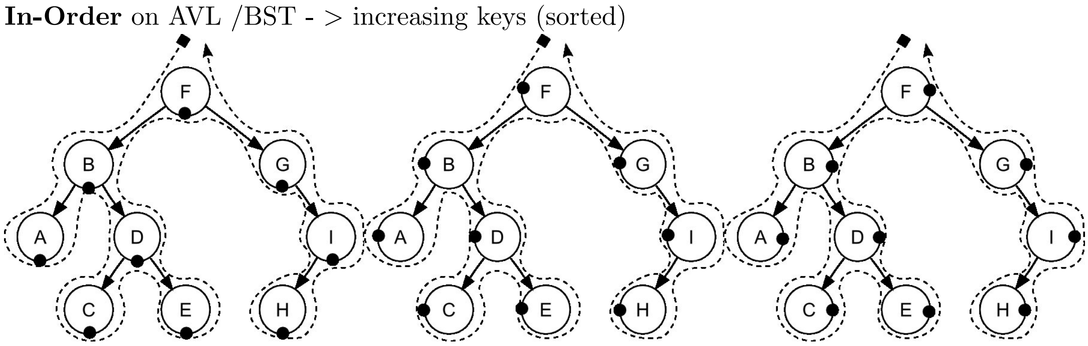
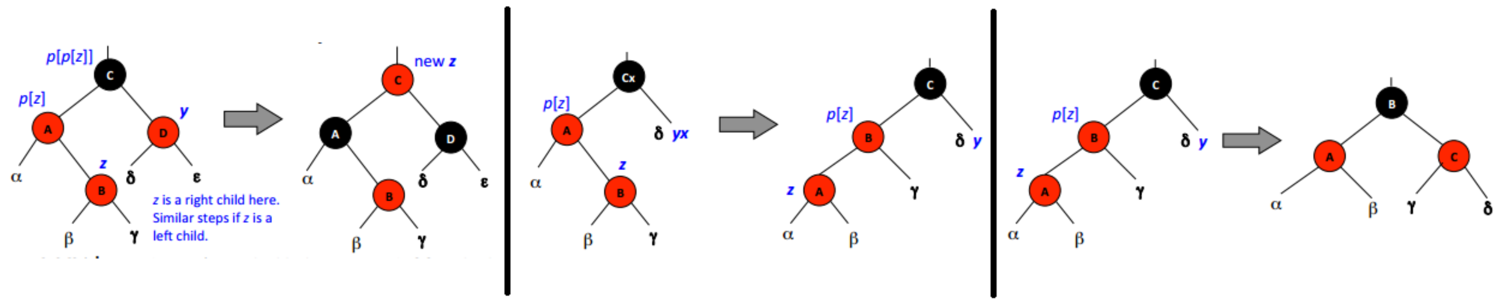
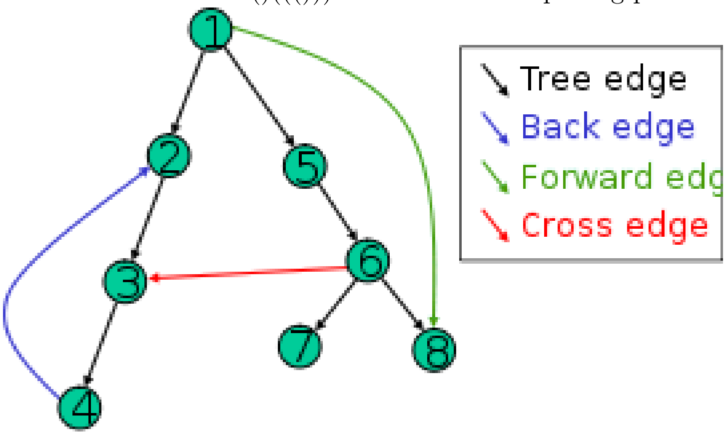
$B-A-W: MergeSort: O(nlogn) HeapSort: O(nlogn) QuickSort: O(nlogn) - O(nlogn) - O(n^2) InsertionSort: O(n) - O(n^2) - O(n^2) BubbleSort: O(n) - O(n^2) - O(n^2)$
Heap OP: RemoveMin()-replace by last node. Bubble down-swap with smallest child.
RemoveMax() -replace by last node. Bubble down-swap with largest child
Insert() - add at the end of array (rightmost last node)
BST OP: BST remove(i) - find smallest child c in rightmost child of i. replace i with c. remove original c.
Unsuccessfull search- $O(1+\alpha)$. Successfull - $\Theta(1+\alpha)$.
Division method: $h(k) = k \ mod \ d, d = 2^r$, r prime, not close to power of 2 or 10.
Multiplication method: $h(k) = Ak \ mod \ 2^w << w - r, 2^{w-1} < A < 2^w$.
Open addressing: Try hash function. If slot taken, take new function and retry.
Linear probing: $h(k,i) = (h'(k) + i) modm$. **Quad:** $h(k,i) = (h'(k) + c_1 i + c_2 i^2) modm$.
Double Hashing: $h(k,i) = (h_1(k) + ih_2(k)) modm$.
Universal Hashing: $\#functions \ h(k) = h(1) \leq H/m, k \in m \ keys$.
Max-heap: Max @ top **Min-heap:** Mean @ top.
Heap as array: Left[i]=A[2i], Right[i]=A[2i+1], Parent[i]=A[i/2] **Operations:** $O(log\ n)$.
MaxHeapify: MaxHeapify(A, i, n)
1. l ← leftNode(i)
2. r ← rightNode(i)
3. if l heap-size[A] and A[l]¿A[i]
4. then largest ← l
5. else largest ← i
6. if r ≤ n and A[r] > A[largest]
7. then largest ← r
8. if largest ≠ i
9. then exchange A[i] ↔ A[largest]
10. MaxHeapify(A, largest)
BuildMaxHeap: Maxheapify(A, i,length(A)) i from length(A)/2 to 1.
Heapsort(A):
1. Build-Max-Heap(A)
2. for i length[A] down to 2
3. exchange A[1] A[i]
4. MaxHeapify(A, 1, i-1)
AVL Tree: BST , $h_{left} - h_{right} \leq 1$
Insert at downmost leftmost child (as BST), $O(logn)$.
Restore AVL property at x:
if x.rightchild is right unbalanced or balanced:
left rotate.
else right rotate then left rotate.
if x.leftchild is left unbalanced or balanced:
right rotate.
else left rotate then right rotate.
continue with x's ancestors.
In-Order on AVL /BST - > increasing keys (sorted)



