Performance Characteristics

Sorting

ALGORITHM	IN PLACE	STABLE	BEST	AVERAGE	WORST	REMARKS
selection sort	X		½ n^2	½ n^2	½ n^2	n exchanges; quadratic in best case
insertion sort	X	X	n	¼ n^2	½ n^2	use for small or partially-sorted arrays
bubble sort	X	X	n	½ n^2	½ n^2	rarely useful; use insertion sort instead
shellsort	X		n log3 n	unknown	c n^3/2	tight code; subquadratic
mergesort		X	½ n lg n	n lg n	n lg n	n log n guarantee; stable
quicksort	X		n lg n	2 n ln n	½ n^2	n log n probabilistic guarantee; fastest in practice
heapsort	X		n †	2 n lg n	2 n lg n	n log n guarantee; in place

Priority Queues

DATA STRUCTURE	INSERT	DEL-MIN	MIN	DEC-KEY	DELETE	MERGE
array	1	n	n	1	1	n
binary heap	log n	log n	1	log n	log n	n
d-way heap	logd n	d logd n	1	logd n	d logd n	n
binomial heap	1	log n	1	log n	log n	log n
Fibonacci heap	1	log n †	1	1 †	log n †	log n

- † n lg n if all keys are distinct

Symbol Tables

DATA STRUCTURE	SEARCH	INSERT	DELETE	SEARCH	INSERT	DELETE
sequential search (in an unordered array)	n	n	n	n	n	n
binary search (in a sorted array)	log n	n	n	log n	n	n
binary search tree (unbalanced)	n	n	n	log n	log n	sqrt(n)
red-black BST (left-leaning)	log n	log n	log n	log n	log n	log n
hash table (separate-chaining)	n	n	n	1 †	1 †	1 †
hash table (linear-probing)	n	n	n	1 †	1 †	1 †

- † uniform hashing assumption

Graph Processing

PROBLEM	ALGORITHM	TIME	SPACE
path	DFS	E + V	V
cycle	DFS	E + V	V
directed cycle	DFS	E + V	V
topological sort	DFS	E + V	V
bipartiteness / odd cycle	DFS	E + V	V
connected components	DFS	E + V	V
strong components	Kosaraju–Sharir	E + V	V
Eulerian cycle	DFS	E + V	E + V
directed Eulerian cycle	DFS	E + V	V
transitive closure	DFS	V (E + V)	V 2
minimum spanning tree	Kruskal	E log E	E + V
minimum spanning tree	Prim	E log V	V
minimum spanning tree	Boruvka	E log V	V
shortest paths (unit weights)	BFS	E + V	V
shortest paths (nonnegative weights)	Dijkstra	E log V	V
shortest paths (negative weights)	Bellman–Ford	V (V + E)	V
all-pairs shortest paths	Floyd–Warshall	V 3	V 2
maxflow–mincut	Ford–Fulkerson	E V (E + V)	V

PROBLEM	ALGORITHM	TIME	SPACE
bipartite matching	Hopcroft–Karp	V ½ (E + V)	V
assignment problem	successive shortest paths	n^3 log n	n^2

Sorting

Selection Sort

```
sort(A)
    n = A.length
    for i in [0:n)
        min = i
        for j in [i + 1:n)
            if A[j] < A[min]
                min = j
         swap(A[i], A[min])
```

Insertion Sort

```
sort(A)
    n = A.length
    for i in [0:n)
```

```
        for j = i; j > 0 and A[j] < A[j - 1]; --j
            swap(A[j], A[j - 1])
```

Bubble Sort

```
sort(A)
    n = A.length
    for i in [0:n)
        exchanges = 0
        for j = n - 1; j > i; --j
            if a[j] < a[j - 1]
                swap(A[j], A[j - 1])
                ++exchanges
        if exchanges == 0
            break
```

Shellsort

```
sort(A)
    n = A.length
    h = 1
    while h < n / 3
        h = 3 * h + 1
    while h >= 1
        for i in [h:n)
            for j = i; j >= h and A[j] < A[j - h]; j -= h
```

```
            swap(A[j], A[j - h])
        h = h / 3
```

Mergesort

```
sort(A)
    Aux = A
    sort(A, Aux, 0, A.length - 1)


sort(A, Aux, lo, hi)
    if hi <= lo
        return
    mid = (hi + lo) / 2
    sort(A, Aux, lo, mid)
    sort(A, Aux, mid + 1, hi)
    merge(A, Aux, lo, mid, hi)


merge(A, Aux, lo, mid, hi)
    for k in [lo:hi]  // closed-interval, includes hi
        Aux[k] = A[k]
    i = lo
    j = mid + 1
    for k in [lo:hi]  // closed-interval, includes hi
        if mid < i
            A[k] = Aux[j++]
        else if hi < j
            A[k] = Aux[i++]
        else if Aux[j] < Aux[i]
            A[k] = Aux[j++]
```

```
        else
            A[k] = Aux[i++]
```

Quicksort

```
sort(A)
    sort(A, 0, A.length − 1)


sort(A, lo, hi)
    if hi <= lo
        return
    j = partition(A, lo, hi)
    sort(A, lo, j − 1)
    sort(A, j + 1, hi)


partition(A, lo, hi)
    i = lo
    j = hi + 1
    v = A[lo]
    while true
        while A[++i] < v
            if i == hi
                break
        while v < A[−−j]
            if j == lo
                break
        if j <= i
            break
        swap(A[i], A[j])
```

```
    swap(A[lo], A[j])
    return j
```

