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`) or via EPFL email (`ali.elazdi@epfl.ch`).

March 25th, 2025

Asymptotic Notation

Big-O

If $\exists c > 0$ and $\exists n_0 > 0$, $0 \le f(n) \le c \cdot g(n)\ \forall n \ge n_0$, then $f(n) = O(g(n))$.

Big-Omega

If $\exists c > 0$ and $\exists n_0 > 0$, $0 \le c \cdot g(n) \le f(n)\ \forall n \ge n_0$, 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 $T(n) = a\,T\left(\frac{n}{b}\right) + f(n)$, where $a \ge 1$, $b > 1$, and $f(n)$ is asymptotically positive. The solution depends on comparing $f(n)$ to $n^{\log_b a}$:

1. **Case 1:** If $f(n) = O(n^{\log_b a - \epsilon})$ for some $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. **Case 2:** If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \log n)$.
3. **Case 3:** If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some $\epsilon > 0$, and if $a\,f\left(\frac{n}{b}\right) \le c\,f(n)$ 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:

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

Insertion Sort

1. **Select the key**
   Begin with the second element (at index 1) as the *key*.
2. **Compare and Shift**
   Compare the key with elements in the sorted section (to its left).
3. **Shift Elements**
   If an element is greater than the key, shift that element one position to the right.
4. **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.
5. **Repeat**
   Move forward to the next element, treating it as the new key, and repeat until the array is sorted.

```
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
```

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

Ideal for small or nearly sorted arrays.

Maximum Subarray Problem

Problem: Find contiguous subarray with largest sum

1. **Divide and Conquer Approach:**
   Divide: Split array at midpoint
   mid = ⌊(low + high)/2⌋

   Conquer: Find maximum subarrays recursively
   1. Left max: in A[low...mid]
   2. Right max: in A[mid + 1...high]
   3. Crossing max: spans the midpoint

   Combine: Return the largest of the three
   max(left_max, right_max, crossing_max)
2. **Finding the Crossing Maximum:**
   1. Find maximum suffix in left half (from mid down to low)
   2. Find maximum prefix in right half (from mid+1 up to high)
   3. 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])          // base case: only one element
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[i..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)*
1. Returns TRUE if the stack is empty.
2. Returns FALSE otherwise.

Push(S, x): *Time: O(1), Space: O(1)*
1. Adds element x to the top of stack S.
2. Increments the stack pointer.

Pop(S): *Time: O(1), Space: O(1)*
1. If Stack-Empty(S), return error "underflow".
2. Otherwise, remove and return the top element.
3. Decrements the stack pointer.

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

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

Strassen's Matrix Multiplication

1. **Divide:** Partition each of A, B, C into four $\frac{n}{2} \times \frac{n}{2}$ submatrices:
$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \cdot \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}.$$
2. **Conquer:** Compute 7 products (recursively on $\frac{n}{2} \times \frac{n}{2}$ matrices):
$$M_1 := (A_{11} + A_{22})(B_{11} + B_{22}),$$
$$M_2 := (A_{21} + A_{22})\,B_{11},$$
$$M_3 := A_{11}\,(B_{12} - B_{22}),$$
$$M_4 := A_{22}\,(B_{21} - B_{11}),$$
$$M_5 := (A_{11} + A_{12})\,B_{22},$$
$$M_6 := (A_{21} - A_{11})\,(B_{11} + B_{12}),$$
$$M_7 := (A_{12} - A_{22})\,(B_{21} + B_{22}).$$
3. **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: $O(n^2)$

Queue Operations

Queue-Empty(Q): *Time: O(1), Space: O(1)*
1. Returns TRUE if the queue is empty (Q.head = Q.tail).
2. Returns FALSE otherwise.

Enqueue(Q, x): *Time: O(1), Space: O(1)*
1. Adds element x to the rear of queue Q.
2. Q[Q.tail] = x
3. Q.tail = Q.tail + 1 (or wrap around if using circular array)

Dequeue(Q): *Time: O(1), Space: O(1)*
1. If Queue-Empty(Q), return error "underflow".
2. Otherwise, remove and return the element at the front.
3. x = Q[Q.head]
4. Q.head = Q.head + 1 (or wrap around)
5. Return x