InOrder: A,B,C,D,E,F,G,H,I-**PreOrder:** F,B,A,D,C,E,G,I,H-**PostOrder:** A,C,E,D,B,H,I,G,F
AVL sort -worst:$O(nlogn)$ if balanced. **BST sort**-best:$O(nlogn)$, worst:$O(n^2)$.
R-B Tree: BST. Root is black. NIL is black. No consecutive red nodes.All black heights are same from node to descendants. If no child-put NIL. **Black-height(x):** # black nodes from x to NIL (count NIL, dont count x).
R-B insert(x): BST insert x as red then restore R-B property.



Sets: In forest representation, root==representative.
Union: By size: smallest into biggest. By height: shortest into tallest.
Path compression: All nodes parent= representative.
Greedy: Local optimal choice, delegate subproblem recursively. For interval, start from beginning and minimize waste of space.
Huffman Trees: Highest frequency letters on top.
Graphs: $|E| = |V| - 1 - >$ it is a tree. Store weights or booleans in **adjacency matrix**.
Color Code: **White:** undiscovered . **Grey:** neighbors unvisited . **Black:** done.
d[u]=smallest # of edges from s to any u.
π[u]=v, v is the predecessor of u.
BFS: O(V+E). **DFS:** Θ(V+E).
Parenthesis Thm: ()((())). Cannot finish exploring parent before child.