Heapsort

- NOTE: uses elements from 1 to n - 1 (rather than 0 to n - 1) to simplify index arithmetic

```
sort(A)
    n = A.length
    for k = n / 2; k >= 1; --k
        sink(A, k, n)
    while n > 1
        swap(A[1], A[n--])
        sink(A, 1, n)


sink(A, k, n)
    while 2 * k <= n
        j = 2 * k
        if j < n and A[j] < A[j + 1]
            ++j
        if A[j] <= A[k]
            break
        swap(A[k], A[j])
        k = j
```

Bucket Sort

- set bucket_size based on size of A, default to 5

```
sort(A, bucket_size)
    if A.length < 10
        // use another sort
    num_buckets = (A.max - A.min / bucket_size) + 1
    Aux // array of arrays num_buckets in size
```

```
    for e in A
        Aux[(e - A.min) / bucket_size].push(e)
    j = 0
    for Arr in Aux
        for e in Arr
            A[j++] = e
```

Counting Sort

```
sort(A)
    if A.length < 10
        // use another sort
    range = A.max - A.min + 1
    Counts // size of range, initialized to 0's
    for e in A
        ++Counts[e - A.min]
    for i in [1:range)
        Counts[i] += Counts[i - 1]
    Aux // size of A
    for a in A
        Aux[--Counts[e - min]] = e
    A = Aux
```

Binary Search

Ordered Array

```
binary_search(A, key)
    lo = 0
    hi = A.length - 1

    while lo <= hi
        mid = (hi + lo) / 2
```

```
        if key < A[mid]
            hi = mid - 1
        else if A[mid] < key
            lo = mid + 1
        else
            return mid

    return -1
```

Binary Search Tree

```
contains(BST, key)
    if key == null
        throw
    return get(BST, key) != null

get(BST, key)
    if key == null
        throw
    if BST == null
        return null
    if key < BST.key
        return get(BST.left, key)
    else if BST.key < key
        return get(BST.right, key)
    return BST.value
```

Binary Search Tree Size

```
size(BST)
    if BST == null
        return 0
    return 1 + size(BST.left) + size(BST.right)
```

Tree Traversal

Recursive

Preorder

```
traverse(root)
    if root == null
        return
    visit(root)
    traverse(root.left)
    traverse(root.right)
```

Inorder

```
traverse(root)
    if root == null
        return
    traverse(root.left)
    visit(root)
    traverse(root.right)
```

Postorder

```
traverse(root)
    if root == null
        return
    traverse(root.left)
    traverse(root.right)
    visit(root)
```

Iterative

Level Order (Breadth First)

```
traverse(root)
    if root == null
        return
    Queue = 0
    Queue.enqueue(root)
    while Queue != 0
        t = Queue.dequeue()
        visit(t)
        if t.left != null
            Queue.enqueue(t.left)
        if t.right != null
            Queue.enqueue(t.right)
```

Preorder

```
traverse(root)
    if root == null
        return
    Stack = 0
    Stack.push(root)
    while Stack != 0
        t = Stack.pop()
        visit(t)
        if t.right != null
            Stack.push(t.right)
```

```
        if t.left != null
            Stack.push(t.left)
```

Inorder

```
traverse(root)
    if root == null
        return
    Stack = 0
    t = root
    while t != null
        while t != null
            if t.right != null
                Stack.push(t.right)
            Stack.push(t)
            t = t.left
        t = Stack.pop()
        while Stack != 0 and t.right == null
            visit(t)
            t = Stack.pop()
        visit(t)
        if Stack == 0
            return
        t = Stack.pop()
```

Postorder

```
traverse(root)
    if root == null
        return
    Stack = 0
    t1 = root
    t2 = root
    while t1 != null
        while t1.left != null
            Stack.push(t1)
```

```
        t1 = t1.left
    while t1.right == null or t1.right == t2
        visit(t1)
        t2 = t1
        if Stack == 0
            return
        t1 = Stack.pop()
    Stack.push(t1)
    t1 = t1.right
```

Tree Printing

Inorder Recursive

```
print_node(item, h)
    for i in [0:h)
        print(" ")
    print(item)
    print("\n")

print_tree(root, h)
    if root == null
        print_node("*", h)
        return
    print_tree(root.right, h + 1)
    print_node(root.value, h)
    print_tree(root.left, h + 1)
```

Level Order (Breadth First)

```
print_tree(root)
    if root == null
        return
```

```
    Queue = 0
    dummy
    Queue.enqueue(root)
    Queue.enqueue(dummy)
    while Queue != 0
        t = Queue.dequeue()
        if t != dummy
            print(t.item)
            if t.left != null
                Queue.enqueue(t.left)
            if t.right != null
                Queue.enqueue(t.right)
         else
            print("\n")
            if Queue != 0
                Queue.enqueue(dummy)
```

