1 Sorting

1.1 Insertion sort

Best case : sorted input $\Theta(n)$

Worst case : reverse sorted $\Theta(n^2)$

```
for i in range(1, len(l)):
    val = l[i]
    j = i - 1
    while j ⩾ 0 and l[j] > val:
        l[j + 1] = l[j]
        j -= 1
    l[j + 1] = val
```

1.2 Merge sort

Runtime complexity : $\Theta(n \log n)$

Not in-place

```
# l1, l2 sorted
def merge(l1, l2):
    i, j = 0, 0
    l = []
    while i < len(l1) and j <
    ↪ len(l2):
        cond = l1[i] < l2[i]
        l.append(l1[i] if cond else
        ↪ l2[j])
        i += cond
        j += not cond
    return l + l1[i:] + l2[j:]

def merge_sort(l):
    if len(l) ⩽ 1: return l
    mid = len(l) // 2
    l1 = merge_sort(l[:mid])
    l2 = merge_sort(l[mid:])
    return merge(l1, l2)
```

1.3 Heapsort

Runtime complexity : $\Theta(n \log n)$

In-place

```
def heap_sort(A):
    build_max_heap(A)
    for i in reversed(range(1,
    ↪ len(A))):
        A[0], A[i] = A[i], A[0]
        max_heapify(A, 0, i)
```

1.4 Quick Sort

Runtime complexity : $\Theta(n^2)$

Best case : subarrays completely balanced $\Theta(n \log n)$

Random version : $O(n \log n)$

In-place

```
# A[p..r] subarray
# last element of array as pivot
def partition(A, p, r):
    x = A[r]
    i = p - 1
    for j in range(p, r):
        if A[j] ⩽ x:
            i += 1
            A[i], A[j] = A[j], A[i]
    A[i + 1], A[r] = A[r], A[i + 1]
    return i + 1

def random_partition(A, p, r):
    i = random(p, r)
    A[r], A[i] = A[i], A[r]
    return partition(A, p, r)

def quicksort(A, p, r):
    if p < r:
        q = partition(A, p, r)
        quicksort(A, p, q - 1)
        quicksort(A, q + 1, r)
```

1.5 Counting sort

Count occurrences of elements in another array of length n, then rewrite elements back into array

