Algorithms in Pseude-Code

COMP2721 Algorithms and Data Structures II

Session 2024

```
input  : a graph G = (V, E)
output: a BFS-forest T = (V, F) of G and a BFS-numbering σ : V → ℕ
1  begin
2      i ← 1; F ← ∅; Q ← emptyQueue;
3      for v ∈ V do mark v unvisited;
4      while there is a unvisited vertex v ∈ V do
5          σ(v) ← i; i ← i + 1;
6          Q ← enqueue(v, Q); mark v visited;
7          while Q is not empty do
8              v ← front(Q); Q ← dequeue(Q);                    /* remove v from Q */
9              for w ∈ N(v) do
10                 if w is unvisited then
11                     F ← F ∪ {{v, w}};
12                     σ(w) ← i; i ← i + 1;
13                     Q ← enqueue(w, Q); mark w visited
```

Algorithm 1: traditional Breadth-First Search

```
input  : a graph G = (V, E)
output: a BFS-forest T = (V, F) of G and a BFS-numbering σ : V → ℕ
1  begin
2      i ← 1; F ← ∅; Q ← emptyQueue;
3      for v ∈ V do mark v unvisited;
4      while there is a unvisited vertex v ∈ V do
5          σ(v) ← i; i ← i + 1;
6          Q ← enqueue(v, Q); mark v visited;
7          while Q is not empty do
8              v ← front(Q);        /* inspect first element without removing it */
9              if v has an unvisited neighbour w then
10                 F ← F ∪ {{v, w}};
11                 σ(w) ← i; i ← i + 1;
12                 Q ← enqueue(w, Q); mark w visited
13             else
14                 Q ← dequeue(Q);                          /* remove first element */
```

Algorithm 2: Breadth-First Search similar to Depth-First Search

```
input  : a graph G = (V, E)
output: a DFS-forest T = (V, F) of G and a DFS-numbering σ : V → ℕ
1  begin
2      i ← 1; F ← ∅; S ← emptyStack;
3      for v ∈ V do mark v unvisited;
4      while there is a unvisited vertex v ∈ V do
5          σ(v) ← i; i ← i + 1;
6          S ← push(v, S); mark v visited;
7          while S is not empty do
8              v ← peek(S);            /* inspect top-element without removing it */
9              if v has an unvisited neighbour w then
10                 F ← F ∪ {{v, w}};
11                 σ(w) ← i; i ← i + 1;
12                 S ← push(w, S); mark w visited
13             else
14                 S ← pop(S);                          /* remove top-element */
```

Algorithm 3: Depth-First Search with stack

```
input  : a graph G = (V, E)
output: a DFS-forest T = (V, F) of G and a DFS-numbering σ : V → ℕ
1  begin
2      i ← 1; F ← ∅;
3      for v ∈ V do mark v unvisited;
4      while there is a unvisited vertex v ∈ V do DFS-visit(v);
5  procedure DFS-visit(v)
6      mark v visited;
7      σ(v) ← i; i ← i + 1;
8      for w ∈ N(v) do
9          if w is unvisited then
10             F ← F ∪ {{v, w}};
11             DFS-visit(w)
```

Algorithm 4: recursive Depth-First Search

```
input  : a directed acyclic graph (dag) G = (V, A)
output: a topological sort σ : V → ℕ of G
1  begin
2      i ← |V|;
3      for v ∈ V do mark v unvisited;
4      while there is a unvisited vertex v ∈ V do TS-visit(v);
5  procedure TS-visit(v)
6      if v is marked stacked then stop;
7      if v is marked unvisited then
8          mark v stacked;
9          for w ∈ N⁺(v) do TS-visit(w);
10         mark v visited;
11         σ(v) ← i; i ← i − 1
```

Algorithm 5: Topological Sort

```
input  : a connected graph G = (V, E) with weights ℓ : E → ℕ
output: a minimum spanning tree T = (V, F) of G
1  begin
2      for v ∈ V do π(v) ← ∞;
3      choose an arbitrary vertex u ∈ V;
4      U ← {u}; F ← ∅;
5      while N(U) ≠ ∅ do
6          for v ∈ N(u) \ U do
7              if π(v) > ℓ({u, v}) then
8                  π(v) ← ℓ({u, v}); p(v) ← u
9          choose v ∈ V \ U with minimal π(v);
10         U ← U ∪ {v}; F ← F ∪ {{p(v), v}}; u ← v
```

Algorithm 6: Minimum Spanning Tree (Prim)

```
input  : a connected graph G = (V, E) with weights ℓ : E → ℕ
output: a minimum spanning tree T = (V, F) of G
1  begin
2      sort the edges e ∈ E by their weights ℓ(e);
3      F ← ∅;
4      for e ∈ E in non-decreasing order do
5          if (V, F ∪ {e}) is acyclic then F ← F ∪ {e};
```

