Algorithms: Time complexity and Proof of correctness

Contents Page

# 1. Representing graphs:

Directed -> Adjacency list

storage: O(V + E)

good for sparse graphs (O(|V|^2))

Searching (List traversal): O(|V|)

Undirected -> Adjacency matrix

Storage: O(||V^2||)

good for dense graphs

Searching: O(1)

# 2.Kruskal's

based on Euclidean distance.

## Complexity [ edit ]

Kruskal's algorithm can be shown to run in $O(E \log E)$ time, or equivalently, $O(E \log V)$ time, where $E$ is the number of edges in the graph and $V$ is the number of vertices, all with simple data structures. These running times are equivalent because:

- $E$ is at most $V^2$ and $\log V^2 = 2 \log V$ is $O(\log V)$.
- Each isolated vertex is a separate component of the minimum spanning forest. If we ignore isolated vertices we obtain $V \leq 2E$, so $\log V$ is $O(\log E)$.

We can achieve this bound as follows: first sort the edges by weight using a comparison sort in $O(E \log E)$ time; this allows the step "remove an edge with minimum weight from $S$" to operate in constant time. Next, we use a disjoint-set data structure to keep track of which vertices are in which components. We need to perform $O(V)$ operations, as in each iteration we connect a vertex to the spanning tree, two 'find' operations and possibly one union for each edge. Even a simple disjoint-set data structure such as disjoint-set forests with union by rank can perform $O(V)$ operations in $O(V \log V)$ time. Thus the total time is $O(E \log E) = O(E \log V)$.

Provided that the edges are either already sorted or can be sorted in linear time (for example with counting sort or radix sort), the algorithm can use a more sophisticated disjoint-set data structure to run in $O(E \alpha(V))$ time, where $\alpha$ is the extremely slowly growing inverse of the single-valued Ackermann function.

## Proof of correctness [ edit ]

The proof consists of two parts. First, it is proved that the algorithm produces a spanning tree. Second, it is proved that the constructed spanning tree is of minimal weight.

### Spanning tree [ edit ]

Let $G$ be a connected, weighted graph and let $Y$ be the subgraph of $G$ produced by the algorithm. $Y$ cannot have a cycle, being within one subtree and not between two different trees. $Y$ cannot be disconnected, since the first encountered edge that joins two components of $Y$ would have been added by the algorithm. Thus, $Y$ is a spanning tree of $G$.

### Minimality [ edit ]

We show that the following proposition **P** is true by induction: If $F$ is the set of edges chosen at any stage of the algorithm, then there is some minimum spanning tree that contains $F$.

- Clearly **P** is true at the beginning, when $F$ is empty: any minimum spanning tree will do, and there exists one because a weighted connected graph always has a minimum spanning tree.
- Now assume **P** is true for some non-final edge set $F$ and let $T$ be a minimum spanning tree that contains $F$.
  - If the next chosen edge $e$ is also in $T$, then **P** is true for $F + e$.
  - Otherwise, $e$ is not in $T$, and $T + e$ has a cycle $C$. This cycle contains edges which do not belong to $F$, since $e$ does not form a cycle in $F$ but does in $T$. Let $f$ be an edge in $C$ but not in $F$. Note that $f$ belongs also to $T$, and by **P** has not be considered by the algorithm, and hence a weight at least as large as $e$. Then $T - f + e$ is a tree, and it has the same or less weight as $T$. So $T - f + e$ is a minimum spanning tree containing $F + e$ and again **P** holds.
- Therefore, by the principle of induction, **P** holds when $F$ has become a spanning tree, which is only possible if $F$ is a minimum spanning tree itself.

# Correctness of Kruskal's Algorithm

**Theorem**: Kruskal's algorithm produces an MST on G = (V, E).

**Proof**: Clearly, the algorithm produces a spanning tree. We need to argue that it is an MST.

Suppose, in contradiction, the algorithm does not produce an MST. Suppose that the algorithm adds edges to the tree T' in order

$$e_1, e_2, \ldots, e_i, \ldots, e_{n-1}.$$

Let i be the value such that $e_1, e_2, \ldots, e_{i-1}$ is a subset of some MST T, but $e_1, e_2, \ldots, e_{i-1}, e_i$ is not a subset of any MST.

Consider $T \cup \{e_i\}$

- $T \cup \{e_i\}$ must have a cycle c involving $e_i$
- In the cycle c there is at least one edge that is not in $e_1, e_2, \ldots, e_{i-1}$ (since the algorithm doesn't pick an edge that creates a cycle and it picked $e_i$).

# Correctness of Kruskal's Algorithm (cont.)

- let e* be the edge in $T \cup \{e_i\}$ that forms a cycle when $e_i$ is added to T that is not in $e_1, e_2, \ldots, e_{i-1}$

  Then $wt(e_i) < wt(e^*)$, otherwise the algorithm would have picked e* next in sorted order when it picked $e_i$ (by assumption that T, with e*, is not an MST because the algorithm does not find an MST).