Priority Queues

Heap Priority Queue

- PQ - an array of keys with a size that is set on construction

```
class MaxPriorityQueue
    PQ
    size

    insert(key)
        PQ[++size] = key
        swim(size)

    pop() // delete max
        max = PQ[1]
        swap(PQ[1], PQ[size--])
        PQ[size + 1] = null
```

```
        sink(1)
        return max

    sink(k)
        while k > 1 and PQ[k / 2] < PQ[k]
            swap(PQ[k / 2], PQ[k])
            k = k / 2

    swim(k)
        while 2 * k <= size
            j = 2 * k
            if j < size and PQ[j] < PQ[j + 1]
                ++j
            if PQ[j] <= PQ[k]
                break
            swap(PQ[k], PQ[j])
            k = j
```

Symbol Tables

Red-Black BST

```
class Node
    key
    value
    left
    right
    size
    color


class RedBlackBST
    Node root

    is_red(node)
```

```
    if node == null
        return false
    return node.color == red


rotate_left(node)
    x = node.right
    node.right = x.left
    x.left = node
    x.color = node.color
    node.color = red
    x.size = node.size
    node.size = 1 + size(node.left) + size(node.right)
    return x


rotate_right(node)
    x = node.left
    node.left = x.right
    x.right = node
    x.color = node.color
    node.color = red
    x.size = node.size
    node.size = 1 + size(node.left) + size(node.right)
    return x


flip_colors(node)
    node.color = red
    node.left.color = black
    node.right.color = black


put(key, value)
    root = put(root, key, value)
    root.color = black


put(node, key, value)
    if node == null
```

```
        return Node(key, value, 1, red)
    if key < node.key
        node.left = put(node.left, key, value)
    else if node.key < key
        node.right = put(node.right, key, value)
    else
        node.value = value

    if !is_red(node.left) and is_red(node.right)
        node = rotate_left(node)
    if is_red(node.left) and is_red(node.left.left)
        node = rotate_right(node)
    if is_red(node.left) and is_red(node.right)
        flip_colors(node)

    node.size = size(node.left) + size(node.right) + 1
    return node
```

Hash Tables

Hashing

- REMEMBER:
    - hash is a template class in C++ that is overloaded for operator()
    - libcxx uses Murmur2 and CityHash
- radix

```
std::numeric_limits<Type>::max() - std::numeric_limits<Type>::min() + 1
```

- floating point numbers

- sign

```
std::numeric_limits<Type>::is_signed
```

- exponent

```
sizeof(type) * std::numeric_limits<unsigned char>::digits - std::numeric_limits<type>::is_signed - std::numeric_limits
```

- mantissa

```
std::numeric_limits<float>::digits
```

- floating point types in libcxx
    - can use a union of the float type and size_t or reinterpret_cast (better to use reinterpret cast)

```
return bits ^ (bits >> 32)

// or
return bits & ~(~0 >> digits) // where digits is numeric_limits<Float_type>::digits
```

- Sedgewick hashing

```
return (hash(x) & 0x7FFFFFFF) % modulo_divisor

// or
return (hash(x) & (~(0UL) >> 1) % modulo_divisor

// or
return hash(x) % modulo_divisor // the masking is not needed because C++ uses std::size_t
```

- Sedgewick string hashing

```
hash(Str)
    h = 0
    for c in Str
        h = (radix * h + c) % modulo_divisor
    return h
```

- Java-style hashing

```
class Example
    c // char
    s // short
    i // int
    l // long
    f // float
    d // double

hash(object)
    h = 17
    h = 31 * h + hash(object.c)
    h = 31 * h + hash(object.s)
    h = 31 * h + hash(object.i)
    h = 31 * h + hash(object.l)
    h = 31 * h + hash(object.f)
    h = 31 * h + hash(object.d)
    return h
```

Hash Table (Separate Chaining)

- ST is an array of LinkedLists
- NOTE: need to handle resizing and re-hashing, etc

```
class HashTable
    ST
    size
```

```
    hash(key)
        return (hash_code(key) & 0x7FFFFFFF) % M


    get(key)
        return ST[hash(key)].get(key)  // get scans linked list for key


    put(key, value)
        ST[hash(key)].pu(key, value)  // put adds key, value node to linked list
```

Hash Table (Linear Probing)

- Keys and Values are arrays with capacity

```
class HashTable
    Keys
    Values
    capacity
    size


    hash(key)
        return (hash_code(key) & 0x7FFFFFFF) % M


    resize(n)
        t = HashTable(n)
        for i in [0:capacity)
            if Keys[i] != null
                t.put(Keys[i], Values[i])
        Keys = t.Keys
        Values = t.Values
        capacity = t.capacity


    put(key, value)
```

```
        if size >= capacity / 2
            resize(2 * capacity)
        i = hash(key)
        while Keys[i] != null
            if Keys[i] == key
                return Values[i]
            i = (i + 1) % capacity
        Keys[i] = key
        Values[i] = value
        ++size


    get(key)
        i = hash(key)
        while Keys[i] != null
            if Keys[i] == key
                return Values[i]
            i = (i + 1) % capacity
        return null
```

Graph Algorithms

Depth First Search

```
class DepthFirstSearch
    Graph
    Marked

    DepthFirstSearch(source)
        dfs(source)

    dfs(v)
        Marked[v] = true
        for each w adjacent to v in Graph
```

```
            if !Marked[w]
                dfs(w)
```

Breadth First Search

```
class BreadthFirstSearch
    Graph
    Marked

    BreadthFirstSearch(source)
        bfs(source)

    bfs(v)
        Queue
        Marked[v] = true
        Queue.enqueue(v)

        while Queue != 0
            u = Queue.dequeue()
            for each w adjacent to u in Graph
                if !Marked[w]
                    Marked[w] = true
                    Queue.enqueue(w)
```

