Asymptotic Complexity

- **Big Theta:** EG. $x^2 = \theta(x^2 + x(lgx)^2)$
($\Leftrightarrow \exists$ D1,D2 > 0, n0 s.t. D1|g(n)| ≥ |f(n)| ≥ D2|g(n)| for n ≥ n0)
EG for $f(n) = 2n^2 + 100$, $g(n) = n^2$ let D1=2, D2=1.
- **Big O:** (<=) EG. $n^2 = O(n^{1000})$
f grows at most as fast as g when x→∞
($\Leftrightarrow \exists$ D>0, n0 s.t. |f(n)| ≤ D|g(n)| for n ≥ n0)
- **Omega:** EG. $n^{1000} = \Omega(n)$
($\Leftrightarrow \exists$D>0, n0 s.t. |f(n)| ≥ D|g(n)| for n ≥ n0

Master Theorem:

$T(n) = aT(n/b) + f(n)$ where $a \geq 1$ and $b > 1$ are constants and $f(n)$ is an asymptotically positive function.

a = # segments divided
n/b = # elements in sub-segments

h = # of levels = $\log_b n = \theta(\log n)$
L = # of leaves = $n^{\log_b a}$

1. If $f(n) = O(n^{log_b a - \epsilon})$ for some constant ϵ>0, then $T(n) = \theta(L)$ (geometrically ↑ing down the tree)
2. If $f(n) = \vartheta(L) \Rightarrow T(n) = \vartheta(L \log n)$ (roughly equal at each lvl)
3. If $f(n) = \Omega(L^{1+\epsilon}) \Rightarrow T(n) = \vartheta(f(n))$
ϵ>0, and f(n) satisfies the **regularity condition** af(n/b) ≤ cf(n) for c<1 and all sufficiently large n.

Peak Finding (1D):

A[i] is peak if it's not smaller than all its neighbour(s)
Brute Force: worst case: **θ(n)** scan left to right
if i!=1,n: A[i]≥A[i-1] and A[i]≥A[i+1]
if i=1: A[1] ≥ A[2]
if i=n: A[n] ≥ A[n-1]
Algo: Start from middle element, compare with neighbours.
If A[n/2-1] > A[n/2]: search for a peak among A[1]… A[n/2-1]
If A[n/2] < A[n/2+1]: search for a peak among A[n/2+1]… A[n]
Else: A[n/2] is a peak!
Worst case: T(n) = T(n/2) + O(1) --> compare A[n/2] w 2 neigh = **O(log n)**

Peak Finding (2D):

Start from middle column j=m/2 n rows, m cols
Find global max on col j at (i,j)
Compare (i,j-1),(i,j)&(i,j+1). Pick Left columns if (i,j-1)>(i,j), vice versa. IF (i,j)>other2: ISSSA peak!!
Recursively start from middle again.
T(n,m) = T(N/2,m) + θ(m) = m log n
[if find max along column]

Document Distance:

Doc D: a seq of words | Word w: a seq of char
Word freq D(w): no. of occurrences of w in D
- Treat each document as a vector of its words
- Dot product & normalise result by measuring distance by angle b/w vectors.
- θ=0: "identical"; same size; permutes of each other
- θ= π/2: don't even share a word

Insertion Sort:

```
For i in range(1,n):
while A[i]<A[i-1]:
    swap A[i] with A[i-1]
    i -= 1
```

Worst case: T(n) = i comparisons & swaps at step i

$$= \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} = \theta(n^2)$$

Merge Sort:

Time:
1. Divide: Θ(n)
2. Recursion: n1=n2=n/2
time = T(n1)+T(n2)=2T(n/2)
3. Merge: Θ(n)
Total: T(n)=2T(n/2)+Θ(n)=**Θ(*n*log*n*)**

- if n=1: done –> return
- Recursively sort A[:n/2] -> L
- Recursively sort A[n/2:] -> R
- Merge L & R -> output A'

Python Cost Model:

L is List, D is Dictionary

`L[i] = x`	Θ(1)
`L.append(x)`	Θ(1)
`L1.extend(L2)`	Θ(|L1|)
`A = L1 + L2`	Θ(|L1|+|L2|)
`x in L`	Θ(|L|)
`D[xi] = yi`	Θ(1)
`x in D`	Θ(1)

Heaps:

Array visualised as a nearly complete bin tree
Max (Min) Heap Property: the key of a node is ≥ (≤) than the keys of its children
parent(i) = $\lfloor i/2 \rfloor$ -> index of node's parent
Left(i) = 2i | right(i) = 2i+1 (index of node's __ child)
Height = O(log n)

Build_max_heap: given an array -> **O(n)**
> go bottom up since leaves do not ever violate max heap property
Index $\lfloor n/2 \rfloor$+1 onwards are leaves of the tree
for i = floor(length[A]/2) downto 1:
  do max_heapify(A,i)

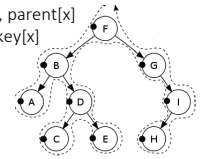
Insert, Extract_max, increase_key: O(log n)
extract_max(S): return element from S largest key, remove it & heapify S to maintain heap property
increase_key(S,x,k'): increase value of x's key to new value, k'

Heapsort: sort an array of size n using heap **O(nlogn)**
(pass in sorted array still transformed to heap & sorted)

Max_heapify(A,i): correct a single violation at root of a subtree in **O(log n)** *{cus height of tree bounded by logn}*
Assume trees rooted at left(i) & right(i) are max heaps, but A[i] violates. Find index of largest element among these 3, if index != i, exchange A[i] w largest element & recurse on subtree (max_heapify(A,largest_index)

Binary Search Trees

- Each node x has key[x]
- Pointers: left[x], right[x], parent[x]
- Property: Key[left[x]] < key[x] & key[x] < key[right[x]]
- Tree Traversals: O(n)
Preorder: FBADCEGIH
Inorder: ABCDEFHI
Postorder: ACEDBHIGF



When you del a root of a BST, the next-largest node replaces that root.

```
Next-larger(x): finds the next element after element x
if right[x] ≠ NIL then
   return findmin(right[x])
else
   y = parent[x]
   while y≠NIL and x==right[y]
   x = y
   y = parent[y]
   return y
```

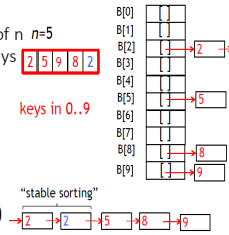
insert(k) (with <3mins check)
find(k) (finds node containing k)
findmin(x) (finds min of tree rooted at x)
deletemin() (finds min of tree and delete it)
All are worst case: **O(n)**

Comparison sorts: Ω(nlogn)

A decision tree can model the execution of ANY comparison sort
One tree for each input size n, a path from root to the leaves is the trace of comparisons the algo perform.
Worst-case running time = height of tree
Proof: Tree must contain >= n! leaves => n! possible answers | A height-h binary tree has <= 2^h leaves
Thus $n! \leq L_n \leq 2^{h_n} \leftrightarrow$
$\log(n!) \leq h_n \rightarrow h_n = \Omega(nlog\ n)$

Counting Sort

Given array A[0],...,A[n-1] of n *n=5* keys to be sorted. The n keys are integers in {0,1,...k-1}
B = array of k empty lists (linked/python lists) – O(k)
for j in range(n): – O(n)
  B[A[j]].append(A[j])
output = []
for i in range(k): – O(n+k)
  output.extend(B[i])



Radix Sort

- Sort on least significant digit/alphabet first
- Given n integers, each int <= M, in base k
Each int has d = $log_k(M)$ digits, a digit is in {0,1,...,k-1}
- Assume counting sort is *auxiliary stable* sort
Counting sort: we need
$\theta(n+k)$ per digit $\Rightarrow \theta((n+k)d) \Rightarrow \theta((n+k)log_k M)$
$\theta(nlog_n M)$ if k=n $\Rightarrow \theta(nc)$ if $M \leq n^c$ *for some* $c > 0$
- if range M of possible values grows at most proportionally with size of problem (n), use counting sort
- if M grows even faster but O(n^c) for some c>1, use radix sort (choose optimal base)

Hash Tables:

Insert O(1) | Search O(n) worst, O(1+α) avg
Del O(1) *if have pointer to obj
Collisions: two keys are hashed onto the same value: h(key1)=h(key2)
Hash table is an array of size m, each element in the table is a linked list (chaining)
Worst case search: **O(n)** one entry has all n elements, essentially searching a linked list
Average case: every entry have α=n/m objects(search list, apply hash function & random access to slot)
O(1+α) = O(1) if α = O(1) ie m= Ω(n)
Chained-hash-insert(A,x): insert x at head of linked list A - **O(1)** compute h(x.key) + 1 insert
Chained-hash-delete(A,x): assume we have pointer to x. Go to x in A[h(x.key)] using address, link predecessor and successor of x in linked list, delete x from list A[h(x.key)] using pointer - **O(1)**
Hashing for non-numerical input: go by some radix
Simple uniform hashing assumption: any given element is equally likely to hash into any of the m slots, indep of where any other element has hashed to.

