items

INSERTION-SORT(A) — cost / times

```
for j = 2 to A.length                         c1   n
    key = A[j]                                c2   n - 1
    // Insert A[j] into the sorted
       sequence A[1..j-1].                    0    n - 1
    i = j - 1                                 c4   n - 1
    while i > 0 and A[i] > key                c5   Σ_{j=2}^{n} t_j
        A[i+1] = A[i]                         c6   Σ_{j=2}^{n} (t_j - 1)
        i = i - 1                             c7   Σ_{j=2}^{n} (t_j - 1)
    A[i+1] = key                              c8   n - 1
```

BUBBLE-SORT(A)

```
do
    swapped = false
    for i = 2 to A.length
        if A[i-1] > A[i]
            temp = A[i]
            A[i] = A[i-1]
            A[i-1] = temp
            swapped = true
while swapped
```

OPTIMIZED-BUBBLE-SORT(A)

```
for j = A.length to 2
    swapped = false
    for i = 2 to j
        if A[i-1] > A[i]
            temp = A[i]
            A[i] = A[i-1]
            A[i-1] = temp
            swapped = true
    if (!swapped)
        break;
```

OPTIMIZED-BUBBLE-SORT(A)

```
n = A.length
do
    swapped = false
    for i = 2 to n
        if A[i-1] > A[i]
            temp = A[i]
            A[i] = A[i-1]
            A[i-1] = temp
            swapped = true
            newLimit = i-1
    n = newLimit
while swapped
```

MERGE-SORT(A, p, r)

copying, merging, recursion (stack)

reqs auxiliary storage

```
if p < r        terminating cond.
                one item list p=r --> False
q = [(p + r)/2]
MERGE-SORT(A, p, q) => Divide and ...
MERGE-SORT(A, q + 1, r) => Divide ...
MERGE(A, p, q, r) => This is the Co...
```

$2T(\frac{n}{2}) + O(n)$

As you can see the highlight of the alg...

```
MERGE(A, p, q, r)      A=[3,4, 1,2] // sub lists must be sorted
                       p q r -->(p=1st index, q= middle index, r= last index)
n1 = q - p + 1         n1=1-0 +1=2 //length of left subarray
n2 = r - q             n2=3-1 =2 //length of right subarray

create arrays L[0 ... n1] and R[0 ... n2]
                       L = [3,4]
for i = 0 to n1-1      R = [1,2]
    L[i] = A[p+i]
for j = 0 to n2-1
    R[j] = A[q+1+j]
                       L=[3,4,inf]
L[n1] = ∞              R=[1,2,inf] so below check will exit
R[n2] = ∞
i = 0
j = 0
for k = p to r         k = resultant array index
    if L[i] ≤ R[j]     k=0 to r=3(end)
        A[k] = L[i]    A =[1,2,3,4] //replacing A
        i = i + 1
    else
        A[k] = R[j]
        j = j + 1
```

quickSort(arr, beg, end)

```
if (beg < end)
    pivotIndex = partition(arr,beg, end)
    quickSort(arr, beg, pivotIndex)
    quickSort(arr, pivotIndex + 1, end)

partition(arr, beg, end)
    set end as pivotIndex
    pIndex = beg - 1
    for i = beg to end-1
        if arr[i] < pivot
            swap arr[i] and arr[pIndex]
            pIndex++
    swap pivot and arr[pIndex+1]
    return pIndex + 1
```

pivot (until sorted) swap split

p/s consider (pos) o n

ITERATIVE-BINARY-SEARCH(A, v, low, high)

```
while low ≤ high
    mid = floor((low + high) / 2)
    if v == A[mid]
        return mid
    else if v > A[mid]
        low = mid + 1
    else high = mid - 1
return NIL
```

RECURSIVE-BINARY-SEARCH(A, v, low, high)

```
if low > high
    return NIL
mid = floor((low + high) / 2)
if v == A[mid]
    return mid
else if v > A[mid]
    return RECURSIVE-BINARY-SEARCH(A, v, mid + 1, high)
else return RECURSIVE-BINARY-SEARCH(A, v, low, mid - 1)
```

$2T(\frac{n}{2}) + O(1)$

P(G, S, vis)
for v ∈ G.V
if vis[v] == T
ret T
else if vis[v] == F
S.push
vis[v] ← T
P(...)
vis[S.top] done
S.pop()

procedure DFS-iterative(G, v) is

```
let S be a stack
S.push(v)
while S is not empty do
    v = S.pop()
    if v is not labeled as discovered then
        label v as discovered
        for all edges from v to w in G.adjacentEdges(v) do
            S.push(w)
```

f(G, S, vis)
for vis ← [False]
for vert v ∈ G.V
if vis[v] == false
S ← []
S.push(v)
vis[v] ← True
P(G, S, vis)