Claim: $T' = T - \{e^*\} \cup \{e_i\}$ is a MST

- T' is a spanning tree since it contains all nodes and has no cycles.
- wt(T') < wt(T), so T is not a MST

This contradiction means our original assumption must be wrong and therefore the algorithm always finds an MST. ■

# 3. Prim's:

## Time complexity [ edit ]

The time complexity of Prim's algorithm depends on the data structures used for the graph and for ordering the edges by weight, which can be done using a priority queue. The following table shows the typical choices:

| Minimum edge weight data structure | Time complexity (total) |
|---|---|
| adjacency matrix, searching | $O(\|V\|^2)$ |
| binary heap and adjacency list | $O((\|V\| + \|E\|) \log \|V\|) = O(\|E\| \log \|V\|)$ |
| Fibonacci heap and adjacency list | $O(\|E\| + \|V\| \log \|V\|)$ |

Prim's algorithm has many applications, such as in the generation of this maze, which applies Prim's algorithm to a randomly weighted grid graph.

A simple implementation of Prim's, using an adjacency matrix or an adjacency list graph representation and linearly searching an array of weights to find the minimum weight edge to add, requires O(|V|$^2$) running time. However, this running time can be greatly improved further by using heaps to implement finding minimum weight edges in the algorithm's inner loop.

A first improved version uses a heap to store all edges of the input graph, ordered by their weight. This leads to an O(|E| log |E|) worst-case running time. But storing vertices instead of edges can improve it still further. The heap should order the vertices by the smallest edge-weight that connects them to any vertex in the partially constructed minimum spanning tree (MST) (or infinity if no such edge exists). Every time a vertex *v* is chosen and added to the MST, a decrease-key operation is performed on all vertices *w* outside the partial MST such that *v* is connected to *w*, setting the key to the minimum of its previous value and the edge cost of (*v*,*w*).

Using a simple binary heap data structure, Prim's algorithm can now be shown to run in time O(|E| log |V|) where |E| is the number of edges and |V| is the number of vertices. Using a more sophisticated Fibonacci heap, this can be brought down to O(|E| + |V| log |V|), which is asymptotically faster when the graph is dense enough that |E| is ω(|V|), and linear time when |E| is at least |V| log |V|. For graphs of even greater density (having at least |V|$^c$ edges for some $c > 1$), Prim's algorithm can be made to run in linear time even more simply, by using a *d*-ary heap in place of a Fibonacci heap.[10][11]

## Proof of correctness [ edit ]

Let *P* be a connected, weighted graph. At every iteration of Prim's algorithm, an edge must be found that connects a vertex in a subgraph to a vertex outside the subgraph. Since *P* is connected, there will always be a path to every vertex. The output *Y* of Prim's algorithm is a tree, because the edge and vertex added to tree *Y* are connected. Let $Y_1$ be a minimum spanning tree of graph P. If $Y_1$=*Y* then *Y* is a minimum spanning tree. Otherwise, let *e* be the first edge added during the construction of tree *Y* that is not in tree $Y_1$, and *V* be the set of vertices connected by the edges added before edge *e*. Then one endpoint of edge *e* is in set *V* and the other is not. Since tree $Y_1$ is a spanning tree of graph *P*, there is a path in tree $Y_1$ joining the two endpoints. As one travels along the path, one must encounter an edge *f* joining a vertex in set *V* to one that is not in set *V*. Now, at the iteration when edge *e* was added to tree *Y*, edge *f* could also have been added and it would be added instead of edge *e* if its weight was less than *e*, and since edge *f* was not added, we conclude that

$$w(f) \geq w(e).$$

Let tree $Y_2$ be the graph obtained by removing edge *f* from and adding edge *e* to tree $Y_1$. It is easy to show that tree $Y_2$ is connected, has the same number of edges as tree $Y_1$, and the total weights of its edges is not larger than that of tree $Y_1$, therefore it is also a minimum spanning tree of graph *P* and it contains edge *e* and all the edges added before it during the construction of set *V*. Repeat the steps above and we will eventually obtain a minimum spanning tree of graph *P* that is identical to tree *Y*. This shows *Y* is a minimum spanning tree. The minimum spanning tree allows for the first subset of the sub-region to be expanded into a smaller subset *X*, which we assume to be the minimum.