DAG: Directed Acyclic Graph (tree is a DAG).**Partial order:** $a > b, b > c \rightarrow a > c$ but maybe a=b. **Total order:** Always $a > b$ or $a < b$.
Topological Sort: On DAG, no back edges. $\Theta(V + E)$. **Strongly Connected Component** if we can reach u from v, for any u and v in a subset of G. G^T has edges flipped (forward=backward).
Compute SCC: DFS(G), then G^T, then DFS(G^T) (in order of deacreasing f[u], as computed by DFS(G)).
d[u]: start time.**f[u]:** finish time.
MST: Connect all edges. Cut respects A - no edge in A crosses it. **Safe edge**: smallest weight edge between A and V-A.
Kruskal:
1. Starts with each vertex in its own component (1 partition / vertex).
2. Repeatedly merges two components into one by choosing a light edge that connects them (i.e., a light edge crossing the cut between them).
3. Scans the set of edges in monotonically increasing order by weight.
4. Uses a disjoint-set data structure to determine whether an edge connects vertices in different components.
Union by rank + path compression : total runtime = O(E log V)
Prim's Algorithm:
1. Builds one tree, so A is always a tree.
2. Starts from an arbitrary root r .
3. At each step, adds a light edge crossing cut $(V_A, V - V_A)$ to A. Where V_A = vertices that A is incident on.
Find a light edge (for Prim):
1. Uses a priority queue Q to find a light edge quickly.
2. Each object in Q is a vertex in V - VA.
3. Key of v is minimum weight of any edge (u, v), where u V_A.
4. Then the vertex returned by Extract-Min is v such that there exists u V and (u, v) is light edge crossing $(V_A, V - V_A)$.
5. Key of v is if v is not adjacent to any vertex in VA.
Prim - add lightest edges to neighbors until all reachable. Kruskal - connect partitions with safe edges until partition= whole graph. **Shortest paths:** d[v]- **shortest path estimate.** (initially = ∞).$\pi[v]$ - predecessor of v (initially = NIL or if none).
Relax an edge (if exists shorter path):
if (d[v] > d[u]+w(edge between u and v)) then d[v]=d[u]+w(edge between u and v), π[v]=u.
Dijkstra:

```
DIJKSTRA( V, E,w, s)
INIT-SINGLE-SOURCE( V, s)
S = {}
Q = V
while Q non empty do
u = EXTRACT-MIN( Q)
S = S U {u}
for each vertex v      Adj[ u] do
RELAX( u,v,w)
```

If we use binary heap, runtime = O(E log V)
Gale-Shapley:

```
matching = empty
while there is a in A not yet matched do
B =pref[a].removeFirst()
if B not yet matched then
                  matching = matching U{(a,B)}
else
                  G = B current match
                  if B prefers a over G then
                          matching = matching -{(G,B)} U {(a,B)}
return matching
```

Residual graph:

```
for each edge e=(u,v) in E
if f(e) < c(e)
then {
                  put a forward edge (u,v) in Residual graph
                  with residual capacity cf(e)=c(e)-f(e)
}
if f(e)>0
then {
                 put a backward edge (v,u) in Residual graph
                 with residual capacity cf(e)=f(e)
}
}
```

Augmenting path: From source to sink via residual graph. Follow lines and add capacity as you go
Final solution: add G + residual graph paths . (take difference of edges flow if backwards + forward)
Ford-Fulkerson:

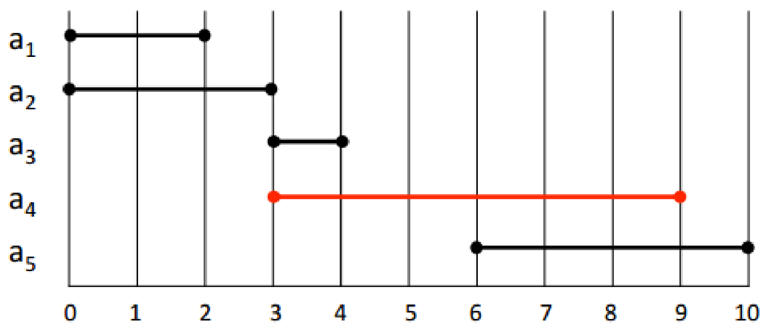
```
f =0
Residual=G
while (there is a source to sink path in Residual){
f.augment(Path)
update Residual based on new f
}
```

Flow through a cut: $|f| = \sum$ f(e) from A to B - $\sum$ f(e) from B to A.
Minimum cut:
1. Run Ford-Fulkerson for max flow.
2. Run BFS /DFS from s on Residual.
3. All reachable vertices define the cut.

