Pseudocode examples

BFS

Queue implementation:

Algorithm 1: BFS(G,s)

```
/* G is hash table, the adjacency list of a graph                        */
/* s is a source vertex in G                                              */
parents ← {}/* empty hash table, parents[v] = v's parent.                 */
dist ← {}/* empty hash table, dist[v] = distance from s.                  */
Q ← empty FIFO queue/* keep track of active nodes                         */
Q.enqueue(s), parents[s] = None, dist[s] = 0 /* initialization            */
while Q is not empty do
    u ← Q.dequeue();
    for v in G[u] do
        /* explore neighbors of active node u                             */
        if v not in parents then
            /* v was so far undiscovered                                  */
            parents[v] = u;
            dist[v] = dist[u] + 1;
            Q.enqueue(v);
return parents, dist
```

Layers implementation:

Algorithm 2: BFS(G,s)

```
/* G is hash table, the adjacency list of a graph */
/* s is a source vertex in G */
parents ← hash table /* parents[v] = v's parent in BFS tree. */
layers ← hash table /* layers[i] = nodes in layer i */
dist ← hash table/* empty hash table, dist[v] = distance from s. */
layers[0] ← s, parents[s] = None, dist[s] = 0, i ← 0 /* initialization */
while layers[i] is not empty do
    layers[i + 1] ← [ ];
    for u in layers[i] do
        for v in G[u] do
            /* explore neighbors of active node u */
            if v not in parents then
                /* v was so far undiscovered */
                parents[v] = u;
                dist[v] = dist[u] + 1;
                layers[i + 1].add(v)/* v is at layer i+1 */
i ← i + 1;
return parents, dist, layers (optional)
```

Algorithm 3: bipartiteBFS(G)

```
/* G is hash table, the adjacency list of a graph */
s ← choose a vertex at random from V/* serves as the source */
parents, dist = BFS(G, s);
C1, C2 ← [ ]/* empty lists, contains the vertices of the two classes in the
    bipartition */
for v in dist do
    /* assign nodes to C1, C2 based on odd/even layers */
    if dist[v] == odd then
        C1.add(v);
    else
        C2.add(v);
return C1, C2
```

DFS

Algorithm 4: DFSWrapper(G, s)

```
/* G adjacency list as nested hash table, s source node            */
parents ← empty hash table/* parents of nodes in the DFS tree (forest)     */
times ← empty hash table /* tuples of <discovery time, finish time>     */
time ← 0/* time counter                                                 */
parents[s] ← None;
DFS(G, s);
return parents, times
```

Algorithm 5: DFS(G, u)

```
time ← time + 1;
times[u][0] = time;
for v in G[u] do
    /* recursively explore u's neighbors                                */
    if v not in parents then
        parents[v] = u;
        DFS(G,v)
time ← time + 1;
times[u][1] = time;
```

Dijkstra

Algorithm 6: Dijkstra(G, s)

```
/* G adjacency list fo weighted directed graph.  G[u][v] = l(u,v), source s */
π ← { }/* hash table, current best list for v */
d ← { }/* hash table, distance of v */
parents ← { }/* parents in shortest paths tree */
Q ←PQ/* priority queue to keep track of min π */
π[s] = 0;
Q.INSERT(< 0, s >);
for v ≠ s in G do
    π[v] = ∞;
    Q.INSERT(< π[v], v >);
while Q is not empty do
    < π[u], u >← EXTRACT-MIN(Q);
    d[u] ← π[u];
    for v in G[u] do
        if π[v] > d[u] + l(u, v) then
            DECREASE-KEY(< π[v], v >, < d[u] + l(u, v), v >);
            π[v] ← d[u] + l(u, v);
            parents[v] ← u;
return d, parents
```

Prim

Algorithm 7: Prim(G)

```
/* G adjacency list of weighted undirected graph G[u][v] = w(u,v) */
parents ← hash table /* parents list in MST, contains current best edge */
wT ← 0 /* total weight of MST, optional */
Tnodes ← empty set /* keep track of nodes fixed in MST */
Q ← empty priority queue /* <weight, v> least edge weight connecting v to MST */
s ← random node as source;
parents[s] ← None, Q.INSERT(< 0, s >);
for v ≠ s in G do
    Q.INSERT(< ∞, v) /* use ∞ for unreachable nodes. Can use really large int instead. */
    parents[v] ← None;
while Q is not empty do
    < weight, u >← Q.EXTRACT − MIN( ) /* next lightest edge */
    wT+ = weight;
    Tnodes.add(u) /* u us added to the MST and is fixed */
    for v in G[u] do
        if v not in Tnodes AND G[v][parents[v]] > G[v][u] then
            Q.DECREASE − KEY(< G[v][parent[v]], v >, < G[v][u], v >);
            parents[v] = u;
return parents, wT
```

Divide and Conquer

Algorithm 8: MergeSort(A, p, r)

```
/* Sorts the subarray A[p:r] in place */
if p == r then
    return A;
q ← ⌊(p+r)/2⌋;
A[p : q] ← MergeSort(A, p, q);
A[q + 1 : r] ← MergeSort(A, q + 1, r);
A[p : r] ← Merge(A, p, q, r);
return A
```

Algorithm 9: BinarySearch($A, p, r, query$)

```
/* find the index of query in the subarray A[p:r] */
q ← ⌊(p+r)/2⌋;
middle ← A[q];
if query == middle then
    return q;
else if query < middle then
    BinarySearch(A, p, q, query);
else
    BinarySearch(A, q+1, r, query);
return query not in A
```

Algorithm 10: Karatsuba(a, b, n)

```
/* multiply n-bit ints a and b */
if n == 1 then
    return ab;
m ← ⌊n/2⌋;
A_1 ← ⌊a/2^m⌋ /* high n/2 bits of a and b */
B_1 ← ⌊b/2^m⌋ /* implemented by bit-shift */
A_0 ← a mod 2^m /* lower n/2 bits of a and b */
B_0 ← b mod 2^m;
x ← Karatsuba(A_1, B_1, m);
y ← Karatsuba(A_0, B_0, m);
z ← Karatsuba(A_1 + A_0, B_1 + B_0, m);
return x^{2m} + (z − x − y)2^m + y;
```

DP algorithms

Algorithm 11: SubsetSum($w_1, w_2, \ldots, w_n, W$)

```
M ← (n+1) × (W+1) /* table (matrix/2D array) */
for w = 0...W do
    /* set border cases */
    M[0][w] = 0;
for j = 1...n do
    for w = 0...W do
        /* Apply recursive formula */
        M[j][w] = max{w_j + M[j−1][w−w_j]; M[j−1][w]};
return M[n][W]
```

Algorithm 12: SubsetSumSolution($M, w = [w_1, w_2, \ldots, w_n], W$)

$S \leftarrow [\,]$ /* set of opt weights */

while $i > 0$ *AND* $j > 0$ **do**

  if $M[i][j] > M[i-1][j]$ **then**

    /* the case where w_i is chosen */

    $S.append(w[i])$;

    $i \leftarrow i - 1$;

    $j \leftarrow j - w$;

  else

    /* w_i is not chosen */

    $i \leftarrow i - 1$;

return S