Algorithms
(Algorithm Code)

Pramod Ganapathi
Department of Computer Science
State University of New York at Stony Brook

August 27, 2024

Contents

What is an algorithm code?

- An algorithm code is also called a pseudocode
- An algorithm code represents an algorithm in a structured, modular, step-by-step format
- An algorithm code is used to show the actual working logic of an algorithm without including unnecessary keywords and information contained in a program code that is not relevant to the problem-solving logic
- An algorithm code is programming-language neutral, i.e., programmers can easily understand an algorithm code irrespective of their favorite programming language

Assignments

```
ALGORITHMCODE()
```

ALGORITHMCODE()

// Assignments . . . .

$a \leftarrow 1$; $a \leftarrow \left(\frac{\lfloor d \rfloor + \lceil e \rceil}{\sqrt{n} + \log_3 k} + a^b \bmod c \right)$ // comma-separated statements

$currenpathcost \leftarrow 0$ // variable names have all small letters

Static arrays

ALGORITHMCODE()

// Static array: Creation and assignment ....

Create an array $A[1 \dots n]$

$A[i] \leftarrow a$ // set ith value of array. time:$\Theta(1)$

$a \leftarrow A[i]$ // get ith value of array. time:$\Theta(1)$

Create an array $B[1 \dots n] \leftarrow [0 \dots 0]$ // $B[i] = 0$ for all $i \in [1, n]$

Create an array $C[1 \dots n] \leftarrow [1 \dots n]$ // $C[i] = i$ for all $i \in [1, n]$

Create an array $D[1 \dots n] \leftarrow B[1 \dots n]$ // $D[i] = B[i]$ for all $i \in [1, n]$

$a \leftarrow \text{Sum}(A[1 \dots n]) + \text{Max}(A[1 \dots n]) + \text{Min}(A[1 \dots n])$

Dynamic arrays

AlgorithmCode()

```
// Dynamic array: Creation and assignment ................................
Create a dynamic array optimalsolutionslist ← [ ]
Create a dynamic array A ← optimalsolutionslist
A.Add(a)          // add element at last. time:Θ(1)*
A[i] ← a          // set ith value of array. time:Θ(1)
a ← A[i]          // get ith value of array. time:Θ(1)
a ← A.First()     // return the first element. time:Θ(1)
a ← A.Last()      // return the last element. time:Θ(1)
```

2-D matrices

```
ALGORITHMCODE()
// 2-D matrix ..................................................
Create a 2-D matrix M[1...m][1...n]
M[i][j] ← a                      // set (i, j)th value of matrix. time:Θ(1)
a ← M[i][j]                      // get (i, j)th value of matrix. time:Θ(1)
```

Conditionals

ALGORITHMCODE()

```
// Conditional: If-else ladder ........................................
if a > 0 then
|   b ← 5
else if a = 0 then
|   b ← 0
|   c ← 0
else
|   b ← −5
```

Conditionals

```
AlgorithmCode()

// Conditional: Compact if-else ladder ....................................
if a > 0 then  b ← 5
else if a = 0 then  { b ← 0; c ← 0 }
else  b ← −5
```

Conditionals

```
AlgorithmCode()

// Conditional ..........................................................
a ← ( b = c ) ? d : e                  // d if condition true, e if condition false
```

Loops

```
AlgorithmCode()
  // Loop: i takes values a, a+1, a+2, so on such that i ≤ n ...............
  for i ← a to n do
  |   print A[i]
  // Loop: i takes values n, n−1, n−2, so on such that i ≥ a ..............
  for i ← n downto a do
  |   print A[i]
```

Loops

```
ALGORITHMCODE()
// Loop: i takes values a, a + k, a + 2k, so on such that i ≤ n .............
for i ← a to n increment k do
|   print A[i]
// Loop: i takes values n, n − k, n − 2k, so on such that i ≥ a .............
for i ← n downto a decrement k do
|   print A[i]
```

Loops

```
AlgorithmCode()
// Loop: i takes values n, n − k, n − 2k, so on such that i ≥ a .............
i ← n  while i ≥ a do
|   i ← i − k
```

Function invocations

```
ALGORITHMCODE()
```

// Function invocation ....

