avior for quicksort occurs when
outine produces one subproblem
and one with 0 elements.

ON procedure, which rearranges the subar-

, for each input element x,
ess than x. It uses this
ment x directly into its
rray.

nts equal to i.

rs less than or equal to i.

sorting by sorting on the least significant digit first.

invariant
sed

rted sequence $A[1 \,..\, j-1]$.

key

Merge Sort:

- proves first that the subroutine Merge correctly takes two sorted subarrays and merges them into a single sorted array. It then uses a Proof by Strong Induction to show that the outer, Merge Sort function properly sorts a given array.
- closely follows the divide-and-conquer paradigm
- Divide: Divide the n-element sequence to be sorted into two subsequences of n/2 elements each.
- Conquer: Sort the two subsequences recursively using merge sort.
- Combine: Merge the two sorted subsequences to produce the sorted answer.

MERGE-SORT(A, p, r)

```
if p < r
    q = ⌊(p + r)/2⌋
    MERGE-SORT(A, p, q)
    MERGE-SORT(A, q + 1, r)
    MERGE(A, p, q, r)
```

MERGE(A, p, q, r)

```
n1 = q - p + 1
n2 = r - q
let L[1 .. n1 + 1] and R[1 .. n2 + 1] be new arrays
for i = 1 to n1
    L[i] = A[p + i - 1]
for j = 1 to n2
    R[j] = A[q + j]
L[n1 + 1] = ∞
R[n2 + 1] = ∞
i = 1
j = 1
for k = p to r
    if L[i] ≤ R[j]
        A[k] = L[i]
        i = i + 1
    else A[k] = R[j]
        j = j + 1
```

Bellman-Ford:

- The Bellman-Ford algorithm runs in time O(VE), since the initialization in line 1 takes, theta(V) time, each of the |V|-1 passes over the edges in lines 2-4 takes theta(E) time, and the for loop of lines 5-7 takes O(E) time.
- solves the single-source shortest-paths problem in the general case in which edge weights may be negative.
- Bellman-Ford algorithm returns a boolean value indicating whether or not there is a negative-weight cycle that is reachable from the source. (returns TRUE if and only if the graph contains no negative-weight cycles that ar
- If there is such a cycle, the algorithm indicates that no solution exists. If the the algorithm produces the shortest paths and their weights.

BELLMAN-FORD(G, w, s)

```
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  for i = 1 to |G.V| - 1
3      for each edge (u, v) ∈ G.E
4          RELAX(u, v, w)
5  for each edge (u, v) ∈ G.E
6      if v.d > u.d + w(u, v)
7          return FALSE
8  return TRUE
```

0-1 Knapsack:

- Uses Dynamic rather than Greedy

Dynamic Programming:

- We typically apply dynamic programming

Dijkstra's:

- solves the single-source shortest-paths problem on a weighted, directed graph G ={V,E} for the case in which all edge weights are nonnegative.
- uses a greedy strategy

DIJKSTRA(G, w, s)

```
INITIALIZE-SINGLE-SOURCE(G, s)
S = ∅
Q = G.V
while Q ≠ ∅
    u = EXTRACT-MIN(Q)
    S = S ∪ {u}
    for each vertex v ∈ G.Adj[u]
        RELAX(u, v, w)
```

BFS:

- Uses FIFO queue
- finds the distance to each reachable vertex in a graph G ={V,E} from a given source vertex s in V.
- BFS below assumes that the input graph G ={V,E} is represented using adjacency lists.

DFS:

- a directed graph is acyclic if and only if a depth-first search yields no "back" edges (Lemma 22.11).
- explores edges out of the most recently discovered vertex that still has unexplored edges leaving it. Once all of 's edges have been explored, the search "backtracks" to explore edges leaving the vertex from which was discovered. This process continues until we have discovered all the vertices that are reachable from the original source vertex. If any undiscovered vertices remain, then depth-first search selects one of them as a new source, and it repeats the search from that source. The algorithm repeats this entire process until it has discovered every vertex.

Bucket sort:

- Linear-time
- assumes that the input is drawn from a uniform distribution
- Bucket sort divides the interval [0,1) into n equal-sized subintervals, or buckets, and then distributes the n input numbers into the buckets.

BUCKET-SORT(A)

```
n = A.length
let B[0 .. n - 1] be a new array
for i = 0 to n - 1
    make B[i] an empty list
for i = 1 to n
    insert A[i] into list B[⌊nA[i]⌋]
for i = 0 to n - 1
    sort list B[i] with insertion sort
concatenate the lists B[0], B[1], ..., B[n - 1] together in order
```

Huffman:

- total running time of HUFFMAN on a set of n characters is O(nlg(n))
- To prove that the greedy algorithm HUFFMAN is correct, we show that the problem of determining an optimal prefix code exhibits the greedy-choice and optimal substructure properties.

HUFFMAN(C)

```
n = |C|
Q = C
for i = 1 to n - 1
    allocate a new node z
    z.left = x = EXTRACT-MIN(Q)
    z.right = y = EXTRACT-MIN(Q)
    z.freq = x.freq + y.freq
    INSERT(Q, z)
return EXTRACT-MIN(Q)    // return the root of the tree
```

Algorithm	Worst-case running time	Average-case/ex running time
Insertion sort	$\Theta(n^2)$	$\Theta(n^2)$
Merge sort	$\Theta(n \lg n)$	$\Theta(n \lg n)$
Heapsort	$O(n \lg n)$	—
Quicksort	$\Theta(n^2)$	$\Theta(n \lg n)$ (exp
Counting sort	$\Theta(k + n)$	$\Theta(k + n)$
Radix sort	$\Theta(d(n + k))$	$\Theta(d(n + k))$
Bucket sort	$\Theta(n^2)$	$\Theta(n)$ (average