Connected Components

```
class ConnectedComponents
    Graph
    Marked
    Ids
    ComponentSize
    count


    ConnectedComponents(Graph)
```

```
        for v in [0:Graph.num_vertices())
            if !Marked[v]
                dfs(v)
                ++count


    dfs(v)
        Marked[v] = true
        Ids[v] = count
        ++ComponentSize[count]
        for each w adjacent to v in Graph
            if !Marked[w]
                dfs(w)


    id(v)
        return Ids[v]


    size(v)
        return ComponentSize[Ids[v]]


    connected(v, w)
        return id(v) == id(w)
```

Reachability

Depth First Paths

```
class DepthFirstPaths
    Graph
    Marked
    EdgeTo
    source
```

```
    DepthFirstPaths(Graph, source)
        dfs(source)


    dfs(v)
        Marked[v] = true
        for each w adjacent to v in Graph
            if !Marked[w]
                EdgeTo[w] = v
                dfs(w)


    has_path_to(v)
        return Marked[v]


    path_to(v)
        if !Marked[v]
            return null
        Path // stack
        for x = v; x != source; x = EdgeTo[x]
            Path.push(x)
        path.push(s)
        return path
```

Breadth First Paths

- before running bfs, init DistanceTo to -1, infinity, etc

```
class BreadthFirstPaths
    Graph
    Marked
    EdgeTo
    DistanceTo
    source


    BreadthFirstPaths(Graph, source)
```

```
    bfs(source)


bfs(s)
    Queue
    DistanceTo[s] = 0
    Marked[s] = true
    Queue.enqueue(s)

    while Queue != 0
        v = Queue.dequeue()
        for each w adjacent to v in Graph
            if !Marked[w]
                EdgeTo[w] = v
                DistanceTo[w] = DistanceTo[v] + 1
                Marked[w] = true
                Queue.enqueue(w)


has_path_to(v)
    return Marked[v]


distance_to(v)
    return DistanceTo[v]


path_to(v)
    if !Marked[v]
        return null
    Path // stack
    for x = v; DistanceTo[x] != 0; x = EdgeTo[x]
        Path.push(x)
```

```
        path.push(v)
        return path
```

Cycle (Undirected)

- Graph - undirected graph
- Marked - boolean array signifying that the vertex has been visited where size = number of vertices
- EdgeTo - integer array marking parent in path
- Cycle - stack containing the cycle if found

```
class Cycle
    Graph
    Marked
    EdgeTo
    Cycle

    Cycle()
        for each vertex v in G
            if !Marked(v)
                dfs(-1, v)

    has_cycle()
        return Cycle != 0

    dfs(u, v)
        Marked[v] = true

        for each w adjacent to v in Graph
            if Cycle != 0
                return
            if !Marked[w]
                EdgeTo[w] = v
                dfs(v, w)
            else if w != u
                for x = v; x != w; x = EdgeTo[x]
                    Cycle.push(x)
```

```
                Cycle.push(w)
                Cycle.push(v)
```

Directed Cycle

- Graph - directed graph
- Marked - boolean array signifying that the vertex has been visited where size = number of vertices
- EdgeTo - integer array marking parent in path
- OnStack - boolean array signifying if the vertex is part of the cycle
- Cycle - stack containing the cycle if found

```
class DirectedCycle
    Graph
    Marked
    EdgeTo
    OnStack
    Cycle

    Cycle()
        for each vertex v in G
            if !Marked(v) and Cycle == 0
                dfs(v)

    has_cycle()
        return Cycle != 0

    dfs(v)
        OnStack[v] = true
        Marked[v] = true

        for each w adjacent to v in Graph
            if Cycle != 0
                return
            else if !Marked[w]
                EdgeTo[w] = v
```

```
                dfs(w)
            else OnStack[w]
                for x = v; x != w; x = EdgeTo[x]
                    Cycle.push(x)
                Cycle.push(w)
                Cycle.push(v)
```

Depth First Order

- Graph - directed graph
- Marked - boolean array that signifies if the vertex has been visited
- PreNumbering - the preorder order number of the vertex
- PostNumbering - the postorder order number of the vertex
- Preorder - a queue containing the vertices in their preorder ordering
- Postorder - a queue containing the vertices in their postorder ordering
- preCounter - used to maintain the current preorder index of the current vertex
- postCounter - used to maintain the current postorder index of the current vertex

```
class DepthFirstOrder
    Graph
    Marked
    PreNumbering
    PostNumbering
    Preorder
    Postorder
    pre_counter
    post_counter

    DepthFirstOrder()
        for each vertex v in G
            if !Marked(v)
                dfs(v)

    dfs(v)
```

```
        Marked[v] = true
        PreNumbering[v] = pre_counter++
        Preorder.enqueue(v)
        for each w adjacent to v in Graph
            if !Marked[w]
                dfs(w)
        Postorder.enqueue(v)
        PostNumbering[v] = post_counter++

    reverse_post()
        Stack
        for v in Postorder
            Stack.push(v)
        return Stack
```

Topological Sort

- Graph - directed graph
- Order - the reverse post order numbering generated by DFS

```
class TopologicalSort
    Graph
    Order

    TopologicalSort()
        if !DirectedCycle(Graph).has_cycle()
            order = DepthFirstOrder(Graph).reverse_post()
```