# Correctness of Prim's Algorithm

Let $T_i$ be the tree after the ith iteration of the while loop

**Lemma**: For all i, $T_i$ is a subtree of some MST of G.

**Proof**: by induction on i

*Basis*: when i = 0, $T_0 = \varnothing$, ok because empty is trivial MST subtree

*IHOP*: Assume $T_i$ is a subtree of some MST M

*Induction Step*: Show that $T_{i+1}$ is a subtree of some MST

Let (u,v) be the edge added in iteration i + 1, then there are 2 cases:

# Correctness of Prim's Algorithm

<u>case 1: (u, v) is an edge of M.</u>
Then clearly $T_{i+1}$ is a subtree of M (ok)

<u>case 2: (u, v) is not an edge of M</u>
We know there is a path p in M from u to v (because M is a ST)

Let (x, y) be the first edge in p with x in $T_i$ and y not in $T_i$. We know this edge exists because the algorithm will not add edge (u,v) to a cycle.

M' = M – {(x, y)} ∪ {(u, v)} is another spanning tree.

Now we note that

$$wt(M') = wt(M) - wt(x, y) + wt(u, v) \leq wt(M)$$

since (u, v) is the minimum weight outgoing edge from $T_i$

Therefore, M' is also a MST of G and $T_{i+1}$ is a subtree of M'.

■

# 4. Breadth-first search

### Time and space complexity

The time complexity can be expressed as $O(|V| + |E|)$, since every vertex and every edge will be explored in the worst case. $|V|$ is the number of vertices and $|E|$ is the number of edges in the graph. Note that $O(|E|)$ may vary between $O(1)$ and $O(|V|^2)$, depending on how sparse the input graph is.[6]

When the number of vertices in the graph is known ahead of time, and additional data structures are used to determine which vertices have already been added to the queue, the space complexity can be expressed as $O(|V|)$, where $|V|$ is the cardinality of the set of vertices. This is in addition to the space required for the graph itself, which may vary depending on the graph representation used by an implementation of the algorithm.

When working with graphs that are too large to store explicitly (or infinite), it is more practical to describe the complexity of breadth-first search in different terms: to find the nodes that are at distance $d$ from the start node (measured in number of edge traversals), BFS takes $O(b^{d+1})$ time and memory, where $b$ is the "branching factor" of the graph (the average out-degree).[7]:81

### Completeness

In the analysis of algorithms, the input to breadth-first search is assumed to be a finite graph, represented explicitly as an adjacency list or similar representation. However, in the application of graph traversal methods in artificial intelligence the input may be an implicit representation of an infinite graph. In this context, a search method is described as being complete if it is guaranteed to find a goal state if one exists. Breadth-first search is complete, but depth-first search is not. When applied to infinite graphs represented implicitly, breadth-first search will eventually find the goal state, but depth-first search may get lost in parts of the graph that have no goal state and never return.[8]

**Theorem 22.5**: (Correctness of BFS)
Let G = (V, E) be a directed or undirected graph, and suppose that BFS is run from a given source vertex $s \in V$. Then, during execution, BFS discovers every vertex $v \neq s$ that is reachable from the source s, and upon termination, $v.d = \delta(s,v)$ for every *reachable* or *unreachable* vertex v.

Proof by contradiction.

Assume that for some vertex v that $v.d \neq \delta(s,v)$ after running BFS. Also, assume that v is the vertex with *minimum* $\delta(s,v)$ that receives an incorrect d value. By Lemma 22.2, it must be that $v.d > \delta(s,v)$.
Case 1: v is not reachable from s. This is a contradiction to the assumption that v is reachable, and the Thm holds.
Case 2: v is reachable from s. Let u be the vertex immediately preceding v on a shortest path from s to v, so that $\delta(s,v) = \delta(s,u) + 1$. Because $\delta(s,u) < \delta(s,v)$ and because v is the vertex with the *minimum* $\delta(s,v)$ that receives an incorrect d value, $u.d = \delta(s,u)$.

So we have $v.d > \delta(s,v) = \delta(s,u) + 1 = u.d + 1$.

Consider the time t when u is dequeued. At time t, v is either white, gray, or black. We can derive a contradiction in each of these cases.

Case 1: v is white. Then in line 12, $v.d = u.d+1$.

Case 2: v is black. Then v was already dequeued, and therefore $v.d \leq u.d$ (by L. 22.3).

Case 3: v is gray. Then v turned gray when it was visited from some vertex w, which was dequeued before u. Then $v.d = w.d + 1$. Since $w.d \leq u.d$ (by L. 22.3), $v.d \leq u.d + 1$.

Each of these cases is a contradiction to $v.d > \delta(s,v)$, so we conclude that $v.d = \delta(s,v)$. ■

# 5.Dijkstra

## Running time [ edit ]

Bounds of the running time of Dijkstra's algorithm on a graph with edges $E$ and vertices $V$ can be expressed as a function of the number of edges, denoted $|E|$, and the number of vertices, denoted $|V|$, using big-O notation. How tight a bound is possible depends on the way the vertex set $Q$ is implemented. In the following, upper bounds can be simplified because $|E|$ is $O(|V|^2)$ for any graph, but that simplification disregards the fact that in some problems, other upper bounds on $|E|$ may hold.

