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 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

COLLECTION OF ALGO-RITHMS PSEUDOCODE

Copyright © 2018 Hengfeng Wei PUBLISHED BY ANT TUFTE-LATEX.GOOGLECODE.COM Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at http://www.apache.org/licenses/ LICENSE-2.0. Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "As is" basis, without warranties or conditions of any kind, either express or implied. See the License for the specific language governing permissions and limitations under

the License.

First printing, May 2018

Contents

Basic Iterative and Recursive Algorithms
Sorting 19
Selection 23
Searching 25
Dynamic Programming 27
Traversal on Trees 29
Stack and Queue 33
Bibliography 35

17

List of Algorithms

1	Horner rule for polynomial evaluation. 17
2	Integer Multiplication. 17
3	Bubblesort (Bubble the smallest element each time.). 19
4	Bubblesort (Bubble the largest element each time.). 19
5	Bubblesort (An improved version; from wiki). 20
6	Hoare partition. 20
7	Selection Sort. 21
8	Counting-Sort in place in $O(n+k)$ time. 21
9	Deterministic search. 25
10	Iterative binary search. 26
11	Recursive binary search. 26
12	Computing $\binom{n}{k}$ recursively. 27
13	Computing $\binom{n}{k}$ by dynamic programming. 28
14	Max-sum subarray. 28
15	Max-sum subarray (Implementation Simplified). 28
16	Recursive DFS pre-order traversal on binary tree. 29
17	Iterative DFS pre-order traversal on binary tree. 29
18	Recursive DFS traversal on a rooted tree stored using the "left-child,
	right-sibling" representation. 30
19	Calculate the sum of depths of all nodes of a tree T . 30
20	Count the number of nodes in <i>T</i> at depth <i>K</i> . 30
21	Check whether a tree <i>T</i> has any leaf at an even depth. 31
22	Calculate the sum of contents of nodes of a tree <i>T</i> at each depth.
23	Count the number of nodes of a tree T at each depth. 32
24	Circular queue. 33
25	Simulating a queue using two stacks S_1 , S_2 . 34

List of Figures

1	3-element sorting algorithm represented by a decision tree. 19	
2	3-element median selection algorithm represented by a decision tree.	23
3	Binary stride. 25	
4 5	Calculate $\binom{4}{2}$ recursively. 27 Pascal triangle for binomial coefficients. 27	

List of Tables

Dedicated to those who appreciate $atural E_T X$ 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.

```
1: procedure Horner(A[0...n], x) \triangleright A : \{a_0...a_n\}
2: p \leftarrow A[n]
3: for i \leftarrow n - 1 downto 0 do
4: p \leftarrow px + A[i]
5: return p
```

Algorithm 2 Integer Multiplication.

```
1: procedure Int-Mult(y,z) \triangleright y,z \ge 0; \ y,z \in \mathbb{Z}
2: if z = 0 then
3: return o
4: return Int-Mult(cy, \lfloor \frac{z}{c} \rfloor) + y(z \mod c)
```

Sorting

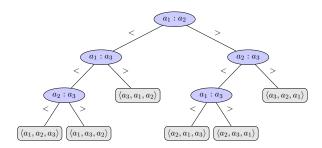


Figure 1: 3-element sorting algorithm represented by a decision tree.

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

```
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).

```
1: procedure Bubblesort(A[1 \cdots n])
2: repeat
3: newn \leftarrow 0
4: for i \leftarrow 1 to n-1 do
5: if A[i-1] > A[i] then
6: SWAP(A[i-1], A[i])
7: newn \leftarrow i
8: n \leftarrow newn
9: until n = 0
```

Quicksort

Hoare-Partition (alg4)

Algorithm 6 Hoare partition.

```
procedure Hoare-Partition(A, l, r)

x \leftarrow A[l]
i \leftarrow l
j \leftarrow r + 1

while TRUE do

while A[++i] < x do

if i = r then

break

while A[--j] > x do

if j = l then

break

if i < j then

Swap(A, l, j)
```

Linear Time Sorting

Algorithm 7 Selection Sort.

```
1: procedure Selection-Sort(A, n)

2: for i \leftarrow 1 to n - 1 do

3: for j \leftarrow i + 1 to n do

4: if A[j] < A[i] then

5: Swap(A[j], A[i])
```

Algorithm 8 Counting-Sort in place in O(n + k) time.

```
1: procedure Counting-Sort(A, k)
```

2: TODO

Selection

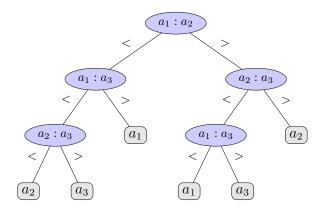


Figure 2: 3-element median selection algorithm represented by a decision tree.

Searching

Algorithm 9 (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 9 Deterministic search.