Running time : $\Theta(n + k)$ when all numbers are between 0 and k

2 Divide & conquer

$T(n)$: time for size n

a : number of sub-problems

$\frac{n}{b}$: size of sub-problems

$D(n)$: time to divide

$C(n)$: time to combine

$T(n) = aT(\frac{n}{b}) + D(n) + C(n)$

2.1 Strassen algorithm

Runtime complexity : $\Theta(n^{\log_2 7})$

$A, B, C : \frac{n}{2} \times \frac{n}{2}$

$$\begin{pmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$

$M_1 := (A_{11} + A_{22})(B_{11} + B_{22})$

$M_2 := (A_{21} + A_{22})B_{11}$

$M_3 := A_{11}(B_{12} - B_{22})$

$M_4 := A_{22}(B_{21} - B_{11})$

$M_5 := (A_{11} + A_{12})B_{22}$

$M_6 := (A_{21} - A_{11})(B_{11} + B_{12})$

$M_7 := (A_{12} - A_{22})(B_{21} + B_{22})$

$C_{11} = M_1 + M_4 - M_5 + M_7$

$C_{12} = M_3 + M_5, \quad C_{21} = M_2 + M_4$

$C_{22} = M_1 - M_2 + M_3 + M_6$

2.2 Master theorem

$a, b >= 1, c \leq 1, \epsilon > 0$ constants

$$T(n) = \left(aT(\frac{n}{b}) + f(n)\right) \in$$

$\Theta(n^{\log_b a})$ if $f(n) \in O(n^{\log_b (a-\epsilon)})$

$\Theta(n^{\log_b a} \log n)$ if $f(n) \in \Theta(n^{\log_b a})$

$\Theta(f(n))$ if $f(n) \in \Omega(n^{\log_b (a+\epsilon)})$

and $a \cdot f(\frac{n}{b}) \leq c \cdot f(n), \forall n > N$

2.3 Max subarray

Runtime (divide and conquer) : $\Theta(n \log n)$

```
def max_from(l, s=0):
    return max((s := s + e, i) for
    ↪ i, e in enumerate(l))

def max_crossing(l1, l2):
    s1, i = max_from(reversed(l1))
    s2, j = max_from(iter(l2))
    return s1 + s2, (i, len(l1) +
    ↪ j)

def max_subarray(l):
    if len(l) == 1:
        return l[0], (0, 0)
    mid = len(l) // 2
    ls = l[:mid], l[mid:]
    s1, s2 = map(max_subarray, ls)
    s3 = max_crossing(*ls)
    return max(s1, s2, s3)
```

Runtime linear : $O(n)$

```
def max_subarray_lin(l):
    M = m = (l[0], (0, 0))
    for i in range(1, len(l)):
        m = max((l[i], (i, i)),
        ↪ (m[0] + l[i], (m[1][0],
        ↪ i)))
        M = max(M, m)
    return M
```

3 Data structures

3.1 Heap

Heap (not garbage-collected storage) : nearly compete binary tree

Max(Min)-Heap property : key of i's children is <=(>=) to i's key, maximum (minimum) element is the root

Height of node : nb of edges on longest simple path from node to a leaf

Height of head : height of root

Store heap in array :

`L[0]` root

`L[(2*i)+1]` left child node

`L[(2*i)+2]` right child node

`L[(i-1)//2]` parent node

3.1.1 Max-Heapify

Runtime complexity : $O(\log n)$

Space complexity : $\Theta(n)$

Maintains the Max-Heap property given a heap such that the subtrees are Max-Heap

```
def max_heapify(A, i, n):
    I = [i, 2*i+1, 2*i+2]
    c = filter(lambda i: i < n, I)
    m = max(c, key=lambda i: A[i])
    if m ≠ i:
        A[i], A[m] = A[m], A[i]
        max_heapify(A, m, n)
```

3.1.2 Build Max-Heap

Runtime complexity : $O(n)$

```
def build_max_heap(A):
    for i in reversed(range(len(A)
    ↪ // 2)):
        max_heapify(A, i, len(A))
```

3.1.3 Priority Queue

Dynamic set S of elements, each element has a key (value regulating its importance)

Insert(S, x) : $O(\log n)$

Maximum(S) : $O(1)$

Pop-Maximum(S) : $O(\log n)$

Increase-Key(S, x, k) : $O(\log n)$

3.2 Stack and Queues

Very efficient, limited support (no search, ...), arrays implementations have fixed capacity

Stack : Last-in, first-out
Push(S, x), Pop(S) : $O(1)$

Queue : First-in, first-out
Enqueue(Q, x), Dequeue(Q) : $O(1)$

3.3 Linked list

Insertion, deletion (double linked) : $O(1)$

Search : $O(n)$

3.4 Binary Search Trees

Property : left element < root, right element >= root

Minimum is leftmost node, maximum is rightmost node $O(h)$

Height h : max number of edges from root to leaf

Search, insert, delete : $O(h)$

In-order : left subtree > root > right subtree

Preorder : root > left > right

Postorder : left > right > root

3.5 Graphs

V : set of vertices, E : set of edges

Edge : ordered pair of vertices

$G = (V, E)$

Adjacency list : Array of $|V|$ linked-lists (one per vertex), `G.Adj[u]` is $\{v : (u, v) \in E\}$
Space = $\Theta(V + E)$, list adjacent vertices = $\Theta(\deg(u))$, test $(u, v) \in E$ = $O(\deg(u))$

