

Algorithms - CheatSheet

IN BA4 - Ola Nils Anders Svensson

Notes by Ali EL AZDI

This is a cheat sheet for the Algorithms midterm exam. For suggestions, contact me on Telegram ([elazdi_al](https://t.me/elazdi_al)) or via EPFL email (ali.elazdi@epfl.ch).

March 25th, 2025

Asymptotic Notation

Big-O
If $\exists c > 0$ and $\exists n_0 > 0$, $0 \leq f(n) \leq c \cdot g(n) \ \forall n \geq n_0$, then $f(n) = O(g(n))$.

Big-Omega
If $\exists c > 0$ and $\exists n_0 > 0$, $0 \leq c \cdot g(n) \leq f(n) \ \forall n \geq n_0$, then $f(n) = \Omega(g(n))$.

Big-Theta
If $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$, then $f(n) = \Theta(g(n))$.

Master Theorem
If $T(n) = aT(\frac{n}{b}) + f(n)$, where $a \geq 1$, $b > 1$, and $f(n)$ is asymptotically positive. The solution depends on comparing $f(n)$ to $n^{\log_b a}$:

- Case 1:** If $f(n) = O(n^{\log_b a - \epsilon})$ for some $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
- Case 2:** If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \log n)$.
- Case 3:** If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some $\epsilon > 0$, and if $\frac{a}{b^d} f(\frac{n}{b}) \leq c f(n)$ for some $c < 1$ and all sufficiently large n , then $T(n) = \Theta(f(n))$.
Common case - if $f(n) = \Theta(n^d)$ for some exponent d :

- If $\frac{a}{b^d} < 1$ (or $d > \log_b a$), then $T(n) = \Theta(n^d)$.
- If $\frac{a}{b^d} = 1$ (or $d = \log_b a$), then $T(n) = \Theta(n^d \log n)$.
- If $\frac{a}{b^d} > 1$ (or $d < \log_b a$), then $T(n) = \Theta(n^{\log_b a})$.

Insertion Sort

- Select the key**
Begin with the second element (at index 1) as the key.
- Compare and Shift**
Compare the key with elements in the sorted section (to its left).
- Shift Elements**
If an element is greater than the key, shift that element one position to the right.
- Insert the Key**
Once an element less than or equal to the key is found (or you reach the start), insert the key immediately after that element.
- Repeat**
Move forward to the next element, treating it as the new key, and repeat until the array is sorted.

Time Complexity: Worst-case $O(n^2)$, Best-case $O(n)$.
Space Complexity: $O(1)$.

Ideal for small or nearly sorted arrays.

```
INSERTION-SORT(A, n)
for j = 2 to n
    key = A[j]
    // Insert A[j] into the sorted sequence A[1..j-1].
    i = j - 1
    while i > 0 and A[i] > key
        A[i + 1] = A[i]
        i = i - 1
    A[i + 1] = key
```

Maximum Subarray Problem

Problem: Find contiguous subarray with largest sum

- Divide and Conquer Approach:**

Divide: Split array at midpoint
mid = $\lfloor (\text{low} + \text{high}) / 2 \rfloor$

Conquer: Find maximum subarrays recursively

 - Left max: in $A[\text{low} \dots \text{mid}]$
 - Right max: in $A[\text{mid} + 1 \dots \text{high}]$
 - Crossing max: spans the midpoint

Combine: Return the largest of the three
max(left.max, right.max, crossing.max)
- Finding the Crossing Maximum:**
 - Find maximum suffix in left half (from mid down to low)
 - Find maximum prefix in right half (from mid+1 up to high)
 - Crossing max = max suffix + max prefix

Time Complexity: $\Theta(n \log n)$ due to $T(n) = 2T(n/2) + \Theta(n)$
Space Complexity: $O(\log n)$ for recursion stack

```
FIND-MAXIMUM-SUBARRAY(A, low, high)
if high == low
    return (low, high, A[low])
else mid = (low + high) / 2
    (left-low, left-high, left-sum) = FIND-MAXIMUM-SUBARRAY(A, low, mid)
    (right-low, right-high, right-sum) = FIND-MAXIMUM-SUBARRAY(A, mid + 1, high)
    (cross-low, cross-high, cross-sum) = FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
    if left-sum > right-sum and left-sum > cross-sum
        return (left-low, left-high, left-sum)
    elseif right-sum > left-sum and right-sum > cross-sum
        return (right-low, right-high, right-sum)
    else return (cross-low, cross-high, cross-sum)
```

```
FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
// Find a maximum subarray of the form A[l..mid].
left-sum = -∞
sum = 0
for i = mid downto low
    sum = sum + A[i]
    if sum > left-sum
        left-sum = sum
        max-left = i
// Find a maximum subarray of the form A[mid + 1..j].
right-sum = -∞
sum = 0
for j = mid + 1 to high
    sum = sum + A[j]
    if sum > right-sum
        right-sum = sum
        max-right = j
// Return the indices and the sum of the two subarrays.
return (max-left, max-right, left-sum + right-sum)
```