$\sum_{k=0}^{n-1} ar^k = a\frac{1-r^n}{1-r}$.
$BST\ amount\ of\ nodes = 2^{h+1} - 1$
$RBT\ h \leq 2\log_2(n+1)$
Memoization: Remember results of subproblem to reuse later. $M[i]$ value.

activity	1	2	3	4	5
predecessor	0	0	2	2	3
Best weight	2	3	4	10	10
V_j+M[p(j)]	2	3	4	10	8
M[j-1]	0	2	3	4	10

(1) Activities sorted by finishing time. (2) Weight equal to the length of activit



Alignment: Insertion/Deletion: letter mapped to empty slot **Count alignments:** $c(m,n) = c(m-1,n) + c(m-1,n-1) + c(m,n-1)$. Use memoization +DP to remember

Levenstein dist: How to transform one string into another with minimal subs, insert/delete.

Edit cost:
$$\delta(x,y) = \begin{cases} 1 & \text{if } x = y \\ -1 & \text{otherwise} \end{cases}$$

ABB−CEE
−BBCCDE

Needleman-Wunch Algorithm

```
for i=0 to m do
  d(i,0)=i*δ(-,-)
for j=0 to n do
  d(0,j)=j*δ(-,-)

for i=1 to m do
     for j=1 to n do
       d(i,j) = max(d(i-1,j)+δ(a_i,-),
                    d(i-1,j-1)+δ(a_i,b_j),
                    d(i,j-1)+δ(-,b_j))
return d(m,n)
```

a=ATTG b=CT

$$\delta(x,y) = \begin{cases} 1 & \text{if } x = y \\ -1 & \text{otherwise} \end{cases}$$

	-	A	T	T	G
-	0	-1	-2	-3	-4
C	-1	-1	-2	-3	-4
T	-2	-2	0	-1	-2

Backtrac

Vertical-insert, Horizontal-Delete,Diagonal-math/substitute.

KNAPSACK $(n, W, w_1, \ldots, w_n, v_1, \ldots, v_n)$

FOR $w = 0$ TO W
  $M[0, w] \leftarrow 0.$

FOR $i = 1$ TO n
  FOR $w = 1$ TO W
  IF $(w_i > w)$ $M[i, w] \leftarrow M[i-1, w].$
  ELSE $M[i, w] \leftarrow \max\{M[i-1, w],\ v_i + M[i-1, w-w_i]\}.$

RETURN $M[n, W]$.

MULTIPLY(x, y, n)

IF $(n = 1)$
  RETURN $x \times y$.
ELSE
  $m \leftarrow \lceil n/2 \rceil.$
  $a \leftarrow \lfloor x/2^m \rfloor$; $b \leftarrow x \bmod 2^m.$
  $c \leftarrow \lfloor y/2^m \rfloor$; $d \leftarrow y \bmod 2^m.$
  $e \leftarrow$ MULTIPLY(a, c, m).
  $f \leftarrow$ MULTIPLY(b, d, m).
  $g \leftarrow$ MULTIPLY(b, c, m).
  $h \leftarrow$ MULTIPLY(a, d, m).
  RETURN $2^{2m} e + 2^m (g + h) + f.$

KARATSUBA-MULTIPLY(x, y, n)

IF $(n = 1)$
  RETURN $x \times y$.
ELSE
  $m \leftarrow \lceil n/2 \rceil.$
  $a \leftarrow \lfloor x/2^m \rfloor$; $b \leftarrow x \bmod 2^m.$
  $c \leftarrow \lfloor y/2^m \rfloor$; $d \leftarrow y \bmod 2^m.$
  $e \leftarrow$ KARATSUBA-MULTIPLY(a, c, m).
  $f \leftarrow$ KARATSUBA-MULTIPLY(b, d, m).
  $g \leftarrow$ KARATSUBA-MULTIPLY$(a-b, c-d, m)$.
  RETURN $2^{2m} e + 2^m (e + f - g) + f.$

$$T(n) = a\,T\left(\frac{n}{b}\right) + f(n) \quad \text{where } a \geq 1,\ b > 1$$

