THE PUBLIC IS MORE FAMILIAR WITH BAD DESIGN THAN GOOD DESIGN. IT IS, IN EFFECT, CONDITIONED TO PREFER BAD DESIGN, BECAUSE THAT IS WHAT IT LIVES WITH. THE NEW BECOMES THREATENING, THE OLD REASSURING.

PAUL RAND

A DESIGNER KNOWS THAT HE HAS ACHIEVED PERFECTION NOT WHEN THERE IS NOTHING LEFT TO ADD, BUT WHEN THERE IS NOTHING LEFT TO TAKE AWAY.

ANTOINE DE SAINT-EXUPÉRY

. . . THE DESIGNER OF A NEW SYSTEM MUST NOT ONLY BE THE IMPLEMENTOR AND THE FIRST LARGE-SCALE USER; THE DESIGNER SHOULD ALSO WRITE THE FIRST USER MANUAL. . . IF I HAD NOT PARTICIPATED FULLY IN ALL THESE ACTIVITIES, LITERALLY HUNDREDS OF IMPROVEMENTS WOULD NEVER HAVE BEEN MADE, BECAUSE I WOULD NEVER HAVE THOUGHT OF THEM OR PERCEIVED WHY THEY WERE IMPORTANT.

DONALD E. KNUTH

HENGFENG WEI

COLLECTION OF ALGORITHMS PSEUDOCODE

ANT

PUBLISHED BY ANT

TUFTE-LATEX.GOOGLECODE.COM

First printing, May 2018

Contents

List of Algorithms

List of Figures

List of Tables

Dedicated to those who appreciate LaTeX

and the work of Edward R. Tufte and Donald E. Knuth.

Introduction

This is a collection of psuedocode for classic algorithms.

Basic Iterative and Recursive Algorithms

Algorithm 1 Horner rule for polynomial evaluation.

```
procedure HORNER(A[0...n], x)                ▷ A : {a_0 ... a_n}
    p ← A[n]
    for i ← n − 1 downto 0 do
        p ← px + A[i]
    return p
```

Algorithm 2 Integer Multiplication.

```
procedure INT-MULT(y, z)                     ▷ y, z ≥ 0; y, z ∈ ℤ
    if z = 0 then
        return 0
    return INT-MULT(cy, ⌊z/c⌋) + y(z mod c)
```

Sorting

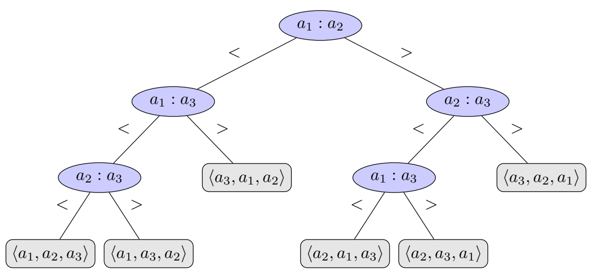

Figure 1: 3-element sorting algorithm represented by a decision tree.

Algorithm 3 Bubblesort (Bubble the smallest element each time.).

procedure BUBBLESORT($A[1 \cdots n]$)
    for $i \leftarrow 1$ **to** $n-1$ **do**
        for $j \leftarrow n$ **downto** $i+1$ **do**
            if $A[j] < A[j-1]$ **then**
                SWAP($A[j], A[j-1]$)

Algorithm 4 Bubblesort (Bubble the largest element each time.).

procedure BUBBLESORT($A[1 \cdots n]$)
    for $i \leftarrow n$ **downto** 2 **do**
        for $j \leftarrow 1$ **to** $i-1$ **do**
            if $A[j] > A[j+1]$ **then**
                SWAP($A[j], A[j+1]$)

Algorithm 5 Bubblesort (An improved version; from wiki).

```
procedure BUBBLESORT(A[1 ··· n])
    repeat
        newn ← 0
        for i ← 1 to n − 1 do
            if A[i − 1] > A[i] then
                SWAP(A[i − 1], A[i])
                newn ← i
        n ← newn
    until n = 0
```

Algorithm 6 Selection Sort.

```
procedure SELECTION-SORT(A, n)
    for i ← 1 to n − 1 do
        for j ← i + 1 to n do
            if A[j] < A[i] then
                SWAP(A[j], A[i])
```

Algorithm 7 COUNTING-SORT in place in $O(n + k)$ time.

```
procedure COUNTING-SORT(A, k)
    TODO
```

Selection

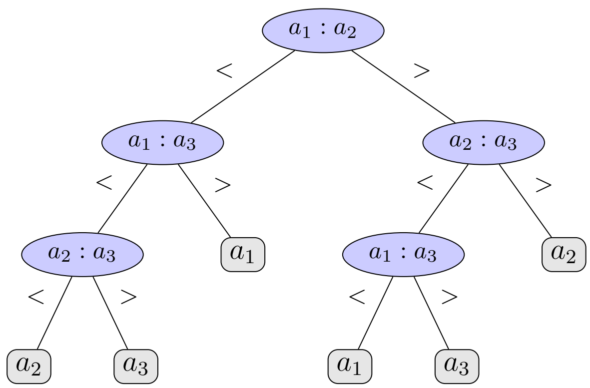

Figure 2: 3-element median selection algorithm represented by a decision tree.