Stack Operations

Stack-Empty(S): *Time:* $O(1)$, *Space:* $O(1)$

- Returns TRUE if the stack is empty.
- Returns FALSE otherwise.

Push(S, x): *Time:* $O(1)$, *Space:* $O(1)$

- Adds element x to the top of stack S.
- Increments the stack pointer.

Pop(S): *Time:* $O(1)$, *Space:* $O(1)$

- If Stack-Empty(S), return error "underflow".
- Otherwise, remove and return the top element.
- Decrements the stack pointer.

Stack Implementation:

- Elements are stored in a simple array
- S.top: Index of the topmost element
- An empty stack has S.top = 0 or S.top = -1 (implementation dependent)

Overall Space Complexity: $O(n)$ for a stack of size n

```
Stack-Empty(S):
    if S.top == -1 or S.top == 0
        return TRUE
    else
        return FALSE

Push(S, x):
    S.top = S.top + 1
    S.array[S.top] = x

Pop(S):
    if S.top == -1 or S.top == 0
        return "underflow"
    else
        x = S.array[S.top]
        S.top = S.top - 1
        return x
```

Strassen's Matrix Multiplication

- Divide:** Partition each of A, B, C into four $\frac{n}{2} \times \frac{n}{2}$ submatrices:
$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \cdot \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$
- Conquer:** Compute 7 products (re-cursively on $\frac{n}{2} \times \frac{n}{2}$ matrices):
$$M_1 := (A_{11} + A_{22})(B_{11} + B_{22})$$
$$M_2 := (A_{21} + A_{22})B_{11}$$
$$M_3 := A_{11}(B_{12} - B_{22})$$
$$M_4 := A_{22}(B_{21} - B_{11})$$
$$M_5 := (A_{11} + A_{12})B_{22}$$
$$M_6 := (A_{21} - A_{11})(B_{11} + B_{12})$$
$$M_7 := (A_{12} - A_{22})(B_{21} + B_{22})$$
- Combine:** Assemble the resulting submatrices to form C :
$$C_{11} = M_1 + M_4 - M_5 + M_7$$
$$C_{21} = M_2 + M_4$$
$$C_{12} = M_3 + M_5$$
$$C_{22} = M_1 + M_3 - M_2 + M_6$$

Time Complexity: $O(n^{\log_2 7}) \approx O(n^{2.81})$
Space Complexity: $O(n^2)$

```
Strassen(A, p, q, r)
n1 = q - p + 1
n2 = r - q
let L[1..n1 + 1] and R[1..n2 + 1] be new arrays
for i = 1 to n1
    L[i] = A[p + i - 1]
for j = 1 to n2
    R[j] = A[q + j]
L[n1 + 1] = ∞
R[n2 + 1] = ∞
i = 1
j = 1
for k = p to r
    if L[i] ≤ R[j]
        A[k] = L[i]
        i = i + 1
    else A[k] = R[j]
        j = j + 1
```

Queue Operations

Queue-Empty(Q): *Time:* $O(1)$, *Space:* $O(1)$

- Returns TRUE if the queue is empty (Q.head = Q.tail).
- Returns FALSE otherwise.

Enqueue(Q, x): *Time:* $O(1)$, *Space:* $O(1)$

- Adds element x to the rear of queue Q.
- Q[Q.tail] = x
- Q.tail = Q.tail + 1 (or wrap around if using circular array)

Dequeue(Q): *Time:* $O(1)$, *Space:* $O(1)$

- If Queue-Empty(Q), return error "underflow".
- Otherwise, remove and return the element at the front.
- x = Q[Q.head]
- Q.head = Q.head + 1 (or wrap around)
- Return x

Queue Implementation:

- Q.head: Index of the front element
- Q.tail: Index where next element will be inserted
- In a circular array, indices wrap around
- Leave one slot empty to distinguish full/empty states

