Algorithms Cheat Sheet

We use the clrscode3e package in LaTeX to typeset pseudocode of most algorithms in **Introduction to Algorithms** (*Third edition*, by *Cormen*, *Leiserson*, *Rivest*, and *Stein*) for quick reference.

1 Sorting

1.1 Bubble Sort (pg.40)

```
BUBBLESORT(A)
for i = 1 to A.length − 1
    for j = A.length downto i + 1
        if A[j] < A[j − 1]
            exchange A[j] with A[j − 1]
```

1.2 Selection Sort (pg.29)

```
SELECTION-SORT(A)
for i = 1 to A.length − 1
    for j = i to A.length
        select the smallest A[j]
        exchange A[j] with A[i]
```

1.3 Insertion Sort (pg.16)

```
INSERTION-SORT(A)
for j = 2 to A.length
    key = A[j]
    // Insert A[j] into sorted 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
```

1.4 Merge Sort (pg.31)

```
MERGE-SORT(A, p, r)
if p < r
    q = ⌊(p + r)/2⌋
    MERGE-SORT(A, p, q)
    MERGE-SORT(A, q + 1, r)
    MERGE(A, p, q, r)
```

```
MERGE(A, p, q, r)
n_i = q − p + 1
n_2 = r − q
new arrays L[1 . . n_i + 1] and R[1 . . n_2 + 1]
for i = 1 to n_1
    L[i] = A[p + i − 1]
for j = 1 to n_2
    R[j] = A[q + j]
L[n_1 + 1] = ∞
R[n_2 + 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
```

1.5 Heap Sort (pg.151)

```
PARENT(i)
return ⌊i/2⌋
```

```
LEFT(i)
return 2i
```

```
RIGHT(i)
return 2i + 1
```

```
MAX-HEAPIFY(A, i)
l = LEFT(i)
r = RIGHT(i)
if l ≤ A.heap-size and A[l] > A[i]
    largest = l
else largets = i
if r ≤ A.heap-size and A[r] > A[largest]
    largest = r
if largest ≠ i
    exchange A[i] with A[largest]
    MAX-HEAPIFY(A, largest)
```

```
BUILD-MAX-HEAP(A)
A.heap-size = A.length
for i = ⌊A.length/2⌋ downto 1
    MAX-HEAPIFY(A, i)
```

```
HEAPSORT(A)
BUILD-MAX-HEAP(A)
for i = A.length downto 2
    exchange A[1] with A[i]
    A.heap-size = A.heap-size − 1
    MAX-HEAPIFY(A, 1)
```

1.6 Quick Sort (pg.170)

```
QuickSort(A, p, r)
if p < r
    q = Partition(A, p, r)
    QuickSort(A, p, q − 1)
    QuickSort(A, q + 1, r)
```

```
Partition(A, p, r)
pivot = A[r]
i = p
j = r − 1
while i ≠ j
    if i < pivot
        i = i + 1
    else exchange A[i] with A[j]
        j = j − 1
exchange A[i] with A[r]
return i
```

1.7 Counting Sort (pg.194)

```
Counting-Sort(A, B, k)
let C[0 .. k] be a new array
for i = 0 to k
    C[i] gets0
for j = 1 to A.length
    C[A[j]] = C[A[j]] + 1
// C[i] now contains number of elements = i.
for i = 1 to k
    C[i] = C[i] + C[i − 1]
// C[i] now contains number of elements ≤ i.
for j = A.length downto 1
    B[C[A[j]]] = A[j]
    C[A[j]] = C[A[j]] − 1
```

1.8 Radix Sort (pg.197)

```
Radix-Sort(A, d)
for i = 1 to d
    use a stable sort to sort array A on digit i.
```

1.9 Bucket Sort (pg.200)

```
Bucket-Sort(A)
let B[0 .. n − 1] be a new array
n = A.length
for i = 0 to n − 1
    make B[i] an empty list
for i = 1 to n
    insert A[i] into list B[⌊nA[i]⌋]
for i = 1 to n − 1
    sort list B[i] with insertion sort
concatenate lists B[0], B[1], . . . , B[n − 1] in order
```

2 Data Structures

2.1 Stacks (pg.233)

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

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

```
Pop(S)
if Stack-Empty(S)
    error "underflow"
else S.top = S.top − 1
    return S[S.top + 1]
```

2.2 Queues (pg.234)

```
Enqueue(Q, x)
Q[Q.tail] = x
if Q.tail == Q.length
    Q.tail = 1
else Q.tail = Q.tail + 1
```

```
Dequeue(Q)
x = Q[Q.head]
if Q.head == Q.length
    Q.head = 1
else Q.head = Q.head + 1
return x
```

2.3 Linked Lists (pg.236)

```
List-Search(L, k)
x = L.head
while x ≠ NIL and x.key ≠ k
    x = x.next
return x
```

```
List-Insert(L, x)
x.next = L.head
if L.head ≠ NIL
    L.head.prev = x
L.head = x
x.prev = NIL
```

```
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
```

2.4 Binary Search Tree (pg.286)

```
TREE-SEARCH(x, k)
if x == NIL or k == x.key
    return x
if k < x.key
    return TREE-SEARCH(x.left, k)
else return TREE-SEARCH(x.right, k)
```

```
ITERATIVE-TREE-SEARCH(x, k)
while x ≠ NIL and k ≠ x.key
    if k < x.key
        x = x.left
    else x = x.right
return x
```