For any implementation of the vertex set $Q$, the running time is in

$$O(|E| \cdot T_{\text{dk}} + |V| \cdot T_{\text{em}}),$$

where $T_{\text{dk}}$ and $T_{\text{em}}$ are the complexities of the *decrease-key* and *extract-minimum* operations in $Q$, respectively. The simplest implementation of Dijkstra's algorithm stores the vertex set $Q$ as an ordinary linked list or array, and extract-minimum is simply a linear search through all vertices in $Q$. In this case, the running time is $O(|E| + |V|^2) = O(|V|^2)$.

It must be noted that if the implementation stores the graph as an adjacency list, the running time for a dense graph i.e. $|E| = O(|V|^2)$ is

$$\Theta((|V|^2) \log |V|).$$

For sparse graphs, that is, graphs with far fewer than $|V|^2$ edges, Dijkstra's algorithm can be implemented more efficiently by storing the graph in the form of adjacency lists and using a self-balancing binary search tree, binary heap, pairing heap, or Fibonacci heap as a priority queue to implement extracting minimum efficiently. To perform decrease-key steps in a binary heap efficiently, it is necessary to use an auxiliary data structure that maps each vertex to its position in the heap, and to keep this structure up to date as the priority queue $Q$ changes. With a self-balancing binary search tree or binary heap, the algorithm requires

$$\Theta((|E| + |V|) \log |V|)$$

time in the worst case (where $\log$ denotes the binary logarithm $\log_2$); for connected graphs this time bound can be simplified to $\Theta(|E| \log |V|)$. The Fibonacci heap improves this to

$$O(|E| + |V| \log |V|).$$

When using binary heaps, the average case time complexity is lower than the worst-case: assuming edge costs are drawn independently from a common probability distribution, the expected number of *decrease-key* operations is bounded by $O(|V| \log(|E|/|V|))$, giving a total running time of[5]:199–200

$$O\left(|E| + |V| \log \frac{|E|}{|V|} \log |V|\right).$$

# Running Time of Dijkstra's SSSP Alg

**Algorithm SSSP-Dijkstra** (G, s)
1. Initialize-Single-Source(G, s)
2. S = ∅
3. Q = V[G]
4. **while** Q ≠ ∅
5. u = Q.Extract-Min()
6. S = S ∪ {u}
7. **for** each outgoing neighbor v of u
8. Relax(u, v, w(u, v))

Steps 1-3: O(V) time
Steps 4-8:
V iterations
Suppose Extract-Min takes $O(X_Q)$ time

Steps 5-8: E iterations overall
(looks at each edge once)
Suppose Relax takes $O(Y_Q)$ time.

Total: $O(VX_Q + EY_Q)$

# Running Time of Dijkstra's SSSP Alg

If G is dense (i.e., $\theta(V^2)$ edges):
Asymptotically speaking, there is no point in saving time using Extract-Min. Store each v.d in the $v^{th}$ entry of an array. Each insert and decrease-key takes O(1) time. Extract-Min takes O(V) time (why?)

Total time: $O(V^2 + E) = O(V^2)$

If G is sparse (i.e., $o(V^2)$ edges):
Try to minimize time with Extract-Min, using min-heap:
Time for Extract-Min & Decrease-Key = O(lgV)

Total time: $O(V \lg V + E \lg V) = O((V + E) \lg V)$

## Proof of correctness [ edit ]

*Proof of Dijkstra's algorithm is constructed by induction on the number of visited nodes.*

*Invariant hypothesis*: For each visited node `v`, `dist[v]` is considered the shortest distance from `source` to `v`; and for each unvisited node `u`, `dist[u]` is assumed the shortest distance when traveling via visited nodes only, from `source` to `u`. This assumption is only considered if a path exists, otherwise the distance is set to infinity. (Note : we do not assume `dist[u]` is the actual shortest distance for unvisited nodes)

The base case is when there is just one visited node, namely the initial node `source`, in which case the hypothesis is trivial.

Otherwise, assume the hypothesis for *n-1* visited nodes. In which case, we choose an edge `vu` where `u` has the least `dist[u]` of any unvisited nodes and the edge `vu` is such that `dist[u] = dist[v] + length[v,u]`. `dist[u]` is considered to be the shortest distance from `source` to `u` because if there were a shorter path, and if `w` was the first unvisited node on that path then by the original hypothesis `dist[w] > dist[u]` which creates a contradiction. Similarly if there was a shorter path to `u` without using unvisited nodes, and if the last but one node on that path were `w`, then we would have had `dist[u] = dist[w] + length[w,u]`, also a contradiction.

