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, April 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($A[0 \ldots n], x$) $\triangleright$ $A : \{a_0 \ldots a_n\}$
    $p \leftarrow A[n]$
    for $i \leftarrow n-1$ **downto** 0 **do**
        $p \leftarrow px + A[i]$
    return p

Algorithm 2 Integer Multiplication.

procedure INT-MULT(y, z) $\triangleright$ $y, z \geq 0;\ y, z \in \mathbb{Z}$
    if $z = 0$ **then**
        return 0
    return INT-MULT($cy, \lfloor \frac{z}{c} \rfloor$) $+ y(z \bmod c)$

Sorting

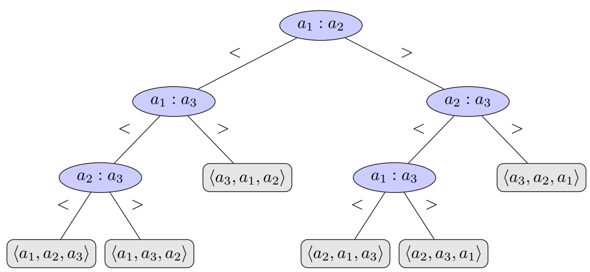

Figure 1: 3-element sorting algorithm represented by a decision tree.

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

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

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

1: **procedure** BUBBLESORT($A[1 \cdots n]$)
2:     **for** $i \leftarrow n$ **downto** 2 **do**
3:         **for** $j \leftarrow 1$ **to** $i-1$ **do**
4:             **if** $A[j] > A[j+1]$ **then**
5:                 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])
```

Selection

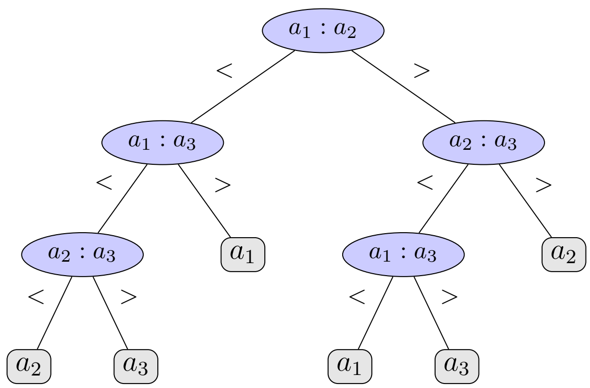

Figure 2: 3-element median selection algorithm represented by a decision tree.

Searching

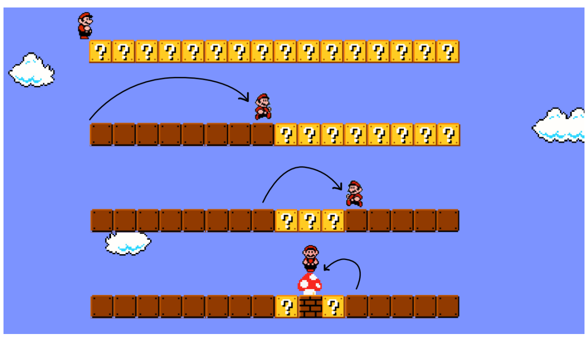

Figure 3: Binary stride.

Algorithm 7 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 8 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 9 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)
```

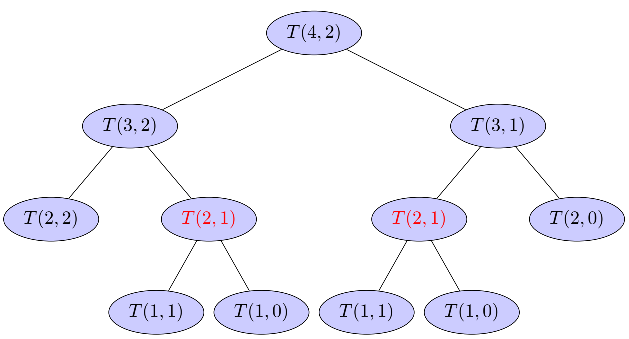

Figure 4: Calculate $\binom{4}{2}$ recursively.

$$
\begin{array}{ccccccccccc}
 & & & & & \binom{0}{0} & & & & & \\
 & & & & \binom{1}{0} & & \binom{1}{1} & & & & \\
 & & & \binom{2}{0} & & \binom{2}{1} & & \binom{2}{2} & & & \\
 & & \binom{3}{0} & & \binom{3}{1} & & \binom{3}{2} & & \binom{3}{3} & & \\
 & \binom{4}{0} & & \binom{4}{1} & & \binom{4}{2} & & \binom{4}{3} & & \binom{4}{4} & \\
\binom{5}{0} & & \binom{5}{1} & & \binom{5}{2} & & \binom{5}{3} & & \binom{5}{4} & & \binom{5}{5}
\end{array}
$$

Figure 5: Pascal triangle for binomial coefficients.

Algorithm 10 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 11 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 12 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 13 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 14 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 15 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 16 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 17 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]
```

Bibliography