Adjancy matrix : $A = |V| \times |V|$ where $a_{ij} = (i, j) \in E \, ? \, 1 : 0$
Space = $\Theta(V^2)$, list adjacent vertices = $\Theta(V)$, test $(u, v) \in E$ = $\Theta(1)$

BFS : $O(V + E)$

DFS : $\Theta(V + E)$

Classification of edges : tree edge = DSF explored (u, v), back edge = (u, v) where u is a descendant of v, forward edge : (u, v) where v is a descendant of u but not a tree edge, cross edge : any other edge

Acyclic : Directed graph G is acyclic $\iff$ DFS yields no back edges

Topological sort : call DFS and compute finishing times, output vertices in decreasing order of finishing times

Strongly connected component (SCC) : is a maximal set of vertices $C \subseteq V$ (in a directed graph), such that $\forall u, v \in C$ both u is reachable from v and v is reachable from u
G^{SCC} is a directed acyclic graph

SCC(G) : call DFS(G) compute finishing times, compute G^T, call DFS(G^T) considering vertices in order of decreasing finishing times and output vertices in each tree of depth-first forest as separate SCC, $\Theta(V + E)$

3.6 Shortest path problem

Single source : from source to every vertex

Single destination : from every vertex to destitution

Single pair : from u to v

All pairs : $\forall u, v \in V$ from u to v

Negative weight : ok as long as no negative-weight cycle reachable from source

Weigt of path $\langle v_0, v_1, \ldots, v_k \rangle$: $\sum_{i=1}^{k} w(v_{i-1}, v_i)$

Bellman-Ford $\Theta(E \cdot V)$: (no negative cycles) each vertex v keep track of $d(v)$ (current upper estimate length shortest path to v) and $\pi(v)$ (the predecessor of v in shortest path)

```
def relax(u, v, w):
    if v.d > u.d + w(u,v):
        v.d = u.d + w(u,v)
        v.π = u

def bellman_ford(G, w, s):
    for v in G.V:
        v.d, v.π = INF, NIL
    s.d = 0 # init
    for i in range(len(G.V) - 1):
        for (u, v) in G.E:
            relax(u, v, w)
```

Negative cycles detection : run one more (V-th) iteration

```
...
    for (u, v) in G.E:
        if v.d > u.d + w(u,v):
            return False
```

Dijkstra : (nonnegative weights), binary heap $O(E \log V)$, $O(V \log V + E)$, start with source $S = \{s\}$, greedily grow S (add to S the vertex closest to S, minimize `u.d + w(u, v)`)

```
... # init
    S = set()
    Q = G.V
    while Q:
        u = extract_min(Q)
        S |= {u}
        for v in G.Adj[u]:
            relax(u, v, w)
            decrease_key(Q, v, v.d)
```

3.6.1 Flow Network

Edge (pipes) has capacity ($c(u, v) \geq 0$) = flow rate upper bound, maximize rate of flow from source s to sink t, no anti-parallel edges

Flow : function $f : V \times V \to \mathbb{R}$ such that $\forall u, v \in V : 0 \leq f(u, v) \leq c(u, v)$ (capacity constraint) and $\forall u \in V \setminus \{s, t\}$: $\sum_{v \in V} f(v, u) = \sum_{v \in V} f(u, v)$ (flow into u = flow out of u, flow conservation)

Flow value : $|f| = \sum_{v \in V} f(s, v) - \sum_{v \in V} f(v, s)$, flow out of source - flow into source

Residual capacity : $c_f(u, v) = c(u, v) - f(u, v)$ if $(u, v) \in E$ (amount of capacity left), $f(v, u)$ if $(v, u) \in E$ (amount of flow that can be reversed), 0 otherwise

Residual network : edges with capacities that represent how we can change the flow on edges, $G_f(V, E_f) = (V, E_f)$ where $E_f = \{(u, v) \in V \times V : c_f(u, v) > 0\}$