Overall Space Complexity: $O(n)$ for a queue of capacity n

```
Queue-Empty(Q):
    return Q.head == Q.tail

Enqueue(Q, x):
    Q[Q.tail] = x
    Q.tail = Q.tail + 1
    if Q.tail == Q.size
        Q.tail = 0

Dequeue(Q):
    if Q.head == Q.tail
        return "underflow"
    x = Q[Q.head]
    Q.head = Q.head + 1
    if Q.head == Q.size
        Q.head = 0
    return x
```

Merge Sort

- Divide:** Split the array evenly into two smaller subarrays, and continue dividing recursively.
- Sort (Recursively):** Apply merge sort recursively on each subarray until each has only one element (base case).
MERGE-SORT(A, p, r)
if $p < r$
 $q = \lfloor (p + r) / 2 \rfloor$ // check for base case
 MERGE-SORT(A, p, q) // divide
 MERGE-SORT($A, q + 1, r$) // conquer
 MERGE(A, p, q, r) // combine
- Merge:** Combine the two sorted subarrays into a single sorted array:
 - Initializing pointers at the start of each subarray.
 - Comparing the elements pointed to, and appending the smaller one into a new array.
 - Advancing the pointer in the subarray from which the element was chosen.
 - Repeating this process until all elements in both subarrays are merged into the sorted array.

Merge Cost Complexity: $O(n)$ per merge operation.
Time Complexity: $O(n \log n)$
Space Complexity: $O(n)$

```
Merge(A, p, q, r)
n1 = q - p + 1
n2 = r - q
let L[1..n1 + 1] and R[1..n2 + 1] be new arrays
for i = 1 to n1
    L[i] = A[p + i - 1]
for j = 1 to n2
    R[j] = A[q + j]
L[n1 + 1] = ∞
R[n2 + 1] = ∞
i = 1
j = 1
for k = p to r
    if L[i] ≤ R[j]
        A[k] = L[i]
        i = i + 1
    else A[k] = R[j]
        j = j + 1
```

Priority Queue

Maintains a dynamic set of elements with associated priority values (keys).
Maximum(S): Return element of S with highest priority (return A[1], complexity $O(1)$)

Insert(S,x): Insert element x into set S

- Increment the heap size
- Insert a new node in the last position in the heap, with key $-\infty$
- Increase the $-\infty$ value to key using Heap-Increase-Key

Extract-Max(S): Remove and return element of S with highest priority

- Make sure heap is not empty
- Make a copy of the maximum element (the root)
- Make the last node in the tree the new root
- Re-heapify the heap, with one fewer node
- Return the copy of the maximum element

Increase-Key(S,x,k): Increase the value of element x's key to the new value k

- Make sure key $\geq A[i]$
- Update $A[i]$'s value to key
- Traverse the tree upward comparing new key to the parent and swapping if necessary

Time Complexity: Insert, Extract-Max, Increase-Key: $O(\log n)$ Maximum: $O(1)$
Space Complexity: $O(n)$

```
PriorityQueue(A)
// A[1] is the root
// A.heap-size is the number of elements in the heap
// A.key[i] is the key of element i
// A.parent[i] is the parent of element i
// A.left[i] is the left child of element i
// A.right[i] is the right child of element i

Insert(S, x):
    A.heap-size = A.heap-size + 1
    A[A.heap-size] = x
    A.key[A.heap-size] = -∞
    bubble-up(A.heap-size)

Extract-Max(S):
    max = A[1]
    A[1] = A[A.heap-size]
    A.heap-size = A.heap-size - 1
    heapify-down(1)
    return max

Increase-Key(S, x, k):
    i = A.index[x]
    A.key[i] = k
    bubble-up(i)
```

Heap

Root is A[1] Left(i) = 2i Right(i) = 2i + 1 Parent(i) = $\lfloor i/2 \rfloor$

Max-Heapify (heapify subtree rooted at i)

- Starting at the root
- Compare A[i], A[Left(i)], A[Right(i)]
- If necessary, swap A[i] with the largest of the two children
- Max-Heapify** the swapped child
- Continue comparing and swapping down the heap until subtree rooted at i is max-heap

Time Complexity: $O(\log(n))$
Space Complexity: $O(1)$

Max-Heap-Insert (insert new key into heap)

- Increase heap size: A.heap-size = A.heap-size + 1
- Set the last element to negative infinity: A[A.heap-size] = $-\infty$
- Call Max-Heap-Increase-Key to update to the correct value

Time Complexity: $O(\log n)$ *Space Complexity:* $O(1)$

Max-Heap-Increase-Key (increase key at position i)

- Ensure new key is larger than current: if key < A[i] then error
- Set A[i] = key
- Compare with parent and swap if necessary: while i > 1 and A[Parent(i)] < A[i]
- Exchange A[i] with A[Parent(i)]
- Set i = Parent(i) and continue upward