Resizing a Hash Table: Complexity:O(n+m+m') |m for visiting every table entry in old table, n for processing every obj, m' for making new table
Resize by doubling: m'=2m whenever n/m hits t
Amortized cost over Nfinal insertion=O(1) for search (if SUH holds) O(1+α) but α is bounded indpt of Nfinal so O(1)
If n>m, we can find ≥ N/l keys that collide for any hash function

Open Addressing: one array entry = one element but can only fill table with at most n=m elements before table doubling
Linear Probing: $h(k,i) = (h'(k) + i)\ mod\ m$
Quadratic Probing:
$$h(k,i) = (h'(k) + c_1 i + c_2 i^2)\ mod\ m$$
Double Hashing:
$$h(k,i) = \left(h_1(k) + i h_2(k)\right)\ mod\ m$$
Use the first mod then compute the second mod if m is different
EG h'(k) = k % n
(h'(k) + ci + ki^2) % m (here u needa compute the first mod first!!)
Uniform Hashing Assumption: *Probe sequence* of each key is equally likely to be any of m! permutations of {0,...,m-1}.
P(first probe finding empty slot)= 1-α
Search: 1/(1−α) probe steps on average for unsuccessful search | 1/α log (1/(1−α)) probe steps on average for successful search
Insertion: 1/(1−α) probe steps on avg
if α is small (much < 1), 1/(1-α) is close to 1.
α should be < 1, use doubling before n = m.

Open Addressing vs Chaining:

- OA: Better cache perf, no pointers to off regions needed when objects are "small" eg int/float
- Chaining: Less sensitive to hash function choices, similar results even when you choose diff hash fn. Less sensitive to high load factors, OA needs α to be small.

Cuckoo Hashing:

- use 2 hash functions, key stored in either T[h1(k)] or T[h2(k)]. Just need to look at at most 2 places – O(1).
Insertion: if T[h1(k)] empty, store. else if T[h2(k)] empty, store. Else, store in T[h1(k)] and move key that was there to its other location.
If α<1, insertion succeeds with high prob.
If Insertion loops: rehash entire table/ x2 table size.
Search is O(1) worst case, Insertion is O(n) worst case, O(1) avg

Representation of Graph:

1. Adjacency lists (of neighbours of each vertex)
Array A of |V| linked list, for each v in vertices, A[v] stores neighbours
Directed graph only stores outgoing neigh, undirected stores edge in 2 places
2. Implicit Representation (as neighbour fn)
Don't store graph at all, implement Adj(u) that returns list of neigh/edges of u, requires no space
3. Incidence lists (of edges from each vertex)
For each vertex v, list its edges: Array A of |V| linked list, for each v in vertices, A[v] store edges. Directed graph only stores outgoing edges, undirected store edge in 2 places
4. Adjacency Matrix (of which pairs are adj)
nxn matrix A = (aij); aij =1 if (i,j) is edge
Space: Adj list: one list node per edge: space is θ(n+m) | Adj matrix: space θ(n^2), better only for dense graph where m is near n^2 (m can be n(n-1))
Time: Add edge: both O(1)|Check edge: mat O(1), Adj list O(n)|Visit all neigh of u: adj list O(neigh), mat θ(n)|Remove edge: find+add
Unweighted graph: maximum n(n-1)/2 edges
BFS -> Queue (FIFO) ; DFS -> Stack (LIFO)

BFS (Array+Adj list):

```
BFS(s,Adj):
  level={s:0}
  parent={s:None}
  i=1
  frontier=[s]
  while frontier:
    next=[]
    for u in frontier:
      for v in Adj[u]:
        if v not in level:
          level[v]=i
          parent[v]=u
          next.append(v)
    frontier=next
    i+=1
```

