

# Introduction to Algorithms

---

Author: Jose 胡冠洲 @ ShanghaiTech

## [Introduction to Algorithms](#)

[Full-ver. Cheatsheet](#)

[Midterm Review Notes \(TeX\)](#)

## Full-ver. Cheatsheet

---

See below (page 2-5).

## Midterm Review Notes (TeX)

---

See below (page 6-).

Thanks Xinyu ❤️ for the wonderful notes!

$$\sum_{k=0}^n k^2 = \frac{n(n+1)(2n+1)}{6}$$

$$\sum_{k=0}^n k^3 = \frac{n^2(n+1)^2}{4}$$

$$\sum_{k=1}^n \frac{1}{k} = \ln n + O(1)$$

\* Interval Scheduling: earliest finishing time.  
 \* Interval Coloring: smallest available resource (d)  
 \* Huffman: freq  $\rightarrow$  new node.

\* Maximum Subarray:  $L+R+crossing$   
 $(L+R)$

\* Closest point pair:  $L+R+crossing$   
 $(\leq 8)$

Sort by y (crossing)  
 Check 5 points below each point.

Greedy

$$\log n! = n \log n - (\log c)n + O(\log n)$$

$$T(n) = aT(n/b) + f(n)$$

1. If  $f(n) = O(n^{\log_b a - \epsilon})$ , then  $T(n) = \Theta(n^{\log_b a})$ .
2. If  $f(n) = \Theta(n^{\log_b a})$ , then  $T(n) = \Theta(n^{\log_b a} \log n)$ .
3. If  $f(n) = \Omega(n^{\log_b a + \epsilon})$ , then  $T(n) = \Theta(f(n))$ .

\* Insertion Sort:

PARTITION( $A, p, r$ )  $O(n^2)$

```

1 x = A[r]
2 i = p - 1      stable
3 for j = p to r - 1
4   if A[j] ≤ x  average  $O(n \log n)$ 
5     i = i + 1  unstable
6   exchange A[i] with A[j]
7 exchange A[i + 1] with A[r]
8 return i + 1   $O(n)$ 

```

COUNTING-SORT( $A, B, k$ )

```

1 let C[0..k] be a new array *stable
2 for i = 0 to k
3   C[i] = 0
4 for j = 1 to A.length ount if k=0n
5   C[A[j]] = C[A[j]] + 1
6 for i = 1 to k      *radix
7   C[i] = C[i] + C[i - 1] sort:
8 for j = A.length downto 1  $O(d(n+k))$ 
9   B[C[A[j]]] = A[j]
10  C[A[j]] = C[A[j]] - 1  stable

```

MAX-HEAPIFY( $A, i$ )  $O(\log n)$

```

1 l = 2i, r = 2i + 1
2 if l ≤ A.heap-size and A[l] > A[i]
3   largest = l
4 else
5   largest = i
6 if r ≤ A.heap-size and A[r] > A[largest]
7   largest = r
8 if largest ≠ i
9   exchange A[i] with A[largest]
10 MAX-HEAPIFY(A, largest)

```

BUILD-MAX-HEAP( $A$ )

```

1 A.heap-size = A.length
2 for i = [A.length/2] downto 1
3   MAX-HEAPIFY(A, i)

```

Complexity:  $O(n)$

HEAPSORT( $A$ )  $unstable. O(n \log n)$

```

1 BUILD-MAX-HEAP(A)
2 for i = A.length downto 2
3   exchange A[1] with A[i]
4   A.heap-size = A.heap-size - 1
5   MAX-HEAPIFY(A, 1)

```

\* Insert. delete. extract-max.

$O(\log n) \Rightarrow$  priority queue.

\* binary search:  $O(\log n)$

TERNARY-SEARCH( $f, l, r, \epsilon$ )

```

1 if r - l < \epsilon
2   return (r + l)/2
3 x = l + (r - l)/3, y = r - (r - l)/3
4 if f(x) < f(y)
5   return TERNARY-SEARCH(f, l, y, \epsilon)
6 if f(x) > f(y)
7   return TERNARY-SEARCH(f, x, r, \epsilon)
8 return TERNARY-SEARCH(f, x, y, \epsilon)

```

MEDIAN-OF-MEDIANS( $A, p, r$ )

```

1 l = [(r - p + 1)/5]
2 let M[1..l] be a new array
3 for k = 1 to l - 1
4   M[k] = A[SELECT(A, 5k + p - 5,
5     5k + p - 1, 3)]
5 M[l] = A[SELECT(A, 5l + p - 5, r,
6   [(r - 5l - p + 6)/2])]
6 m = SELECT(M, 1, l, [l/2])
7 if m == l
8   return [(5l + p - 5 + r)/2]
9 return 5m + p - 3

```

\* randomized select:  
 expected  $O(n)$

SELECT( $A, p, r, i$ )

```

1 if r - p + 1 ≤ 5
2   INSERTION-SORT(A[p..r])
3   return p - 1 + i
4 exchange A[r] with
5   A[MEDIAN-OF-MEDIANS(A, p, r)]
6 q = PARTITION(A, p, r)
7 k = q - p + 1
8 if i == k
9   return q
9 elseif i < k
10  return SELECT(A, p, q - 1, i)
11 else
12  return SELECT(A, q + 1, r, i - k)

```

$$S(n) = S(n/5) + S(n) + O(n)$$

Complexity:  $O(n)$

To multiply binary numbers  $a$  and  $b$ , split their bits in half,  $a = 2^{n/2} \cdot a_1 + a_0$ ,  $b = 2^{n/2} \cdot b_1 + b_0$ , then  $ab = 2^n \cdot a_1 b_1 + 2^{n/2} \cdot (a_1 b_0 + a_0 b_1) + a_0 b_0 = 2^n \cdot a_1 b_1 + 2^{n/2} \cdot ((a_1 + a_0)(b_1 + b_0) - a_1 b_1 - a_0 b_0) + a_0 b_0$ .

\* FFT:  $w_n = e^{\frac{2\pi i}{n}}$   $A^{[0]} = a_0 + a_2 x + \dots + a_{n-2} x^{\frac{n}{2}-1}$

BST-FIND( $x, k$ )  $A^{[0]} = a_0 + a_2 x + \dots + a_{n-2} x^{\frac{n}{2}-1}$

```

1 if x == NIL or k == x.key
2   return x
3 if k < x.key
4   return FIND(x.left, k)
5 return FIND(x.right, k)

```

BST-SUCCESSOR( $x$ )  $O(n \log n)$

```

1 if x.right ≠ NIL
2   return left most node in x.right
3 y = x.p
4 while y ≠ NIL and x == y.right
5   x = y, y = y.p
6 return y

```

\* tips:  $\log \rightarrow D\&C_1$  二分答案. a. b. 排序. 区间 a-b. x. 有意义的尝试.

最小  $\rightarrow$  D&C / greedy greedy 没明: ① 至少写同其他算法. ② 约束条件 (interval coloring).

图论复杂度低  $\rightarrow$  DFS timestamp.

COUNT-CROSSING-INVERSIONS( $A, p, q, r$ )

```

1 n1 = q - p + 1
2 n2 = r - q
3 let L[0..n1 - 1] and R[0..n2 - 1]
be new arrays
4 num = 0
5 for i = 0 to n1 - 1
6   L[i] = A[p + i]
7 for j = 0 to n2 - 1
8   R[j] = A[q + 1 + j]
9 i = 0      * mergesort: stable.
10 j = 0      O(n log n). 2n memory
11 k = p
12 while i < n1 and j < n2
13   if L[i] > R[j]
14     num = num + 1
15     A[k] = R[j]
16     j = j + 1
17 else
18   num = num + j
19   A[k] = L[i]
20   i = i + 1
21 k = k + 1  L[n1] = R[n2] = 0
22 if j == n2      * merge 无判断
23   num = num + (n1 - i - 1) * n2
24   for r = 0 to n1 - i - 1
25     A[k + r] = L[i + r]
26 else
27   num = num - j
28   for r = 0 to n2 - j - 1
29     A[k + r] = R[j + r]
30 return num

```

L+R+ crossing. O(n log n)

BST-INSERT( $T, k$ )

```

1 x = T.root
2 if FIND(x, k) == NIL
3   FIND(x, k) reaches node y
4   if y == NIL  (empty tree)
5     insert in y
6   elseif z.key < y.key
7     insert in y.left
8   else
9     insert in y.right

```

\* transplant.

BST-DELETE( $T, z$ ) 反指针. O(n)

```

1 if z.left == NIL
2   TRANSPLANT(T, z, z.right)
3 elseif z.right == NIL
4   TRANSPLANT(T, z, z.left)
5 else
6   left most node in z.right
7   y = MINIMUM(z.right)
8   if y.p ≠ z
9     TRANSPLANT(T, y, y.right)
10    y.right = z.right
11    y.right.p = y
12    TRANSPLANT(T, z, y)
13    y.left = z.left
14    y.left.p = y

```

\* Rotation: 反向操作. O(1)



\* Heuristic hash functions:

Division method:  $h(k) = k \bmod m$ .  $m$  prime. not close to  $2^t$ .

Multiplication method:  $h(k) = \lfloor m(\alpha k \bmod 1) \rfloor$ ,  $\alpha \in (0, 1)$

**AVL-Insert:** BST-INSERT first, then find the lowest unbalanced node and use  $O(1)$  (one or two) rotations to restore balance.

**AVL-Delete:** BST-DELETE first, then bottom up find unbalanced node and use  $O(h)$  rotations to restore balance.

$O(\log n)$

**Hash table**  $h: U \rightarrow \{0, 1, \dots, m-1\}$   
Call  $\alpha = n/m$  load factor. The average time for each operation is  $O(\alpha)$ .

**Universal hash family  $H$ :**  $\forall$  keys  $x \neq y$ ,  $P_{h \in H}[h(x) = h(y)] \leq 1/m$ .

**Constructing:** Choose a prime number  $p$  such that  $p > m$ , and all keys  $< p$ .  $h_{ab}(k) = ((ak + b) \bmod p) \bmod m$ .  $H_{pm} = \{h_{ab} \mid a \in \{1, 2, \dots, p-1\}, b \in \{0, 1, \dots, p-1\}\}$

**Open addressing**

**Linear probing:**  $h(k, i) = (h'(k) + i) \bmod m$ .  $\rightarrow m$  probe sequences

**Quadratic probing:**  $h(k, i) = (h'(k) + c_1 i + c_2 i^2) \bmod m$ .

**Double hashing:**  $h(k, i) = (h_1(k) + ih_2(k)) \bmod m$ . make  $m$  prime.  $m' = m - 1$ , and set  $h_1(k) = k \bmod m$ ,  $h_2(k) = 1 + (k \bmod m') / m^{2^t}$ .  $h_2$  odd.  $O(m^2)$  probe  
 $h: U \times \{0, \dots, m-1\} \rightarrow \{0, \dots, m-1\}$  sequences

**BFS( $G, s$ )**

```

1 for each vertex  $u \in G.V - \{s\}$ 
2    $u.\text{color} = \text{WHITE}$ 
3    $u.d = \infty$ 
4    $u.\pi = \text{NIL}$ 
5    $s.\text{color} = \text{GRAY}$ 
6    $s.d = 0$ 
7    $s.\pi = \text{NIL}$ 
8    $Q = \emptyset$  * bipartiteness:
9   ENQUEUE( $Q, s$ ) odd layer red.
10  while  $Q \neq \emptyset$  even layer blue.
11     $u = \text{DEQUEUE}(Q)$  check color
12    for each  $v \in G.\text{Adj}[u]$  match
13      if  $v.\text{color} == \text{WHITE}$ 
14         $v.\text{color} = \text{GRAY}$ 
15         $v.d = u.d + 1$ 
16         $v.\pi = u$ 
17        ENQUEUE( $Q, v$ )
18     $u.\text{color} = \text{BLACK}$ 

```

**Complexity:**  $O(|V| + |E|)$ .

**TOPOLOGICAL-SORT( $G$ )** \* if  $v \prec w$ .  
 $v.f > w.f$ .

```

1 DFS( $G$ )
2 as each vertex is finished, insert it
   onto the front of a linked list
3 return the linked list of vertices

```

DAG.