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

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

```
TREE-SUCCESSOR(x)
while x.right ≠ NIL
    return TREE-MINIMUM(x.right)
y = x.p
while y ≠ NIL and y.right == x
    x = y
    y = y.p
return y
```

```
TREE-PREDECESSOR(x)
// Symmetric to TREE-SUCCESSOR
```

```
TREE-INSERT(T, z)
y = NIL
x = T.root
while x ≠ NIL
    y = x
    if z.key < x.key
        x = x.left
    else x = x.right
z.p = y
if y == NIL
    T.root = z      // tree T was empty
elseif z.key < y.key
    y.left = z
else y.right = z
```

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

```
TREE-DELETE(T, z)
if z.left == NIL
    TRANSPLANT(T, z, z.right)
elseif z.right == NIL
    TRANSPLANT(T, z, z.left)
else y = TREE-MINIMUM(z.right)
    if y.p ≠ z
        TRANSPLAT(T, y, y.right)
        y.right = z.right
        y.right.p = y
    TRANSPLANT(T, z, y)
    y.left = z.left
    y.left.p = y
```

3 Graph Algorithms

3.1 Elementary Graph Algorithms(pg.589)

```
BFS(G, s)
for each vertex u ∈ G.V − {s}
    u.color = WHITE
    u.d = ∞
    u.π = NIL
s.color = GRAY
s.d = 0
s.π = NIL
Q = ∅
ENQUEUE(Q, s)
while Q ≠ ∅
    u = DEQUEUE(Q)
    for each v ∈ G.Adj[u]
        if v.color == WHITE
            v.color = GRAY
            v.d = u.d + 1
            v.π = u
            ENQUEUE(Q, v)
    u.color = BLACK
```

```
DFS(G)
for each vertex u ∈ G.V
    u.color = WHITE
    u.π = NIL
time = 0
for each vertex u ∈ G.V
    if u.color == WHITE
        DFS-VISIT(G, u)
```

```
DFS-VISIT(G, u)
time = time + 1    // white vertex u is discovered
u.d = time
u.color = GRAY
for each v ∈ G.Adj[u]    // explore edge (u, v)
    if v.color == WHITE
        v.π = u
        DFS-VISIT(G, v)
u.color = BLACK    // blacken u; it is finished
time = time + 1
u.f = time
```

```
TOPOLOGICAL-SORT(G)
call DFS(G) to compute v.f for each vertex v
as each vertex is finished, insert it onto the front
of a linked list
return the linked list of vertices
```

3.2 Minimum Spanning Trees (pg.624)

```
GENERIC-MST(G, w)
A = ∅
while A does not form a spanning tree
    find an edge (u, v) that is safe for A
    A = A ∪ {(u, v)}
return A
```

```
MST-KRUSKAL(G, w)
A = ∅
for each vertex v ∈ G.V
    // several disjoint sets of elements
    MAKE-SET(v)
sort edges of G.E into nondecreasing order by weight
for each edge (u, v) ∈ G.E (in nondecreasing order)
    // FIND-SET(u) returns which set that contains u
    if FIND-SET(u) ≠ FIND-SET(v)
        A = A ∪ {(u, v)}
        UNION(u, v)
return A
```

```
MST-PRIM(G, w, r)
for each u ∈ G.V
    // v.key is min w connecting v to a vertex in tree
    u.key = ∞
    u.π = NIL
r.key = 0
Q = G.V
while Q ≠ ∅
    u = EXTRACT-MIN(Q)
    for each v ∈ G.Adj[u]
        if v ∈ Q and w(u, v) < v.key
            v.π = u
            v.key = w(u, v)
```

3.3 Single-Source Shortest Paths (pg.643)

```
INITIALIZE-SINGLE-SOURCE(G, s)
for each vertex v ∈ G.V
    v.d = ∞
    v.π = NIL
s.d = 0
```

```
RELAX(u, v, w)
if v.d > u.d + w(u, v)
    v.d = u.d + w(u, v)
    v.π = u
```

```
BELLMAN-FORD(G, w, s)
INITIALIZE-SINGLE-SOURCE(G, s)
for i = 1 to |G.V| − 1
    for each edge (u, v) ∈ G.E
        RELAX(u, v, w)
// returns TRUE ⟺ G contains no negative-weight
cycles reachable from s
for each edge (u, v) ∈ G.E
    if v.d > u.d + w(u, v)
        return FALSE
return TRUE
```

```
DAG-SHORTEST-PATHS(G, w, s)
topologically sort the vertices of G
INITIALIZE-SINGLE-SOURCE(G, s)
for each vertex u, taken in topologically sorted order
    for each vertex v ∈ G.Adj[u]
        RELAX(u, v, w)
```

```
DIJKSTRA(G, w, s)
INITIALIZE-SINGLE-SOURCE(G, s)
S = ∅
Q = G.V
while Q ≠ ∅
    u = EXTRACT-MIN(Q)
    S = S ∪ {u}
    for each vertex v ∈ G.Adj[u]
        RELAX(u, v, w)
```