Searching

Algorithm 8 (see Problem $5 - 2$ of [1]) searches for a value x in an unsorted array A consisting of n elements by checking $A[1], A[2], \cdots, A[n]$ in order until either it finds $A[i] = x$ or it reaches the end of the array.

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009. ISBN 0262033844, 9780262033848

Algorithm 8 Deterministic search.

```
procedure DETERMINISTIC-SEARCH(A[1···n], x)
    i ← 1
    while i ≤ n do
        if A[i] = x then
            return true
        i ← i + 1
    return false
```

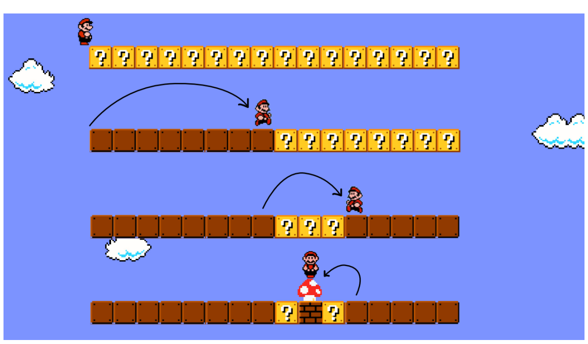

Figure 3: Binary stride.

Algorithm 9 Iterative binary search.

```
procedure BinarySearch(A[0 ··· n − 1], x)
    L ← 0
    R ← n − 1
    while L ≤ R do
        m ← L + (R − L)/2
        if A[m] < x then
            L ← m + 1
        else if A[m] > x then
            R ← m − 1
        else
            return m
    return −1
```

Algorithm 10 Recursive binary search.

```
procedure BinarySearch(A, L, R, x)
    if R < L then
        return −1
    m ← L + (R − L)/2
    if A[m] = x then
        return m
    else if A[m] > x then
        return BinarySearch(A, L, m − 1, x)
    else
        return BinarySearch(A, m + 1, R, x)
```

Dynamic Programming

Algorithm 11 Computing $\binom{n}{k}$ recursively.

```
procedure BINOM(n, k)                    ▷ Required: n ≥ k ≥ 0
    if k = 0 ∨ n = k then
        return 1
    return BINOM(n − 1, k) + BINOM(n − 1, k − 1)
```

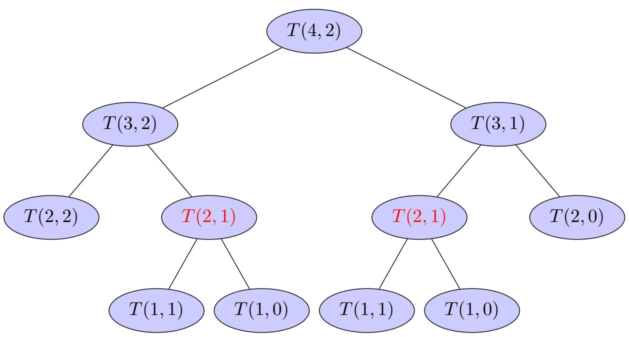

Figure 4: Calculate $\binom{4}{2}$ recursively.

$$\binom{0}{0}$$

$$\binom{1}{0} \quad \binom{1}{1}$$

$$\binom{2}{0} \quad \binom{2}{1} \quad \binom{2}{2}$$

$$\binom{3}{0} \quad \binom{3}{1} \quad \binom{3}{2} \quad \binom{3}{3}$$

$$\binom{4}{0} \quad \binom{4}{1} \quad \binom{4}{2} \quad \binom{4}{3} \quad \binom{4}{4}$$

$$\binom{5}{0} \quad \binom{5}{1} \quad \binom{5}{2} \quad \binom{5}{3} \quad \binom{5}{4} \quad \binom{5}{5}$$

Figure 5: Pascal triangle for binomial coefficients.

Algorithm 12 Computing $\binom{n}{k}$ by dynamic programming.

```
procedure BINOM(n, k)                          ▷ Required: n ≥ k ≥ 0
    for i ← 0 to n − k do
        B[i][0] ← 1
    for i ← 1 to k do
        B[i][i] ← 1
    for j ← 1 to k do
        for d ← 1 to n − k do
            i ← j + d
            B[i][j] ← B[i − 1][j] + B[i − 1][j − 1]
    return B[n][k]
```

$$(n-k+1)+(k)+k(n-k) = nk-k^2+n+1$$

Algorithm 13 Max-sum subarray.

```
procedure MSS(A[1 ··· n])
    MSS[0] ← 0
    for i ← 1 to n do
        MSS[i] ← max {MSS[i − 1] + A[i], 0}
    return max_{1≤i≤n} MSS[i]
```