Space → O(E)

BST Search(T, key)
root == NULL → false
root == root.data → true
key < root.data → BST(root.left, key)

complete BT: root, left, right (max)

ALWAYS look down and check if heap is broken

BST Delete

```
struct node *removeNode(struct node *root, int val)
{   if(root == NULL)
        return NULL;

    if(root->key < val)
        root->right = removeNode(root->right,val);

    else if(root->key > val)
        root->left = removeNode(root->left,val);
    else
    {
        if(root->left == NULL && root->right == NULL)
        {
            free(root);
            return NULL;
        }
        else if(root->left == NULL)
        {
            struct node *temp = root->right;
            free(root);
            return temp;
        }
        else if(root->right == NULL)
        {
            struct node *temp = root->left;
            free(root);
            return temp;
        }
        else
        {
            int rightMin = getRightMin(root->right);
            root->key = rightMin;
            root->right = removeNode(root->right,rightMin);
        }
```

pre, inorder, suc

HEAPSORT(A) — O(nlogn)

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

BUILD-MAX-HEAP(A) — O(n)

```
A.heap-size = A.length
for i = ⌊A.length/2⌋ downto 1
    MAX-HEAPIFY(A, i)
```

MAX-HEAPIFY(A, i) — O(logn) — extract largest child to bottom

```
l = LEFT(i)
r = RIGHT(i)
if l ≤ A.heap-size and A[l] > A[i]
    largest = l
else largest = i
if r ≤ A.heap-size and A[r] > A[largest]
    largest = r
if largest ≠ i
    exchange A[i] with A[largest]
    MAX-HEAPIFY(A, largest)
```

$T(n) \le T(2n/3) + \Theta(1)$

HEAP-EXTRACT-MAX(A)

```
if A.heap-size < 1
    error "heap underflow"
max = A[1]
A[1] = A[A.heap-size]
A.heap-size = A.heap-size - 1
MAX-HEAPIFY(A, 1)
return max
```

PARENT(i)
return ⌊i/2⌋

LEFT(i)
return 2i

RIGHT(i)
return 2i + 1

applications
- Job sched, shared c...
- priority event
- printer queue

MAX-HEAP-INSERT(A, key) — O(logn)

```
A.heap-size = A.heap-size + 1
A[A.heap-size] = -∞
HEAP-INCREASE-KEY(A, A.heap-size, key)
```

insert to last space

HEAP-INCREASE-KEY(A, i, key)

```
if key < A[i]
    error "new key is smaller than current key"
A[i] = key
while i > 1 and A[PARENT(i)] < A[i]
    exchange A[i] with A[PARENT(i)]
    i = PARENT(i)
```

parent check

delete: replace last, heapify()

HASH-INSERT(T, k)

```
i = 0
repeat
    j = h(k, i)
    if T[j] == NIL
        T[j] = k
        return j
    else i = i + 1
until i == m
error "hash table overflow"
```

HASH-SEARCH(T, k)

```
i = 0
repeat
    j = h(k, i)
    if T[j] == k
        return j
    i = i + 1
until T[j] == NIL or i == m
return NIL
```

perfect → O(1)

key = index, H(k) = k

$e^{\frac{-n(n-1)}{2d}}$ slots

space waste (universe of keys actually stored)

DAccess: O(1), search: O(1); Hash: O(n)

insert, del O(1)

disadv: universe of keys

head → O(1)
else → O(n)

LIST-INSERT(L, x)

```
x.next = L.head
if L.head ≠ NIL
    L.head.prev = x
L.head = x
x.prev = NIL
```

LIST-DELETE(L, x)

```
if x.prev ≠ NIL
    x.prev.next = x.next
else L.head = x.next
if x.next ≠ NIL
    x.next.prev = x.prev
```

INORDER-TREE-WALK(x)

```
if x ≠ NIL
    INORDER-TREE-WALK(x.left)
    print x.key
    INORDER-TREE-WALK(x.right)
```

TREE-MAXIMUM(x)

```
while x.right ≠ NIL
    x = x.right
return x
```

TREE-MINIMUM(x)

```
while x.left ≠ NIL
    x = x.left
return x
```

TREE-SEARCH(x, k)

```
if x == NIL or k == x.key
    return x
if k < x.key
    return TREE-SEARCH(x.left, k)
else return TREE-SEARCH(x.right, k)
```