**Complexity:**  $O(|V| + |E|)$

\* 2-SAT:

if  $x \rightsquigarrow x$ ,  $\neg x \rightsquigarrow x$ . unsatisfiable.

assign value to reachable nodes.

repeat for remaining nodes

DFS( $G$ ) \* 8-queens \* maze.

```

1 for each vertex  $u \in G.V$ 
2    $u.\text{color} = \text{WHITE}$ 
3    $u.\pi = \text{NIL}$ 
4    $time = 0$ 
5   for each vertex  $u \in G.V$ 
6     if  $u.\text{color} == \text{WHITE}$ 
7       DFS-VISIT( $G, u$ )

```

DFS-VISIT( $G, u$ )

```

1   time = time + 1
2    $u.d = time$ 
3    $u.\text{color} = \text{GRAY}$ 
4   for each  $v \in G.\text{Adj}[u]$ 
5     if  $v.\text{color} == \text{WHITE}$ 
6        $v.\pi = u$ 
7       DFS-VISIT( $G, v$ )
8    $u.\text{color} = \text{BLACK}$ 
9   time = time + 1
10   $u.f = time$ 

```

**Complexity:**  $O(|V| + |E|)$ .

**Nesting Property:** if  $u \pi = v$ .

KASARAJU'S-SCC( $G$ )  $[u.d, u.f] \subset [v.d, v.f]$

```

1 DFS( $G$ )
2 compute  $G^T$ 
3 DFS( $G^T$ ), but in the main loop
   in order of decreasing  $u.f$ 
4 output every DFS tree

```

**Complextiy:**  $O(|V| + |E|)$

MST-PRIM( $G, w$ ) unique path.  
no edges.

```

1  $H = \{\text{an arbitrary node } v \in G.V\}$ 
2  $F = \emptyset, R = \emptyset$ 
3 for each vertex  $u \in G.\text{Adj}[v]$ 
4    $R = R \cup \{(u, v)\}$ 
5 while  $H \neq G.V$ 
6    $e = \text{EXTRACT-MIN}(R)$  priority queue
7   let  $e = (u, v)$ , with  $v \notin H$ 
8    $H = H \cup \{v\}$ 
9    $F = F \cup \{e\}$ 
10  for each vertex  $u \in G.\text{Adj}[v]$ 
11    if  $u \notin H$ 
12       $R = R \cup \{(u, v)\}$ 

```

**Complexity:**  $O(|E| \log |V|)$

MST-KRUSKAL( $G, w$ )

```

1  $A = \emptyset$ 
2 for each vertex  $v \in G.V$ 
3   MAKE-SET( $v$ )
4   sort  $G.E$  in nondecreasing order of  $e.w$ 
5   for each edge  $(u, v) \in G.E$ 
6     if FIND-SET( $u$ )  $\neq$  FIND-SET( $v$ )
7        $A = A \cup \{(u, v)\}$ 
8       UNION( $u, v$ )
9   return  $A$ 

```

**Complexity:**  $O(|E| \log |V|)$

\* MST principle:  $\frac{1}{2}$  partial MST.

CCH.  $e = \min$  weight edge

between  $C_s$ .  $\Rightarrow$  HU's  $\Rightarrow$  partial MST.



INITIALIZE-SINGLE-SOURCE( $G, s$ )

1 for each vertex  $v \in G.V$

2  $v.d = \infty$

3  $v.\pi = \text{NIL}$

4  $s.d = 0$

\* triangle inequality:  $S(s, v) \leq S(s, u) + S(u, v)$

RELAX( $u, v, w$ ) \* shortest subpaths.

1 if  $v.d > u.d + w(u, v)$

2  $v.d = u.d + w(u, v)$

3  $v.\pi = u$

BELLMAN-FORD( $G, w, s$ )

1 INITIALIZE-SINGLE-SOURCE( $G, s$ )

2 for  $i = 1$  to  $|G.V| - 1$

3 for each edge  $(u, v) \in G.E$

4 RELAX( $u, v, w$ )

5 for each edge  $(u, v) \in G.E$

6 if  $v.d > u.d + w(u, v)$

7 return FALSE

8 return TRUE

**Complexity:**  $O(|V||E|)$

DIJKSTRA( $G, w, s$ )

1 INITIALIZE-SINGLE-SOURCE( $G, s$ )

2  $S = \emptyset$

3  $Q = G.V$

4 while  $Q \neq \emptyset$

5  $u = \text{EXTRACT-MIN}(Q)$

6  $S = S \cup \{u\}$

7 for each vertex  $v \in G.\text{Adj}[u]$

8 RELAX( $u, v, w$ )

**Complexity:**  $O(|E| \log |V|)$  using a heap  
 $O(|V| \log |V| + |E|)$  using a Fibonacci heap

EXTEND-SHORTEST-PATHS( $L, W$ )

1  $n = L.\text{rows}$

2 let  $L' = (l'_{ij})$  be a new  $n \times n$  matrix

3 for  $i = 1$  to  $n$

4 for  $j = 1$  to  $n$

5  $l'_{ij} = \infty$

6 for  $k = 1$  to  $n$

7  $l'_{ij} = \min(l'_{ij}, l_{ik} + w_{kj})$

8 return  $L'$  repeated doubling.

**Complexity:**  $O(n^3 \log n)$   $n \neq 2^k \Rightarrow n = \sum 2^{j-1}$   
(binary)

FLOYD-WARSHALL( $W$ )

1  $n = W.\text{rows}$

2  $D^{(0)} = W$

3 for  $k = 1$  to  $n$

4 let  $D^{(k)} = (d_{ij}^{(k)})$

be a new  $n \times n$  matrix

5 for  $i = 1$  to  $n$

6 for  $j = 1$  to  $n$

7  $d_{ij}^{(k)} = \min(d_{ij}^{(k-1)},$

$d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$

8 return  $D^{(n)}$

**Complexity:**  $O(n^3)$

\* Arborescences:  $w^*(u, v) = w(u, v) - S(u, v)$

\* Johnson's:  $h(u, v) = S(u, v)$  reweighing

$\hat{w}(u, v) = w(u, v) + h(u) - h(v)$

$d_{uv} = \hat{w}(u, v) + h(v) - h(u)$



把 H-min 问题转化为树的 root 1st  
直到 H-min.

相向 degree(对称度) root key  
大根堆的最小 root 为对称度。

直到对称度。

decrease-key: O(1)

violating min heap property.  
→ cut out (移除) not list

mark: → 一个被 cut out

第 1 个被移除 out 的本身

eff cut out

cascading cut 直到 root.

delete: decrease-key → -o.

extract-min. O(logn)

string matching.

Bruce-force: O(nm)

Knuth-Morris-Pratt (KMP).

failure function f[i]: len

of longest matching proper

prefix & suffix of SI. jj

failure function f[i]: len

of longest matching proper

prefix & suffix of SI. jj

failure function f[i]: len

of longest matching proper

prefix & suffix of SI. jj

failure function f[i]: len

of longest matching proper

prefix & suffix of SI. jj

failure function f[i]: len

of longest matching proper

prefix & suffix of SI. jj

failure function f[i]: len

of longest matching proper

prefix & suffix of SI. jj

failure function f[i]: len

of longest matching proper

failure function f[i]: len

function KMP(S,T) O(mn)  
m=len(S), n=len(T)  
while (len) and (j<m) in T  
if T[j]=S[j]

i=j+1, j=j+1

else if j=1

i=i+1, j=j+1

else constant time to cut

j=f[i-1]+i-1+q-q

if j=m+1

return i-m

return "S not in T"

O(mn) most (average) case

O(mn) worst case

function Rabin-Karp(S,T,d,q)

n=len(S), m=len(T)

he d<sup>m</sup> mod q

pe<sub>0</sub>, to<sub>0</sub>

for i=1 to m

P<sub>i</sub>=(d·P<sub>i-1</sub>+SI<sub>i</sub>)mod q

to<sub>i</sub>=(d<sup>i</sup>·SI<sub>i</sub>)mod q

for i=1 to m

P<sub>i</sub>=(d·P<sub>i-1</sub>+SI<sub>i</sub>)mod q

to<sub>i</sub>=(d<sup>i</sup>·SI<sub>i</sub>)mod q

for i=0 to n-m

if P<sub>i</sub>=P<sub>i+m</sub>

if S[1..m]=T[1..m]

print S then break

if S[1..m]=T[1..m]

break

if S[1..m]=T[1..m]

break

if S[1..m]=T[1..m]

break

(SQL)  $\text{SELECT COUNT(*) FROM table WHERE column = 'value'}$   
 $\text{SELECT COUNT(*) FROM table WHERE column IN ('value1', 'value2')}$   
 $\text{SELECT COUNT(*) FROM table WHERE column NOT IN ('value1', 'value2')}$   
 $\text{SELECT COUNT(*) FROM table WHERE column > value}$   
 $\text{SELECT COUNT(*) FROM table WHERE column < value}$   
 $\text{SELECT COUNT(*) FROM table WHERE column >= value}$   
 $\text{SELECT COUNT(*) FROM table WHERE column <= value}$   
 $\text{SELECT COUNT(*) FROM table WHERE column = ANY (VALUES 'value1', 'value2')}$   
 $\text{SELECT COUNT(*) FROM table WHERE column NOT ANY (VALUES 'value1', 'value2')}$   
 $\text{SELECT COUNT(*) FROM table WHERE column > ANY (VALUES 'value1', 'value2')}$   
 $\text{SELECT COUNT(*) FROM table WHERE column < ANY (VALUES 'value1', 'value2')}$   
 $\text{SELECT COUNT(*) FROM table WHERE column >= ANY (VALUES 'value1', 'value2')}$   
 $\text{SELECT COUNT(*) FROM table WHERE column <= ANY (VALUES 'value1', 'value2')}$   
 $\text{SELECT COUNT(*) FROM table WHERE column = ALL (VALUES 'value1', 'value2')}$   
 $\text{SELECT COUNT(*) FROM table WHERE column NOT ALL (VALUES 'value1', 'value2')}$   
 $\text{SELECT COUNT(*) FROM table WHERE column > ALL (VALUES 'value1', 'value2')}$   
 $\text{SELECT COUNT(*) FROM table WHERE column < ALL (VALUES 'value1', 'value2')}$   
 $\text{SELECT COUNT(*) FROM table WHERE column >= ALL (VALUES 'value1', 'value2')}$   
 $\text{SELECT COUNT(*) FROM table WHERE column <= ALL (VALUES 'value1', 'value2')}$

(SQL)  $\text{SELECT COUNT(*) FROM table WHERE column = 'value'}$   
 $\text{SELECT COUNT(*) FROM table WHERE column IN ('value1', 'value2')}$   
 $\text{SELECT COUNT(*) FROM table WHERE column NOT IN ('value1', 'value2')}$   
 $\text{SELECT COUNT(*) FROM table WHERE column > value}$   
 $\text{SELECT COUNT(*) FROM table WHERE column < value}$   
 $\text{SELECT COUNT(*) FROM table WHERE column >= value}$   
 $\text{SELECT COUNT(*) FROM table WHERE column <= value}$   
 $\text{SELECT COUNT(*) FROM table WHERE column = ANY (VALUES 'value1', 'value2')}$   
 $\text{SELECT COUNT(*) FROM table WHERE column NOT ANY (VALUES 'value1', 'value2')}$   
 $\text{SELECT COUNT(*) FROM table WHERE column > ANY (VALUES 'value1', 'value2')}$   
 $\text{SELECT COUNT(*) FROM table WHERE column < ANY (VALUES 'value1', 'value2')}$   
 $\text{SELECT COUNT(*) FROM table WHERE column >= ANY (VALUES 'value1', 'value2')}$   
 $\text{SELECT COUNT(*) FROM table WHERE column <= ANY (VALUES 'value1', 'value2')}$   
 $\text{SELECT COUNT(*) FROM table WHERE column = ALL (VALUES 'value1', 'value2')}$   
 $\text{SELECT COUNT(*) FROM table WHERE column NOT ALL (VALUES 'value1', 'value2')}$   
 $\text{SELECT COUNT(*) FROM table WHERE column > ALL (VALUES 'value1', 'value2')}$   
 $\text{SELECT COUNT(*) FROM table WHERE column < ALL (VALUES 'value1', 'value2')}$   
 $\text{SELECT COUNT(*) FROM table WHERE column >= ALL (VALUES 'value1', 'value2')}$   
 $\text{SELECT COUNT(*) FROM table WHERE column <= ALL (VALUES 'value1', 'value2')}$