Algorithm 14 Max-sum subarray (Implementation Simplified).

```
procedure MSS(A[1 ··· n])
    mss ← 0
    MSS ← 0
    for i ← 1 to n do
        MSS ← max {MSS + A[i], 0}
        mss ← max {mss, MSS}
    return mss
```

Traversal on Trees

DFS on Trees

Algorithm 15 Recursive DFS pre-order traversal on binary tree.

```
procedure RECURSIVE-DFS(t)
    print t.key

    if t.left ≠ NIL then
        RECURSIVE-DFS(t.left)
    if t.right ≠ NIL then
        RECURSIVE-DFS(t.right)

RECURSIVE-DFS(T.root)
```

Algorithm 16 Iterative DFS pre-order traversal on binary tree.

```
procedure ITERATIVE-DFS(t)
    S.PUSH(t)                                   ▷ S : stack

    while S ≠ ∅ do
        v ← S.POP()
        print v.key

        if v.right ≠ NIL then
            S.PUSH(v.right)
        if v.left ≠ NIL then
            S.PUSH(v.left)

ITERATIVE-DFS(T.root)
```

Algorithm 17 Recursive DFS traversal on a rooted tree stored using the "left-child, right-sibling" representation.

```
procedure Recursive-DFS(t)
    print t.key

    if t.left-child ≠ NIL then
        Recursive-DFS(t.left-child)
    if t.right-sibling ≠ NIL then
        Recursive-DFS(t.right-sibling)

Recursive-DFS(T.root)
```

Algorithm 18 Calculate the sum of depths of all nodes of a tree T.

```
procedure Sum-of-Depths()
    return Sum-of-Depths(T, 0)

procedure Sum-of-Depths(r, depth)                ▷ r: root of a tree
    if r is a leaf then
        return depth
    sum ← depth
    for all child vertex v of r do
        sum ← sum + Sum-of-Depths(v, depth + 1)
    return sum
```

Algorithm 19 Count the number of nodes in T at depth K.

```
procedure Nodes-at-Depth()
    return Nodes-at-Depth(T, K)

procedure Nodes-at-Depth(r, k)                   ▷ r: root of a tree
    if k = 0 then
        return 1
    if r is a leaf then
        return 0
    num ← 0
    for all child vertex v of r do
        num ← num + Nodes-at-Depth(v, k − 1)
    return num
```

Algorithm 20 Check whether a tree T has any leaf at an even depth.

```
procedure LEAF-AT-EVEN-DEPTH()
    return LEAF-AT-DEPTH(T, even = 0)

procedure LEAF-AT-DEPTH(r, parity)                ▷ r: root of a tree
    if r is a leaf then
        return 1 − parity
    result ← 0
    for all child vertex v of r do
        result ← result ∨ LEAF-AT-DEPTH(v, 1 − parity)
    return result
```

BFS on Trees

Algorithm 21 Calculate the sum of contents of nodes of a tree T at each depth.

```
procedure SUM-AT-DEPTH(r)                         ▷ r: root of the tree T
    r.depth ← 0
    Q ← ∅
    ENQUEUE(Q, r)
    while Q ≠ ∅ do
        u ← DEQUEUE(Q)
        sumAtDepth[u.depth] += u.content
        for all child vertex v of u do
            v.depth ← u.depth + 1
            ENQUEUE(Q, v)
```

Algorithm 22 Count the number of nodes of a tree T at each depth.

```
procedure NODES-AT-DEPTH(r)                    ▷ r: root of the tree T
    r.depth ← 0
    Q ← ∅
    ENQUEUE(Q, r)
    while Q ≠ ∅ do
        u ← DEQUEUE(Q)
        nodesAtDepth[u.depth] += 1
        for all child vertex v of u do
            v.depth ← u.depth + 1
            ENQUEUE(Q, v)
    return argmax_K nodesAtDepth[k]
```

Stack and Queue

Stack

Queue

Algorithm 23 Circular queue.

```
procedure ENQUEUE(Q, x)
    if Q.head = Q.tail + 1 then
        return "OVERFLOW"

    Q[Q.tail] = x

    if Q.tail = Q.length then
        Q.tail = 1
    else
        Q.tail = Q.tail + 1

procedure DEQUEUE(Q)
    if Q.head = Q.tail then
        return "UNDERFLOW"

    x = Q[Q.head]

    if Q.head = Q.length then
        Q.head = 1
    else
        Q.head = Q.head + 1

    return x
```

Stack and Queue

Algorithm 24 Simulating a queue using two stacks S_1, S_2.

```
procedure Enq(x)
    Push(S_1, x)

procedure Deq()
    if S_2 = ∅ then
        while S_1 ≠ ∅ do
            Push(S_2, Pop(S_1))

    Pop(S_2)
```

Bibliography

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009. ISBN 0262033844, 9780262033848.