recurrence	solution
`T(n) = T(n/2) + O(1)`	$O(\log n)$
`T(n) = T(n/2) + O(n)`	$O(n)$
`T(n) = 2T(n/2) + O(1)`	$O(n)$
`T(n) = 2T(n/2) + O(n)`	$O(n \log n)$
`T(n) = T(n-1) + O(1)`	$O(n)$
`T(n) = T(n-1) + O(n)`	$O(n^2)$
`T(n) = 2T(n-1) + O(1)`	$O(2^n)$

Tower of Hanoi: $T(n) = 2T(n-1) + 1, T(1) = 0.$

Searching a linked list: $T(n) = T(n-1) + 1, T(0) = 0.$

Insertion sort: $T(n) = T(n-1) + n, T(0) = 0.$

Binary search: $T(n) = T(n/2) + 1, T(1) = 0$ makes sens

Mergesort: $T(n) = 2T(n/2) + n, T(1) = 0$ makes sense

Quicksort: $T(n) = (2/n)\sum_{i<n} T(i) + n - 1, T(1) = 0.$

Running time		**Input size**			
Function	*Notation*	10	100	1000	10^7
Constant	1	1	1	1	1
Logarithmic	$\lg n$	1	2	3	7
Linear	n	1	10	100	10^6
"Linearithmic"	$n \lg n$	1	20	300	7×10^6
Quadratic	n^2	1	100	10000	10^{12}
Cubic	n^3	1	1000	10^6	10^{18}
Exponential	2^n	1	10^{27}	10^{298}	$10^{3010296}$

	Best	Average	Worst	STABLE	IN-PLACE	SWAPS	COMPARISONS
Selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	No (linked-lists Yes)	Yes	depends	comparisons same for any list n^2
Insertion sort	O(n)	$O(n^2)$	$O(n^2)$	Yes	Yes	swaps = #inversions sorted-list = 0 inversions reverse = $\frac{n(n-1)}{2}$ inversions	n-1 + #inversions
merge sort	O(nlogn)	O(nlogn)	O(nlogn)	Yes	No (linked-lists Yes)	*The minimum possible number of comparisons occurs when all the elements in L1 are smaller than the first element * The maximum possible number of comparisons occurs when we sort of have to zig-zag between L1 and L2	
quicksort	O(nlogn)	O(nlogn)	$O(n^2)$	No	Almost	hash load factor n/m occupied entries / capacity	if median = pivot then good
heapsort	O(nlogn)	O(nlogn)	O(nlogn)	No	Yes	Use **Quickselect** to find the 3rd largest element of the list 25, 65, 50, 21, 2, 67, 70, 31, 15, 8. • Pivot = 25, partition into [21,2,15,8] 25 [65 50 67 70 31]; • Move into the right hand side; • Pivot = 65, partition into [50 31] 65 [67 70]; • 3rd largest element is the pivot; • Stop. Average = θ(n)	
Treesort	O(nlogn)	O(nlogn)	$O(n^2)$				
Hashing	Add/Find/Delete in time O(1) avg / O(n) worst						

$1024! \ll 100^{1024} \log_{1024}(n) \ll 10^{1024} \log_3(n) \ll 10^{1/2}\, n \ll 1/5\, n \log_5(n^3) \ll 12n^{5/4} \ll 8n^2 \ll 6n^7 \ll 10^{n-4} \ll 1024n!$

$\log n < (\log n)^2 < \sqrt{n} < n < n \log n < n(\log n)^2 < n^2 < n^2 < \cdots < (1.5)^2 < 2^2 < n! < n^2.$

f is O(g): f smaller than g (f grows at most as fast as g)
f is Ω(g): f is larger than g, and g is O(f) (f grows at least as fast as g)
f is Θ(g): f is equal to g, f is both O(g) and Ω(g) (f grows at the same rate as g)