(SQL)  $\text{SELECT COUNT(*) FROM table WHERE column = 'value'}$   
 $\text{SELECT COUNT(*) FROM table WHERE column IN ('value1', 'value2')}$   
 $\text{SELECT COUNT(*) FROM table WHERE column NOT IN ('value1', 'value2')}$   
 $\text{SELECT COUNT(*) FROM table WHERE column > value}$   
 $\text{SELECT COUNT(*) FROM table WHERE column < value}$   
 $\text{SELECT COUNT(*) FROM table WHERE column >= value}$   
 $\text{SELECT COUNT(*) FROM table WHERE column <= value}$   
 $\text{SELECT COUNT(*) FROM table WHERE column = ANY (VALUES 'value1', 'value2')}$   
 $\text{SELECT COUNT(*) FROM table WHERE column NOT ANY (VALUES 'value1', 'value2')}$   
 $\text{SELECT COUNT(*) FROM table WHERE column > ANY (VALUES 'value1', 'value2')}$   
 $\text{SELECT COUNT(*) FROM table WHERE column < ANY (VALUES 'value1', 'value2')}$   
 $\text{SELECT COUNT(*) FROM table WHERE column >= ANY (VALUES 'value1', 'value2')}$   
 $\text{SELECT COUNT(*) FROM table WHERE column <= ANY (VALUES 'value1', 'value2')}$   
 $\text{SELECT COUNT(*) FROM table WHERE column = ALL (VALUES 'value1', 'value2')}$   
 $\text{SELECT COUNT(*) FROM table WHERE column NOT ALL (VALUES 'value1', 'value2')}$   
 $\text{SELECT COUNT(*) FROM table WHERE column > ALL (VALUES 'value1', 'value2')}$   
 $\text{SELECT COUNT(*) FROM table WHERE column < ALL (VALUES 'value1', 'value2')}$   
 $\text{SELECT COUNT(*) FROM table WHERE column >= ALL (VALUES 'value1', 'value2')}$   
 $\text{SELECT COUNT(*) FROM table WHERE column <= ALL (VALUES 'value1', 'value2')}$

(SQL)  $\text{SELECT COUNT(*) FROM table WHERE column = 'value'}$   
 $\text{SELECT COUNT(*) FROM table WHERE column IN ('value1', 'value2')}$   
 $\text{SELECT COUNT(*) FROM table WHERE column NOT IN ('value1', 'value2')}$   
 $\text{SELECT COUNT(*) FROM table WHERE column > value}$   
 $\text{SELECT COUNT(*) FROM table WHERE column < value}$   
 $\text{SELECT COUNT(*) FROM table WHERE column >= value}$   
 $\text{SELECT COUNT(*) FROM table WHERE column <= value}$   
 $\text{SELECT COUNT(*) FROM table WHERE column = ANY (VALUES 'value1', 'value2')}$   
 $\text{SELECT COUNT(*) FROM table WHERE column NOT ANY (VALUES 'value1', 'value2')}$   
 $\text{SELECT COUNT(*) FROM table WHERE column > ANY (VALUES 'value1', 'value2')}$   
 $\text{SELECT COUNT(*) FROM table WHERE column < ANY (VALUES 'value1', 'value2')}$   
 $\text{SELECT COUNT(*) FROM table WHERE column >= ANY (VALUES 'value1', 'value2')}$   
 $\text{SELECT COUNT(*) FROM table WHERE column <= ANY (VALUES 'value1', 'value2')}$   
 $\text{SELECT COUNT(*) FROM table WHERE column = ALL (VALUES 'value1', 'value2')}$   
 $\text{SELECT COUNT(*) FROM table WHERE column NOT ALL (VALUES 'value1', 'value2')}$   
 $\text{SELECT COUNT(*) FROM table WHERE column > ALL (VALUES 'value1', 'value2')}$   
 $\text{SELECT COUNT(*) FROM table WHERE column < ALL (VALUES 'value1', 'value2')}$   
 $\text{SELECT COUNT(*) FROM table WHERE column >= ALL (VALUES 'value1', 'value2')}$   
 $\text{SELECT COUNT(*) FROM table WHERE column <= ALL (VALUES 'value1', 'value2')}$

(SQL)  $\text{SELECT COUNT(*) FROM table WHERE column = 'value'}$   
 $\text{SELECT COUNT(*) FROM table WHERE column IN ('value1', 'value2')}$   
 $\text{SELECT COUNT(*) FROM table WHERE column NOT IN ('value1', 'value2')}$   
 $\text{SELECT COUNT(*) FROM table WHERE column > value}$   
 $\text{SELECT COUNT(*) FROM table WHERE column < value}$   
 $\text{SELECT COUNT(*) FROM table WHERE column >= value}$   
 $\text{SELECT COUNT(*) FROM table WHERE column <= value}$   
 $\text{SELECT COUNT(*) FROM table WHERE column = ANY (VALUES 'value1', 'value2')}$   
 $\text{SELECT COUNT(*) FROM table WHERE column NOT ANY (VALUES 'value1', 'value2')}$   
 $\text{SELECT COUNT(*) FROM table WHERE column > ANY (VALUES 'value1', 'value2')}$   
 $\text{SELECT COUNT(*) FROM table WHERE column < ANY (VALUES 'value1', 'value2')}$   
 $\text{SELECT COUNT(*) FROM table WHERE column >= ANY (VALUES 'value1', 'value2')}$   
 $\text{SELECT COUNT(*) FROM table WHERE column <= ANY (VALUES 'value1', 'value2')}$   
 $\text{SELECT COUNT(*) FROM table WHERE column = ALL (VALUES 'value1', 'value2')}$   
 $\text{SELECT COUNT(*) FROM table WHERE column NOT ALL (VALUES 'value1', 'value2')}$   
 $\text{SELECT COUNT(*) FROM table WHERE column > ALL (VALUES 'value1', 'value2')}$   
 $\text{SELECT COUNT(*) FROM table WHERE column < ALL (VALUES 'value1', 'value2')}$   
 $\text{SELECT COUNT(*) FROM table WHERE column >= ALL (VALUES 'value1', 'value2')}$   
 $\text{SELECT COUNT(*) FROM table WHERE column <= ALL (VALUES 'value1', 'value2')}$

# CS140 Midterm Review

Zhang Xinyu

November 21, 2017

## 1 Mathematical Background

### 1.1 Summations

#### 1.1.1 Sums of squares and cubes

$$\sum_{k=0}^n k^2 = \frac{n(n+1)(2n+1)}{6}$$
$$\sum_{k=0}^n k^3 = \frac{n^2(n+1)^2}{4}$$

#### 1.1.2 Harmonic series

$$\sum_{k=1}^n \frac{1}{k} = \ln n + O(1)$$

### 1.2 Stirling's approximation

$$\log n! = n \log n - (\log e)n + O(\log n)$$

## 2 Analysis of algorithms

### 2.1 Master theorem

Let  $a \geq 1$  and  $b > 1$  be constants, let  $f(n)$  be a function, and let  $T(n)$  be defined on the nonnegative integers by the recurrence

$$T(n) = aT(n/b) + f(n)$$

where we interpret  $n/b$  to mean either  $\lfloor n/b \rfloor$  or  $\lceil n/b \rceil$ . Then  $T(n)$  has the following asymptotic bounds:

1. If  $f(n) = O(n^{\log_b a - \epsilon})$  for some constant  $\epsilon > 0$ , then  $T(n) = \Theta(n^{\log_b a})$ .
2. If  $f(n) = \Theta(n^{\log_b a})$ , then  $T(n) = \Theta(n^{\log_b a} \log n)$ .
3. If  $f(n) = \Omega(n^{\log_b a + \epsilon})$  for some constant  $\epsilon > 0$ , and if  $af(n/b) \leq cf(n)$  for some constant  $c < 1$  and all sufficiently large  $n$ , then  $T(n) = \Theta(f(n))$ .

**Note:** In cases 1 (case 3),  $f(n)$  needs to be smaller (larger) than  $n^{\log_b a}$  by a **polynomial factor**  $n^\epsilon$ , so we can't always apply Master theorem.

E.g.  $T(n) = 2T(n/2) + n \log n$ ,  $\log_b a = 1$ ,  $f(n) \neq \Theta(n)$  and  $f(n) \neq \Omega(n^{1+\epsilon})$ , can't use Master theorem to solve this recurrence.

## 2.2 Big O analysis

### 2.2.1 Asymptotic notation

notation	set of functions form	limit form
$O(g(n))$	$\{f(n) : \exists c > 0, n_0 > 0, \forall n \geq n_0, 0 \leq f(n) \leq cg(n)\}$	$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} < \infty$
$\Omega(g(n))$	$\{f(n) : \exists c > 0, n_0 > 0, \forall n \geq n_0, 0 \leq cg(n) \leq f(n)\}$	$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} > 0$
$\Theta(g(n))$	$\{f(n) : \exists c_1 > 0, c_2 > 0, n_0 > 0, \forall n \geq n_0, 0 \leq c_1g(n) \leq f(n) \leq c_2g(n)\}$	$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = c, 0 < c < \infty$
$o(g(n))$	$\{f(n) : \forall c > 0, \exists n_0 > 0, \forall n \geq n_0, 0 \leq f(n) < cg(n)\}$	$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0$
$\omega(g(n))$	$\{f(n) : \forall c > 0, \exists n_0 > 0, \forall n \geq n_0, 0 \leq cg(n) < f(n)\}$	$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \infty$

### 2.2.2 Big O properties

1.  $f(n) = O(g(n)), f(n) = \Omega(g(n)) \implies f(n) = \Theta(g(n))$
2.  $O, \Omega, \Theta$  are **transitive**.
3.  $\Theta$  is **symmetric**.
4.  $O(1)$  is the set of constants.

## 3 Sorting

### 3.1 Comparison sorts

$\Omega(n \log n)$  lower bound on time complexity

#### 3.1.1 Insertion sort

INSERTION-SORT( $A$ )

```

1  for  $j = 2$  to  $A.length$ 
2       $key = A[j]$ 
3      // Insert  $A[j]$  into the sorted sequence  $A[1..j-1]$ .
4       $i = j - 1$ 
5      while  $i > 0$  and  $A[i] > key$ 
6           $A[i + 1] = A[i]$ 
7           $i = i - 1$ 
8       $A[i + 1] = key$ 

```

**Complexity:**  $O(n^2)$

**Pros:**

Simple algorithm with little bookkeeping overhead.

Requires only one unit of extra storage, for swap.

Fast if input is nearly sorted.

Effective for small  $n$  (e.g.  $n \leq 20$ ), due to low overhead, and  $O(n^2)$  being not too big for small  $n$ .

**Cons:** not practical for large input size because  $O(n^2)$  is too high

### 3.1.2 Mergesort

MERGE( $A, p, q, r$ )

```

1   $n_1 = q - p + 1$ 
2   $n_2 = r - q$ 
3  let  $L[1..n_1 + 1]$  and  $R[1..n_2 + 1]$  be new arrays
4  for  $i = 1$  to  $n_1$ 
5     $L[i] = A[p + i - 1]$ 
6  for  $j = 1$  to  $n_2$ 
7     $R[j] = A[q + j]$ 
8   $L[n_1 + 1] = \infty$ 
9   $R[n_2 + 1] = \infty$ 
10  $i = 1$ 
11  $j = 1$ 
12 for  $k = p$  to  $r$ 
13   if  $L[i] \leq R[j]$ 
14      $A[k] = L[i]$ 
15      $i = i + 1$ 
16   else
17      $A[k] = R[j]$ 
18      $j = j + 1$ 
```

MERGE-SORT( $A, p, r$ )

```

1  if  $p < r$ 
2     $q = \lfloor (p + r)/2 \rfloor$ 
3    MERGE-SORT( $A, p, q$ )
4    MERGE-SORT( $A, q + 1, r$ )
5    MERGE( $A, p, q, r$ )
```

**Complexity:**  $O(n \log n)$

**Pros:** much faster for large  $n$ ; can be parallelized

**Cons:** needs extra storage in  $L, R, 2n$  storage for sorting  $n$  numbers

**Combining mergesort and insertion sort:**

Divide problem into small groups ( $\approx 20$  elements each).