$output \leftarrow$ AnotherAlgorithm$(a, A[1 \ldots n])$
return *output*

AnotherAlgorithm$(a, A[1 \ldots n])$

return $A[a]$

Singly linked lists

```
ALGORITHMCODE()

// Singly linked list (SLL) ................................................
Create a SinglyLinkedList L
a ← L.First()            // return the first element. time:Θ(1)
a ← L.Last()             // return the last element. time:Θ(1)
L.AddFirst(a)            // add element at first. time:Θ(1)
L.AddLast(a)             // add element at last. time:Θ(1)
a ← L.RemoveFirst()      // remove element at first. time:Θ(1)
a ← L.RemoveLast()       // remove element at last. time:Θ(n)
```

Circularly singly linked lists

```
AlgorithmCode()
// Circularly singly linked list (CSLL) ....................................
Create a CircularlySinglyLinkedList L
a ← L.First()                       // return the first element. time:Θ(1)
a ← L.Last()                        // return the last element. time:Θ(1)
L.AddFirst(a)                       // add element at first. time:Θ(1)
L.AddLast(a)                        // add element at last. time:Θ(1)
a ← L.RemoveFirst()                 // remove element at first. time:Θ(1)
a ← L.RemoveLast()                  // remove element at last. time:Θ(n)
L.Rotate()                          // move first element to last. time:Θ(1)
```

Doubly linked lists

```
ALGORITHMCODE()
// Doubly linked list (DLL) ............................................
Create a DoublyLinkedList L
a ← L.First()          // return the first element. time:Θ(1)
a ← L.Last()           // return the last element. time:Θ(1)
L.AddFirst(a)          // add element at first. time:Θ(1)
L.AddLast(a)           // add element at last. time:Θ(1)
a ← L.RemoveFirst()    // remove element at first. time:Θ(1)
a ← L.RemoveLast()     // remove element at last. time:Θ(1)
```

Stacks

ALGORITHMCODE()

```
// Stack (implemented using dynamic array or SLL) ........................
Create a stack S
S.Push(a)                          // add element at top. time:Θ(1)
a ← S.Pop()                     // remove element at top. time:Θ(1)
a ← S.Top()                     // return element at top. time:Θ(1)
```

Queues

```
AlgorithmCode()
  // Queue (implemented using a circular dynamic array or SLL) .............
  Create a queue Q
  Q.Enqueue(a)                          // add element at last. time:Θ(1)
  a ← Q.Dequeue()                  // remove element at first. time:Θ(1)
  a ← Q.Top()                         // return element at top. time:Θ(1)
```

Deques

```
AlgorithmCode()

// Double-ended queue (deque) ........................................
Create a deque D
D.AddFirst(a)                       // add element at first. time:Θ(1)
D.AddLast(a)                        // add element at last. time:Θ(1)
a ← D.RemoveFirst()                 // remove element at first. time:Θ(1)
a ← D.RemoveLast()                  // remove element at last. time:Θ(1)
```

Balanced search trees

```
ALGORITHMCODE()
// Balanced search tree with size n ......................................
Create a BalancedSearchTree T
T.Add(a)                                // add element. time:O(log n)
T.Remove(a)                             // remove element. time:O(log n)
a ← T.Search(a)                         // check if element exists. time:O(log n)
T.InOrderTraversal()                    // inorder traversal. time:Θ(n)
T.LevelOrderTraversal()                 // levelorder traversal. time:Θ(n)
```

Hash sets

```
AlgorithmCode()
// Hash set (assuming perfect hash function) ..............................
Create a hash set H to elements in unordered/unsorted fashion
H.Add(a)                                  // add element. time:O(1)*
H.Remove(a)                               // remove element. time:O(1)*
a ← H.Search(a)                           // check if element exists. time:O(1)*
```

Hash maps

AlgorithmCode()

```
// Hash map (assuming perfect hash function) ..............................
Create a hash map H to store unordered/unsorted (key, value) pairs
a ← H.GetValue(k)          // return the value for key k. time:O(1)*
a ← H[k]                   // return the value for key k. time:O(1)*
H.Add(⟨k, v⟩)              // add a pair. time:O(1)*
H[k] ← v                   // add a pair. time:O(1)*
⟨k, v⟩ ← H.Remove(k)       // return the pair with key k. time:O(1)*
⟨k, v⟩ ← H.Search(k)       // search for pair with key k. time:O(1)*
```

Priority queues

```
AlgorithmCode()

// Minimum-heap with size n ..............................................
Create a min-heap H to store (key, value) pairs
H.Add(⟨k, v⟩)                                  // add a pair. time:O(log n)
⟨k, v⟩ ← H.Min()                     // return pair with min key. time:Θ(1)
⟨k, v⟩ ← H.RemoveMin()           // remove pair with min key. time:O(log n)
```