SS	-Running time on a **linked list** is not much different from performance on an array. The differences involve swap versus insertion, but comparisons dominate the running time anyway. -There is no better way to find the maximum using a list than what we have done above. Except for heapsort. -The algorithm is very insensitive to the input. Its best and worst case running time are very similar. **The best case is when the list is already sorted; the worst is when every swap is needed**, and this occurs when the input permutation has no fixed point. However the **number of comparisons is the same for every input**. -it makes the smallest possible number of swaps of any comparison-based sorting algorithm, so may be useful if data moves are VERY expensive
IS	-We can reduce the number of data moves if we use a **linked list**. However, searching to find the right insertion point still takes time in $\Theta(n^2)$ in the worst case. insertion sort has running time that is very sensitive to the input. **The best case occurs when the input is already sorted, and the worst when the input is in reverse sorted order.**
MS	-splitting is very easy, most of the work is in combining -if **linked lists** are used, it can be done in place. In any case the number of comparisons is in Θ(n) -Mergesort is not very sensitive to the input order. If the input is already sorted, the merge operation does the fewest possible comparisons. The worst case occurs when the input looks like 5,1,7,3,6,2,8,4 (every possible comparison is made at every level of the merge).
QS	-most of the work is in the splitting, combining is very easy. -quicksort is almost never used on a **linked list**. It is too difficult to quickly find a good pivot element. -The worst case number of comparisons occurs when the pivot is always at the end of the sublist. For example, if we always choose the first element as the pivot and the input is in sorted order, this will happen. The running time is then $\Theta(n^2)$. The best case occurs when the pivot turns out to be the median element, so that the left and right subarrays are balanced at each level of the recursion, and this gives running time in Θ(n log n) as with mergesort.
D.TREE	-This is a binary tree whose leaves are the outputs of the algorithm. There are n! possible outputs or leaves -The number of comparisons required to obtain the output at a leaf is the length of the path from the root to that leaf (the depth of the leaf node). -The worst case number of comparisons is the maximum depth (the height) of the tree. - Going from the root, the number of nodes at each level at most doubles. if the maximum depth (height) is h then the number of leaves is at most 2^h
Q	-If we implement Q using an unsorted list, we obtain a selection sort. Insertion takes Θ(1) time but deletion time is Θ(n). The total is quadratic. -If we use a sorted list to implement Q, we obtain insertion sort. Insertion takes Θ(n) time but deletion is Θ(1). The total is quadratic. -We can do better with an implementation in which insertion and deletion each take time O(log n). The simplest is the binary heap.
Heap	-is left-complete (every level except perhaps the last is full and the last level is left-filled) -has the partial order property (on every path from the root, the keys decrease) -Building a heap using n successive insertions takes time in O(n log n) since the tree has height in O(log n) -Deleting the root n times, restoring the heap property each time, takes time in O(n log n) -most efficient implementation: build a complete binary tree without the heap property, then recursively heapify the left and right subtrees, and then let the root swap down to the right position.

BST	-The number of comparisons required to find the key is the depth of the leaf containing that key. Worst-case run-time: θ(logn) -The best case is when we find the element on the first try (it is in the middle position) - each node is ≥ every node in the left subtree and ≤ every node in the right subtree -The running time of all basic operations is proportional to the number of nodes visited -In the worst case, finding/removing/inserting take time in Θ(height) - the average search cost in a BST built by random insertions is Θ(log n) .	**BST vs Quicksort** The cost of constructing the tree is the same as the number of comparisons used by quicksort in sorting the file. This is equal to the internal path length, the sum of all depths of nodes

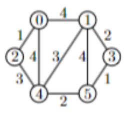
	Matrix	Details	List	Details	TRAVERSALS
Checking if an arc exists	Θ(1)	Directly access the entry in the matrix at row u and column v to check if 1	Θ(d)	Search through the list of neighbors of node u to see if v is present.	**DFS** sparse list: Θ(n+m), dense list / MATRIX: Θ(n^2) **(v,w) is a tree arc OR a forward arc** seen[v] < seen[w] < done[w] < done[v] and v is an ancestor of w **(v,w) is a back arc** seen[v] < done[v] < seen[w] < done[w] and v is NOT an ancestor of w **(v,w is a cross arc)** seen[w] < done[w] < seen[v] < done[v]
Outdegree of node	Θ(n)	Count the number of 1s in the row corresponding to node u.	Θ(1)	Directly return the length of the list of neighbors of node u.	
Indegree of node	Θ(n)	Count the number of 1s in the column corresponding to node v.	Θ(n+m)	Search through all lists to count occurrences of node v. With a maintained count, you can return it directly.	**BFS** sparse list: Θ(n+m), dense list / MATRIX: Θ(n^2) **BFS does not produce forward arcs.** **(v,w) is a tree arc** colour[w] = White and d[w] = d[v] + 1 The levels of w and v differ by 1 **(v,w) is a back arc** colour[w] = Black and d[w] <= d[v] - 1 **(v,w) is a cross arc** **d[w] < d[v] - 1 and colour[w] = Black**
adding an arc	Θ(1)	Set the entry in the matrix at row u and column v to 1.	Θ(1)	Append node v to the list of neighbors of node u	
deleting an arc	Θ(1)	Go to the column i, turn all 1's into 0,	Θ(d)	Locate source node, scan through this list to find the target node and remove it.	**PFS = Ω(n^2)** **TARJANS** O(n+m) **TOPOLOGICAL** O(n+m) DAG (A graph of Order n with a topological order has n connected components) **DIJKSTRA** O(m log n) Queue = Θ(n+m logn) **BELLMAN FORD** Θ(n*m) **FLOYDS O(m log n)** **PRIMS O(m log n)** **KRUSKAL O(m log n)**
adding a node	Θ(n)	Requires creating a new matrix with one additional row and column and copying the existing entries	Θ(1)	Add a new list (initially empty) for the new node.	
deleting a node	Θ(n^2)	Requires creating a new matrix without the row and column of the deleted node and copying the remaining entries.	Θ(n+m)	Remove the list of the node and remove all occurrences of the node from other lists. In the worst case, this involves scanning through all lists.	