Use insertion sort to sort each group.

Combine the sorted groups using merge.

### 3.1.3 Quicksort

PARTITION( $A, p, r$ )

```

1   $x = A[r]$ 
2   $i = p - 1$ 
3  for  $j = p$  to  $r - 1$ 
4    if  $A[j] \leq x$ 
5       $i = i + 1$ 
6      exchange  $A[i]$  with  $A[j]$ 
7  exchange  $A[i + 1]$  with  $A[r]$ 
8  return  $i + 1$ 
```

**Complexity:**  $O(n)$

```

QUICKSORT( $A, p, r$ )
1  if  $p < r$ 
2       $q = \text{PARTITION}(A, p, r)$ 
3       $\text{QUICKSORT}(A, p, q - 1)$ 
4       $\text{QUICKSORT}(A, q + 1, r)$ 

```

**Complexity:** depends on how **balanced** the pivot is.

Best case:  $u = v, O(n \log n)$

Worst case:  $u = n - 1, v = 0, O(n^2)$

Average case:  $O(n \log n)$  (use **recursion tree** to see if  $u = 0.9n, v = 0.1n - 1$ )

**Pros:** generally a little faster than mergesort; needs no additional storage; can be parallelized

**Comparing mergesort and quicksort:**

In mergesort, the division is trivial, but combination step takes  $O(n)$  time.

Quicksort spends more time finding a nice division of the problem, to make the combination step faster.

Overall time complexity of mergesort and Quicksort are the same (assuming random input).

## 3.2 Sorting in linear time

### 3.2.1 Counting sort

Assumes all input values are integers in the range 0 to  $k$ , for some  $k$ . Used as a stand-alone algorithm, and also as a subroutine in other algorithms, e.g., radix sort, which ensures  $k$  is small when counting sort is fast.

```

COUNTING-SORT( $A, B, k$ )
1  let  $C[0..k]$  be a new array
2  for  $i = 0$  to  $k$ 
3       $C[i] = 0$ 
4  for  $j = 1$  to  $A.length$ 
5       $C[A[j]] = C[A[j]] + 1$ 
6 //  $C[i]$  now contains the number of elements equal to  $i$ .
7  for  $i = 1$  to  $k$ 
8       $C[i] = C[i] + C[i - 1]$ 
9 //  $C[i]$  now contains the number of elements less than or equal to  $i$ .
10 for  $j = A.length$  downto 1
11      $B[C[A[j]]] = A[j]$ 
12      $C[A[j]] = C[A[j]] - 1$ 

```

**Complexity:**  $\Theta(n + k)$ . If  $k = O(n)$ ,  $\Theta(n)$ .

**Pros:** stable

**Cons:** becomes increasingly inefficient as  $k$  gets larger

### 3.2.2 Radix sort

RADIX-SORT( $A, d$ )

```

1  for  $i = 1$  to  $d$ 
2      use a stable sort to sort array  $A$  on digit  $i$ 

```

**Complexity:**

1.  $n$   $d$ -digit numbers, where each digit is between 0 and  $k - 1$ ,  $\Theta(d(n + k))$

2.  $n$   $b$ -bit numbers, break each number into blocks of  $r \leq b$  bits,  $\Theta(\frac{b}{r}(n + 2^r))$
3. setting  $r = \min(\lfloor \log n \rfloor, b)$  minimizes the running time  $\Theta(\frac{b}{r}(n + 2^r))$

### 3.2.3 Bucket sort

Works for real value inputs in  $[0, 1]$ . Can shift and scale inputs so they always fall in this range.

BUCKET-SORT( $A$ )

- 1 let  $B[0..n - 1]$  be a new array
- 2  $n = A.length$
- 3 **for**  $i = 0$  **to**  $n - 1$
- 4     make  $B[i]$  an empty list
- 5 **for**  $i = 1$  **to**  $n$
- 6     insert  $A[i]$  into list  $B[\lfloor nA[i] \rfloor]$
- 7 **for**  $i = 0$  **to**  $n - 1$
- 8     sort list  $B[i]$  with insertion sort
- 9 concatenate the lists  $B[0], B[1], \dots, B[n - 1]$  together in order

**Complexity:**  $\Theta(n)$  if inputs are **randomly distributed**.

*Proof.* Total running time  $T(n)$  equals  $\Theta(n)$  time distributing input values to buckets plus  $\sum_{i=0}^{n-1} O(n_i^2)$  time sorting all the buckets using insertion sort.

$$\begin{aligned} E[T(n)] &= E \left[ \Theta(n) + \sum_{i=0}^{n-1} O(n_i^2) \right] \\ &= \Theta(n) + \sum_{i=0}^{n-1} E[O(n_i^2)] \\ &= \Theta(n) + \sum_{i=0}^{n-1} O(E[n_i^2]) \end{aligned}$$

Let  $X_{ij} = 1$  if the  $j$ th input value falls into the  $i$ th bucket, 0 otherwise.  $n_i = \sum_{j=1}^n X_{ij}$  is the number of inputs in the  $i$ th bucket.

$$\begin{aligned} E[n_i^2] &= E \left[ \left( \sum_{j=1}^n X_{ij} \right)^2 \right] \\ &= E \left[ \sum_{j=1}^n \sum_{k=1}^n X_{ij} X_{ik} \right] \\ &= E \left[ \sum_{j=1}^n X_{ij}^2 + \sum_{1 \leq j \leq n} \sum_{\substack{1 \leq k \leq n \\ k \neq j}} X_{ij} X_{ik} \right] \\ &= \sum_{j=1}^n E[X_{ij}^2] + \sum_{1 \leq j \leq n} \sum_{\substack{1 \leq k \leq n \\ k \neq j}} E[X_{ij} X_{ik}] \end{aligned}$$

As  $E[X_{ij}^2] = 1^2 \cdot \frac{1}{n} + 0^2 \cdot \left(1 - \frac{1}{n}\right) = \frac{1}{n}$ ,  $E[X_{ij}X_{ik}] = E[X_{ij}]E[X_{ik}] = \frac{1}{n} \cdot \frac{1}{n} = \frac{1}{n^2}$ ,

$$\begin{aligned} E[n_i^2] &= \sum_{j=1}^n \frac{1}{n} + \sum_{1 \leq j \leq n} \sum_{\substack{1 \leq k \leq n \\ k \neq j}} \frac{1}{n^2} \\ &= n \cdot \frac{1}{n} + n(n-1) \cdot \frac{1}{n^2} \\ &= 1 + \frac{n-1}{n} \\ &= 2 - \frac{1}{n} \end{aligned}$$

Thus  $E[T(n)] = \Theta(n) + \sum_{i=0}^{n-1} O(E[n_i^2]) = \Theta(n) + \sum_{i=0}^{n-1} O\left(2 - \frac{1}{n}\right) = \Theta(n)$ .  $\square$

### 3.3 Heapsort

#### 3.3.1 Heap

**Depth** of a node: the length of the path from the node to the root

**Height** of a tree: the largest depth of any node

**Complete binary trees**: all layers, except possibly the last, are full; nodes in the bottom layer are as far left as possible

height =  $h \iff$  # of nodes  $\in [2^h, 2^{h+1} - 1]$

# of nodes =  $n \iff$  height =  $\lfloor \log_2 n \rfloor$

**Heap**: a complete binary tree where each node contains a value; the value of any node is larger (smaller) than the value of any other node in its **subtree** in a max (min) heap.

#### 3.3.2 Heapify

LEFT( $i$ )

1 **return**  $2i$

RIGHT( $i$ )

1 **return**  $2i + 1$

MAX-HEAPIFY( $A, i$ )

```

1   l = LEFT(i)
2   r = RIGHT(i)
3   if l ≤ A.heap-size and A[l] > A[i]
4       largest = l
5   else
6       largest = i
7   if r ≤ A.heap-size and A[r] > A[largest]
8       largest = r
9   if largest ≠ i
10      exchange A[i] with A[largest]
11      MAX-HEAPIFY(A, largest)
```

**Complexity**:  $O(h - d)$  where  $h$  is the height of the heap and  $d$  is the depth of the node

### 3.3.3 Building a heap

Label nodes left to right, top to bottom, heapify **internal nodes** in **reverse label order** (right to left, bottom up).

BUILD-MAX-HEAP( $A$ )

```
1  $A.\text{heap-size} = A.\text{length}$ 
2 for  $i = \lfloor A.\text{length}/2 \rfloor$  downto 1
3     MAX-HEAPIFY( $A, i$ )
```

**Complexity:**  $O(n)$

### 3.3.4 Heapsort

HEAPSORT( $A$ )

```
1 BUILD-MAX-HEAP( $A$ )
2 for  $i = A.\text{length}$  downto 2
3     exchange  $A[1]$  with  $A[i]$ 
4      $A.\text{heap-size} = A.\text{heap-size} - 1$ 
5     MAX-HEAPIFY( $A, 1$ )
```

**Complexity:**  $O(n \log n)$

**Cons:** generally slower than others due to cache effects; cannot be parallelized

## 4 Search

### 4.1 Binary search

BINARY-SEARCH( $A, x, l, r$ )

```
1  $m = \lfloor (l + r)/2 \rfloor$ 
2 if  $x == A[m]$ 
3     return  $m$ 
4 if  $l == r$ 
5     return NIL
6 if  $x < A[m]$ 
7     return BINARY-SEARCH( $A, x, l, m - 1$ )
8 return BINARY-SEARCH( $A, x, m + 1, r$ )
```

**Complexity:**  $O(\log n)$

### 4.2 Ternary search

A function  $f$  is unimodal if there is an  $x$  s.t.  $f$  is monotonically decreasing for  $y \leq x$ , and monotonically increasing for  $y > x$ . Thus  $f$  has a unique minimum at  $x$ . Can also consider symmetric case where  $f$  first increases then decreases. In this case  $f$  has a unique maximum. Our goal is to find  $x$ .

```

TERNARY-SEARCH( $f, l, r, \epsilon$ )
1  if  $r - l < \epsilon$ 
2    return  $(r + l)/2$ 
3   $x = l + (r - l)/3$ 
4   $y = r - (r - l)/3$ 
5  if  $f(x) < f(y)$ 
6    return TERNARY-SEARCH( $f, l, y, \epsilon$ )
7  if  $f(x) > f(y)$ 
8    return TERNARY-SEARCH( $f, x, r, \epsilon$ )
9  return TERNARY-SEARCH( $f, x, y, \epsilon$ )

```

**Complexity:**  $O(\log(\frac{r-l}{\epsilon}))$

### 4.3 Selection using median of medians

```

MEDIAN-OF-MEDIANS( $A, p, r$ )
1   $l = \lceil(r - p + 1)/5\rceil$ 
2  let  $M[1..l]$  be a new array
3  for  $k = 1$  to  $l - 1$ 
4     $M[k] = A[\text{SELECT}(A, 5k + p - 5, 5k + p - 1, 3)]$ 
5   $M[l] = A[\text{SELECT}(A, 5l + p - 5, r, \lceil(r - 5l - p + 6)/2\rceil)]$ 
6   $m = \text{SELECT}(M, 1, l, \lfloor l/2 \rfloor)$ 
7  if  $m == l$ 
8    return  $\lfloor(5l + p - 5 + r)/2\rfloor$ 
9  return  $5m + p - 3$ 

```

```

SELECT( $A, p, r, i$ )
1  if  $r - p + 1 \leq 5$ 
2    INSERTION-SORT( $A[p..r]$ )
3    return  $p - 1 + i$ 
4  exchange  $A[r]$  with  $A[\text{MEDIAN-OF-MEDIANS}(A, p, r)]$ 
5   $q = \text{PARTITION}(A, p, r)$ 
6   $k = q - p + 1$ 
7  if  $i == k$ 
8    return  $q$ 
9  elseif  $i < k$ 
10   return SELECT( $A, p, q - 1, i$ )
11 else
12   return SELECT( $A, q + 1, r, i - k$ )

```

**Complexity:**  $S(n) = S(n/5) + S(u) + O(n)$  where  $u$  is the size of whichever partition we recurse on

$O(n)$  finding medians of  $\lceil(n/5)\rceil$  groups of 5 numbers and partitioning the array