Algorithm 7: Minimum Spanning Tree (Kruskal)

input : a graph $G = (V, E)$ with weights $\ell : E \to \mathbb{N}$ and a source $s \in V$
output: the distances $d(s, v)$ for all $v \in V$ and
a shortest path tree $T = (V, F)$ of G with source s

```
1  begin
2      for v ∈ V do π(v) ← ∞;
3      d(s, s) ← 0; π(s) ← 0; U ← {s}; u ← s; F ← ∅;
4      while N(U) ≠ ∅ do
5          for v ∈ N(u) \ U do
6              if π(v) > π(u) + ℓ({u, v}) then
7                  π(v) ← π(u) + ℓ({u, v}); p(v) ← u
8          choose v ∈ V \ U with minimal π(v);
9          d(s, v) ← π(v);
10         U ← U ∪ {v}; F ← F ∪ {{p(v), v}}; u ← v
```

Algorithm 8: Single-Source Shortest Path (DIJKSTRA)

input : a directed graph $G = (V, E)$, a root $r \in V$, and weights $w : E \to \mathbb{N}$
output: $F \subseteq E$ such that (V, F) is a minimum arborescense of G rooted at r, if there is one

```
1  procedure minimumArborescence(G, r, w)
2      F_0 ← ∅; w(r) ← 0;
3      for v ∈ V \ {r} do
4          w(v) ← ∞;
5          for u ∈ N_G^-(v) do
6              if w(u, v) < w(v) then w(v) ← w(u, v); x ← u;
7          if w(v) = ∞ then exit;
8          else F_0 ← F_0 ∪ {(x, v)};
9      if (V, F_0) has a cycle (U, C) then
10         shrink; F_1 ← minimumArborescence(G', r, w'); expand
11     else F ← F_0;
12     return F
13 procedure shrink
14     V' ← {u} ∪ V \ U; E' ← ∅;
15     for v ∈ V do w'(u, v) ← ∞; w'(v, u) ← ∞;
16     for xv ∈ E do
17         if v ∈ V \ U then
18             if x ∈ V \ U then
19                 E' ← E' ∪ {(x, v)}; w'(x, v) ← w(x, v) − w(v); p(x, v) ← xv
20             else
21                 E' ← E' ∪ {(u, v)};
22                 if w(x, v) − w(v) < w'(u, v) then
23                     w'(u, v) ← w(x, v) − w(v); p(u, v) ← (x, v)
24         if x ∈ V \ U then
25             E' ← E' ∪ {(x, u)};
26             if w(x, v) − w(v) < w'(xu) then
27                 w'(x, u) ← w(x, v) − w(v); p(x, u) ← (x, v)
28     G' ← (V', E')
29 procedure expand
30     F ← C;
31     for xv ∈ F_1 do
32         F ← F ∪ {p(x, v)};
33         if v = u then y ← x;
34     for x ∈ U do
35         if (x, y) ∈ C then F ← F \ {(x, y)};
```

Algorithm 9: Minimum Arborescence

```
input : an integer n > 0 (number of matrices A_i to multiply) and an array d[0..n]
        (sizes, A_i is a d[i-1] × d[i] matrix)
output: N[1,n] the minimum number of scalar multiplications
1  begin
2      for i ← 1 to n do N[i,i] ← 0;
3      for b ← 1 to n − 1 do
4          for i ← 1 to n − b do
5              k ← i + b; m ← d[i − 1] ∗ d[k]; N[i,k] ← ∞;
6              for j ← i to k − 1 do
7                  N[i,k] ← min(N[i,k], N[i,j] + m ∗ d[j] + N[j + 1,k])
```

Algorithm 10: Matrix Chain-Product

```
input : a directed graph G = (V,E) with weights ℓ : E → ℤ such that G has no cycle of
        negative length, w.l.o.g. V = {1,2,...,n}
output: the distances d_n(u,v) for all u,v ∈ V
1  begin
2      for u ∈ V do
3          for v ∈ V do
4              if u = v then
5                  d_0(u,v) ← 0
6              else
7                  if uv ∈ E then
8                      d_0(u,v) ← ℓ({u,v})
9                  else
10                     d_0(u,v) ← ∞
11     for k ← 1 to n do
12         for u ∈ V do
13             for v ∈ V do
14                 d_k(u,v) ← min(d_{k−1}(u,v), d_{k−1}(u,k) + d_{k−1}(k,v))
```