Strong Components (Kosaraju-Sharir)

- Graph - directed graph
- Marked - boolean array
- Id - component identifiers

- count - number of strong components

```
class StrongComponents
    Graph
    Marked
    Id
    count


    StrongComponents(Graph)
        for s in DepthFirstOrder(Graph.reverse()).reverse_post()
            if !Marked(s)
                dfs(s)
                ++count


    dfs(v)
        Marked[v] = true
        Id[v] = count
        for each w adjacent to v in Graph
            if !Marked[w]
                dfs(w)


    strongly_connected(v, w)
        return Id[v] == Id[w]
```

Minimum Spanning Tree (Prim)

- Graph - edge weighted graph
- PQ - Index min priority queue

```
class MST
    Graph
    EdgeTo
    DistanceTo
```

```
    Marked
    PQ


    MST(Graph)
        PQ.insert(0, 0.0)
        while !PQ.empty()
            visit(PQ.pop())


    visit(v)
        Marked[v] = true
        for each e adjacent to v in Graph  // uses an Edge object
            w = e.other(v)
            if Marked[w]
                continue
            if e.weight() < DistanceTo[w]
                EdgeTo[w] = e
                DistanceTo[w] = e.weight()
                if PQ.contains(w)
                    PQ.change_key(w, DistanceTo[w])
                else
                    PQ.insert(w, DistanceTo[w])
```

Minimum Spanning Tree (Kruskal)

- Graph - edge weighted graph
- Queue - a queue of Edge objects

```
class MST
    Graph
    Queue


    MST(Graph)
        PQ // min priority queue
        for e in Graph
```

```
            PQ.push(e)
        uf = UnionFind(Graph)
        while Queue != 0 and Queue.size < Graph.num_vertices() - 1
            e = PQ.pop()
            v, w = vertices in e
            if uf.connected(v, w)
                continue
            uf.create_union(v, w)
            Queue.enqueue(e)
```

Shortest Paths (Djikstra)

- Graph - an edge-weighted directed graph
- EdgeTo - array of directed edge objects
- DistanceTo - the weighted distance to an edge from the source, initialized to infinity and 0.0 for the source
- PQ - index min priority queue

```
class ShortestPaths
    Graph
    EdgeTo
    PQ


    ShortestPaths(Graph, s)
        PQ.push(s, 0.0)
        while PQ != 0
            relax(PQ.pop())


    relax(v)
        for each e adjacent to v in Graph  // uses a DirectedEdge object
            w = e.destination()
            if DistanceTo[v] + e.weight() < DistanceTo[w]
                DistanceTo[w] = DistanceTo[v] + e.weight()
                EdgeTo[w] = e
                if PQ.contains(w)
```

```
                PQ[w] = DistanceTo[w]
            else
                PQ.push(w, DistanceTo[w])


    path_to(v)
        if !has_path_to(w)
            return null
        Path // Stack of DirectedEdges
        for e = EdgeTo[v]; e != null; e = EdgeTo[e.from()]
            Path.push(e)
        return Path
```

Shortest Paths (Bellman-Ford)

```
class ShortestPaths
    Graph
    DistanceTo
    EdgeTo
    InQueue
    Queue
    Cycle
    cost


    ShortestPaths(Graph, s)
        Queue.enqueue(s)
        OnQueue[s] = true
        while Queue != 0 and !has_negative_cycle()
            v = Queue.dequeue()
            OnQueue[v] = false
            relax(v)


    relax(v)
        for each e adjacent to v in Graph  // uses a DirectedEdge object
            w = e.destination()
```

```
        if DistanceTo[v] + e.weight() < DistanceTo[w]
            DistanceTo[w] = DistanceTo[v] + e.weight()
            EdgeTo[w] = e
            if !OnQueue[w]
                Queue.enqueue(w)
                OnQueue[w] = true
        if cost++ % Graph.num_vertices() == 0
            find_negative_cycle()


find_negative_cycle()
    Spt // EdgeWeightedDigraph of size EdgeTo.length
    for v in [0:EdgeTo.length)
        if EdgeTo[v] != null
            Spt.add_edge(EdgeTo[v])
    CF // EdgeWeightedCycleFinder from Spt
    cycle = CF.cycle()


has_negative_cycle()
    return Cycle != 0
```

Dijkstra All-Pairs Shortest Paths

- AllShortestPaths - an array of Dijkstra shortest paths objects

```
class AllPairsShortestPaths
    Graph
    AllShortestPaths


    AllPairsShortestPaths(Graph)
        for v in [0:Graph.num_vertices())
            AllShortestPaths[v] = ShortestPaths(G, v)
```

```
    path(s, t)
        return AllShortestPaths[s].path_to( t)


    distance_to(s, t)
        return AllShortestPaths[s].distance_to(t)
```

String Algorithms

LSD String Sort

- A is an array of strings
- radix - 256 for 8-bit char
- very similar to counting sort

```
sort(A, w) // sort A on leading w characters
    n = A.length
    Aux // size of n
    for d = w - 1; d >= 0; --d
        Counts // size of radix + 1
        for i in [0:n)
            ++Counts[A[i][d] + 1];
        for r in [0:radix)
            Counts[r + 1] += Counts[r]
        for i in [0:n)
            Aux[Counts[A[i][d]++] = A[i]
```

```
        for i in [0:n)
            A[i] = Aux[i]
```