After processing `u` it will still be true that for each unvisited nodes `w`, `dist[w]` will be the shortest distance from `source` to `w` using visited nodes only, because if there were a shorter path that doesn't go by `u` we would have found it previously, and if there were a shorter path using `u` we would have updated it when processing `u`.

# 6.Bellman-ford

Proof of correctness:

The correctness of the algorithm can be shown by induction:

**Lemma**. After *i* repetitions of *for* loop,

- if Distance(*u*) is not infinity, it is equal to the length of some path from *s* to *u*; and
- if there is a path from *s* to *u* with at most *i* edges, then Distance(u) is at most the length of the shortest path from *s* to *u* with at most *i* edges.

**Proof**. For the base case of induction, consider `i=0` and the moment before *for* loop is executed for the first time. Then, for the source vertex, `source.distance = 0`, which is correct. For other vertices *u*, `u.distance = infinity`, which is also correct because there is no path from *source* to *u* with 0 edges.

For the inductive case, we first prove the first part. Consider a moment when a vertex's distance is updated by `v.distance := u.distance + uv.weight`. By inductive assumption, `u.distance` is the length of some path from *source* to *u*. Then `u.distance + uv.weight` is the length of the path from *source* to *v* that follows the path from *source* to *u* and then goes to *v*.

For the second part, consider a shortest path *P* (there may be more than one) from *source* to *u* with at most *i* edges. Let *v* be the last vertex before *u* on this path. Then, the part of the path from *source* to *v* is a shortest path from *source* to *v* with at most *i-1* edges, since if it were not, then there must be some strictly shorter path from *source* to *v* with at most *i-1* edges, and we could then append the edge *uv* to this path to obtain a path with at most *i* edges that is strictly shorter than *P*—a contradiction. By inductive assumption, `v.distance` after *i*–1 iterations is at most the length of this path from *source* to *v*. Therefore, `uv.weight + v.distance` is at most the length of *P*. In the $i^{th}$ iteration, `u.distance` gets compared with `uv.weight + v.distance`, and is set equal to it if `uv.weight + v.distance` is smaller. Therefore, after *i* iterations, `u.distance` is at most the length of *P*, i.e., the length of the shortest path from *source* to *u* that uses at most *i* edges.

If there are no negative-weight cycles, then every shortest path visits each vertex at most once, so at step 3 no further improvements can be made. Conversely, suppose no improvement can be made. Then for any cycle with vertices *v*[0], ..., *v*[*k*–1],

```
v[i].distance <= v[i-1 (mod k)].distance + v[i-1 (mod k)]v[i].weight
```

Summing around the cycle, the *v*[*i*].distance and *v*[*i*–1 (mod *k*)].distance terms cancel, leaving

```
0 <= sum from 1 to k of v[i-1 (mod k)]v[i].weight
```

I.e., every cycle has nonnegative weight.

## Complexity of Bellman-Ford Algorithm

- **Initialization = O(V)**
- **Decrease-key is called (|V| - 1) · |E| times**
- **Test for negative-weight cycle = O(E)**

- **Total: O(VE) -- so more expensive than Dijkstra's, but also more general, since it detects graphs with negative weight cycles and finds single-source shortest path on graphs with negative-weight edges.**