TREE-INSERT(T, z)

```
y = NIL
x = T.root
while x ≠ NIL
    y = x
    if z.key < x.key
        x = x.left
    else x = x.right
z.p = y
if y == NIL
    T.root = z          // tree T was empty
elseif z.key < y.key
    y.left = z
else y.right = z
```

ITERATIVE-TREE-SEARCH(x, k)

```
while x ≠ NIL and k ≠ x.key
    if k < x.key
        x = x.left
    else x = x.right
return x
```

TREE-SUCCESSOR(x)

```
if x.right ≠ NIL
    return TREE-MINIMUM(x.right)
y = x.p
while y ≠ NIL and x == y.right
    x = y
    y = y.p
```

adj mat $O(V^2)$

BFS(G, s) — O(V+E)

```
for each vertex u ∈ G.V - {s}
    u.color = WHITE
    u.d = ∞
    u.π = NIL
s.color = GRAY
s.d = 0
s.π = NIL
Q = ∅
ENQUEUE(Q, s)
while Q ≠ ∅
    u = DEQUEUE(Q)
    for each v ∈ G.Adj[u]
        if v.color == WHITE
            v.color = GRAY
            v.d = u.d + 1
            v.π = u
            ENQUEUE(Q, v)
    u.color = BLACK
```

queue, print, pop, explore, fully

visit, explore, suspend → stack, explore new

Space → O(V)

DFS(G) — O(V+E)

```
for each vertex u ∈ G.V
    u.color = WHITE
    u.π = NIL
time = 0
for each vertex u ∈ G.V
    if u.color == WHITE
        DFS-VISIT(G, u)
```

DFS-VISIT(G, u)

```
time = time + 1
u.d = time
u.color = GRAY
for each v ∈ G.Adj[u]
    if v.color == WHITE
        v.π = u
        DFS-VISIT(G, v)
u.color = BLACK
time = time + 1
u.f = time
```

Space O(V)

start time = [1, ∞, ...]
end time = [∞, ∞, ...]

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

```
left-sum = -∞
sum = 0
for i = mid downto low
    sum = sum + A[i]
    if sum > left-sum
        left-sum = sum
        max-left = i
right-sum = -∞
sum = 0
for j = mid + 1 to high
    sum = sum + A[j]
    if sum > right-sum
        right-sum = sum
        max-right = j
return (max-left, max-right, left-sum + right-sum)
```

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

```
if high == low
    return (low, high, A[low])          // base case: only one element
else mid = ⌊(low + high)/2⌋
    (left-low, left-high, left-sum) =
        FIND-MAXIMUM-SUBARRAY(A, low, mid)
    (right-low, right-high, right-sum) =
        FIND-MAXIMUM-SUBARRAY(A, mid + 1, high)
    (cross-low, cross-high, cross-sum) =
        FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
    if left-sum ≥ right-sum and left-sum ≥ cross-sum
        return (left-low, left-high, left-sum)
    elseif right-sum ≥ left-sum and right-sum ≥ cross-sum
        return (right-low, right-high, right-sum)
    else return (cross-low, cross-high, cross-sum)
```

T(n) = 2T(n/2) + O(n)

Time Complexity : O(nlogn)

v prop desc. u: d[u] < d[v] < f[v] < f[u]

min cost path → unique
mst → may be not

acyclic (DFS → x back edges)
tree ⇒ MST | connected

MST-Prim(G, w, r) → connected

Greedy

1. for each vertex u ∈ G.V — start with a node, get the minimum connected edge and add to the tree, find the next connected min edge to the tree
2. u.key = ∞
3. u.π = NIL
4. r.key = 0
5. Q = G.V
6. while Q ≠ ∅
7. u = EXTRACT-MIN(Q)
8. for each v ∈ G.Adj[u]
9. if v ∈ Q and w(u, v) < v.key
10. v.π = u
11. v.key = w(u, v)

tree with min edge

Prim's Algorithm for Minimum Spanning Tree
Add Node 0 to the tree
Repeat:
1. Check all edges from nodes in tree to nodes outside the tree
2. Find the smallest among such edges
3. Add this edge and its endpoint to tree
End

$O(E \log V) \rightarrow t(pq)$
$O(E+V) \rightarrow spc$

- travelling Salesman (NP)