Time Complexity: $O(\log n)$ *Space Complexity:* $O(1)$

Build-Max-Heap (build a max-heap from an array)

- Start from the last non-leaf node at index $\frac{n}{2} - 1$
- Move upwards to the root (index 0) and:
 - Max-Heapify** the current node
 - Ensure the subtree rooted here satisfies max-heap property
 - Repeat until the root node is processed
- After completion, array A represents a valid max heap

Time Complexity: $O(n)$ *Space Complexity:* $O(1)$

Heap Sort

- Build a Max Heap:**
 - Convert the given array into a max heap
 - Start from the last non-leaf node and heapify upwards
 - Ensure each parent node is greater than its children
- Extract Maximum Elements:**
 - Swap the root (maximum value) with the last element
 - Reduce heap size by one to exclude the last element
 - Heapify the root to maintain max heap property
 - Repeat until heap size becomes 1
- Final Sorted Array:**
 - After extraction, the sorted array in ascending order is obtained
 - Maximum elements are placed at the end

Time Complexity: $O(n \log n)$ *Space Complexity:* $O(1)$

```
Max-Heapify(A, i, n)
l = LEFT(i)
r = RIGHT(i)
if l ≤ n and A[l] > A[i]
    largest = l
else largest = i
if r ≤ n and A[r] > A[largest]
    largest = r
if largest ≠ i
    exchange A[i] with A[largest]
    MAX-HEAPIFY(A, largest, n)

MAX-HEAP-INSERT(A, key, n)
n = n + 1
A[n] = -∞
HEAP-INCREASE-KEY(A, n, key)

BUILD-MAX-HEAP(A, n)
for i = [n/2] downto 1
    MAX-HEAPIFY(A, i, n)

HEAPSORT(A, n)
BUILD-MAX-HEAP(A, n)
for i = n downto 2
    exchange A[1] with A[i]
    MAX-HEAPIFY(A, 1, i - 1)
```

Data Structure Operations Summary

Queue Operations			
Operation	Description	Time	Space
Queue-Empty(Q)	Returns TRUE if queue is empty, FALSE otherwise	$O(1)$	$O(1)$
Enqueue(Q, x)	Adds element x to rear of queue Q	$O(1)$	$O(1)$
Dequeue(Q)	Removes and returns front element from queue Q	$O(1)$	$O(1)$

Stack Operations			
Operation	Description	Time	Space
Stack-Empty(S)	Returns TRUE if stack is empty, FALSE otherwise	$O(1)$	$O(1)$
Push(S, x)	Adds element x to top of stack S	$O(1)$	$O(1)$
Pop(S)	Removes and returns top element from stack S	$O(1)$	$O(1)$

Linked List Operations			
Operation	Description	Time	Space
List-Search(L, k)	Returns pointer to first node with key k, NULL if not found	$O(n)$	$O(1)$
List-Max-Heap-Insert(L, x)	Inserts node x at beginning of list L	$O(1)$	$O(1)$
List-Delete(L, x)	Removes node x from list L	$O(n)$ find, $O(1)$ del	$O(1)$

Heap Operations			
Operation	Description	Time	Space
Max-Heapify(A, i)	Maintains max-heap property at node i	$O(\log n)$	$O(1)$
Build-Max-Heap(A)	Converts array A into max heap	$O(n)$	$O(1)$
Heap-Sort(A)	Sorts array A using heap	$O(n \log n)$	$O(1)$
Max-Heap-Insert(A, k)	Inserts key k into heap A	$O(\log n)$	$O(1)$
Heap-Extract-Max(A)	Returns and removes largest element from heap A	$O(\log n)$	$O(1)$
Heap-Increase-Key(A, i, k)	Increases key at index i to new value k	$O(\log n)$	$O(1)$

BST Operations			
Operation	Description	Time	Space
BST-Search(T, k)	Finds node with key k in tree T	$O(h)$	$O(h)$
BST-Minimum(T)	Returns node with smallest key in T	$O(h)$	$O(1)$
BST-Maximum(T)	Returns node with largest key in T	$O(h)$	$O(1)$
BST-Successor(x)	Returns node with smallest key greater than x's key	$O(h)$	$O(1)$
BST-Insert(T, z)	Inserts node z into BST T	$O(h)$	$O(1)$
BST-Delete(T, z)	Removes node z from BST T	$O(h)$	$O(1)$
BST-Inorder(T)	Visits all nodes in sorted order	$O(n)$	$O(h)$
BST-Preorder(T)	Visits root before its children	$O(n)$	$O(h)$
BST-Postorder(T)	Visits children before root	$O(n)$	$O(h)$