$S(n/5)$  finding the median of the  $\lceil(n/5)\rceil$  medians

$u \leq 7n/10$ ,  $S(n) = O(n)$ , but the hidden constant term is **large**

### 4.4 Randomized selection

```

RANDOMIZED-PARTITION( $A, p, r$ )
1   $i = \text{RANDOM}(p, r)$ 
2  exchange  $A[r]$  with  $A[i]$ 
3  return PARTITION( $A, p, r$ )

```

```

RANDOMIZED-SELECT( $A, p, r, i$ )
1  if  $p == r$ 
2    return  $A[p]$ 
3   $q = \text{RANDOMIZED-PARTITION}(A, p, r)$ 
4   $k = q - p + 1$ 
5  if  $i == k$       // the pivot value is the answer
6    return  $A[q]$ 
7  elseif  $i < k$ 
8    return RANDOMIZED-SELECT( $A, p, q - 1, i$ )
9  else
10   return RANDOMIZED-SELECT( $A, q + 1, r, i - k$ )

```

**Complexity:** expected linear time

## 5 Binary search trees

### 5.1 Operations

**Complexity:** all  $O(h)$

#### 5.1.1 Find

```

FIND( $x, k$ )
1  if  $x == \text{NIL}$  or  $k == x.key$ 
2    return  $x$ 
3  if  $k < x.key$ 
4    return FIND( $x.left, k$ )
5  return FIND( $x.right, k$ )

```

ITERATIVE-FIND( $x, k$ )

```

1  while  $x \neq \text{NIL}$  and  $k \neq x.key$ 
2    if  $k < x.key$ 
3       $x = x.left$ 
4    else
5       $x = x.right$ 
6  return  $x$ 

```

### 5.1.2 Insert

```
INSERT( $T, k$ )
1   $x = T.root$ 
2  if FIND( $x, k$ ) == NIL
3      allocate a new node  $z$ 
4       $z.key = k$ 
5       $z.left = \text{NIL}$ 
6       $z.right = \text{NIL}$ 
7       $y = \text{NIL}$ 
8      while  $x \neq \text{NIL}$ 
9           $y = x$ 
10         if  $z.key < x.key$ 
11              $x = x.left$ 
12         else
13              $x = x.right$ 
14      $z.p = y$ 
15     if  $y == \text{NIL}$ 
16          $T.root = z$       // tree  $T$  was empty
17     elseif  $z.key < y.key$ 
18          $y.left = z$ 
19     else
20          $y.right = z$ 
```

### 5.1.3 Successor

```
MINIMUM( $x$ )
1  while  $x.left \neq \text{NIL}$ 
2       $x = x.left$ 
3  return  $x$ 
```

```
SUCCESSOR( $x$ )
1  if  $x.right \neq \text{NIL}$ 
2      return MINIMUM( $x.right$ )
3   $y = x.p$ 
4  while  $y \neq \text{NIL}$  and  $x == y.right$ 
5       $x = y$ 
6       $y = y.p$ 
7  return  $y$ 
```

### 5.1.4 Delete

```
TRANSPLANT( $T, u, v$ )
1  if  $u.p == \text{NIL}$ 
2       $T.root = v$ 
3  elseif  $u == u.p.left$ 
4       $u.p.left = v$ 
5  else
6       $u.p.right = v$ 
7  if  $v \neq \text{NIL}$ 
8       $v.p = u.p$ 
```

```

DELETE( $T, z$ )
1  if  $z.left == \text{NIL}$ 
2    TRANSPLANT( $T, z, z.right$ )
3  elseif  $z.right == \text{NIL}$ 
4    TRANSPLANT( $T, z, z.left$ )
5  else
6     $y = \text{MINIMUM}(z.right)$ 
7    if  $y.p \neq z$ 
8      TRANSPLANT( $T, y, y.right$ )
9       $y.right = z.right$ 
10      $y.right.p = y$ 
11   TRANSPLANT( $T, z, y$ )
12    $y.left = z.left$ 
13    $y.left.p = y$ 

```

## 5.2 Rotations

LEFT-ROTATE( $T, x$ )

```

1   $y = x.right$           // set  $y$ 
2   $x.right = y.left$       // turn  $y$ 's left subtree into  $x$ 's right subtree
3  if  $y.left \neq T.nil$ 
4     $y.left.p = x$ 
5   $y.p = x.p$             // link  $x$ 's parent to  $y$ 
6  if  $x.p == T.nil$ 
7     $T.root = y$ 
8  elseif  $x == x.p.left$ 
9     $x.p.left = y$ 
10 else
11    $x.p.right = y$ 
12    $y.left = x$           // put  $x$  on  $y$ 's left
13    $x.p = y$ 

```

RIGHT-ROTATE( $T, y$ )

```

1   $x = y.left$           // set  $x$ 
2   $y.left = x.right$       // turn  $x$ 's right subtree into  $y$ 's left subtree
3  if  $x.right \neq T.nil$ 
4     $x.right.p = y$ 
5   $x.p = y.p$             // link  $y$ 's parent to  $x$ 
6  if  $y.p == T.nil$ 
7     $T.root = x$ 
8  elseif  $y == y.p.left$ 
9     $y.p.left = x$ 
10 else
11    $y.p.right = x$ 
12    $x.right = y$           // put  $y$  on  $x$ 's right
13    $y.p = x$ 

```

**Complexity:**  $O(1)$

### 5.3 AVL trees

**AVL invariant:**  $bal(x) = ht(x.r) - ht(x.l)$  at all nodes is -1, 0 or 1

**Insert:** BST-INSERT first, then find the **lowest unbalanced node** and use  $O(1)$  (one or two) rotations to restore balance.

**Delete:** BST-DELETE first, then **bottom up** find unbalanced node and use  $O(h)$  rotations to restore balance.

**Complexity:**  $O(h) = O(\log n)$

## 6 Hashing

### 6.1 Direct addressing

- Suppose we want to store  $(key, value)$  pairs, where keys come from a finite universe  $U = \{0, 1, \dots, m-1\}$
- Use an array of size  $m$ . All operations take  $O(1)$  time.
  - $\text{INSERT}(k, v)$  Store  $v$  in array position  $k$ .
  - $\text{FIND}(k)$  Return the value in array position  $k$ .
  - $\text{DELETE}(k)$  Clear the value in array position  $k$ .
- Cons:
  - If  $m$  is large, then we need to use a lot of memory. Uses  $O(|U|)$  space.
  - If only need to store few values, lots of space wasted.

### 6.2 Hash tables

- A randomized data structure to efficiently implement a dictionary.
- Support find, insert, and delete operations all in expected  $O(1)$  time, but in the worst case, all operations are  $O(n)$ . The worst case is provably very unlikely to occur.
- Do not support efficient min/max or predecessor/successor functions. All these take  $O(n)$  time on average.
- A practical, efficient alternative to binary search trees if only find, insert and delete needed.
- Consists of the following:
  - A universe  $U$  of keys.
  - An array  $T$  of size  $m$ .
  - A hash function  $h : U \rightarrow \{0, 1, \dots, m - 1\}$ .
- Assuming  $h(k)$  takes  $O(1)$  time to compute, all operations still take  $O(1)$  time. Uses  $O(m)$  space, much less than direct addressing if  $m \ll |U|$ .
  - $\text{INSERT}(k, v)$  Hash key to  $h(k)$ . Store  $v$  in  $T[h(k)]$ .
  - $\text{FIND}(k)$  Return the value in  $T[h(k)]$ .
  - $\text{DELETE}(k)$  Delete the value in  $T[h(k)]$ .
- Collisions unavoidable when  $|U| > m$  by Pigeonhole Principle.

### 6.3 Closed addressing

- Every entry in hash table points to a linked list. Keys that hash to the same location get added to the linked list.
- Suppose the longest list has length  $\hat{n}$ , and average length list is  $\bar{n}$ . Each operation takes worst case  $O(\hat{n})$  time. An operation on a random key takes  $O(\bar{n})$  time.
  - $\text{INSERT}(k)$  Add  $k$  to the linked list in  $T[h(k)]$ .
  - $\text{FIND}(k)$  Search the linked list in  $T[h(k)]$  for  $k$ .
  - $\text{DELETE}(k)$  Delete  $k$  from the linked list in  $T[h(k)]$ .
- Suppose the hash table contains  $n$  items, and has size  $m$ . Call  $\alpha = n/m$  **load factor**. The average time for each operation is  $O(\alpha)$ . However, even with uniform hashing, in the worst case, all keys can hash to the same location, so the worst case performance is still  $O(n)$ .

#### 6.3.1 Heuristic hash functions

- Assume the keys are natural numbers. Convert other data types to numbers.
- **Division method:**  $h(k) = k \bmod m$ , often choose  $m$  a prime number not too close to a power of 2.
- **Multiplication method:**  $h(k) = \lfloor m(kA \bmod 1) \rfloor$ , where  $A$  is a constant in the range  $(0, 1)$ .

#### 6.3.2 Universal hashing

- No matter what the  $n$  input keys are, every operation takes optimal  $O(n/m)$  time in expectation, for a size  $m$  hash table.
- **Universal hash family  $H$ :**  $\forall$  keys  $x \neq y$ ,  $P_{h \in H}[h(x) = h(y)] \leq 1/m$ .
- Let  $S$  be a set of  $n$  keys, and let  $x \in S$ . If  $h \in H$  is chosen at random, then the expected number of  $y \in S$  such that  $h(x) = h(y)$  is  $n/m$ .
- **Constructing a universal hash family**
  - Choose a prime number  $p$  such that  $p > m$ , and all keys  $< p$ .
  - $h_{ab}(k) = ((ak + b) \bmod p) \bmod m$
  - $H_{pm} = \{h_{ab} \mid a \in \{1, 2, \dots, p-1\}, b \in \{0, 1, \dots, p-1\}\}$

### 6.4 Open addressing

Define a new hash function  $h : U \times \{0, 1, \dots, m-1\} \rightarrow \{0, 1, \dots, m-1\}$  and a **probe sequence**  $(h(k, 0), h(k, 1), \dots, h(k, m-1))$ .

### 6.4.1 Operations

HASH-INSERT( $T, k$ )

```

1    $i = 0$ 
2   repeat
3        $j = h(k, i)$ 
4       if  $T[j] == \text{NIL}$  or  $T[j] == \text{DELETED}$ 
5            $T[j] = k$ 
6           return  $j$ 
7   else
8        $i = i + 1$ 
9   until  $i == m$ 
10  error "hash table overflow"

```

HASH-SEARCH( $T, k$ )

```

1    $i = 0$ 
2   repeat
3        $j = h(k, i)$ 
4       if  $T[j] == k$ 
5           return  $j$ 
6        $i = i + 1$ 
7   until  $T[j] == \text{NIL}$  or  $i == m$ 
8   return  $\text{NIL}$ 

```

HASH-DELETE( $T, k$ )

```

1    $T[\text{HASH-SEARCH}(T, k)] = \text{DELETED}$ 

```

Now search time depends both on current number of table entries, and also number of past deleted ones, so **closed addressing** is more common when keys need to be deleted. Since each table entry stores one key, a table of size  $m$  can store at most  $m$  keys. However, since each table entry stores just the key and no pointers, then for the same amount of storage, an open addressing hash table can store more entries than a closed addressing table. Thus, it has lower load factor.

### 6.4.2 Probe sequences

- Two Properties

- $h(k, 0), h(k, 1), \dots, h(k, m - 1)$  covers all table entries  $0, 1, \dots, m - 1$ .
- Given a random key  $k$ ,  $h(k, 0), h(k, 1), \dots, h(k, m - 1)$  is a random sequence in the  $m!$  permutations of  $\{0, \dots, m - 1\}$ .

- Techniques to compute probe sequences

- **Linear probing:**  $h(k, i) = (h'(k) + i) \bmod m$ , where  $h'$  is an ordinary hash function.  $h'(k)$  determines the entire probe sequence, so there's only  $m$  different probe sequences. Poor performance due to **primary clustering**. An empty slot with  $i$  filled slots before it gets filled with probability  $(i + 1)/m$ , so long runs of filled slots, that tend to get even longer.
- **Quadratic probing:**  $h(k, i) = (h'(k) + c_1 i + c_2 i^2) \bmod m$ , where  $c_1, c_2$  are constants. No primary clustering, but **secondary clustering**, where  $h(k_1, 0) = h(k_2, 0)$  implies both probe sequences are equal.