Ford-Fulkerson Method'54 $O(E \cdot |f_{\max}|)$: initialize flow f to 0, while $\exists$ augmenting path p in residual network G_f : augment flow f along p by bottleneck

Cut : a partition of V into S and $T = V \setminus S$ such that $s \in S$ and $t \in T$

Net flow across cut : $f(S,T) = \sum_{u \in S, v \in T} f(u,v) - \sum_{u \in S, v \in T} f(v,u)$ (flow leaving S − flow entering S)

For any cut : $|f| = f(S,T)$

Capacity : $c(S,T) = \sum_{u \in S, v \in T} c(u,v)$

for any flow, cut : $|f| = f(S,T) \leq c(S,T)$

max-flow = min-cut

number of possible cuts : $2^{|V|-2}$

Min-cut : set S of all nodes which can be reached from s in the final residual network

Equivalences : f is max-flow $\iff$ G_f has no augmenting path $\iff$ $|f| = c(S,T)$ for min-cut (S,T)

3.7 Disjoint-set

Aka. "union find", maintain collection $S = \{S_1, \ldots, S_k\}$ of disjoint dynamic sets, each set defined by a representative (member of the set)

Operations : `make-set(x)` (add a new set $S_i = \{x\}$ to S), `union(x, y)` ($S = (S - S_x - S_y) \cup (S_x \cup S_y)$), `find(x)` (representative of set containing x)

Connected components of Graph : for each vertex `make-set(v)`, for each edge if `find-set(u)` $\neq$ `find-set(v)`:`union(u, v)`, linked list weighted-union heuristic $O(V \log V + E)$, forest union-by-rank $O((V+E)\alpha(V)) \approx O(V+E)$

Weigted-union heuristic : always append the smaller list to the larger list (break ties arbitrarily), sequence of m operations on n elements take $O(m + n \log n)$ time

Forest of trees : one tree per set, root is representative, each node only points to parent, make-set (single-node tree), find (follow pointers to root), union (make one root a child of another) $O(m \cdot \alpha(n))$

Great heuristics : union by rank (root of the smaller (rank) tree becomes child of root of larger tree), don't use size, use rank (upper bound on height of node)

Spanning tree : acyclic set T of edges, spanning (connects all vertices)

Cut property : let $(S, V \setminus S)$ a cut, T a tree on S which is part of MST, e a crossing edge of minimum weight, $\implies$ $\exists$ MST of G containing e and T

Prim Min spanning tree (MST) $O(E \log V)$: start with any vertex v and build tree T from v, greedily grow T (add to T a min weight crossing edge with respect to cut induced by T)

Kruskal $O(E \log V)$: start from empty forest T, greedily maintain forest T (add cheapest edge that does not create cycle)

3.8 Hash

Direct-Address Tables : every item has unique id, array/table with position for each item, $O(1)$ insertion deletion and search, space $O(|U|)$, small fraction of possible items

Hash Tables : space proportional to number k of keys stored $\Theta(k)$, search insertion deletion $O(1)$ average time, item with key k sorted in slot $h(k)$, $h : U \to \{0, 1, \ldots, m-1\}$ hash function

Hash function properties : efficient computable, uniform keys distribution, deterministic ($h(k)$ always equal to $h(k)$), example : $h(k) = k \mod m$

Collisions : two items with keys k_i, k_j have $h(k_i) = h(k_j)$, place all items with same hash into same (double) linked list, insertion deletion and expected search $O(1)$, space $O(m+k)$

4 Dynamic programming

Remember calculations already made to save enormous amount of computation

Top-down memoization : solve recursively and store each result in table

Bottom-up : sort subproblems, solve smaller first, already have solved smaller ones when solving a subproblem

5 Probabilistic analysis

Indicator Random Variables : event A, $I\{A\} = 1$ if A occurs and 0 if A does not occur

$X_A = I\{A\} \implies E[X_A] = P[A]$

Linearity of expectation : $E[aX + bY] = aE[X] + bE[Y]$