	SSPS	APSP	-VE WEIGHTS
BFS	For unweighted graphs: n+m	For unweighted graphs: n(n+m)	NO
DIJKSTRA	Queue = Θ(n+m logn)	n*m logn	NO
BELLMAN	slower runtime than dijkstra. n*m	n2*m	YES
FLOYS	Yes. n3	Yes. n3	YES
Generally, unweighted digraph (dense) = Dijkstra For SSSP or APSP, use bellman ford on a sparse digraph, but for dense digraphs use Floyd.			

DFS can be used to determine whether or not a digraph is a DAG. If the traversal finds no back arcs, G is a DAG

The diameter of a strongly connected digraph G is the maximum of d(u, v) over all nodes u, v ∈ V(G). If the digraph is not strongly connected the diameter is undefined though can be set to ∞.

Example 31.3. An application of Floyd's algorithm on the graph on the left is given below. The initial cost matrix is as follows.



$$\begin{bmatrix} 0 & 4 & 1 & \infty & 4 & \infty \\ 4 & 0 & \infty & 2 & 3 & 4 \\ 1 & \infty & 0 & \infty & 3 & \infty \\ \infty & 2 & \infty & 0 & \infty & 1 \\ 4 & 3 & 3 & \infty & 0 & 2 \\ \infty & 4 & \infty & 1 & 2 & 0 \end{bmatrix}$$

In the matrices below, we list the entries that change in bold after each iteration of the outer for-loop, that is, after x has been 0, 1, and so on.

$$\underset{x=0}{\begin{bmatrix} 0 & 4 & 1 & \infty & 4 & \infty \\ 4 & 0 & \mathbf{5} & 2 & 3 & 4 \\ 1 & \mathbf{5} & 0 & \infty & 3 & \infty \\ \infty & 2 & \infty & 0 & \infty & 1 \\ 4 & 3 & 3 & \infty & 0 & 2 \\ \infty & 4 & \infty & 1 & 2 & 0 \end{bmatrix}} \quad \underset{x=1}{\begin{bmatrix} 0 & 4 & 1 & \mathbf{6} & 4 & \mathbf{8} \\ 4 & 0 & 5 & 2 & 3 & 4 \\ 1 & 5 & 0 & \mathbf{7} & 3 & \mathbf{9} \\ \mathbf{6} & 2 & \mathbf{7} & 0 & \mathbf{5} & 1 \\ 4 & 3 & 3 & \mathbf{5} & 0 & 2 \\ \mathbf{8} & 4 & \mathbf{9} & 1 & 2 & 0 \end{bmatrix}} \quad \underset{x=2}{\begin{bmatrix} 0 & 4 & 1 & 6 & 4 & 8 \\ 4 & 0 & 5 & 2 & 3 & 4 \\ 1 & 5 & 0 & 7 & 3 & 9 \\ 6 & 2 & 7 & 0 & 5 & 1 \\ 4 & 3 & 3 & 5 & 0 & 2 \\ 8 & 4 & 9 & 1 & 2 & 0 \end{bmatrix}}$$

$$\underset{x=3}{\begin{bmatrix} 0 & 4 & 1 & 6 & 4 & \mathbf{7} \\ 4 & 0 & 5 & 2 & 3 & \mathbf{3} \\ 1 & 5 & 0 & 7 & 3 & \mathbf{8} \\ 6 & 2 & 7 & 0 & 5 & 1 \\ 4 & 3 & 3 & 5 & 0 & 2 \\ \mathbf{7} & \mathbf{3} & \mathbf{8} & 1 & 2 & 0 \end{bmatrix}} \quad \underset{x=4}{\begin{bmatrix} 0 & 4 & 1 & 6 & 4 & \mathbf{6} \\ 4 & 0 & 5 & 2 & 3 & 3 \\ 1 & 5 & 0 & 7 & 3 & \mathbf{5} \\ 6 & 2 & 7 & 0 & 5 & 1 \\ 4 & 3 & 3 & 5 & 0 & 2 \\ \mathbf{6} & 3 & \mathbf{5} & 1 & 2 & 0 \end{bmatrix}} \quad \underset{x=5}{\begin{bmatrix} 0 & 4 & 1 & 6 & 4 & 6 \\ 4 & 0 & 5 & 2 & 3 & 3 \\ 1 & 5 & 0 & \mathbf{6} & 3 & 5 \\ 6 & 2 & \mathbf{6} & 0 & \mathbf{3} & 1 \\ 4 & 3 & 3 & \mathbf{3} & 0 & 2 \\ 6 & 3 & 5 & 1 & 2 & 0 \end{bmatrix}}$$

Example 30.1. An application of Bellman–Ford algorithm with starting node 4 when the nodes are processed in the order from 0 to 4.