```
1: procedure Deterministic-Search(A[1 \cdots n], x)
2: i \leftarrow 1
3: while i \leq n do
4: if A[i] = x then
5: return true
6: i \leftarrow i + 1
7: return false
```

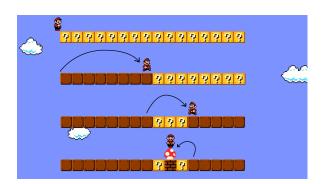


Figure 3: Binary stride.

Algorithm 10 Iterative binary search.

```
1: procedure BINARYSEARCH(A[0 \cdots n-1], x)
        L \leftarrow 0
        R \leftarrow n - 1
 3:
       while L \leq R do
 4:
           m \leftarrow L + (R - L)/2
 5:
           if A[m] < x then
                L \leftarrow m+1
 7:
            else if A[m] > x then
 8:
                R \leftarrow m-1
 9:
            else
10:
                return m
11:
        return -1
12:
```

Algorithm 11 Recursive binary search.

```
1: procedure BINARYSEARCH(A, L, R, x)
      if R < L then
3:
          return -1
      m \leftarrow L + (R - L)/2
      if A[m] = x then
 5:
 6:
          return m
      else if A[m] > x then
 7:
          return BinarySearch(A, L, m - 1, x)
 8:
 9:
          return BinarySearch(A, m + 1, R, x)
10:
```

Dynamic Programming

Algorithm 12 Computing $\binom{n}{k}$ recursively.

```
1: procedure BINOM(n,k)
```

ightharpoonup Required: $n \ge k \ge 0$

2: **if** $k = 0 \lor n = k$ **then**

3: return 1

4: **return** BINOM(n-1,k) + BINOM(n-1,k-1)

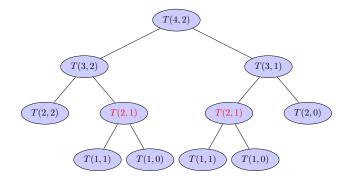


Figure 4: Calculate $\binom{4}{2}$ recursively.

$$\begin{pmatrix} 0 \\ 0 \end{pmatrix}$$

$$\begin{pmatrix} 1 \\ 0 \end{pmatrix} \quad \begin{pmatrix} 1 \\ 1 \end{pmatrix}$$

$$\begin{pmatrix} 2 \\ 0 \end{pmatrix} \quad \begin{pmatrix} 2 \\ 1 \end{pmatrix} \quad \begin{pmatrix} 2 \\ 2 \end{pmatrix}$$

$$\begin{pmatrix} 3 \\ 0 \end{pmatrix} \quad \begin{pmatrix} 3 \\ 1 \end{pmatrix} \quad \begin{pmatrix} 3 \\ 2 \end{pmatrix} \quad \begin{pmatrix} 3 \\ 3 \end{pmatrix}$$

$$\begin{pmatrix} 4 \\ 0 \end{pmatrix} \quad \begin{pmatrix} 4 \\ 1 \end{pmatrix} \quad \begin{pmatrix} 4 \\ 2 \end{pmatrix} \quad \begin{pmatrix} 4 \\ 3 \end{pmatrix} \quad \begin{pmatrix} 4 \\ 4 \end{pmatrix}$$

$$\begin{pmatrix} 5 \\ 0 \end{pmatrix} \quad \begin{pmatrix} 5 \\ 1 \end{pmatrix} \quad \begin{pmatrix} 5 \\ 2 \end{pmatrix} \quad \begin{pmatrix} 5 \\ 3 \end{pmatrix} \quad \begin{pmatrix} 5 \\ 4 \end{pmatrix} \quad \begin{pmatrix} 5 \\ 5 \end{pmatrix}$$

Figure 5: Pascal triangle for binomial coefficients.

Algorithm 13 Computing $\binom{n}{k}$ by dynamic programming.

```
1: procedure BINOM(n,k)
                                                                   \triangleright Required: n \ge k \ge 0
         for i \leftarrow 0 to n - k do
 2:
              B[i][0] \leftarrow 1
 3:
         for i \leftarrow 1 to k do
 4:
              B[i][i] \leftarrow 1
 5:
         for j \leftarrow 1 to k do
 6:
              for d \leftarrow 1 to n - k do
 7:
                   i \leftarrow j + d
 8:
                   B[i][j] \leftarrow B[i-1][j] + B[i-1][j-1]
 9:
         return B[n][k]
10:
```

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

Algorithm 14 Max-sum subarray.