MST-Kruskal(G, w) — matrix → disconnected ✓

Greedy

go through sorted E → no cycles

1. A ← ∅
2. For each vertex v ∈ G.V
3. MAKE-SET(v) — single vertex trees
4. sort the edges of G.E in nondecreasing order of weight — differes from increasing, can include same values
5. for each edge (u, v) ∈ G.E, in order of nondecreasing weight
6. if FIND-SET(u) ≠ FIND-SET(v) } check if it is in same tree
7. A ← A ∪ {(u, v)} — first time
8. UNION(u, v)
9. return A

- Net

larger element set is decided as parent representative
by rank 3 element chain

$O(V^2) \rightarrow t(basic)$
$O(E \log V) \rightarrow t(Union)$
$O(E+V) \rightarrow spc$

SMT

weighted, undirected, connected, acyclic, including upto only terminals

Generic MST(G, w)
A ← ∅
while A not form MST,
Safe edge(u,v)
A ← A ∪ (u,v)
return A

cut ↓ light edge
do not cross

application
- Comm Nets
- cct des
- highway system layout

$\# MST = n^{(n-2)}$ } complete

INITIALIZE-SINGLE-SOURCE(G, s) $O(V)$
1. for each vertex v ∈ G.V
2. v.d = ∞
3. v.π = NIL
4. s.d = 0

RELAX(u, v, w) $O(1)$
1. if v.d > u.d + w(u, v)
2. v.d = u.d + w(u, v)
3. v.π = u

Cut ⇒ (S, V−S) partition of V into 2 disjoint sets
Cross ⇒ edge (u,v) in two sep cuts
respect ⇒ given subset edges, if no edges crosses the cut } respect
light edge ⇒ crossing cut edge with min cost
Safe edge ⇒ must be connected

weighted, directed
sp → no cycles
SSSP $(+, \theta$ } x d.c; but (−)) dp

$d[V] = [0\ \infty\ \infty]$
$\pi[V] = [nil, nil \ldots]$

BELLMAN-FORD (G, w, s)

== BFS

1. INITIALIZE-SINGLE-SOURCE(G, s)
2. for i = 1 to |G.V|−1 — you have to relax for [V-1] times (max number of edges btw longest
3. for each edge (u, v) ∈ G.E — for every edge, any order
4. RELAX(u, v, w)
5. for each (u, v) ∈ G.E — e more time
6. if v.d > u.d + w(u, v)
7. return FALSE
8. return TRUE

① edge list
② (v−1) repeat & relax
③ one more to

~~detective negative weight cycle~~

time complexity − O(|E||V|)
space − O(V)

if cycle total weight is positive it's fine

$O(VE)$

DIJKSTRA(G, w, s) . Assume: Non Negative Weights

Greedy

1. INITIALIZE-SINGLE-SOURCE(G, s) — every other v.d = inf, predecessor v.p = null, source.d = 0
2. S = ∅ — visited list
3. Q = G.V — unvisited list
4. while Q ≠ ∅
u = EXTRACT-MIN(Q) — get the smallest unvisited vertex distance from src, choose that
S = S ∪ {u}
for each v ∈ G.Adj[u]
RELAX(u, v, w)

Q = {s, a, b, c, d} (min pq) ~~FIFO~~
select min, relax
Greedy part

1) adj matrix Naive
a. O(v * v) → space
2) adj list → O(V) space
binary heap (pq)
i. O(v + ElogV) = O(ElogV) — time
b. fibonacci heap
c. sets

Data Structure	Worst Case			Average Case		
	Insert	Search	Delete	Insert	Search	Delete
Array	O(1)	O(n)	O(n)	O(1)	O(n)	O(n)
Sorted Array	O(n)	O(logn)	O(n)	O(n)	O(logn)	O(n)
Binary Search Tree	O(n) O(h)	O(n)	O(n)	O(logn)	O(logn) balanced	O(logn)
Hash Table (Perfect Hash)	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)
Direct Access Table	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)