- **Double hashing:**  $h(k, i) = (h_1(k) + ih_2(k)) \bmod m$ .  $h_2(k)$  needs to be relatively prime to  $m$  to make sure entire table searched. One way to ensure this is make  $m$  a power of 2, and  $h_2$  always odd. Can also make  $m$  prime,  $m' = m - 1$ , and set  $h_1(k) = k \bmod m$ ,  $h_2(k) = 1 + (k \bmod m')$ . Since each distinct  $(h_1(k), h_2(k))$  leads to distinct probe sequence, double hashing can produce  $O(m^2)$  probe sequences. Performs quite well in practice.

## 6.5 Perfect hashing

- Ensures **no colisions** for a **fixed set of keys**.
- Allows  $\text{FIND}(k)$  and  $\text{DELETE}(k)$  in  $O(1)$  time. Does not support  $\text{INSERT}(k)$ .
- Uses two levels of universal hashing.
  - Use first layer hash function  $h$  to hash key to a location in  $T$ .
  - Each location  $j$  in  $T$  points to a hash table  $S_j$  with hash function  $h_j$ .
  - If  $n_j$  keys hash to location  $j$ , the size of  $s_j$  is  $m_j = n_j^2$ .
- Suppose we store  $n$  keys in a hash table of size  $m = n^2$  using universal hashing, then there is a  $< 1/2$  probability of collision. If collisions occur, pick another random hash function from the universal family. In expectation, do this twice before finding a hash function causing no collisions.
- Suppose we store  $n$  keys in a hash table of size  $m = n$ . Then the secondary hash tables use space  $E[\sum_{j=0}^{m-1} n_j^2] \leq 2n$ , where  $n_j$  is the number of keys hashing to location  $j$ . Overall the sapce use is  $O(m + \sum_{j=0}^{m-1} n_j^2) = O(m)$ .

## 6.6 Bloom filters

- Can implement a set. It only keeps track of which keys are present, not any values associated to keys.
- Supports insert and find operations. Doesn't support delete operations. Deletes can be done by storing a count of how many keys hashed to that location, and incline/decline the counts when inserting or deleting. But this uses more memory. Also, what if the counts overflow?
- Use less memory than hash tables or other ways of implementing sets.
- **Approximate:** can produce **false positives**, no **false negatives**. False positive probability

$$f = \left[ 1 - \left( 1 - \frac{1}{m} \right)^{nk} \right]^k \approx \left( 1 - e^{-\frac{nk}{m}} \right)^k$$

where  $k$  is the number of hash functions,  $m$  is the size of the table and  $n$  is the number of keys inserted. False positive probability minimized by  $k = \frac{m \ln 2}{n}$ , which leads to  $f = (1/2)^k \approx 0.6185 \frac{m}{n}$ , where  $m/n$  is the average number of bits per item. So error rate decreases exponentially in space usage.

- **Neat trick** Given Bloom filters for sets  $S_1, S_2$ , we can create Bloom filter for  $S_1 \cap S_2$  and  $S_1 \cup S_2$  just by bitwise ANDing or ORing  $S_1$  and  $S_2$ 's filters.
- Consists of:
  - An array  $A$  of size  $m$ , initially all 0's.
  - $k$  **independent** hash functions  $h_1, \dots, h_k : U \rightarrow \{1, \dots, m\}$ .

BLOOM-FILTER-INSERT( $A, H, x$ )

```

1  for each  $h \in H$ 
2       $A[h(x)] = 1$ 
```

BLOOM-FILTER-SEARCH( $A, H, x$ )

```

1  for each  $h \in H$ 
2      if  $A[h(x)] == 0$ 
3          return FALSE
4  return TRUE
```

## 7 Divide and conquer

### 7.1 Multiplication

#### 7.1.1 Gauss's method for multiplying complex numbers

$(a + bi)(c + di) = x + yi$ , where  $x = ac - bd$ ,  $y = ad + bc = (a + b)(c + d) - ac - bd$

4 multiplications + 2 additions  $\rightarrow$  3 multiplications + 5 additions

#### 7.1.2 Karatsuba multiplication

To multiply binary numbers  $a$  and  $b$ , split their bits in half,  $a = 2^{n/2} \cdot a_1 + a_0$ ,  $b = 2^{n/2} \cdot b_1 + b_0$ , then  $ab = 2^n \cdot a_1 b_1 + 2^{n/2} \cdot (a_1 b_0 + a_0 b_1) + a_0 b_0 = 2^n \cdot a_1 b_1 + 2^{n/2} \cdot ((a_1 + a_0)(b_1 + b_0) - a_1 b_1 - a_0 b_0) + a_0 b_0$ .

**Complexity:**  $S(n) = 3S(n/2) + O(n)$ ,  $S(n) = \Theta(n^{\log 3}) = O(n^{1.59})$

#### 7.1.3 Strassen's algorithm for block matrix multiplication

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \times \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

$$P_1 = A_{11} \times (B_{12} - B_{22}) \quad P_2 = (A_{11} + A_{12}) \times B_{22} \quad P_3 = (A_{21} + A_{22}) \times B_{11}$$

$$P_4 = A_{22} \times (B_{21} - B_{11}) \quad P_5 = (A_{11} + A_{22}) \times (B_{11} + B_{22})$$

$$P_6 = (A_{12} - A_{22}) \times (B_{21} + B_{22}) \quad P_7 = (A_{11} - A_{21}) \times (B_{11} + B_{12})$$

$$C_{11} = P_5 + P_4 - P_2 + P_6 \quad C_{12} = P_1 + P_2 \quad C_{21} = P_3 + P_4 \quad C_{22} = P_5 + P_1 - P_3 - P_7$$

**Complexity:**  $S(n) = 7S(n/2) + O(n^2)$ ,  $S(n) = \Theta(n^{\log 7}) = O(n^{2.81})$

#### 7.1.4 FFT for polynomial multiplication

Details see Recitation2.pdf

**Complexity:**  $O(n \log n)$

## 7.2 Counting inversions

COUNT-CROSSING-INVERSIONS( $A, p, q, r$ )

```

1   $n_1 = q - p + 1$ 
2   $n_2 = r - q$ 
3  let  $L[0..n_1 - 1]$  and  $R[0..n_2 - 1]$  be new arrays
4   $num = 0$ 
5  for  $i = 0$  to  $n_1 - 1$ 
6     $L[i] = A[p + i]$ 
7  for  $j = 0$  to  $n_2 - 1$ 
8     $R[j] = A[q + 1 + j]$ 
9   $i = 0$ 
10  $j = 0$ 
11  $k = p$ 
12 while  $i < n_1$  and  $j < n_2$ 
13  if  $L[i] > R[j]$ 
14     $num = num + 1$ 
15     $A[k] = R[j]$ 
16     $j = j + 1$ 
17  else
18     $num = num + j$ 
19     $A[k] = L[i]$ 
20     $i = i + 1$ 
21     $k = k + 1$ 
22  if  $j == n_2$ 
23     $num = num + (n_1 - i - 1) * n_2$ 
24    for  $r = 0$  to  $n_1 - i - 1$ 
25       $A[k + r] = L[i + r]$ 
26  else
27     $num = num - j$ 
28    for  $r = 0$  to  $n_2 - j - 1$ 
29       $A[k + r] = R[j + r]$ 
30  return  $num$ 
```

COUNT-INVERSIONS( $A, p, r$ )

```

1  if  $p == r$ 
2    return 0
3  if  $r == p + 1$ 
4    if  $A[p] > A[r]$ 
5      return 1
6  else
7    return 0
8   $q = \lfloor (p + r / 2) \rfloor$ 
9  return COUNT-INVERSIONS( $A, p, q$ ) + COUNT-INVERSIONS( $A, q + 1, r$ ) +
   COUNT-CROSSING-INVERSIONS( $A, p, q, r$ )
```

**Complexity:**  $T(n) = 2T(n/2) + O(n)$ ,  $T(n) = O(n \log n)$

### 7.3 Maximum subarray

FIND-MAX-CROSSING-SUBARRAY( $A, low, mid, high$ )

```

1  left-sum =  $-\infty$ 
2  sum = 0
3  for  $i = mid$  downto  $low$ 
4      sum = sum +  $A[i]$ 
5      if sum > left-sum
6          left-sum = sum
7          max-left = i
8  for  $j = mid + 1$  to  $high$ 
9      sum = sum +  $A[j]$ 
10     if sum > right-sum
11         right-sum = sum
12         max-right = j
13 return (max-left, max-right, left-sum + right-sum)

```

FIND-MAXIMUM-SUBARRAY( $A, low, high$ )

```

1  if  $high == low$ 
2      return ( $low, high, A[low]$ )
3  else
4      mid =  $\lfloor (low + high)/2 \rfloor$ 
5      ( $left-low, left-high, left-sum$ ) = FIND-MAXIMUM-SUBARRAY( $A, low, mid$ )
6      ( $right-low, right-high, right-sum$ ) = FIND-MAXIMUM-SUBARRAY( $A, mid + 1, high$ )
7      ( $cross-low, cross-high, cross-sum$ ) = FIND-MAX-CROSSING-SUBARRAY( $A, low, mid, high$ )
8      if left-sum  $\geq$  right-sum and left-sum  $\geq$  cross-sum
9          return ( $left-low, left-high, left-sum$ )
10     elseif right-sum  $\geq$  left-sum and right-sum  $\geq$  cross-sum
11         return ( $right-low, right-high, right-sum$ )
12     return ( $cross-low, cross-high, cross-sum$ )

```

### 7.4 Closest point pair

## 8 Greedy algorithms

### 8.1 Interval scheduling

### 8.2 Interval coloring

### 8.3 Huffman coding

HUFFMAN( $C$ )

```

1  n = | $C$ |
2  Q =  $C$ 
3  for  $i = 1$  to  $n - 1$ 
4      allocate a new node  $z$ 
5       $z.left = x = \text{EXTRACT-MIN}(Q)$ 
6       $z.right = y = \text{EXTRACT-MIN}(Q)$ 
7       $z.freq = x.freq + y.freq$ 
8      INSERT( $Q, z$ )
9  return EXTRACT-MIN( $Q$ )    // return the root of the tree

```

## 9 Graph algorithms

### 9.1 Graphs

#### 9.1.1 Basic definitions

- $G = (V, E)$ , where  $V$  is the set of **vertices (nodes)**,  $E \subseteq V \times V$  is the set of **edges**.

$$|E| = O(|V|^2)$$

- An edge  $e = (u, v)$  consists of two **endpoints**,  $u$  and  $v$  are **adjacent (neighbors)**. Two edges are **adjacent** if they share an endpoint.
- A **path** is a set of adjacent edges. A graph is **connected** if there's a path between any two nodes. If a graph is not connected, it consists of a set of **connected components**. **Distance** between two nodes is the smallest number of edges between the nodes in any path.
- A **cycle** is a path that starts and ends at the same vertex. A graph is **acyclic** if it doesn't have any cycles.
- **Degree** of a vertex  $v$ ,  $d(v)$ , is the number of neighbors  $v$  has.
- **Handshaking theorem:** Let  $G = (V, E)$  be an undirected graph,  $\sum_{v \in V} d(v) = 2|E|$ .

#### 9.1.2 Types of graphs

- **directed** (follow edge in direction indicated), **undirected** (follow edge in either direction)
- **simple** (no parallel edges between nodes, no self loops), **cyclic** (following edges from a node, can get back to the same node)
- **explicit** (edges are listed), **implicit** (edges generated on the fly, e.g. in search trees)
- **weighted** (edges have values attached), **unweighted** (all edges have weight 1)
- **dense** (many edges in graph), **sparse** ( $O(n)$  edges in graph with  $n$  nodes)
- **embedded** (vertices have coordinates, e.g., in Euclidean space), **topological** (edges only indicate connections)
- **labeled** (nodes have labels (names) attached)

#### 9.1.3 Trees

- A **tree** is a connected, acyclic undirected graph.
- A tree with  $n$  vertices has  $n - 1$  edges.
- Removing any edge from a tree **disconnects** it into two **subtrees**.
- Adding any edge to a tree creates a **cycle**.
- There is a **unique path** between any two nodes in a tree.
- A **forest** is a collection of trees.

