# 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

```
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

```
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

```
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

```
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

```
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

```
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

```
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

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

### 1.9 Bucket sort

```
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

```
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

```
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

```
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

```
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