weights are non-negative. The array version suits dense graphs (E close to V^2); the binary-heap version suits sparse graphs (E much smaller than V^2).

BELLMAN-FORD'S TIME COMPLEXITY

$$O(V \cdot E)$$

Each of the V−1 rounds scans all E edges once; the extra negative-cycle check costs another O(E).

FLOYD-WARSHALL RECURRENCE

$$d_k[i][j] = \min\left(d_{k-1}[i][j],\ d_{k-1}[i][k] + d_{k-1}[k][j]\right)$$

d_k[i][j] is the length of the shortest path from i to j whose intermediate vertices all lie in {1,...,k}. d_0 is the input: d_0[i][i] = 0, d_0[i][j] = w(i,j) where a direct edge exists, and infinity otherwise. The answer is d_V.

FLOYD-WARSHALL COMPLEXITY

$$O(V^3) \text{ time}, \qquad O(V^2) \text{ space}$$

Three nested loops over all V vertices. It is correct with negative edge weights as long as there is no negative-weight cycle, and it detects one if present. Replacing min with OR and + with AND gives the transitive-closure (reachability) algorithm, also O(V^3).

Programming in Python

FLOOR DIVISION AND REMAINDER IN PYTHON

$$a = b\,(a \mathbin{//} b) + (a \bmod b), \qquad 0 \le a \bmod b < b \ \text{ if } b > 0, \quad b < a \bmod b \le 0 \ \text{ if } b < 0$$

Holds for integers with b nonzero; `a // b` is the floor of the true quotient, so a nonzero remainder takes the sign of b.

NUMBER OF VALUES IN A RANGE

$$\text{len}(\textbf{range}(a, b, k)) = \max\left(0,\ \left\lceil \frac{b-a}{k} \right\rceil\right)$$

Holds for integers a, b and any nonzero step k, positive or negative; for example len(range(3, 20, 4)) = ceil(17/4) = 5 and len(range(5, 0, -2)) = ceil(2.5) = 3.

SET OPERATION SIZES

$$|A \cup B| = |A| + |B| - |A \cap B|, \qquad |A \triangle B| = |A| + |B| - 2\,|A \cap B|$$

For finite sets; A △ B is the symmetric difference, written `A ^ B` in Python. Use these to count `len(A | B)` or `len(A ^ B)` without listing the elements.

NUMBER OF CALLS MADE BY NAIVE FIBONACCI RECURSION

$$C(n) = 1 + C(n-1) + C(n-2), \quad C(0) = C(1) = 1 \implies C(n) = 2F(n+1) - 1$$

For the `fib` above, which returns directly for n < 2 and otherwise calls itself on n-1 and n-2, with F(0)=0, F(1)=1. C(n) counts every call, the first one included; for example fib(6) makes C(6) = 2·13 − 1 = 25 calls.

FROM IAE ACADEMY

Preparing for GATE CS or DA?

IAE Academy teaches GATE CS and DA in Ballygunge, Kolkata, and online. Counselling is free: ask us about batches, fees or your preparation.

FREE COUNSELLING

Chat on WhatsApp

Call +91 84204 36524

Open every day, 10:30 am to 7:30 pm.

Scan to chat on WhatsApp

GATE CS AND DA COURSES

Classroom batches in Ballygunge and online batches. Fees and details are on the course pages.

Scan: courses

See the courses

FREE ON IAEACADEMY.IN

Notes and solved previous-year questions for Data Structures and Algorithms, topic by topic.

Scan: the notes

Open the notes

Found this useful? Send it to a friend preparing for GATE.

Send to a friend on WhatsApp

IAE Academy free study material. The formulas come from the free Data Structures and Algorithms notes on iaeacademy.in. Prepared and published by IAE Academy, Kolkata. Free to read and free to share, unaltered and with attribution.

IAE ACADEMY INSTITUTE FOR ENGINEERS

GATE coaching in Kolkata since 2012 · EC, EE, IN, CS and IT, DA, BT and CE · in the classroom and online

227A Rash Behari Ave, Ballygunge, Kolkata 700019

+91 84204 36524 · iaeacademy.in · Updated 27 October 2026

Get directions