# 7. Floyd-Warshall:

## Analysis [ edit ]

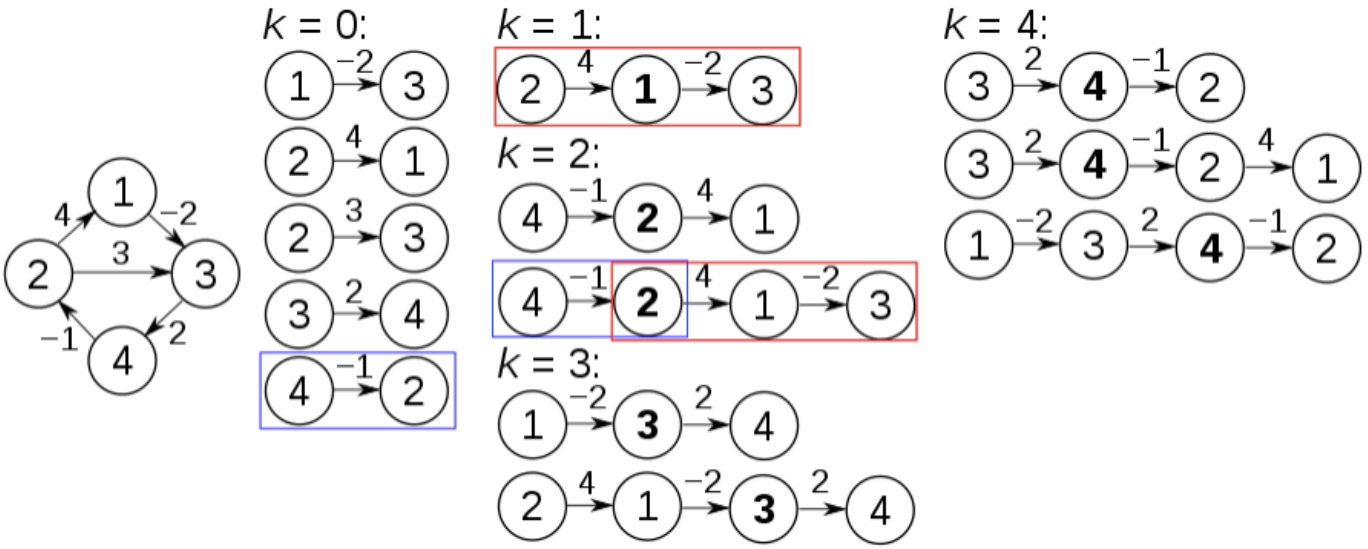
Let $n$ be $|V|$, the number of vertices. To find all $n^2$ of $\mathrm{shortestPath}(i, j, k)$ (for all $i$ and $j$) from those of $\mathrm{shortestPath}(i, j, k-1)$ requires $2n^2$ operations. Since we begin with $\mathrm{shortestPath}(i, j, 0) = \mathrm{edgeCost}(i, j)$ and compute the sequence of $n$ matrices $\mathrm{shortestPath}(i, j, 1)$, $\mathrm{shortestPath}(i, j, 2)$, ..., $\mathrm{shortestPath}(i, j, n)$, the total number of operations used is $n \cdot 2n^2 = 2n^3$. Therefore, the complexity of the algorithm is $\Theta(n^3)$.

## Behavior with negative cycles [ edit ]

A negative cycle is a cycle whose edges sum to a negative value. There is no shortest path between any pair of vertices $i$, $j$ which form part of a negative cycle, because path-lengths from $i$ to $j$ can be arbitrarily small (negative). For numerically meaningful output, the Floyd–Warshall algorithm assumes that there are no negative cycles. Nevertheless, if there are negative cycles, the Floyd–Warshall algorithm can be used to detect them. The intuition is as follows:

- The Floyd–Warshall algorithm iteratively revises path lengths between all pairs of vertices $(i, j)$, including where $i = j$;
- Initially, the length of the path $(i, i)$ is zero;
- A path $[i, k, \ldots, i]$ can only improve upon this if it has length less than zero, i.e. denotes a negative cycle;
- Thus, after the algorithm, $(i, i)$ will be negative if there exists a negative-length path from $i$ back to $i$.

Hence, to detect negative cycles using the Floyd–Warshall algorithm, one can inspect the diagonal of the path matrix, and the presence of a negative number indicates that the graph contains at least one negative cycle.[9] To avoid numerical problems one should check for negative numbers on the diagonal of the path matrix within the inner for loop of the algorithm.[10] Obviously, in an undirected graph a negative edge creates a negative cycle (i.e., a closed walk) involving its incident vertices. Considering all edges of the above example graph as undirected, e.g. the vertex sequence 4 – 2 – 4 is a cycle with weight sum –2.




| $k=0$ | | $j$ | | | |
|---|---|---|---|---|---|
| | | **1** | **2** | **3** | **4** |
| $i$ | **1** | 0 | ∞ | −2 | ∞ |
| | **2** | 4 | 0 | 3 | ∞ |
| | **3** | ∞ | ∞ | 0 | 2 |
| | **4** | ∞ | −1 | ∞ | 0 |

| $k=1$ | | $j$ | | | |
|---|---|---|---|---|---|
| | | **1** | **2** | **3** | **4** |
| $i$ | **1** | 0 | ∞ | −2 | ∞ |
| | **2** | 4 | 0 | **2** | ∞ |
| | **3** | ∞ | ∞ | 0 | 2 |
| | **4** | ∞ | −1 | ∞ | 0 |

| $k=2$ | | $j$ | | | |
|---|---|---|---|---|---|
| | | **1** | **2** | **3** | **4** |
| $i$ | **1** | 0 | ∞ | −2 | ∞ |
| | **2** | 4 | 0 | 2 | ∞ |
| | **3** | ∞ | ∞ | 0 | 2 |
| | **4** | **3** | −1 | **1** | 0 |

| $k=3$ | | $j$ | | | |
|---|---|---|---|---|---|
| | | **1** | **2** | **3** | **4** |
| $i$ | **1** | 0 | ∞ | −2 | **0** |
| | **2** | 4 | 0 | 2 | **4** |
| | **3** | ∞ | ∞ | 0 | 2 |
| | **4** | 3 | −1 | 1 | 0 |

| $k=4$ | | $j$ | | | |
|---|---|---|---|---|---|
| | | **1** | **2** | **3** | **4** |
| $i$ | **1** | 0 | **−1** | −2 | 0 |
| | **2** | 4 | 0 | 2 | 4 |
| | **3** | **5** | **1** | 0 | 2 |
| | **4** | 3 | −1 | 1 | 0 |