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

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

Merge Sort

1. **Divide**: Split the array evenly into two smaller subarrays, and continue dividing recursively.
2. **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                        // check for base case
    q = ⌊(p + r)/2⌋             // divide
    MERGE-SORT(A, p, q)         // conquer
    MERGE-SORT(A, q + 1, r)     // conquer
    MERGE(A, p, q, r)           // combine
```

3. **Merge**: Combine the two sorted subarrays into a single sorted array:
   (a) Initializing pointers at the start of each subarray.
   (b) Comparing the elements pointed to, and appending the smaller one into a new array.
   (c) Advancing the pointer in the subarray from which the element was chosen.
   (d) 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
1. Increment the heap size
2. Insert a new node in the last position in the heap, with key $-\infty$
3. Increase the $-\infty$ value to key using Heap-Increase-Key

Extract-Max(S): Remove and return element of S with highest priority
1. Make sure heap is not empty
2. Make a copy of the maximum element (the root)
3. Make the last node in the tree the new root
4. Re-heapify the heap, with one fewer node
5. 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
1. Make sure key $\ge$ A[i]
2. Update A[i]'s value to key
3. 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: $O(n)$

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)
1. Starting at the root
2. Compare A[i], A[Left(i)], A[Right(i)]
3. If necessary, swap A[i] with the largest of the two children
4. **Max-Heapify** the swapped child
5. 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)
1. Increase heap size: A.heap-size = A.heap-size + 1
2. Set the last element to negative infinity: A[A.heap-size] = $-\infty$
3. 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)
1. Ensure new key is larger than current: if key < A[i] then error
2. Set A[i] = key
3. Compare with parent and swap if necessary: while i > 1 and A[Parent(i)] < A[i]
4. Exchange A[i] with A[Parent(i)]
5. 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) 1. Start from the last non-leaf node at index $\frac{n}{2} - 1$
2. Move upwards to the root (index 0) and:
a. **Max-Heapify** the current node
b. Ensure the subtree rooted here satisfies max-heap property
3. Repeat until the root is processed
4. After completion, array A represents a valid max heap

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

Heap Sort

1. Build a Max Heap:
a. Convert the given array into a max heap
b. Start from the last non-leaf node and heapify upwards
c. Ensure each parent node is greater than its children

2. Extract Maximum Elements:
a. Swap the root (maximum value) with the last element
b. Reduce heap size by one to exclude the last element
c. Heapify the root to maintain max heap property
d. Repeat until heap size becomes 1

3. Final Sorted Array:
a. After extraction, the sorted array in ascending order is obtained
b. 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)
```

```
HEAP-INCREASE-KEY(A, i, key)
if key < A[i]
    error "new key is smaller than current key"
A[i] = key
while i > 1 and A[PARENT(i)] < A[i]
    exchange A[i] with A[PARENT(i)]
    i = PARENT(i)
```

```
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	$O(1)$	$O(1)$

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-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	$O(\log n)$	$O(1)$

Stack Operations

Operation	Description	Time	Spac
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	$O(1)$	$O(1)$

BST Operations

Operation	Description	Time	Spac
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	Spac
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-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-Preorder

1. Visit current node
2. Recursively traverse left
3. Recursively traverse right

(Root first, then children)

```
PREORDER-TREE-WALK(x)
  if x ≠ NIL
      print key[x]
      PREORDER-TREE-WALK(x.left)
      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-Postorder

1. Recursively traverse left
2. Recursively traverse right
3. Visit current node

(Children first, then root)

```
POSTORDER-TREE-WALK(x)
  if x ≠ NIL
      POSTORDER-TREE-WALK(x.left)
      POSTORDER-TREE-WALK(x.right)
      print key[x]
```

Time: $O(n)$
Space: $O(h)$

BST-Transplant

Replace subtree at u with v:
1. If u is root, set v as root
2. If u is left child, make v left child of u's parent
3. Else make v right child of u's parent
4. Set v's parent to u's parent

```
TRANSPLANT(T, u, v)
  if u.p == NIL
      T.root = v
  elseif u == u.p.left
      u.p.left = v
  else u.p.right = v
  if v ≠ NIL
      v.p = u.p
```

Time: $O(1)$
Space: $O(1)$

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
  elseif 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 ≠ z
          // 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, z, y)
      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 $F[0 \dots n]$ initialized to NIL
2. Base cases: $F[0] = 0$, $F[1] = 1$
3. Recursive 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)
  Let r = [0...n] be a new array
  for i = 0 to n
      r[i] ← −∞
  return MEMOIZED-FIB-AUX(n, r)
```

```
MEMOIZED-FIB-AUX(n, r)
  if r[n] ≥ 0
      return r[n]
  if n = 0 or n = 1
      ans ← 1
  else
      ans ← MEMOIZED-FIB-AUX(n − 1, r)+
            MEMOIZED-FIB-AUX(n − 2, r)
  r[n] ← ans
  return r[n]
```

Bottom-Up (Tabulation):

1. Create array $F[0 \dots n]$
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)
  Let r = [0...n] be a new array
  r[0] ← 1
  r[1] ← 1
  for i = 2 to n
      r[i] ← r[i − 1] + r[i − 2]
  return r[n]
```