## 9.2 Representations of graphs

### 9.2.1 Adjacency list

An array of linked lists. Each array location represents a vertex. Linked list at a location gives (directed) neighbors of vertex. Size is  $O(|E|)$ , i.e., proportional to number of edges. Uses less memory if  $G$  doesn't have many edges, but to see if  $(i, j)$  is an edge, need to scan  $i$ 's list in  $O(|V|)$  time.

### 9.2.2 Adjacency matrix

$|V|$  rows and  $|V|$  columns. Entry  $(i, j) = 1$  if  $i$  is adjacent to  $j$  in an undirected graph and if there is an edge from  $i$  to  $j$  in a directed graph, 0 otherwise. Size is  $O(|V|^2)$ . Uses more memory, but can check if  $(i, j)$  is an edge in  $O(1)$  time.

## 9.3 BFS

Use a queue  $Q$  storing nodes. Each node  $v$  has three fields:

- $v.d$  giving  $v$ 's distance to  $s$
- $v.color \in \{\text{WHITE, GRAY, BLACK}\}$  showing whether  $v$  is **undiscovered**, **being processed** or **processed**
- $v.\pi$  giving  $v$ 's parent

$\text{BFS}(G, s)$

```

1  for each vertex  $u \in G. V - \{s\}$ 
2       $u.color = \text{WHITE}$ 
3       $u.d = \infty$ 
4       $u.\pi = \text{NIL}$ 
5       $s.color = \text{GRAY}$ 
6       $s.d = 0$ 
7       $s.\pi = \text{NIL}$ 
8       $Q = \emptyset$ 
9      ENQUEUE( $Q, s$ )
10     while  $Q \neq \emptyset$ 
11          $u = \text{DEQUEUE}(Q)$ 
12         for each  $v \in G. Adj[u]$ 
13             if  $v.color == \text{WHITE}$ 
14                  $v.color = \text{GRAY}$ 
15                  $v.d = u.d + 1$ 
16                  $v.\pi = u$ 
17                 ENQUEUE( $Q, v$ )
18          $u.color = \text{BLACK}$ 

```

**Complexity:** Using an adjacency list, for each vertex,  $O(1)$  to enqueue it and  $O(d(v))$  to examine all its neighbors.  $T(n) = \sum_{v \in V} O(1 + d(v)) = O(V + \sum_{v \in V} d(v)) = O(V + 2E) = O(V + E)$ .

The graph  $(V, \{(v, v.\pi)\}_{v \in V})$  is a forest called the **BFS forest**. If all nodes are reachable from  $s$ , then the forest has one tree. Each tree in the forest is connected, and each node has a unique path to the root of the tree. Say the nodes with  $v.d = L$  are at **layer  $L$**  of the BFS tree. All these nodes are **distance  $L$**  from the **source node  $s$** .

## 9.4 DFS

Visit unvisited neighbors of a vertex when possible, otherwise **backtrack** to its parent. Use **timestamps** to track when events occur. The timestamps are used in many other algorithms using DFS as a subroutine.

Each node  $v$  has four fields:

- $v.\text{color} \in \{\text{WHITE}, \text{GRAY}, \text{BLACK}\}$  showing whether  $v$  is **undiscovered**, **being processed** or **processed**
- $v.\pi$  giving  $v$ 's parent
- $v.d$  giving the time when  $v$  is **discovered**
- $v.f$  giving the time when  $v$  is **finished**, i.e., itself and its non-parent neighbors have been processed

$\text{DFS}(G)$

```

1  for each vertex  $u \in G.V$ 
2       $u.\text{color} = \text{WHITE}$ 
3       $u.\pi = \text{NIL}$ 
4       $time = 0$ 
5  for each vertex  $u \in G.V$ 
6      if  $u.\text{color} == \text{WHITE}$ 
7           $\text{DFS-VISIT}(G, u)$ 

```

$\text{DFS-VISIT}(G, u)$

```

1   $time = time + 1$            // white vertex  $u$  has just been discovered
2   $u.d = time$ 
3   $u.\text{color} = \text{GRAY}$ 
4  for each  $v \in G.\text{Adj}[u]$       // explore edge  $(u, v)$ 
5      if  $v.\text{color} == \text{WHITE}$ 
6           $v.\pi = u$ 
7           $\text{DFS-VISIT}(G, v)$ 
8   $u.\text{color} = \text{BLACK}$            // blacken  $u$ ; it is finished
9   $time = time + 1$ 
10  $u.f = time$ 

```

**Complexity:** Using an adjacency list, similar as BFS,  $O(V + E)$ .

The graph  $(V, \{(v, v.\pi)\}_{v \in V})$  is a forest called the **DFS forest**.

**Nesting property** of discovery and finish time

$u$  is a descendant of  $v \implies [u.d, u.f] \subset [v.d, v.f]$

$u$  and  $v$  are not descendants of each other  $\implies [u.d, u.f]$  and  $[v.d, v.f]$  are **disjoint**

**Classify** each edge  $e = (u, v) \in E$  as

**tree edge** ( $e$  is in the DFS tree)

**back edge** ( $v$  is an ancestor of  $u$  in the DFS tree)

**forward edge** ( $v$  is a descendant of  $u$  in the DFS tree)

**cross edge** (otherwise)

## 9.5 Applications of BFS and DFS

### 9.5.1 Connected components

Start with an arbitrary node  $s$  in  $V$  and run **BFS** or **DFS** from  $s$ . Let  $V'$  be all the black nodes.  $V$  forms one connected component. If  $V - V' \neq \emptyset$ , set  $V = V - V'$ , and repeat the above process to find more connected components.

**Complexity:**  $O(V + E)$ . Each node  $v$  is processed once, i.e., takes  $O(1 + d(v))$  time.

### 9.5.2 Testing bipartiteness

Is-BIPARTITE( $G$ )

```

1   $s = \text{any vertex } v \in G. V$ 
2   $\text{BFS}(G, s)$ 
3  for each vertex  $u \in G. V$ 
4    if  $u. d$  is even
5       $u. color = \text{RED}$ 
6  for each edge  $(u, v) \in G. E$ 
7    if  $u. color == v. color$ 
8      return FALSE
9  return TRUE

```

**Complexity:**  $O(V + E)$

### 9.5.3 Satisfiability (2SAT)

Write expressions in **conjunctive normal form** (CNF), i.e., as the ANDs of ORs. In  $k$ -CNF, each clause has  $k$  literals. Brute force solution  $O(2^k)$ .

Given a 2-CNF formula  $\phi$ , convert it into a directed graph  $G$ . There are two nodes, called  $x$  and  $\neg x$ , for each variable  $x$  in  $\phi$ . For each clause of the form  $(\neg a \vee b)$  in  $\phi$ , create an edge  $(a, b)$ .

If there's a path from a node  $u$  to  $v$ , then there's also a path from  $\neg v$  to  $\neg u$ .

Suppose there is a variable  $x$  such that there is a path from  $x$  to  $\neg x$  and a path from  $\neg x$  to  $x$  in  $G$ . Then  $\phi$  is not satisfiable.

2-SAT( $G$ )

```

1  for each vertex  $x \in G. V$ 
2     $x. value = \text{NIL}$ 
3    if EXIST-PATH( $x, \neg x$ ) and EXIST-PATH( $\neg x, x$ )
4      return FALSE
5   $Q = G. V$ 
6  while  $Q \neq \emptyset$ 
7     $x = \text{any vertex such that EXIST-PATH}(x, \neg x) == \text{FALSE}$ 
8     $R = \{\text{all the vertices reachable from } x \text{ found by BFS or DFS}\}$ 
9    for each vertex  $v \in R$ 
10   if  $v. value == \text{NIL}$ 
11      $v. value = \text{TRUE}$ 
12      $Q = Q - \{v\}$ 
13   if  $\neg v. value == \text{NIL}$ 
14      $\neg v. value = \text{FALSE}$ 
15      $Q = Q - \{\neg v\}$ 
16 return TRUE

```

**Complexity:**  $O(n(m + n))$ , where  $n, m$  are the number of literals and clauses, resp.

### 9.5.4 8 queens puzzle

- Use a directed graph to search for solutions.
- Each vertex represents a legal placement of  $k \leq 8$  queens. The root of the tree represents the empty board. At most  $8^8$  nodes, actually far less.

- Edge  $(u, v)$  means placement  $v$  is an extension of  $u$ .
- Nodes on level 8 of solution graph are solutions to puzzle.
- Algorithm
  - Run DFS, starting from the root node (empty board).
  - If current node has  $k > 0$  child nodes, run DFS from each child node in order. After all DFS's finished, backtrack to parent node.
  - If node were exploring has 0 child nodes, backtrack to parent.
  - If we reach a level 8 node, record it as a solution.
- Uses  $O(1)$  memory. Only need to keep track of the parent nodes in the DFS branch were searching.
- Huge time complexity since DFS needs to go through all nodes of the solution tree, but still much faster than brute force search.

#### 9.5.5 Solving a maze

- Represent the maze by an undirected graph.
- Nodes are **junctions**, where multiple paths branch off. Also two nodes for the start and finish.
- Edges between nodes that dont have intermediate junctions.
- Algorithm
  - Run DFS on graph from the start node.
  - If we visit finish node, chain of parent nodes back to start node is maze solution.
  - If DFS returns without visiting finish node, maze has no solution.

#### 9.5.6 Partial orders

- A **partial order** is an (incomplete) ordering on a set.
- Can be represented by a directed graph, where vertices are the elements in the set, and an edge  $(v, w)$  means  $v < w$  in the order.
- Graph must be **acyclic** for the partial order to be valid, called a **directed acyclic graph (DAG)**.
- **Topological sort** of the DAG representing the partial order: Find a **total order** consistent with the partial order. There may be multiple consistent orderings, we can return any one.
- Suppose  $v < w$  in a partial order corresponding to  $G$ . Then  $v.f > w.f$ .

**TOPOLOGICAL-SORT( $G$ )**

- 1 call DFS( $G$ ) to compute finishing times  $v.f$  for each vertex  $v$
- 2 as each vertex is finished, insert it onto the front of a linked list
- 3 **return** the linked list of vertices

**Complexity:**  $O(V + E)$

### 9.5.7 Strongly connected components

- A maximal set of nodes  $C$  in a directed graph  $G$  such that for all  $v, w \in C$ , both paths from node  $v$  to  $w$  and from  $w$  to  $v$  exist.
- Suppose we contract all the nodes in each SCC of  $G$  to a single node. This forms a component graph  $G'$ , then  $G'$  is a DAG.
- The SCCs output in the second DFS occur in the **topological sort order** of the component graph.

STRONGLY-CONNECTED-COMPONENTS( $G$ )

- 1 call DFS( $G$ ) to compute finishing time  $u.f$  for each vertex  $u$
- 2 compute  $G^T$  // with all edges in  $G$  reversed
- 3 call DFS( $G^T$ ), but in the main loop of DFS, consider the vertices in order of decreasing  $u.f$  (as computed in line 1)
- 4 output the vertices of each tree in the depth-first forest formed in line 3 as a separate strongly connected component

**Complextiy:**  $O(V + E) = 2O(V + E) + O(E)$

### 9.5.8 Biconnected components

- **Articulation point:** a node whose removal separates the graph into multiple parts.
- **Biconnected graph:** graph without articulation points, are either a single edge or have  $\geq 2$  **edge disjoing paths** between any pair of nodes.
- **Biconnected component:** a maximal biconnected subgraph.

BICONNECTED-COMPONENTS( $G$ )

- 1  $S = \emptyset$  //  $S$  is a stack different from the one used to implement DFS itself
- 2  $C = \emptyset$  //  $C$  is the set of edges in a biconnected component
- 3  $SCC = \emptyset$
- 4 **during** DFS( $G$ )
  - 5 **for** each edge  $e$  encountered
  - 6  $S.push(e)$
  - 7 **if** DFS backtracks to an edge  $(u, v)$  where  $u$  is an articulation point
  - 8 **repeat**
  - 9  $e = S.pop()$
  - 10  $C = C \cup \{e\}$
  - 11 **until**  $e == (u, v)$
  - 12  $SCC = SCC \cup \{C\}$
  - 13  $C = \emptyset$
- 14 **return** SCC

**Complexity:**  $O(V + E)$  if the articulation points are known.

## 9.6 MST

Start with no edges, and add one edge per stage. At every stage, edges form a **partial MST**, which does not have any **cycles** and is a **forest**. Stop when adding any edge creates a cycle.