Algorithm 11: All Pair Shortest Path (Floyd/Warshall)

fixed : A CFG $G = \langle T, V, P, S\rangle$ in Chomsky normal form is not part of the input.
input : a string $x_1x_2\ldots x_n \in T^*$
output: a boolean value indicating whether $x_1x_2\ldots x_n$ can be generated by G

```
1  begin
2      for i ← 1 to n do
3          V(i,i) ← {A ∈ V | (A → x_i) ∈ P}
4      for b ← 1 to n − 1 do
5          for i ← 1 to n − b do
6              k ← i + b; V(i,k) ← ∅;
7              for j ← i to k − 1 do
8                  for (A → BC) ∈ P do
9                      if B ∈ V(i,j) and C ∈ V(j+1,k) then V(i,k) ← V(i,k) ∪ {A} ;
10     if S ∈ V(1,n) then accept x_1x_2...x_n else reject x_1x_2...x_n;
```

Algorithm 12: CYK parsing of context-free languages (Cocke/Younger/Kasami)

input : an integer $n > 2$ (number of cities) and a 2-dimensional array $d[1..n, 1..n]$ of pairwise distances
output: a permutation $\pi : \{1,\ldots,n\} \to \{1,\ldots,n\}$ minimising $d[\pi(n),\pi(1)] + \sum_{i=2}^{n} d[\pi(i-1),\pi(i)]$

```
1  begin
2      for c ∈ {2,...,n} do ℓ({1,c},c) ← d[1,c]; p({1,c},c) ← 1 ;
3      for k ← 2 to n − 1 do
4          for S' ∈ C(n−1, k) do
5              for c ∈ S' do
6                  S ← S' ∪ {1}; ℓ(S,c) ← ∞;
7                  for a ∈ S' \ {c} do
8                      ℓ' ← ℓ(S \ {c}, a) + d[a,c];
9                      if ℓ' < ℓ(S,c) then ℓ(S,c) ← ℓ'; p(S,c) ← a ;
10     ℓ'' ← ∞;
11     for a ∈ {2,...,n} do
12         ℓ' ← ℓ({1,...,n}, a) + d[a,1];
13         if ℓ' < ℓ'' then ℓ'' ← ℓ'; π(n) ← a;
14     S ← {1,...,n};
15     for i ← n downto 2 do
16         π(i−1) ← p(S, π(i)); S ← S \ {π(i)}
```

Algorithm 13: Travelling Sales Person (Bellman/Held/Karp)

```
input : an array A[1..n] containing a sequence of n numbers (a_1, a_2, ..., a_n)
output: the array A containing a permutation (a'_1, a'_2, ..., a'_n) of the input sequence
        such that a'_1 ⩽ a'_2 ⩽ ... ⩽ a'_n
1  begin
2      quicksort(A, 1, n)
3  procedure quicksort(A, p, r)
4      if p < r then
5          q ← partition(A, p, r);
6          quicksort(A, p, q − 1); quicksort(A, q + 1, r)
7  function partition(A, p, r)
8      x ← A[p]; i ← p; j ← r; k ← r;
9      while i < k do
10         if A[j] ⩽ x then
11             A[i] ← A[j]; i ← i + 1; j ← i
12         else
13             A[k] ← A[j]; k ← k − 1; j ← k
14     A[j] ← x; return j
```

Algorithm 14: quicksort

```
input : an array A[1..n] containing a sequence of n numbers (a_1, a_2, ..., a_n)
output: the array A containing a permutation (a'_1, a'_2, ..., a'_n) of the input sequence
        such that a'_1 ⩽ a'_2 ⩽ ... ⩽ a'_n
1  begin
2      mergesort(A, 1, n)
3  procedure mergesort(A, p, r)
4      if p < r then
5          q ← ⌊(p + r)/2⌋;
6          mergesort(A, p, q); mergesort(A, q + 1, r);
7          merge(A, p, q, r)
8  procedure merge(A, p, q, r)
9      n_L ← q − p + 1; n_R ← r − q;
10     create arrays L[1..n_L + 1] and R[1..n_R + 1];
11     for i ← 1 to n_L do L[i] ← A[p + i − 1];
12     for j ← 1 to n_R do R[j] ← A[q + j];
13     L[n_L + 1] ← ∞; R[n_R + 1] ← ∞;
14     i ← 1; j ← 1;
15     for k ← p to r do
16         if L[i] ⩽ R[j] then A[k] ← L[i]; i ← i + 1 ;
17         else A[k] ← R[j]; j ← j + 1
```