Complexity: Time: $O(n)$, Space: $O(n)$

Cut-Rod Problem:

Find optimal way to cut rod to maximize revenue

Top-Down (Memoization):

1. Create memo array $r[0 \dots n]$ with $r[0] = 0$
2. For uncalculated $r[j]$, compute:
$r[j] = \max_{1 \le i \le j}(p[i] + r[j-i])$
3. Return $r[n]$

```
MEMOIZED-CUT-ROD-AUX(p, n, r)
  if r[n] ≥ 0
      return r[n]
  if n == 0
      q = 0
  else q = −∞
      for i = 1 to n
          q = max(q, p[i] + MEMOIZED-CUT-ROD-AUX(p, n − i, r))
  r[n] = q
  return q
```

```
MEMOIZED-CUT-ROD(p, n)
  let r[0..n] be a new array
  for i = 0 to n
      r[i] = −∞
  return MEMOIZED-CUT-ROD-AUX(p, n, r)
```

Bottom-Up (Tabulation):

1. Create array $r[0 \dots n]$ with $r[0] = 0$
2. For $j = 1$ to n:
- Compute $r[j] = \max_{1 \le i \le j}(p[i] + r[j-i])$
3. Return $r[n]$

```
EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
  let r[0..n] and s[0..n] be new arrays
  r[0] = 0
  for j = 1 to n
      q = −∞
      for i = 1 to j
          if q < p[i] + r[j − i]
              q = p[i] + r[j − i]
              s[j] = i
      r[j] = q
  return r and s
```

Complexity: Time: $O(n^2)$, Space: $O(n)$

Matrix Chain Multiplication:

Find optimal parenthesization to minimize multiplications

Bottom-Up (Tabulation):

1. Create table $m[1 \dots n, 1 \dots n]$ 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 $m[i,j] = \min_{i \le k < j}\{m[i,k] + m[k+1,j] + p_{i-1}p_kp_j\}$
- Store k in $s[i,j]$ that achieved minimum cost
4. Return $m[1,n]$

```
MATRIX-CHAIN-ORDER(p)
  n = p.length − 1
  let m[1..n, 1..n] and s[1..n, 1..n] be new tables
  for i = 1 to n
      m[i, i] = 0
  for ℓ = 2 to n          // ℓ is the chain length
      for i = 1 to n − ℓ + 1
          j = i + ℓ − 1
          m[i, j] = ∞
          for k = i to j − 1
              q = m[i, k] + m[k + 1, j] + p_{i−1} p_k p_j
              if q < m[i, j]
                  m[i, j] = q
                  s[i, j] = k     ⇐ s stores the optimal choice
  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

Longest Common Subsequence

Problem: Find the longest subsequence common to two sequences

LCS-Length:

- Build tables for length $c[0 \dots m, 0 \dots n]$ and direction $b[1 \dots m, 1 \dots n]$
- 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, o..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 − 1, j)
  else PRINT-LCS(b, X, i, j − 1)
```

Example Recording of Solution:

i \ j	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0 ↑	0 ↑	0 ↑	1 ↖	1 ←	1 ↖	1 ←
2	0	0 ↑	1 ↖	1 ←	1 ↑	1 ↑	1 ↑	2 ↖
3	0	0 ↑	1 ↑	1 ↑	2 ↖	2 ←	2 ↖	2 ↑
4	0	1 ↖	1 ↑	1 ↑	2 ↑	2 ↑	2 ↑	2 ↑
5	0	1 ↑	1 ↑	1 ↑	2 ↖	2 ↑	3 ↖	3 ←
6	0	1 ↑	2 ↖	2 ←	2 ↑	2 ↑	3 ↑	4 ↖

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 $K_1, \dots, K_n$, probabilities $p_1, \dots, p_n$ for successful searches
- Optional: Probabilities $q_0, \dots, q_n$ for unsuccessful searches
- Create tables $e[1 \dots n+1, 0 \dots n]$ 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: $e[i,j] = \min_{i \le r \le j}\{e[i,r-1] + e[r+1,j] + w[i,j]\}$
- 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.

```
LIST-SEARCH(L,K)
  x ← L.head
  while x ≠ nil and x.key ≠ k
      x ← x.next
  return x
```

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.

```
LIST-INSERT(L,X)
  x.next ← L.head
  if L.head ≠ nil
      L.head.prev ← x
  L.head ← x
  x.prev = NIL
```

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.

```
LIST-DELETE(L,X)
  if x.prev ≠ nil
      x.prev.next ← x.next
  else L.head ← x.next
  if x.next ≠ nil
      x.next.prev ← x.prev
```

Time Complexity: $O(n)$ for finding the node, $O(1)$ for deletion
Space Complexity: $O(1)$