**MST principle:** Let  $G$  be a weighted graph. Suppose  $H$  is a **subset of some MST** of  $G$ .  $C$  is **any connected component** of  $H$ .  $S$  is set of edges connecting  $C$  to **any other component** of  $H$ .  $e$  is the **min weight edge** in  $S$ . Then  $H \cup \{e\}$  is also a **subset of some MST** of  $G$ .

### 9.6.1 Prim's algorithm

MST-PRIM( $G, w$ )

```

1 //  $H$  contains the nodes in the partial MST
2  $H = \{\text{an arbitrary node } v \in G. V\}$ 
3 //  $F$  contains the edges in the partial MST
4  $F = \emptyset$ 
5 //  $R$  contains the edges touching  $F$  and not creating a cycle, initially containing all edges from  $G. V$ 
6  $R = \emptyset$ 
7 for each vertex  $u \in G. Adj[v]$ 
8      $R = R \cup \{(u, v)\}$ 
9 while  $H \neq G. V$ 
10     $e = \text{GET-MIN-WEIGHT-EDGE}(R)$ 
11    let  $e = (u, v)$ , with  $v \notin H$ 
12     $H = H \cup \{v\}$ 
13     $F = F \cup \{e\}$ 
14    for each vertex  $u \in G. Adj[v]$ 
15      if  $u \notin H$ 
16           $R = R \cup \{(u, v)\}$ 
```

**Complexity:**  $O(E \log V) = O(1) + O(V) + O(V \log V) + O(E \log V)$ ,  $R$  implemented by priority queue.

### 9.6.2 Kruskal's algorithm

MST-KRUSKAL( $G, w$ )

```

1  $A = \emptyset$ 
2 for each vertex  $v \in G. V$ 
3     MAKE-SET( $v$ )
4 sort the edges of  $G. E$  into nondecreasing order by weight  $w$ 
5 for each edge  $(u, v) \in G. E$ , taken in nondecreasing order by weight
6   if FIND-SET( $u$ )  $\neq$  FIND-SET( $v$ )
7      $A = A \cup \{(u, v)\}$ 
8     UNION( $u, v$ )
9 return  $A$ 
```

**Complexity:**  $O(E \log V) = O(V) + O(E \log V) + EO(\log V)$

### 9.6.3 Arborescences

- **Directed spanning trees.** Given a directed graph and a root vertex  $r$ , there is a unique directed path from  $r$  to every other vertex.
- BFS or DFS can find some arborescence, if one exists.
- A subgraph  $T$  of  $G$  is an arborescence rooted at node  $r \iff T$  has no directed cycles and every node  $v \neq r$  has exactly one incoming edge
- **Edge reweighting**
  - For each vertex  $v$ , let  $\delta(v)$  be the minimum cost of any incoming edge.
  - For each of its incoming edges  $(u, v)$ , set  $w^*(u, v) = w(u, v) - \delta(v)$ .
  - Call the reweighted graph  $G'$ . All edge weights in  $G'$  are nonnegative.

- An arborescence is minimum weight for  $G'$  if and only if it is minimum weight for  $G$ .
- Each vertex in  $G'$  has at least one incoming edge with weight 0.
- If a set of weight 0 edges form an arborescence, then it has min weight.

$\text{ARBORESCENCE}(G, w)$

```

1 for each vertex  $v \in G. V$ 
2    $v.\delta = \min(\text{weight of incoming edge } (u, v))$ 
3   for each incoming edge  $(u, v)$ 
4      $w(u, v) = w(u, v) - v.\delta$ 
5    $Q = \emptyset$ 
6   for each vertex  $v \in G. V$ 
7      $Q = Q \cup \{\text{one of } (u, v) \text{ such that } w(u, v) = 0\}$ 
8   if  $Q$  is an arborescence
9     return  $Q$ 
10   $C = \text{a directed cycle found by BFS or DFS}$ 
11   $v = \text{CONTRACT}(C)$ 
12  return  $\text{ARBORESCENCE}(G - C + \{v\}, w) + C - \{\text{one edge of } C\}$ 

```

**Complexity:**  $O(VE)$

## 9.7 Single source shortest paths (SSSP)

### 9.7.1 Relaxations

$\text{INITIALIZE-SINGLE-SOURCE}(G, s)$

```

1 for each vertex  $v \in G. V$ 
2    $v.d = \infty$ 
3    $v.\pi = \text{NIL}$ 
4    $s.d = 0$ 

```

$\text{RELAX}(u, v, w)$

```

1 if  $v.d > u.d + w(u, v)$ 
2    $v.d = u.d + w(u, v)$ 
3    $v.\pi = u$ 

```

### 9.7.2 Negative weight cycles

- A cycle in the graph such that the sum of all weights on the cycle is negative.
- If a graph has a negative weight cycle reachable from the source, then shortest paths are not well defined, because we can repeatedly go around the cycle to get arbitrarily short paths.

### 9.7.3 Properties of shortest paths

- Suppose a graph doesn't contain any negative weight cycles, and all nodes are reachable from the source  $s$ .
- **Triangle inequality:** The length of a shortest path from  $s$  to  $v$ ,  $\delta(s, v) \leq \delta(s, u) + w(u, v)$
- **Shortest subpaths:** Let  $p = \langle (s, v_0), (v_0, v_1), \dots, (v_k, v) \rangle$  be a shortest path from  $s$  to some node  $v$ , then for every  $v_i \in \{v_0, \dots, v_k\}$ ,  $\langle (s, v_0), (v_0, v_1), \dots, (v_{i-1}, v_i) \rangle$  is a shortest path from  $s$  to  $v_i$ .

- **Shortest path tree:** There exists a tree  $T$  rooted at  $s$  such that the shortest path from  $s$  to any node  $v$  lies in  $T$ . Each node  $v$  has a parent in the tree. By following parent pointers starting from  $v$ , we find shortest path from  $s$  to  $v$ .

#### 9.7.4 Bellman-Ford algorithm

Returns FALSE if there is a negative weight cycle reachable from  $s$  in the graph. Settles nodes in **order of number of edges** on the shortest path.

```
BELLMAN-FORD( $G, w, s$ )
1 INITIALIZE-SINGLE-SOURCE( $G, s$ )
2 for  $i = 1$  to  $|G.V| - 1$ 
3   for each edge  $(u, v) \in G.E$ 
4     RELAX( $u, v, w$ )
5   for each edge  $(u, v) \in G.E$ 
6     if  $v.d > u.d + w(u, v)$ 
7       return FALSE
8 return TRUE
```

**Complexity:**  $O(VE)$

#### 9.7.5 Dijkstra's algorithm

Assumes all weights are nonnegative. Settles nodes in **order of distance** of node from source.  $Q$  is a priority queue.

```
DIJKSTRA( $G, w, s$ )
1 INITIALIZE-SINGLE-SOURCE( $G, s$ )
2  $S = \emptyset$ 
3  $Q = G.V$ 
4 while  $Q \neq \emptyset$ 
5    $u = \text{EXTRACT-MIN}(Q)$ 
6    $S = S \cup \{u\}$ 
7   for each vertex  $v \in G.Adj[u]$ 
8     RELAX( $u, v, w$ )
```

**Complexity:**  $O(E \log V)$  using a binary (min) heap,  $O(V \log V + E)$  using a Fibonacci heap

### 9.8 All pairs shortest paths (APSP)

- Want to find the shortest paths from each node to all the other nodes.
- Represent this by an  $n \times n$  matrix  $L$ , where  $L_{i,j}$  is the length of shortest path from  $i$  to  $j$ , and  $n = |V|$ .
- Can also output a predecessor matrix  $\Pi$ , where  $\Pi_{i,j}$  is the last node before  $j$  on the shortest path from  $i$ .
- Assume in following that there are no negative weight cycles. Algorithms can be modified to detect negative weight cycles.
- Can solve APSP by running SSSP  $n$  times.
  - If all weights are nonnegative, running Dijkstra  $|V|$  times takes  $O(V^2 \log V + VE)$  time.
  - For general weights, can run Bellman-Ford  $|V|$  times, in  $O(V^2 E) = O(V^4)$  time.

### 9.8.1 Recursive shortest paths

- Given nodes  $i, j$ , and  $1 \leq m < n$ , let  $l_{i,j}^{(m)}$  be the length of the shortest path from  $i$  to  $j$  using at most  $m$  edges, or  $\infty$  if there is no such path. Set  $l_{i,j}^{(0)} = 0$  for  $i = j$ , and  $l_{i,j}^{(0)} = \infty$  otherwise.
- $l_{i,j}^{(m)} = \min_{1 \leq k \leq n} \{l_{i,k}^{(m-1)} + w(k,j)\}$
- The shortest path distances are given by  $\{l_{i,j}^{(n-1)}\}_{i,j}$

### 9.8.2 Matrix multiplication APSP

EXTEND-SHORTEST-PATHS( $L, W$ )

```

1   $n = L.\text{rows}$ 
2  let  $L' = (l'_{ij})$  be a new  $n \times n$  matrix
3  for  $i = 1$  to  $n$ 
4    for  $j = 1$  to  $n$ 
5       $l'_{ij} = \infty$ 
6      for  $k = 1$  to  $n$ 
7         $l'_{ij} = \min(l'_{ij}, l_{ik} + w_{kj})$ 
8  return  $L'$ 
```

APSP( $W$ )

```

1   $n = W.\text{rows}$ 
2   $L^{(1)} = W$ 
3   $m = 1$ 
4  while  $m < n - 1$ 
5    let  $L^{(2m)}$  be a new  $n \times n$  matrix
6     $L^{(2m)} = \text{EXTEND-SHORTEST-PATHS}(L^{(m)}, L^{(m)})$ 
7     $m = 2m$ 
8  return  $L^{(m)}$ 
```

**Complexity:** similar to matrix multiplication,  $O(n^3 \log n)$  using **repeated doubling**

### 9.8.3 Floyd-Warshall algorithm

Consider the shortest path from  $i$  to  $j$  using only nodes  $1, \dots, k$  as **intermediate nodes**, which are nodes on the path besides  $i$  and  $j$ .

Let  $d_{i,j}^{(k)}$  be the minimum distance from  $i$  to  $j$  using  $1, \dots, k$  as intermediate nodes. The shortest path distance from  $i$  to  $j$  equals  $d_{i,j}^{(n)}$ . Define  $d_{i,j}^{(0)} = w_{i,j}$ .

FLOYD-WARSHALL( $W$ )

```

1   $n = W.\text{rows}$ 
2   $D^{(0)} = W$ 
3  for  $k = 1$  to  $n$ 
4    let  $D^{(k)} = (d_{ij}^{(k)})$  be a new  $n \times n$  matrix
5    for  $i = 1$  to  $n$ 
6      for  $j = 1$  to  $n$ 
7         $d_{ij}^{(k)} = \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$ 
8  return  $D^{(n)}$ 
```

**Complexity:**  $O(n^3)$ , can also compute the predecessor matrix in  $O(n^3)$  time.

#### 9.8.4 Johnson's algorithm

**JOHNSON( $G, w$ )**

```

1  compute  $G'$ , where  $G'.V = G.V \cup \{s\}$ ,  $G'.E = G.E \cup \{(s, v) : v \in G.V\}$ , and  $w(s, v) = 0$  for all  $v \in G.V$ 
2  if BELLMAN-FORD( $G', w, s$ ) == FALSE
3      print "the input graph contains a negative-weight cycle"
4  else
5      for each vertex  $v \in G'.V$ 
6          set  $h(v)$  to the value of  $\delta(s, v)$  computed by the Bellman-Ford algorithm
7      for each edge  $(u, v) \in G'.E$ 
8           $\hat{w}(u, v) = w(u, v) + h(u) - h(v)$ 
9      let  $D = (d_{uv})$  be a new  $n \times n$  matrix
10     for each vertex  $u \in G.V$ 
11         run DIJKSTRA( $G, \hat{w}, u$ ) to compute  $\hat{\delta}(u, v)$  for all  $v \in G.V$ 
12         for each vertex  $v \in G.V$ 
13              $d_{uv} = \hat{\delta}(u, v) + h(v) - h(u)$ 
14     return  $D$ 
```

**Complexity:**  $O(V^2 \log V + VE) = O(VE) + O(V^2 \log V + VE)$