Priority Queue Operations			
Operation	Description	Time	Space
Maximum(S)	Returns element with highest priority	$O(1)$	$O(1)$
Extract-Max(S)	Removes and returns element with highest priority	$O(\log n)$	$O(1)$
Insert(S, x)	Inserts element x into set S	$O(\log n)$	$O(1)$
Increase-Key(S, x, k)	Increases priority of element x to k	$O(\log n)$	$O(1)$

Binary Search Trees (BST)

BST-Search

1. Start at root
2. If NULL, return NULL
3. If key = root's key, return root
4. If key < root's key, search left
5. If key > root's key, search right

TREE-SEARCH(x, k)

if x = NIL or k == key[x]
return x
if k < x.key
return TREE-SEARCH(x.left, k)
else return TREE-SEARCH(x.right, k)
Time: O(log n) avg, O(h) worst
Space: O(h)

BST-Preorder

1. Visit current node
2. Recursively traverse left
3. Recursively traverse right
(Root first, then children)

PREORDER-TREE-WALK(x)

1. if x ≠ NIL
2. print key[x]
3. PREORDER-TREE-WALK(x.left)
4. PREORDER-TREE-WALK(x.right)
Time: O(n)
Space: O(h)

Properties:

- Left subtree: all keys < node's key
- Right subtree: all keys > node's key
- Left and right subtrees are also BSTs
- A Node has: key (value), left & right (child pointers), parent (optional)
- Tree height h: length of longest path from root to leaf

BST-Minimum

1. Start at root
2. If NULL, return NULL
3. Follow left pointers until no left child
4. Return leftmost node

TREE-MINIMUM(x)

while x.left ≠ NIL
x = x.left
return x
Time: O(h)
Space: O(1)

BST-Maximum

1. Start at root
2. If NULL, return NULL
3. Follow right pointers until no right child
4. Return rightmost node

TREE-MAXIMUM(x)

while x.right ≠ NIL
x = x.right
return x
Time: O(h)
Space: O(1)

BST-Successor

1. If right subtree exists:
Return minimum in right subtree
2. Otherwise:
Find first ancestor where node is in left subtree

TREE-SUCCESSOR(x)

if x.right ≠ NIL
return TREE-MINIMUM(x.right)
y = x.p
while y ≠ NIL and x == y.right
x = y
y = y.p
return y
Time: O(h)
Space: O(1)

BST-Inorder

1. Recursively traverse left
2. Visit current node
3. Recursively traverse right
(Visits nodes in sorted order)

INORDER-TREE-WALK(x)

if x ≠ NIL
INORDER-TREE-WALK(x.left)
print key[x]
INORDER-TREE-WALK(x.right)
Time: O(n)
Space: O(h)

BST-Delete

1. If z has no left: transplant right
2. If z has no right: transplant left
3. With both children:
a. Find successor y
b. Handle y's children
c. Replace z with y

TREE-DELETE(T, z)

if z.left = NIL
TRANSPLANT(T, z, z.right) // z has no left child
else if z.right = NIL
TRANSPLANT(T, z, z.left) // z has just a left child
else // z has two children.
y = TREE-MINIMUM(z.right) // y is z's successor
if y.p ≠ NIL
// y lies within z's right subtree but is not the root of this
TRANSPLANT(T, y, y.right)
y.right = z.right
y.right.p = y
// Replace z by y
TRANSPLANT(T, y, y.parent)
y.left = z.left
y.left.p = y
Time: O(h)
Space: O(1)

Dynamic Programming

Problem:

Optimal solutions to overlapping subproblems

Fibonacci Sequence:

Top-Down (Memoization):

1. Create memo array F[0...n] initialized to NIL
2. Base cases: F[0] = 0, F[1] = 1
3. Recursively with memo:
- Return F[n] if already computed
- Otherwise compute F[n] = F[n - 1] + F[n - 2]
- Store result in F[n] and return

MEMOIZED-FIB(n)

1. Let r = [0...n] be a new array
2. for i = 0 to n
3. r[i] ← ∞
4. return MEMOIZED-FIB-AUX(n, r)

MEMOIZED-FIB-AUX(n, r)

1. if r[n] ≥ 0
2. return r[n]
3. if n = 0 or n = 1
4. ans ← 1
5. else
6. ans ← MEMOIZED-FIB-AUX(n - 1, r) + MEMOIZED-FIB-AUX(n - 2, r)
7. r[n] ← ans
8. return r[n]