Algorithm	Best Time Complexity	Average Time Complexity	Worst Time Complexity	Worst Space Compl
Linear Search	O(1)	O(n)	O(n)	O(1)
Binary Search	O(1)	O(log n)	O(log n)	O(1)
Bubble Sort	O(n)	O(n^2)	O(n^2)	O(1)
Selection Sort	O(n^2)	O(n^2)	O(n^2)	O(1)
Insertion Sort	O(n)	O(n^2)	O(n^2)	O(1)
Merge Sort	O(nlogn)	O(nlogn)	O(nlogn)	O(n)
Quick Sort	O(nlogn)	O(nlogn)	O(n^2)	O(log n)
Heap Sort	O(nlogn)	O(nlogn)	O(nlogn)	O(n)
Bucket Sort	O(n+k)	O(n+k)	O(n^2)	O(n)
Radix Sort	O(nk)	O(nk)	O(nk)	O(n+k)
Tim Sort	O(n)	O(nlogn)	O(nlogn)	O(n)
Shell Sort	O(n)	O((nlog(n))^2)	O((nlog(n))^2)	O(1)

merge req space

Space

	Access	Search	Insertion	Deletion	
Array	O(1)	O(n)	O(n)	O(n)	O(n)
Stack	O(n)	O(n)	O(1)	O(1)	O(n)
Queue	O(n)	O(n)	O(1)	O(1)	O(n)
Singly-Linked List	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	O(n)	O(n)	O(1)	O(1)	O(n)
Hash Table	N/A	O(1)	O(1)	O(1)	O(n)
Binary Search Tree	O(n)	O(n)	O(n)	O(n)	O(n)
B-Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Red-Black Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)

not perfect

CUT-ROD(p, n)
1. if n == 0
2. return 0
3. q = −∞
4. for i = 1 to n
5. q = max(q, p[i] + CUT-ROD(p, n − i))
6. return q

MEMOIZED-CUT-ROD-AUX(p, n, r)
1. if r[n] ≥ 0
2. return r[n]
3. if n == 0
4. q = 0
5. else q = −∞
6. for i = 1 to n
7. q = max(q, p[i] + MEMOIZED-CUT-ROD-AUX(p, n − i, r))
8. r[n] = q
9. return q

MEMOIZED-CUT-ROD(p, n)
1. let r[0..n] be a new array
2. for i = 0 to n
3. r[i] = −∞
4. return MEMOIZED-CUT-ROD-AUX(p, n, r)

BOTTOM-UP-CUT-ROD(p, n)
1. let r[0..n] be a new array
2. r[0] = 0
3. for j = 1 to n
4. q = −∞
5. for i = 1 to j
6. q = max(q, p[i] + r[j − i])
7. r[j] = q
8. return r[n]

1 sub, n check choices
n−1 0

EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
1. let r[0..n] and s[0..n] be new arrays
2. r[0] = 0
3. for j = 1 to n
4. q = −∞
5. for i = 1 to j
6. if q < p[i] + r[j − i]
7. q = p[i] + r[j − i]
8. s[j] = i
9. r[j] = q
10. return r and s

PRINT-CUT-ROD-SOLUTION(p, n)
1. (r, s) = EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
2. while n > 0
3. print s[n]
4. n = n − s[n]

RECURSIVE-ACTIVITY-SELECTOR(s, f, k, n)
1. m = k + 1
2. while m ≤ n and s[m] < f[k] // find the first activity in S_k to finish
3. m = m + 1
4. if m ≤ n
5. return $\{a_m\}$ ∪ RECURSIVE-ACTIVITY-SELECTOR(s, f, m, n)
6. else return ∅

GREEDY-ACTIVITY-SELECTOR(s, f) $\Theta(n)$ if sorted
1. n = s.length
2. A = $\{a_1\}$
3. k = 1
4. for m = 2 to n
5. if s[m] ≥ f[k]
6. A = A ∪ $\{a_m\}$
k = m ; return A

~~matrix chain~~
2 → j−1

Knapsack problem dynamic prog

```
for j from 0 to W do:
    m[0, j] := 0

for i from 1 to n do:
    for j from 0 to W do:
        if w[i] > j then:
            m[i, j] := m[i-1, j]
        else:
            m[i, j] := max(m[i-1, j], m[i-1, j-w[i]] + v[i])
```

0-1 O(nW)
fractional → O(nlogn)

- DS → store & org data, set of op perform on them
- ADT → abs. DS, set of ops, impl independent } encaps, abs. n

Operations → set / type (primary)

set: · Name · 1 · Process · 0

type (primary): construct, access, manipulate

- algo → comp task (vary I) → deterministic: same O for same I; non: diff O for same I
- factors affecting run(time) of prog → Size of I, cpu speed, mem, processor; Nature
  - ↳ consider HW indep: { speed, space, simple } adv / disadv } algos, avail cpu, reqd by Algo

Sorting algo
- properties
  - online/offline (can be accepted while sorting)
  - stable/unstable (order) → 5, 2, 2, 7 → 2 2 5 7
  - inplace/outplace