## Path reconstruction [ edit ]

The Floyd–Warshall algorithm typically only provides the lengths of the paths between all pairs of vertices. With simple modifications, it is possible to create a method to reconstruct the actual path between any two endpoint vertices. While one may be inclined to store the actual path from each vertex to each other vertex, this is not necessary, and in fact, is very costly in terms of memory. Instead, the shortest-path tree can be calculated for each node in $\Theta(|E|)$ time using $\Theta(|V|)$ memory to store each tree which allows us to efficiently reconstruct a path from any two connected vertices.

## Comparison with other shortest path algorithms [ edit ]

The Floyd–Warshall algorithm is a good choice for computing paths between all pairs of vertices in dense graphs, in which most or all pairs of vertices are connected by edges. For sparse graphs with non-negative edge weights, a better choice is to use Dijkstra's algorithm from each possible starting vertex, since the running time of repeated Dijkstra ($O(|E||V| + |V|^2 \log |V|)$ using Fibonacci heaps) is better than the $O(|V|^3)$ running time of the Floyd–Warshall algorithm when $|E|$ is significantly smaller than $|V|^2$. For sparse graphs with negative edges but no negative cycles, Johnson's algorithm can be used, with the same asymptotic running time as the repeated Dijkstra approach.

There are also known algorithms using fast matrix multiplication to speed up all-pairs shortest path computation in dense graphs, but these typically make extra assumptions on the edge weights (such as requiring them to be small integers).[14][15] In addition, because of the high constant factors in their running time, they would only provide a speedup over the Floyd–Warshall algorithm for very large graphs.

**Proof of Correctness:** This can be shown using induction.

**Induction Hypothesis**
We know the lengths of the shortest paths between all pairs of vertices such that only k-paths, i..e, except endpoints, the highest-labeled vertex on the path is labeled k, for some $k \leq m$, are considered. In i-ith iteration of the loop we computed all these.

**Proof:** For m=1, the basis is correct since we have only direct edges as paths. For general m, the shortest path between any pair can have $v_m$ at most one and then the paths to and from $v_m$ are k-paths, for $k \leq m-1$. Since we sum the

paths to and from $v_m$ and compare it with the best path found so far in the algorithm, we are done.

# 8. Topological sort:

## Algorithms [edit]

The usual algorithms for topological sorting have running time linear in the number of nodes plus the number of edges, asymptotically, $O(|V| + |E|)$.

# 9. Warshall's transitive closure:

## Warshall's Algorithm (pseudocode and analysis)

**ALGORITHM** *Warshall*$(A[1..n, 1..n])$

```
//Implements Warshall's algorithm for computing the transitive closure
//Input: The adjacency matrix A of a digraph with n vertices
//Output: The transitive closure of the digraph
R^(0) ← A
for k ← 1 to n do
    for i ← 1 to n do
        for j ← 1 to n do
            R^(k)[i, j] ← R^(k−1)[i, j] or (R^(k−1)[i, k] and R^(k−1)[k, j])
return R^(n)
```

**Time efficiency: $\Theta(n^3)$**

**Space efficiency: Matrices can be written over their predecessors**

# 10.Knap-Sack Problem:

## Complexity of 0-1 Knapsack Solution

***Running time is dominated by 2 nested for-loops, where the outer loop iterates n times and the inner one iterates at most W times.***

The running time of this algorithm is $O$(nW)

**The running time of the 0-1Knapsack algorithm depends on a parameter W that is not proportional to the size of the input.**

**An algorithm whose running time depends on the magnitude of a number given in the input, not the size of the input set, is called a *pseudo-polynomial* time algorithm.**

This solution will therefore run in $O(nW)$ time and $O(nW)$ space.

However, if we take it a step or two further, we should know that the method will run in the time between $O(nW)$ and $O(2^n)$. From **Definition A**, we can know that there is no need for computing all the weights when the number of items and the items themselves that we chose are fixed. That is to say, the program above computes more than expected because that the weight changes from 0 to W all the time. All we need to do is to compare m[i-1, j] and m[i-1, j-w[i]] + v[i] for m[i, j], and when m[i-1, j-w[i]] is out of range, we just give the value of m[i-1, j] to m[i, j]. From this perspective, we can program this method so that it runs recursively!

Fractional Knapsack Algorithm:

## Fractional Knapsack Algorithm

The running time of the Fractional Knapsack algorithm is $O$(nlgn).
Why?