BFS (Queue+Adj list):

```
BFS(G,s):
  for each vertex:
    u.color=WHITE
    u.d=∞ ; u.parent=NIL
  s.color=GRAY
  s.d=0
  s.parent=NIL
  Q=∅
  ENQUEUE(Q,s)
  while Q != ∅:
    u=DEQUEUE(Q)
    for each v in Adj[u]:
      v.color=GRAY; v.d=u.d+1
      v.parent=u
      ENQUEUE(Q,v)
    u.color=BLACK
```

Shortest paths from s to v:

Length=level[v]; is ∞ (if not reachable from s)
follow v->parent[v]->...->s

Depth-First Search (depth up to V-1, complete graph)

- No 2 vertices will have same start/end time
- Type of edge, u -> v: (no forward in undir)
back: v.d < u.d < u.f < v.f (v is ancestor of u)
forward: u.d < v.d < v.f < u.f (visit descendent)
cross: v.d < v.f < u.d < u.f
Tree: first time v is visited

Graph has a cycle iff DFS has a back edge

```
DFS(G):
  for each vertex u:
    u.color=WHITE; u.prnt=NIL
    time=0
  for each vertex u
    if u.colour==WHITE
      DFS-VISIT(G,u)
```

```
DFS-VISIT(G,u):
  time+=1
  u.d=time // u was white
  u.color=GRAY
  for v in Adj[u]: //explore edge
    if v.colour==WHITE:
      v.parent=u
      DFS-VISIT(G,v)
  u.colour=BLACK //finished
  time+=1
```

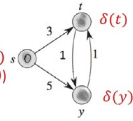
Θ(n+m)
For both directed & undirected

Topological Sort: (SSSD For DAG)

1. Run DFS(G), compute finishing times of nodes θ(n+m)
2. Output nodes in decreasing order of finishing times (can get to θ(n))
after u.color=black, store finishtime[time]=u.
then for (j=2n;j>0;j--) {if (finishtime[j] not empty) output finishtime[j]}

Single Source Shortest Path (hav weights, diff from BFS)

Shortest path not necessarily unique but value function is unique | Subpaths of shortest paths are also shortest paths
$\delta(Vo,Vj) + \delta(Vj,Vn) = \delta(Vo,Vn)$
Triangle inequality: $\delta(s,t) <= \delta(s,u) + \delta(u,t)$
Negative-weight cycle: shortest path cannot be defined | Positive-weight cycle: never occurs in any shortest path
Relax an edge(u,v): if $\delta(u)+w(u,v) < \delta(v)$ then $\delta(v) := \delta(u)+w(u,v)$; $v.\pi := u$
Bellman Eqns: $\delta(s)=0$;
$\delta(t)=\min\{\delta(s)+3, \delta(y)+1\}$;
$\delta(y)=\min\{\delta(s)+5, \delta(t)+1\}$



Bellman-Ford

```
for v in V:                          initialisation
  v.d=∞; v.π=nil                     θ(|V|)
s.d=0
do n-1 times:                        θ(|V||E|)
  for each edge(u,v) in E:
    relax(u,v)
for each edge(u,v) in E:             detect neg cyc
  if v.d > u.d + w(u,v)              θ(|E|)
    negative cycle
no neg cycle
```

Overall complexity =θ(|V||E|)

Dense graphs: $|E| \sim |V|^2$
Sparse graphs: $|E| \sim |V|$

★ if no neg cyc, at termination for all v: v.d=δ(s,v)
★ Can be exponentially many relaxations hence order matters

Dijkstra's Algo

Applies to graphs with only **NONNEGATIVE WEIGHTS**

Uses relaxation to improve estimated dist of non-established vertices via newly promoted vertex; terminates when all vertices have established distances

```
for v in V:
  v.d=∞; v.π=nil
s.d=0; S=∅
while S≠V:
  u=v with v.d≤w.d for all w not in S
  S=S∪{u}
  for each edge(u,v) in E, v not in S:
    relax(u,v)
```

initialise: θ(|V|)

Overall complexity=$\theta(|V|^2)$

$|V|(|V|+|V|)$ or $O(|V|^2 + [E])$
- while loop thru all v
- calculate min + loop thru all neigh

(max. no of neigh for v = |V| - 1)

If priority queues used for storing & retrieving v.d, $O(|V|\log(|V|)+|E|)$

Dij for shortest path relies on fact that once we visit a vertex we wont ever find a shorter path to it, which requires non-neg weights

Avl trees

Hashing (applications) + Formulas

DP

P&NP