Bottom-Up (Tabulation):

1. Create array F[0...n]
2. Base cases: F[0] = 0, F[1] = 1
3. For i = 2 to n:
- Compute F[i] = F[i - 1] + F[i - 2]
4. Return F[n]
BOTTOM-UP-FIB(n)

1. Let r = [0...n] be a new array
2. r[0] ← 1
3. r[1] ← 1
4. for i = 2 to n
5. r[i] ← r[i - 1] + r[i - 2]
6. return r[n]
Complexity: Time: O(n), Space: O(n)

Matrix Chain Multiplication:

Find optimal parenthesization to minimize multiplications

Bottom-Up (Tabulation):

1. Create table m[1...n, 1...n] with m[i, i] = 0
- m[i, j] stores minimal cost of multiplying matrices i through j
2. For l = 2 to n (chain length):
3. For i = 1 to n - l + 1:
- Set j = i + l - 1
- Compute m[i, j] = min_{i ≤ k < j} {m[i, k] + m[k + 1, j] + p_{i-1}p_kp_j}
- Store k in s[i, j] that achieved minimum cost
4. Return m[1, n]
MATRIX-CHAIN-ORDER(p)

1. n = p.length - 1
2. let m[1...n, 1...n] and s[1...n, 1...n] be new tables
3. for i = 1 to n
4. m[i, i] = 0
5. for l = 2 to n // l is the chain length
6. for i = 1 to n - l + 1
7. j = i + l - 1
8. m[i, j] = ∞
9. for k = i to j - 1
10. q = m[i, k] + m[k + 1, j] + p_{i-1}p_kp_j
11. if q < m[i, j]
12. m[i, j] = q
13. s[i, j] = k // s stores the optimal choice
14. return m and s
Complexity:
- Time: O(n^3)
- Space: O(n^2) for m and s tables
- Table s[i, j] stores index of last matrix in first parenthesized group

Optimal Binary Search Tree

Problem:

Construct a BST with minimum expected search cost given access probabilities

Optimal-BST Algorithm:

Input: Keys K_1, ..., K_n, probabilities p_1, ..., p_n for successful searches
• Optional: Probabilities q_0, ..., q_n for unsuccessful searches
• Create tables e[1...n + 1, 0...n] for expected costs
• Create w[i, j] for sum of probabilities from i to j
• Create root[i, j] to record optimal roots
• Fill tables bottom-up by increasing subproblem size
• For each subproblem, try all possible roots and pick the minimum cost
• Formula: e[i, j] = min_{i ≤ r ≤ j} {e[i, r - 1] + e[r + 1, j] + w[i, j]}
• The root of the overall optimal tree is in root[1, n]
OPTIMAL-BST(p, q, n)
let e[1...n + 1, 0...n], w[1...n + 1, 0...n], and root[1...n, 1...n] be new tables
for i = 1 to n + 1
e[i, i - 1] = 0
w[i, i - 1] = 0
for l = 1 to n
for i = 1 to n - l + 1
j = i + l - 1
e[i, j] = ∞
w[i, j] = w[i, j - 1] + p_j
for r = i to j
t = e[i, r - 1] + e[r + 1, j] + w[i, j]
if t < e[i, j]
e[i, j] = t
root[i, j] = r
return e and root
Complexity:
• Time: O(n^3)
• Space: O(n^2) for tables e, w, and root
• The algorithm computes optimal costs for all possible subtrees

Longest Common Subsequence

Problem:

Find the longest subsequence common to two sequences

LCS-Length:

- Build tables for length c[0...m, 0...n] and direction b[1...m, 1...n]
- Initialize first row and column to zeros
- For each cell (i, j) in the table:
If characters match, take diagonal value + 1
Otherwise, take maximum from above or left
- Table c[m, n] contains the LCS length
- Table b records decisions for reconstruction
LCS-LENGTH(X, Y, m, n)
let b[1...m, 1...n] and c[0...m, 0...n] be new t
for i = 1 to m
c[i, 0] = 0
for j = 0 to n
c[0, j] = 0
for i = 1 to m
for j = 1 to n
if x_i == y_j
c[i, j] = c[i - 1, j - 1] + 1
b[i, j] = "↖"
else if c[i - 1, j] ≥ c[i, j - 1]
c[i, j] = c[i - 1, j]
b[i, j] = "↑"
else c[i, j] = c[i, j - 1]
b[i, j] = "←"
return c and b

Print-LCS:

- Recursively trace back through direction table b
- Follow diagonal arrows and print characters
- Skip cells with up or left arrows
- Stops when reaching first row or column
PRINT-LCS(b, X, i, j)
if i == 0 or j == 0
return
if b[i, j] == "↖"
PRINT-LCS(b, X, i - 1, j - 1)
print x_i
elseif b[i, j] == "↑"
PRINT-LCS(b, X, i, j - 1)
else PRINT-LCS(b, X, i - 1, j)
Example Recording of Solution:

0 1 2 3 4 5 6 7
1 0 0 0 0 0 0 0
2 0 0 0 0 0 1 1 1 2 1
3 0 0 1 1 1 2 2 2 3 2 1
4 0 1 1 1 1 2 1 2 2 2 2 1
5 0 1 1 1 1 2 2 2 2 1 3 1 4 1
6 0 1 2 2 2 2 2 2 2 1 3 1 4 1
Complexity:
- Time: O(mn) for two sequences of lengths m and n
- Space: O(mn) for the tables
- Space can be optimized to O(min(m, n)) if only length is needed
- Print-LCS takes O(m + n) time to reconstruct the solution

Optimal Binary Search Tree

Problem:

Construct a BST with minimum expected search cost given access probabilities

Optimal-BST Algorithm:

Input: Keys K_1, ..., K_n, probabilities p_1, ..., p_n for successful searches
• Optional: Probabilities q_0, ..., q_n for unsuccessful searches
• Create tables e[1...n + 1, 0...n] for expected costs
• Create w[i, j] for sum of probabilities from i to j
• Create root[i, j] to record optimal roots
• Fill tables bottom-up by increasing subproblem size
• For each subproblem, try all possible roots and pick the minimum cost
• Formula: e[i, j] = min_{i ≤ r ≤ j} {e[i, r - 1] + e[r + 1, j] + w[i, j]}
• The root of the overall optimal tree is in root[1, n]
OPTIMAL-BST(p, q, n)
let e[1...n + 1, 0...n], w[1...n + 1, 0...n], and root[1...n, 1...n] be new tables
for i = 1 to n + 1
e[i, i - 1] = 0
w[i, i - 1] = 0
for l = 1 to n
for i = 1 to n - l + 1
j = i + l - 1
e[i, j] = ∞
w[i, j] = w[i, j - 1] + p_j
for r = i to j
t = e[i, r - 1] + e[r + 1, j] + w[i, j]
if t < e[i, j]
e[i, j] = t
root[i, j] = r
return e and root
Complexity:
• Time: O(n^3)
• Space: O(n^2) for tables e, w, and root
• The algorithm computes optimal costs for all possible subtrees

Linked List

Linear data structure where each node contains:
1. key/data: The value stored in the node
2. next: A pointer to the next node in the sequence
3. prev: A pointer to the previous node (in doubly linked lists)
List-Search (find a node with a given key)
1. Start from the head of the linked list.
2. Traverse the list by following the next pointers.
3. Compare each node's key with the target key.
4. Return the node if the key is found.
5. Return NULL if the end of the list is reached without finding the key.
Time Complexity: O(n) where n is list length Space Complexity: O(1)
List-Insert (insert a new node at the beginning)
1. Create a new node with the given key.
2. Set the next pointer of the new node to point to the current head.
3. If implementing a doubly linked list, set the prev pointer of the current head to the new node.
4. Update the head pointer to point to the new node.
5. If the list was empty, update the tail pointer as well.
Time Complexity: O(1) Space Complexity: O(1)
List-Delete (remove a node from the list)
1. Find the node to be deleted (may require traversal).
2. If the node is the head, update the head pointer to the next node.
3. Otherwise, update the next pointer of the previous node to skip the node being deleted.
4. For doubly linked lists, also update the prev pointer of the next node.
5. Free the memory allocated for the deleted node.
6. Handle edge cases: empty list, deleting the only node, or deleting the tail.
Time Complexity: O(n) for finding the node, O(1) for deletion
Space Complexity: O(1)
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

Longest Common Subsequence

Problem:

Find the longest subsequence common to two sequences

LCS-Length:

- Build tables for length c[0...m, 0...n] and direction b[1...m, 1...n]
- Initialize first row and column to zeros
- For each cell (i, j) in the table:
If characters match, take diagonal value + 1
Otherwise, take maximum from above or left
- Table c[m, n] contains the LCS length
- Table b records decisions for reconstruction
LCS-LENGTH(X, Y, m, n)
let b[1...m, 1...n] and c[0...m, 0...n] be new t
for i = 1 to m
c[i, 0] = 0
for j = 0 to n
c[0, j] = 0
for i = 1 to m
for j = 1 to n
if x_i == y_j
c[i, j] = c[i - 1, j - 1] + 1
b[i, j] = "↖"
else if c[i - 1, j] ≥ c[i, j - 1]
c[i, j] = c[i - 1, j]
b[i, j] = "↑"
else c[i, j] = c[i, j - 1]
b[i, j] = "←"
return c and b

Print-LCS:

- Recursively trace back through direction table b
- Follow diagonal arrows and print characters
- Skip cells with up or left arrows
- Stops when reaching first row or column
PRINT-LCS(b, X, i, j)
if i == 0 or j == 0
return
if b[i, j] == "↖"
PRINT-LCS(b, X, i - 1, j - 1)
print x_i
elseif b[i, j] == "↑"
PRINT-LCS(b, X, i, j - 1)
else PRINT-LCS(b, X, i - 1, j)
Example Recording of Solution:

0 1 2 3 4 5 6 7
1 0 0 0 0 0 0 0
2 0 0 0 0 0 1 1 1 2 1
3 0 0 1 1 1 2 2 2 3 2 1
4 0 1 1 1 1 2 1 2 2 2 2 1
5 0 1 1 1 1 2 2 2 2 1 3 1 4 1
6 0 1 2 2 2 2 2 2 2 1 3 1 4 1
Complexity:
- Time: O(mn) for two sequences of lengths m and n
- Space: O(mn) for the tables
- Space can be optimized to O(min(m, n)) if only length is needed
- Print-LCS takes O(m + n) time to reconstruct the solution

Optimal Binary Search Tree

Problem:

Construct a BST with minimum expected search cost given access probabilities

Optimal-BST Algorithm:

Input: Keys K_1, ..., K_n, probabilities p_1, ..., p_n for successful searches
• Optional: Probabilities q_0, ..., q_n for unsuccessful searches
• Create tables e[1...n + 1, 0...n] for expected costs
• Create w[i, j] for sum of probabilities from i to j
• Create root[i, j] to record optimal roots
• Fill tables bottom-up by increasing subproblem size
• For each subproblem, try all possible roots and pick the minimum cost
• Formula: e[i, j] = min_{i ≤ r ≤ j} {e[i, r - 1] + e[r + 1, j] + w[i, j]}
• The root of the overall optimal tree is in root[1, n]
OPTIMAL-BST(p, q, n)
let e[1...n + 1, 0...n], w[1...n + 1, 0...n], and root[1...n, 1...n] be new tables
for i = 1 to n + 1
e[i, i - 1] = 0
w[i, i - 1] = 0
for l = 1 to n
for i = 1 to n - l + 1
j = i + l - 1
e[i, j] = ∞
w[i, j] = w[i, j - 1] + p_j
for r = i to j
t = e[i, r - 1] + e[r + 1, j] + w[i, j]
if t < e[i, j]
e[i, j] = t
root[i, j] = r
return e and root
Complexity:
• Time: O(n^3)
• Space: O(n^2) for tables e, w, and root
• The algorithm computes optimal costs for all possible subtrees

Linked List

Linear data structure where each node contains:
1. key/data: The value stored in the node
2. next: A pointer to the next node in the sequence
3. prev: A pointer to the previous node (in doubly linked lists)
List-Search (find a node with a given key)
1. Start from the head of the linked list.
2. Traverse the list by following the next pointers.
3. Compare each node's key with the target key.
4. Return the node if the key is found.
5. Return NULL if the end of the list is reached without finding the key.
Time Complexity: O(n) where n is list length Space Complexity: O(1)
List-Insert (insert a new node at the beginning)
1. Create a new node with the given key.
2. Set the next pointer of the new node to point to the current head.
3. If implementing a doubly linked list, set the prev pointer of the current head to the new node.
4. Update the head pointer to point to the new node.
5. If the list was empty, update the tail pointer as well.
Time Complexity: O(1) Space Complexity: O(1)
List-Delete (remove a node from the list)
1. Find the node to be deleted (may require traversal).
2. If the node is the head, update the head pointer to the next node.
3. Otherwise, update the next pointer of the previous node to skip the node being deleted.
4. For doubly linked lists, also update the prev pointer of the next node.
5. Free the memory allocated for the deleted node.
6. Handle edge cases: empty list, deleting the only node, or deleting the tail.
Time Complexity: O(n) for finding the node, O(1) for deletion
Space Complexity: O(1)
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