MSD String Sort

- uses same technique as above but in reverse and calculates the character at a position of a string by first checking the length and otherwise returning -1

Trie Symbol Table

- radix - 256 for 8-bit chars
- root - Node
- keys are strings

```
class Node
    value
    Subtries // an array of Nodes of size radix


class Trie
    root

    get(key)
        x = get(root, key, 0)
        if x == null
            return null
        return x.value


    get(node, key, d)  // dth key character for subtrie
        if node == null
            return null
        if d == key.length
            return x
        return get(x.Subtries[key[c]], key, d + 1)
```

```
    put(key, value)
        root = put(root, key, value, 0)


    put(node, key, value, d)
        if x == null
            x = Node
        if d == key.length
            x.value = value
            return x
        x.Subtries[key[d]] = put(x.Subtries[key[d]], key, value, d + 1)
        return x
```

TST

```
class Node
    char
    left
    mid
    right
    value


class TST
    root


    get(Key)
        x = get(root, Key, 0)
        if x == null
            return null
        return x.value


    get(node, Key, d)
```

```
        if node == null
            return null
        c = Key[d]
        if c < node.char
            return get(node.left, Key, d)
        else if node.char < c
            return get(node.right, Key, d)
        else if d < Key.length - 1
            return get(node.mid, Key, d + 1)
        return node


    put(Key, value)
        root = put(root, Key, value, 0)


    put(node, Key, value, d)
        c = Key[d]
        if node == null
            node = Node
            node.char = c
        if c < node.char
            node.left = put(node.left, Key, value, d)
        else if node.char < c
            node.right = put(node.right, Key, value, d)
        else if d < Key.length - 1
            node.mid = put(node.mid, Key, value, d + 1)
        else
            node.value = value
        return node
```

Substring Search (Knuth-Morris-Pratt)

- String - the pattern to search for
- DFA - two-dimensional array of integers (representing characters) of size radix X String.length
- radix - 256 for 8-bit chars

```
class SubstringSearch
    Pattern
    DFA


    SubstringSearch(String)
        DFA[Pattern[0]][0] = 1
        x = 0
        for j in [1:Pattern.length)
            for c in [0:radix)
                DFA[c][j] = DFA[c][x]
            DFA[Pattern[j]][j] = j + 1
            x = DFA[Pattern[j]][x]


    search(Text)
        m = Pattern.length
        n = Text.length
        i = 0
        j = 0
        while i < n and j < m
            j = DFA[Text[i]][j]
            ++i
        if j == m
            return i - m  // found - hit end of pattern
        else
            return n  // not found - hit end of text
```

Substring Search (Boyer-Moore)

- RightOccurence - integer (represents chars) of size radix, initialized to -1
- radix - 256 for 8-bit chars

```
class SubstringSearch
    RightOccurence
    Pattern
```

```
    SubstringSearch(String)
        for i in [0:Pattern.length)
            RightOccurrence[Pattern[j]] = j


    search(Text)
        m = Pattern.length
        n = Text.length
        skip = 0
        i = 0
        for i = 0; i <= n - m; i += skip
            skip = 0
            for j = m - 1; j >= 0; --j
                if Pattern[j] != Text[i + j]
                    skip = j - RightOccurrence[Text[i + j]]
                    if skip < 1
                        skip = 1
                    break
                if skip == 0
                    return i
        return n
```

Substring Search (Rabin-Karp)

- radix - 256 for 8-bit char
- rm - radix ^ (pattern_length - 1) % large_prime

```
class SubstringSearch
    pattern_hash
    pattern_length
    large_prime
    rm


    SubstringSearch(String)
```

```
    for i in [1:pattern_length − 1)
        rm = (radix ∗ rm) % large_prime
    pattern_hash = hash(Pattern, pattern_length)


hash(key, m)
    h = 0
    for j in [0:pattern_length)
        h = (radix ∗ h + key[j]] % large_prime
    return h


search(Text)
    n = Text.length
    text_hash = hash(Text, m)
    if pattern_hash == text_hash
        return 0
    for i in [pattern_length:n)
        text_hash = (text_hash + large_prime − rm ∗ Text[i − m] % large_prime) % large_prime
        text_hash = (text_hash ∗ radix + Text[i]) % large_prime
        if text_hash == pattern_hash
            return i − pattern_length + 1
    return n
```

Regular Expression Pattern Matching

- Regex - char array matching input regular expression
- Digraph - epsilon transitions
- num_states - length of the regular expression

```
class NFA
    Regex
    Digraph
    num_states
```

```
NFA(String)
    Ops // stack
    for i in [0:num_states)
        lp = i
        if Regex[i] == '(' or Regex[i] == '|'
            Ops.push(i)
        else if Regex[i] == ')'
            or = Ops.pop()
            if re[or] == '|'
                lp = Ops.pop()
                Digraph.add_edge(lp, or + 1)
                Digraph.add_edge(or, i)
            else
                lp = or
        if i < num_states - 1 and Regex[i + 1] == '*' // lookahead
            Digraph.add_edge(lp, i + 1)
            Digraph.add_edge(i + 1, lp)
        if Regex[i] == '(' or Regex[i] == '*' or Regex[i] == ')'
            Digraph.add_edge(i, i + 1)


recognizes(Text)
    PC // bag of integers
    dfs // directed DFS of the Digraph from 0
    for v in [0:Digraph.num_vertices())
        if dfs.marked(v)
            PC.add(v)
    for i in [0:Text.length)
        Match // bag of integers
        for v in PC
            if v < num_states
                if Regex[v] == Text[i] or Regex[v] == '.'
                    Match.add(v + 1)
        PC // re-initialize
        dfs // directed DFS of the Digraph for all vertices in Match
        for v in [0:Digraph.num_vertices())
            if dfs.marked(v)
                PC.add(v)
    for v in PC
        if v == num_states
```

```
            return true
        return false
```