```
1: procedure MSS(A[1 \cdots n])

2: MSS[0] \leftarrow 0

3: for i \leftarrow 1 to n do

4: MSS[i] \leftarrow \max \{MSS[i-1] + A[i], 0\}

5: return \max_{1 \le i \le n} MSS[i]
```

Algorithm 15 Max-sum subarray (Implementation Simplified).

```
1: procedure MSS(A[1 \cdots n])

2: mss \leftarrow 0

3: MSS \leftarrow 0

4: for i \leftarrow 1 to n do

5: MSS \leftarrow max \{MSS + A[i], 0\}

6: mss \leftarrow max \{mss, MSS\}

7: return mss
```

Traversal on Trees

DFS on Trees

```
Algorithm 16 Recursive DFS pre-order traversal on binary tree.
   procedure Recursive-DFS(t)
       print t.key
      if t.left \neq NIL then
          Recursive-DFS(t.left)
      if t.right \neq NIL then
          RECURSIVE-DFS(t.right)
   Recursive-DFS(T.root)
```

```
Algorithm 17 Iterative DFS pre-order traversal on binary tree.
```

```
procedure Iterative-DFS(t)
    S.Push(t)
                                                                 \triangleright S: stack
   while S \neq \emptyset do
       v \leftarrow S.Pop()
       print v.key
       if v.right \neq NIL then
           S.Push(v.right)
       if v.left \neq NIL then
           S.Push(v.left)
Iterative-DFS(T.root)
```

Algorithm 18 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 \neq NIL then
      RECURSIVE-DFS(t.left-child)
   if t.right-sibling \neq NIL then
      RECURSIVE-DFS(t.right-sibling)
RECURSIVE-DFS(T.root)
```

Algorithm 19 Calculate the sum of depths of all nodes of a tree *T*.

```
1: procedure Sum-of-Depths()
      return Sum-of-Depths(T,0)
3: procedure Sum-of-Depths(r, depth)
                                            ▷ r: root of a tree
     if r is a leaf then
4:
         return depth
5:
     sum \leftarrow depth
     for all child vertex v of r do
7:
         sum \leftarrow sum + Sum - of-Depths(v, depth + 1)
8:
     return sum
```

Algorithm 20 Count the number of nodes in *T* at depth *K*.

```
1: procedure Nodes-at-Depth()
       return Nodes-AT-Depth(T, K)
                                                    ⊳ r: root of a tree
3: procedure Nodes-At-Depth(r, k)
      if k = 0 then
          return 1
      if r is a leaf then
6:
          return o
7:
      num \leftarrow 0
8:
      for all child vertex v of r do
          num \leftarrow num + \text{Nodes-at-Depth}(v, k - 1)
      return num
11:
```

Algorithm 21 Check whether a tree *T* has any leaf at an even depth.

```
1: procedure Leaf-at-Even-Depth()
      return LEAF-AT-DEPTH(T, even = 0)
3: procedure Leaf-at-Depth(r, parity)
                                                         \triangleright r: root of a tree
      if r is a leaf then
          return 1 - parity
5:
      result \leftarrow 0
6:
      for all child vertex v of r do
7:
          result \leftarrow result \lor Leaf-at-Depth(v, 1 - parity)
8:
      return result
9:
```

BFS on Trees

Algorithm 22 Calculate the sum of contents of nodes of a tree *T* at each depth.

```
\triangleright r: root of the tree T
 1: procedure SUM-AT-DEPTH(r)
        r.depth \leftarrow 0
 2:
        Q \leftarrow \emptyset
        Enqueue(Q,r)
        while Q \neq \emptyset do
 5:
            u \leftarrow \text{Dequeue}(Q)
 6:
            sumAtDepth[u.depth] += u.content
 7:
            for all child vertex v of u do
 8:
                 v.depth \leftarrow u.depth + 1
 9:
                 Enqueue(Q, v)
10:
```

Algorithm 23 Count the number of nodes of a tree *T* at each depth.

```
1: procedure Nodes-at-Depth(r)
                                                               \triangleright r: root of the tree T
        r.depth \leftarrow 0
 2:
        Q \leftarrow \emptyset
 3:
        Enqueue(Q,r)
 4:
        while Q \neq \emptyset do
 5:
             u \leftarrow \text{Dequeue}(Q)
 6:
             nodesAtDepth[u.depth] += 1
 7:
             for all child vertex v of u do
 8:
                 v.depth \leftarrow u.depth + 1
 9:
                 Enqueue(Q, v)
10:
        \mathbf{return} \ \mathrm{argmax}_{K} nodes AtDepth[k]
11:
```

Stack and Queue

Stack

Queue

```
Algorithm 24 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 25 Simulating a queue using two stacks S_1, S_2.
```

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

procedure Deq()

if S_2 = \emptyset then
while S_1 \neq \emptyset do
Push(S_2, Pop(S_1))
```

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.