Algorithm 15: mergesort

```
1  function empty
2      allocate memory for an array h[0..maxHeapSize];
3      return (0,h)
4  function isEmpty(n, h)
5      if n = 0 then
6          return true
7      else
8          return false
9  function minimum(n, h)
10     return h[1]
11 function swap((n, h), i, j)
12     t ← h[i]; h[i] ← h[j]; h[j] ← t;
13     return (n, h)
14 function toggleUp((n, h), i)
15     j ← ⌊i/2⌋;
16     if i > 1 and h[j] > h[i] then
17         return toggleUp(swap((n, h), i, j), j)
18     else
19         return (n, h)
20 function toggleDown((n, h), i)
21     j ← 2 ∗ i;
22     if j + 1 ⩽ n and h[j + 1] < h[j] then j ← j + 1;
23     if j ⩽ n and h[j] < h[i] then
24         return toggleDown(swap((n, h), i, j), j)
25     else
26         return (n, h)
27 function insert((n, h), t)
28     h[n] ← t; n ← n + 1;
29     return toggleUp((h, n), n)
30 function deleteMin((n, h))
31     h[1] ← h[n]; n ← n − 1;
32     return toggleDown((h, n), 1)
```

Algorithm 16: binary heap

```
1  function empty
2      return nil
3  function isEmpty(h)
4      if h = nil then
5          return true
6      else
7          return false
8  function minimum(h)
9      m ← ∞;
10     while h ≠ nil do
11         if h.key < m then m ← h.key; i ← h.info;
12         h ← h.sibling
13     return (m, i)
14 function link(s, t)
15     if s.key < t.key then
16         t.sibling ← s.child;
17         s.child ← t;
18         s.degree ← s.degree + 1;
19         return s
20     else
21         s.sibling ← t.child;
22         t.child ← s;
23         t.degree ← t.degree + 1;
24         return t
25 function merge(g, h)
26     if isEmpty(g) then return h;
27     if isEmpty(h) then return g;
28     if g.degree < h.degree then
29         g.sibling ← merge(g.sibling, h);
30         return g
31     if g.degree > h.degree then
32         h.sibling ← merge(g, h.sibling);
33         return h
34     x ← merge(g.sibling, h.sibling);
35     y ← link(g, h); y.sibling ← nil;
36     return merge(x, y)
```

Algorithm 17: binomial heap

```
1  function insert((k, i), h)
2      allocate a new cell at t;
3      t.degree ← 0;
4      t.key ← k;
5      t.info ← i;
6      t.child ← nil;
7      t.sibling ← nil;
8      return merge(t, h)
9  function deleteMin(h)
10     (m, i) ← minimum(h);
11     if h.key = m then
12         x ← h.sibling;
13         t ← h;
14         t.sibling ← nil
15     else
16         x ← h; z ← h;
17         while z.sibling.key ≠ m do z ← z.sibling;
18         t ← z.sibling;
19         z.sibling ← z.sibling.sibling
20     s ← t.child;
21     y ← nil;
22     while s.sibling ≠ nil do
23         z ← s;
24         z.sibling ← y;
25         y ← z;
26         s ← s.sibling
27     deallocate the cell at t;
28     return merge(x, y)
```

Algorithm 18: binomial heap—continued

```
   input  : integers n ⩾ m ⩾ 0 and arrays T[1..n] (text) and P[1..m] (pattern)
   output: the set S ⊆ {0, 1, ..., n − m} of valid shifts
1  begin
2      S ← ∅;
3      for s ← 0 to n − m do
4          found ← true;
5          for i ← 1 to m do
6              found ← found ∧ (P[i] = T[s + i])
7          if found then S ← S ∪ {s};
```

Algorithm 19: String Matching—Naive Algorithm

```
input  : integers n ⩾ m ⩾ 0 and arrays T[1..n] (text) and P[1..m] (pattern)
output: the set S ⊆ {0, 1, ..., n − m} of valid shifts
1  begin
2      computeTransitionFunction;
3      S ← ∅; q ← 0;
4      for i ← 1 to n do
5          q ← δ(q, T[i]);
6          if q = m then S ← S ∪ {i − m};
7  procedure computeTransitionFunction
8      for q ← 0 to m do
9          for a ∈ Σ do
10             k ← min{m, q + 1};
11             while P_k ⋣ P_q a do k ← k − 1;
12             δ(q, a) ← k
```

Algorithm 20: String Matching—DFA Matcher

```
input  : integers n ⩾ m ⩾ 0 and arrays T[1..n] (text) and P[1..m] (pattern)
output: the set S ⊆ {0, 1, ..., n − m} of valid shifts
1  begin
2      computePrefixFunction;
3      S ← ∅; q ← 0;
4      for i ← 1 to n do
5          while q > 0 and P[q + 1] ≠ T[i] do q ← π(q);
6          if P[q + 1] = T[i] then q ← q + 1;
7          if q = m then S ← S ∪ {i − m}; q ← π(q);
8  procedure computePrefixFunction
9      π[1] ← 0; k ← 0;
10     for q ← 2 to m do
11         while k > 0 and P[k + 1] ≠ P[q] do k ← π[k];
12         if P[k + 1] = P[q] then k ← k + 1;
13         π[q] ← k
```

Algorithm 21: Knuth/Morris/Pratt Algorithm