Huffman Compression

- radix - 256 for 8-bit char

```
class Node
    char
    frequency
    left
    right


compress(String)
    Input // char array of String
    Frequency
    for i in [0:Inpute.length)
        ++Frequency[Input[i]]
    root = build_trie(Frequency)
    ST // array of strings of size radix
    build_code(ST, root, "")
    Encoded = ""
    for i in [0:Input.length)
        Code = ST[Input[i]]
        for j in Code
            Encoded += j
    return Encoded


build_code(ST, node, String)
    if node.is_leaf()
        ST[node.char] = String
        return
    build_code(ST, node.left, s + '0')
    build_code(ST, node.right, s + '1')
```

```
build_trie(Frequency)
    PQ // min priority queue of trie nodes
    for c in [0:radix)
        if Frequency[c] > 0
            PQ.insert(Node(c, Frequency[c], null, null)
    while PQ.size() > 1
        x = PQ.pop()
        y = PQ.pop
        parent = Node('\0', x.frequency + y.frequency, x, y)
        PQ.insert(parent)
    return PQ.pop()
```

LZW Compression

- radix - 256 for 8-bit char
- num_codewords - 4096 (number of codewords or 2^12)
- width - 12 (codeword width)

```
compress(String)
    ST // TST integer
    for i in [0:radix)
        ST.put("" + i, i)
    code = radix + 1
    Encoded = ""
    while String.length
        S = ST.longest_prefix_of(String) // maximum prefix match
        Encoded += ST.get(S)
        t = S.length
        if t < String.length and code < num_codewords
            ST.put(String[0:t], code++)  // closed-interval, includes t
        String = String[t:)
    return Encoded
```

General Algorithms

Longest Common Subsequence

- MaxLengths - 2D array, Str1.length x Str2.length, initialized to -1

```
class LCS
    MaxLengths
    Str1
    Str2


    LCS(Str1, Str2)


    lcs()
        return lcs(Str1.length - 1, Str2.length - 1)


    lcs(i, j)
        if i == 0 or j == 0
            return 0
        if MaxLengths[i][j] != -1
            return MaxLengths[i][j]
        if Str1[i] == Str2[j]
            result = 1 + lcs(i - 1, j - 1)
        else
            result = max(lcs(i - 1, j), lcs(i, j - 1))
        MaxLengths[i][j] = result
        return result
```

Suffix Arrays

- good for actually returning value of longest common subsequence, longest common prefix, and longest repeated substring

Knapsack

- Options - 2D array of ints, num_items + 1 x max_weight + 1
- Solution - 2D array of bools, num_items + 1 x max_weight + 1
- num_items - Profits.length (or Weights.length)
- max_weight - constraint on weight allowed in knapsack
- NOTE: also assumes a reference to Profits and Weights is maintained
- NOTE: would also memoize get_max_profit and get_total_weight

```
class Knapsack
    Options
    Solution
    num_items
    max_weight


    Knapsack(Profits, Weights)
        for i in [1:num_items]  // closed-interval, includes num_items
            option1 = Options[i - 1][j]
            option2 = Int.min()
            if Weights[i - 1] <= j
                option2 = Profits[i - 1] + Options[i - 1][j - Weights[i - 1]]
            Options[i][j] = max(option1, option2)
            Solution[i][j] = option1 < option2;


    get_items_to_take()
        ItemsToTake  // set of integers
        n = num_items
        w = max_weight
        while n > 0
            if Solution[n][w]
                ItemsToTake.push(n)
                w = w - Weights[n - 1]
            --n
        return ItemsToTake
```

```
    get_max_profit()
        return Options[num_items][max_weight]


    get_total_weight()
        ItemsToTake  // set of integers
        n = num_items
        w = max_weight
        total_weight = 0
        while n > 0
            if Solution[n][w]
                total_weight += Weights[n - 1]
                w = w - Weights[n - 1]
            --n
        return total_weight
```

Maximum Subarray

```
max_subarray(A)
    max = 0
    local_max = 0
    for e in A
        local_max = max(0, local_max + e)
        max = max(max, local_max)
    return max
```

Divide-and-Conquer

Levenshtein Distance

- Distances - prefix distances, 2D array size of Str1 x size of Str2 initialized to -1

```
class Levenshtein
    Distances
    Str1
    Str2



    Levenshtein(Str1, Str2)
        distances(Str1.length − 1, Str2.length − 1)



    get_edit_distances(i, j)
        if i < 0
            return j + 1
        if j < 0
            i + 1
        if Distances[i][j] == −1
            if Str1[i] == Str2[i]
                Distances[i][j] = get_edit_distances(i − 1, j − 1)
            else
                a = get_edit_distances(i − 1, j − 1)
                b = get_edit_distances(i − 1, j)
                c = get_edit_distances(i, j − 1)
                Distances[i][j] = min(a, b, c)
        return Distances[i][j]
```