- recursion = algo calls itself dir/indir to solve smaller ver of its task
  - div → conquer → combine
  - 1) def sup problems with smaller instances
  - 2) base case (terminating case)
  - terminating cond

DP

↑ tabular or ↓ catch memo

① opt substructure
- opt soln contains within opt soln to sub probs
- opt profit from picking i items ↳ can use to find i+1 th item

② overlapping subprobs (space b) ← independent ← bd by poly not (2^n)

Greedy → ⑧ Greedy choice prop
- can achieve globally opt soln by making locally opt solns } sub prob
- assume the by method not mentioned
- fractional → compute val/kg (P_i/ω_i), sort (b), take as much can | if more 2 fraction

Array vs linked list

- ↓ x search
- only complete BT
- arbitrary insert & delete not efficient

dp vs divide & conquer

do not share resources, solve each sub prob seperately

- interval } activity
- sched } selection

DP steps

1) Characterize structure of optimal soln
2) def val of optimal soln recursively
3) compute optimal soln
4) construct ″

Hashtables

- open(not locked) hashing(val stored)=>chaining
- closed hashing,open addressing(indexes vary)=>cluster
- $h(k,i) = (h'(k) + i) \bmod m$
- $h(k,i) = (h'(k) + c_1 i + c_2 i^2) \bmod m$
- $h(k,i) = (h_1(k) + i\,h_2(k)) \bmod m$

Chaining

openaddr → linear, quad, double hash

$\alpha = \frac{n}{m}$ ← #elem / size; Avg size, keys

primary clustering
- tendency for a collision resolution scheme such as linear prob to create long runs of filled slots near hash pos of keys.
- primary hash ind =x, resolve to x+1,.. =>prim cluster
- once cluster formed it is like to grow from similar hash collisions. performance decrease(resolving further collisions and searching keys within the cluster will take longer)

Secondary clustering
- tendency for a collision resolution scheme such as quad prob to create long runs of filled slots (quadratically) away from the original hash pos.
- primary hash ind =x, resolve to x+1,4,9...
- less likely to occur than primary clustering since the inserted items' hash values must exactly match the previous collided hash value to form secondary clustering

double hashing

even collide into same hash index the probe seq is diff

no prim x secondx

linear & quad prop seq depend only on probe seq index

Chaining	open address
delete ↓	delete → mark as deleted (or rearrange)
search ↓	search
Access	

$h(k,i) = (h'(k) + c i^2) \bmod m$

- efficiency hashtab → good hash fn, increase size (less collisions), better collision resolution techs
- del → mark as del, not null
- des hash fn → k mod m (m prime ll not closer to 2^n) division
  - → $m[kA \bmod m]$, $0<A<1$, $m=2^n$ Mult
  - → Universal → fam of fns with math prop

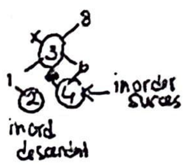
	Chain	Open address
•	implement easy	× comput t ↑, After full ×
•	↓ mem for ↑ datasize	↓ mem for data

- used when freq & # keys are known
- cache perf ↓ → ↑ same table everything

Root
- ↓ ancestor, ○ ↓ descendent, ○ leaf/no ch
- ↕ depth = # edges root → x
- ↕ height = # edges x → leaf

- K-ary tree (at most k children) → # nodes depth = k^d; → # internal nodes = $\frac{k^h - 1}{k-1}$

- distinct BST from unique keys = $\frac{(2n)!}{n!(n+1)!}$?
- successor → min right; predecessor → max left



Tree → part of DFS tree (undirected)

F → descendent but not DFS tree

B → ancest ″ } cycle (DFS)

C → no ances / des → ″

- deg = 3, in, out; simple path → no rep V (no circle) cy

dense $|E| \approx |V|^2$

sparse $|E| << |V|^2$

Strongly connected ⇒ every V reachable from every V

$|E| = |V| - 1$ tree; $|E| \geq |V| - 1$ tree; $|E| < \frac{n(n-1)}{2}$ } undirected

	space	t
adj list	O(V+E)	O(degree(u)) ⇒ V
adj mat	O(V^2)	O(1)

weight = [2, 2, 6, 4, 5] min

profit = [12, 25, 24, 15, 14] max

n = 5 items

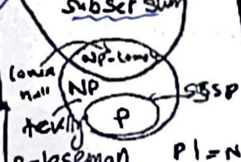
c = 10 restriction

#	W	0	1	2	3	4	5	6	7	8	9	10
5	5	0	0	0	0	0	14	14	14	14	14	14
4	4	0	0	0	0	15	15	15	15	15	29	29
3	6	0	0	0	0	15	15	24	24	29	29	39
2	2	0	0	25	25	25	25	40	40	49	49	49
1	2	0	0	25	25	37	37	40	40	52	52	64

