Algorithms - CheatSheet

IN BA4 - Martin Werner Licht

Notes by Ali EL AZDI

This cheat sheet provides a concise summary of key algorithms and concepts. 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 $f(n) = \Theta(g(n))$.

Insertion Sort

```
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
```

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.

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 large

1. **Divide and Conquer Approach:**
   Divide: Split array at midpoint
   mid $= \lfloor(\text{low} + \text{high})/2\rfloor$

   Conquer: Find maximum subarrays recursively
   1. Left max: in $A[\text{low}...\text{mid}]$
   2. Right max: in $A[\text{mid} + 1...\text{high}]$
   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(

```
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):
1. Returns TRUE if the stack is empty.
2. Returns FALSE otherwise.

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

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

Time Complexity: $O(1)$ for all operations

Space Complexity: $O(n)$

Queue

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

Enqueue(Q, x):
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):
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 if using circular array)
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 implementation, indices wrap around using modulo arithmetic
4. For a queue with capacity n, we leave one slot empty to distinguish between full and empty states

Time Complexity: $O(1)$ for all operations *Space Complexity:* $O(n)$

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 $T(n) = \Theta(n^{\log_b a})$.

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(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
```

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

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: $O(n^2)$

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 to preserve heap property
4. **Max-Heapify** the swapped child
5. Continue this process of comparing and swapping down the heap, until subtree rooted at i is max-heap

```
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)
```

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, which is located at index $\frac{n}{2} - 1$ (in a zero-indexed array).
2. Move upwards to the root (index 0) and perform the following:
a. **Max-Heapify** the current node.
b. Ensure that the subtree rooted at this node satisfies the max-heap property.
3. Repeat this process until the root node is processed.
4. After the entire process, the array A represents a valid max heap.

```
BUILD-MAX-HEAP(A, n)
  for i = ⌊n/2⌋ downto 1
      MAX-HEAPIFY(A, i, n)
```

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 move upwards to the root, heapifying each node.
c. Ensure that each parent node is greater than its child nodes.

```
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)
```

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

3. Final Sorted Array:
a. After extracting maximum elements one by one, the sorted array is obtained.
b. The array is sorted in ascending order since the maximum element is placed at the end.

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

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)
1. x ← L.head
2. while x ≠ nil and x.key ≠ k
3.     x ← x.next
4. return x
```

Time Complexity: $O(n)$ *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)
1. x.next ← L.head
2. if L.head ≠ nil
3.     L.head.prev ← x
4. L.head ← x
5. 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)
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
```

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

Binary Search Trees (BST)

Properties

- The left subtree of a node contains only nodes with keys less than the node's key
- The right subtree of a node contains only nodes with keys greater than the node's key
- Both the left and right subtrees are also binary search trees

Node Structure:
Each node in a BST typically contains:
1. **key**: The value stored in the node
2. **left**: A pointer to the left child
3. **right**: A pointer to the right child
4. **parent**: A pointer to the parent node (optional)

BST-Search
(find a node with a given key)

1. Start from the root of the tree.
2. If the root is NULL, return NULL.
3. If the key equals the root's key, return the root.
4. If the key is less than the root's key, recursively search the left subtree.
5. If the key is greater than the root's key, recursively search the right subtree.

```
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 Complexity: $O(\log n)$ average, $O(n)$ worst case *Space Complexity:* $O(\log n)$ for recursion

BST-Minimum
(find the minimum key in the tree)

1. Start from the root of the tree.
2. If the root is NULL, return NULL.
3. Traverse the tree by following the left child pointers until reaching a node with no left child.
4. Return this leftmost node, which contains the minimum key.

```
TREE-MINIMUM(x)
  while x.left ≠ NIL
      x = x.left
  return x
```

Time Complexity: $O(h)$ where h is the height of the tree *Space Complexity:* $O(1)$

BST-Maximum
(find the maximum key in the tree)

1. Start from the root of the tree.
2. If the root is NULL, return NULL.
3. Traverse the tree by following the right child pointers until reaching a node with no right child.
4. Return this rightmost node, which contains the maximum key.

```
TREE-MAXIMUM(x)
  while x.right ≠ NIL
      x = x.right
  return x
```

Time Complexity: $O(h)$ where h is the height of the tree *Space Complexity:* $O(1)$

BST-Successor
(find the node with the next larger key)

1. If the node has a right subtree, return the minimum node in that right subtree.
2. Otherwise, traverse up the tree using parent pointers.
3. Find the first ancestor for which the given node is in its left subtree.
4. Return this ancestor as the successor.
5. If no such ancestor exists, there is no successor.

```
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 Complexity: $O(h)$ where h is the height of the tree *Space Complexity:* $O(1)$

BST-Predecessor
(find the node with the next smaller key)

1. If the node has a left subtree, return the maximum node in that left subtree.
2. Otherwise, traverse up the tree using parent pointers.
3. Find the first ancestor for which the given node is in its right subtree.
4. Return this ancestor as the predecessor.
5. If no such ancestor exists, there is no predecessor.