Permutations

Standard 1

```
permutations(A)
    permutations({}, A)



permutations(P, S)
    n = S.length
    if n == 0
```

```
        Out.push(P)
        return
    for i in [0:n)
        permutations(P + S[i], S[0:i) + S[i + 1:))
```

Standard 2

```
permutations(A)
    permutations(A, A.length)


permutations(A, n)
    if n == 1
        Out.push(A)
        return
    for i in [0:n)
        swap(A[i], A[n − 1])
        permutations(A, n − 1)
        swap(A[i], A[n − 1])
```

K

```
permutations(A, k)
    permutations(A, A.length, k)


permutations(A, n, k)
    if k == 0
        Out.push(A)
        return
    for i in [0:n)
        swap(A[i], A[n − 1])
        permutations(A, n − 1, k − 1)
        swap(A[i], A[n − 1])
```

Lexicographical

```
has_next(A)
    n = A.length
    k = 0
    for k = n - 2; k >= 0; --k
        if A[k] < A[k + 1]
            break
    if k == -1
        return false
    j = n - 1
    while A[j] < A[k]
        --j
    r = n - 1
    s = k + 1
    while r > s
        swap(A[r], A[s]
        --r
        ++s
    return true

permutations(A)
    Out.push(A)
    while has_next(A)
        Out.push(A)
```

Combinations

Standard 1

```
combinations(A)
    combinations({}, A)


combinations(P, S)
```

```
    if S.length <= 0
        return
    Out.push(P + S[0])
    combinations(P + S[0], S[1:))
    combinations(P, S[1:))
```

Standard 2

```
combinations(A)
    combinations({}, A)



combinations(P, S)
    Out.push(P)
    for i in [0:S.length)
        combinations(P + S[i], S[i + 1:))
```

K on n elements

```
combinations(A, pos, next, k, n)
    if pos == k
        Out.push(A)
        return
    for i in [next:n)
        s[pos] = i
        combinations(A, pos + 1, i + 1, k, n)
```

K Lexicographic 1

```
combinations(A, k)
    combinations({}, A, k)



combinations(P, S, k)
```

```
    if S.length < k
        return
    if k == 0
        Out.push(P)
        return
    combinations(P + S[0], S[1:), k - 1)
    combinations(P, S[1:), k)
```

K Lexicographic 2

```
combinations(A, k)
    combinations({}, A, k)


combinations(P, S, k)
    if k == 0
        Out.push(P)
        return
    for i in [0:S.length)
        combinations(P + S[i], S[i + 1:), k - 1)
```

Partitions

```
partition(n)
    partition({}, n, n)


partition(P, n, max)
    if n == 0
        Out.push(P)
        return
    for i = min(n, max); i >= 1; --i
        partition(P + i, n - i, i)
```

Backtracking

n-Queens

- call to Out.push(A) assumes the results can be one-dimensional arrays with index of queen for each row

```
is_consistent(A, n)
    for i in [0:n)
        if A[i] == A[n]  // same column
            return false
        if A[i] - A[n] == n - i  // same major diagonal
            return false
        if A[n] - A[i] == n - i  // same minor diagonal
            return false
    return true


enumerate(n)
    A // size of n
    enumerate(A, 0)


enumerate(A, k)
    n = A.length
    if k == n
        Out.push(A)
        return
    for i in [0:n)
        A[k] = i
        if is_consistent(A, k)
            enumerate(A, k + 1)
```

Sudoku

- Board - 9 x 9 board of values with 0's in unspecified spots

```
solve(Board)
    return solve(Board, 0, 0)


solve(Board, i, j)
    if i == Board.length and j + 1 == Board[i].length  // reached solution
        return true
    if i == Board.length  // move to next row
        i = 0
        ++j
    if Board[i][j] != 0
        return solve(Board, i + 1, j)
    for val in [1:Board.size]  // closed-interval, includes Board.size
        if is_valid(Board, i, j, val)
            Board[i][j] = val
            if solve(Board, i + 1, j)
                return true
    Board[i][j] = 0
    return false


is_valid(Board, row, column, val)
    for A in Board  // check same column
        if A[column] == val
            return false
    for e in Board[row]  // check same row
        if e == val
            return false
    r_sz = 3
    for a in [0:r_sz)
        for b in [0:r_sz)
            if Board[r_sz * (row / r_sz) + a][r_sz * (column / r_sz) + b] == val
```

```
            return false
    return true
```

Lexicographical Compare

```
compare(A, B)
    i = 0
    j = 0
    while i < A.length && j < B.length
        if A[i] < B[j]
            return true
        if B[j] < A[i]
            return false
        ++i
        ++j
    return i == A.length and j < B.length
```

Horner's Method

```
horners_method(Coefficients, x)
    result = 0
    for i = Coefficients.length - 1; i >= 0; --i
        result = result * x + Coefficients[i]
    return result
```

Matrix Operations

Matrix x Matrix

```
multiply(A, B)
    m1 = A.length
```

```
n1 = A[0].length
m2 = b.length
n2 = b[0].length
if n1 != m2
    throw
C // m1 x n2
for i in [0:m1)
    for j in [0:n2)
        for k in [0:n1)
            C[i][j] += A[i][k] * B[k][j]
return C
```