max { not up, including prof + capacity − weight profit }

rod ⇒ $r_n = \max_{1 \leq i \leq n} (p_i + r_{n-i})$

0/1 ⇒ $P(i,k) = \begin{cases} 0 & i = n,\ w_i > k \\ P_n & i = n,\ w_i \leq k \\ P(i+1,k) & i < n,\ w_i > k \\ \max\{P(i+1,k),\ P_i + P(i+1, k-w_i)\} & i < n,\ w_i \leq k \end{cases}$



travelling salesman

P != NP

one exist ⇒ all exist

NP: soln guessed & verified in polyn t

NP comp: comp nal (subset) probs; no effic algos has been found; cannot prove it exist not

$f(n) = \Theta(g(n)) = \{ \exists$ const $c_1, c_2, n_0 > 0$ st $\forall n \geq n_0,\ 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \}$

↑ tighter bd

$\lim_{n \to \infty} \frac{f}{g}$

O	1 or 0 $< \infty$ cannot be const	True for $a_n g(n)$ (Worst)	
o	0	cannot be const	
Θ	const $\in \mathbb{R}$	cannot 0 or ∞	tight [bd on worst, $\not\Rightarrow$ bd on every I, $\Theta(n) \neq \Theta(n^2)$]
ω	∞	cannot be const	
Ω	∞, 1	> 0	best

$n! \leq n^n$

$\log n! \leq n \log n$

$\ln n \leq n$

$n^k < c^n$

$\log_2 n = \boxed{\frac{\log_3 n}{\log_3 2}}$

base don't matter

$\log n \leq n\ \forall n \geq 1$

$(\log n)^k \leq n^\alpha$

Master

$T(n) = \begin{cases} c & \text{if } n < d \\ a\,T(\frac{n}{b}) + f(n) & \text{if } n \geq d \end{cases}$ $\boxed{a \geq 1,\ b > 1}$

sub probs solved, # divide, $O(n)$, State ϵ

if $f(n)$ is $O(n^{\log_b a - \epsilon})$ (bounded) $\Rightarrow T(n) = \Theta(n^{\log_b a})$ $k > 0$

$f(n) = \Theta(n^{\log_b a} \log^k n) \Rightarrow T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$

$f(n) = \Omega(n^{\log_b a + \epsilon}) \Rightarrow T(n) = \Theta(f(n))$

provided $a f(\frac{n}{b}) \leq \delta f(n)$ for some $\delta < 1$ for suff large n

polynomials fine but others

$T(n) = a\,T(\frac{n}{b}) + O(n^d)$

① $a, b, f(n)$

② $n^{\log_b a}$

③ compare ↳ case

$T(n) = \begin{cases} O(n^d) & \text{if } d > \log_b a \\ O(n^d \log n) & \text{if } d = \log_b a \\ O(n^{\log_b a}) & \text{if } d < \log_b a \end{cases}$

- fun call itself → direct / Indirect
  f1(){ f1() } f1(){ f2() } f2(){ f1() }
  Nested f(f(n-1))
- pending op at recur call → tail / head
- fun calling pttrn → linear / tree
  once f(n-1) f(n-1)+f(n-2)

infix → postfix

A+B*(C+D) → ABCD+*+

same prec → pop [+ -][* /][^]