If $f(n) = O(n^c)$ where $c < \log_b a$ => $T(n) = \Theta\left(n^{\log_b a}\right)$ Case (1)

$f(n) = \Theta\left(n^c \log^k n\right)$ where $c = \log_b a$ => $T(n) = \Theta\left(n^c \log^{k+1} n\right)$ Case (2)

$f(n) = \Omega(n^c)$ where $c > \log_b a$, $af\left(\frac{n}{b}\right) \leq kf(n)$ for some constant $k < 1$ => $T(n) = \Theta(f(n))$ Case (3)

Binary counter: Amounts of bits flipped
Value=$\sum_{i=0}^{k-1} A[i]2^i$
Intuition: previous cost of flipping + cost to arrive to current state from previous (i.e. undo bits from last call) Ex: 5:101 =8 means 6:110=10

```
Increment (A,k)
i = 0
while i <k and A[i]=1 do
        A[i] = 0
        i++
if i < k then
        A[i]=1
```

Amortized cost: Amount charged per operation. We can prepay (i.e. pay for pop when pushing instead of paying separately)
Ex: set bit to 1 = 2$. Set bit to zero = 0$ since we prepaid.
Aggregate Analysis: cost(i)=i if i-1 is a power of 2, cost(i)=1 otherwise.
Contraction: Randomized. (1)Take edge randomly from u to v.(2) Merge u and v into w. (3) Update edges: keep parallel edges, but delete self loops (delete edges between u and v) (4) repeat until 2 nodes left, v_1, v_2 (5) Return all nodes that were contracted to form v_1.
Runtime: $\Theta(n^2 \log n)$ iterations, each takes $\Omega(m)$ time.
3-SAT Choose values of 3 booleans s.t. they make as much clauses true as possible. A random assignment of 3-SAT will satisfy on avg. $7/8k$, where k is the amount of clauses. **Lemma** $P(satisfy \quad \geq \; 7/8k \;\; clauses) \geq 1/(8k)$

STRASSEN(n, A, B)

IF $(n = 1)$ RETURN $A \times B$.
Partition A and B into 2-by-2 block matrices.
$P_1 \leftarrow$ STRASSEN$(n/2, A_{11}, (B_{12} - B_{22}))$.
$P_2 \leftarrow$ STRASSEN$(n/2, (A_{11} + A_{12}), B_{22})$.
$P_3 \leftarrow$ STRASSEN$(n/2, (A_{21} + A_{22}), B_{11})$.
$P_4 \leftarrow$ STRASSEN$(n/2, A_{22}, (B_{21} - B_{11}))$.
$P_5 \leftarrow$ STRASSEN$(n/2, (A_{11} + A_{22}) \times (B_{11} + B_{22}))$.
$P_6 \leftarrow$ STRASSEN$(n/2, (A_{12} - A_{22}) \times (B_{21} + B_{22}))$.
$P_7 \leftarrow$ STRASSEN$(n/2, (A_{11} - A_{21}) \times (B_{11} + B_{12}))$.
$C_{11} = P_5 + P_4 - P_2 + P_6$.
$C_{12} = P_1 + P_2$.
$C_{21} = P_3 + P_4$.
$C_{22} = P_1 + P_5 - P_3 - P_7$.
RETURN C.

Monte-Carlo Poly-time, likely to find right answer (ex:contraction)**Las Vegas**Will find correct answer, likely to be poly-time (ex:rand. quicksort, 3-sat).
Quicksort: Slow on small lists -¿ use another one on subproblems (insert. sort). **Randomized QuickSort:** select random pivot between left and right bounds.
Indicator rv: $I(A) = 1$ iff A occurs, 0 otherwise. $E(I\{A\}) = P(A)$
Deterministic: Same output with same input. **Probabilistic:** Different output with same input.