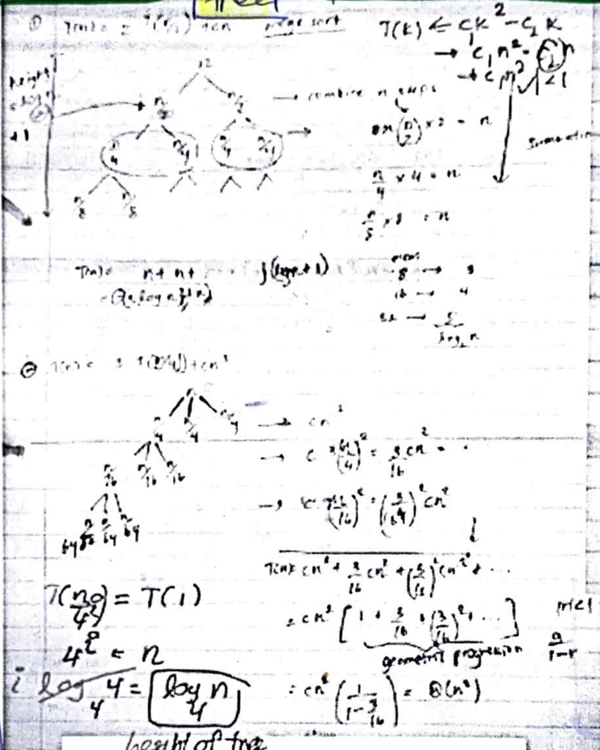
( ) → pop until (

postfix → infix

ABCD+*+

pop last 2 perform op → push stack

tree



$T(k) \leq ck^2 - c_1 k$

$\to c_1 n^2 - c_1 n$

$-c\,n^2 / \sqrt{2}$

$T(\frac{n}{4^i}) = T(1)$

$4^i = n$

$i = \log_4 4^i = \boxed{\log_4 n}$ height of tree

$x^{\log_b n} = n^{\log_b x}$

$x^0 + x^1 + x^2 + \cdots + x^n = \frac{x^{n+1} - 1}{x - 1}$ for $x \neq 1$

$x^0 + x^1 + x^2 + \cdots = \frac{1}{1-x}$ for $|x| < 1$

substitn method

Substitution method (induction)

should be proved exactly, not $cn\log n + n$ ✗

$T(n) = 2\,T(n/2) + an$

guess $T(n) = O(n \log n) \Rightarrow T(n) \leq cn \log n\ \exists c > 0\ \forall n \geq n_0$

assume above true for $m < n$

$1, 2, \ldots, n-1$ assumed true, n prove

let $m = \frac{n}{2} < n$

then $T(\frac{n}{2}) \leq c\frac{n}{2} \log \frac{n}{2}$ true

$T(n) = 2\,T(\frac{n}{2}) + an$

$\leq 2c\frac{n}{2} \log \frac{n}{2} + an$

$= cn[\log n - \log 2] + an$

$= cn\log n$ $n(1-c) \leq 0$, $c \geq 1$

$\therefore T(n) \leq cn \log n$ by induction $T(n) = O(n \log n)$ $\forall n \geq n_0$

$c_0^b n^b \leq (n+a)^b \leq 2^b n^b$

$T(n) = T(\sqrt{n}) + n$

$= T(n^{1/2}) + n$

$= T(n^{1/2^2}) + 2n$

$T(n) = T(n^{\frac{1}{2^k}}) + kn$

$n = 2^m$

$T(2^m) = T(2^{\frac{m}{2^k}}) + k 2^m$

$T(2)$

$m = 2^k \Rightarrow k = \log m = \log_2 \log_2 n$

$= \log \log n + kn$

$\Theta(n)$

MASTER

$T(n) = a\,T(n-b) + f(n)$, $a > 0$, $b > 0$, $f(n) = O(n^k)$; $k \geq 0$

$a < 1 \Rightarrow O(n^k)$

$a = 1 \Rightarrow O(n^{k+1})$

$a > 1 \Rightarrow O(a^{n/b} n^k)$

$T(n) = a\,T(\frac{n}{b}) + f(n)$, $a \geq 1$, $b > 1$, $f(n) = n^k \log^p n$; $p \geq 0$ or Θ

extended

① $\log_b a > k \Rightarrow \Theta(n^{\log_b a})$

② $\log_b a = k \Rightarrow p > -1 \Rightarrow \Theta(n^k \log^{p+1} n)$

$p = -1 \Rightarrow \Theta(n^k \log \log n)$

$p < -1 \Rightarrow \Theta(n^k)$

③ $\log_b a < k \Rightarrow p \geq 0 \Rightarrow \Theta(n^k \log^p n)$

$p < 0 \Rightarrow O(n^k)$

$f(n) \Rightarrow$ polynomial $\Rightarrow$ else ✗ works

$f(n)/n^{\log_b a} < n^\epsilon$ for any $\epsilon > 0$ $\therefore$ diff not polynom master ✗

$T(n) = T(n-1) + n$

$T(n) = \begin{cases} 1 & ; n = 0 \\ T(n-1) + n & ; n > 0 \end{cases}$

$T(n) = T(n-1) + n$

$= T(n-2) + (n-1) + n$

$\vdots$

$= T(n-k) + [n-(k-1)] + \cdots + (n-1) + n$

$n - k = 0$, $k = n \Rightarrow T(0) + \underbrace{1 + 2 + \cdots + n}_{n(n+1)/2}$