This algorithm uses a greedy approach, not a dynamic programming technique, to find the optimal solution. Which other algorithms did we study that use a greedy approach to find an optimal solution?

```
FractionalKnapsack (T, W)
1.  for each item i in T do
2.      x_i = 0
3.      v_i = b_i/w_i    {value index of item}
4.   w = 0
5.  while w < W do
6.      remove from S item i with highest v_i
7.      a = min{w_i, W-w}
8.      x_i = a
9.      w = w + a
```

## Solution technique [ edit ]

The continuous knapsack problem may be solved by a greedy algorithm, first published in 1957 by George Dantzig,[2][3] that considers the materials in sorted order by their values per unit weight. For each material, the amount $x_i$ is chosen to be as large as possible:

- If the sum of the choices made so far equals the capacity $W$, then the algorithm sets $x_i = 0$.
- If the difference $d$ between the sum of the choices made so far and $W$ is smaller than $w_i$, then the algorithm sets $x_i = d$.
- In the remaining case, the algorithm chooses $x_i = w_i$.

Because of the need to sort the materials, this algorithm takes time $O(n \log n)$ on inputs with $n$ materials.[1][2] However, by adapting an algorithm for finding weighted medians, it is possible to solve the problem in time $O(n)$.[2]

# 11. Longest Common Subsequence Problem:

### Bottom-Up LCS DP

Running time = O(mn) (constant time for each entry in C[ ])

This algorithm finds the value of the LCS, but how can we keep track of the characters in the LCS?

We need to keep track of which neighboring table entry gave the optimal solution to a sub-problem (break ties arbitrarily).
if $x_i = y_j$ the answer came from the upper left (diagonal)
if $x_i \neq y_j$ the answer came from above or to the left, whichever value is larger (if equal, default to above).

## Complexity of LCS Algorithm

The running time of the LCS algorithm is O(mn), since each table entry takes O(1) time to compute.

The running time of the Print-LCS algorithm is O(m + n), since one of m or n is decremented in each stage of the recursion.

## Complexity [ edit ]

For the general case of an arbitrary number of input sequences, the problem is NP-hard.[1] When the number of sequences is constant, the problem is solvable in polynomial time by dynamic programming (see *Solution* below). Assume you have $N$ sequences of lengths $n_1, \ldots, n_N$. A naive search would test each of the $2^{n_1}$ subsequences of the first sequence to determine whether they are also subsequences of the remaining sequences; each subsequence may be tested in time linear in the lengths of the remaining sequences, so the time for this algorithm would be

$$O\left(2^{n_1} \sum_{i>1} n_i\right).$$

For the case of two sequences of *n* and *m* elements, the running time of the dynamic programming approach is O(*n* × *m*).[2] For an arbitrary number of input sequences, the dynamic programming approach gives a solution in

$$O\left(N \prod_{i=1}^{N} n_i\right).$$

There exist methods with lower complexity,[3] which often depend on the length of the LCS, the size of the alphabet, or both.

Notice that the LCS is not necessarily unique; for example the LCS of "ABC" and "ACB" is both "AB" and "AC". Indeed, the LCS problem is often defined to be finding *all* common subsequences of a maximum length. This problem inherently has higher complexity, as the number of such subsequences is exponential in the worst case,[4] even for only two input strings.

## Reduce the comparison time:

1. **Reduce strings to hashes**
   A hash function or checksum can be used to reduce the size of the strings in the sequences.
2.

### Reduce the required space [ edit ]

If only the length of the LCS is required, the matrix can be reduced to a $2 \times \min(n, m)$ matrix with ease, or to a $\min(m, n) + 1$ vector (smarter) as the dynamic programming approach only needs the current and previous columns of the matrix. Hirschberg's algorithm allows the construction of the optimal sequence itself in the same quadratic time and linear space bounds.[7]

# 12.Cutting Rod problem:

### 1.1.2 Memoization (top down approach)

One way we can do this is by writing the recursion as normal, but store the result of the recursive calls, and if we need the result in a future recursive call, we can use the precomputed value. The answer will be stored in `r[n]`.

```
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)
```

```
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
```

**Runtime:** $\Theta(n^2)$. Each subproblem is solved exactly once, and to solve a subproblem of size $i$, we run through $i$ iterations of the for loop. So the total number of iterations of the for loop, over all recursive calls, forms an arithmetic series, which produces $\Theta(n^2)$ iterations in total.

### 1.1.3 Bottom up approach

Here we proactively compute the solutions for smaller rods first, knowing that they will later be used to compute the solutions for larger rods. The answer will once again be stored in `r[n]`.

```
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]
```

Often the bottom up approach is simpler to write, and has less overhead, because you don't have to keep a recursive call stack. Most people will write the bottom up procedure when they implement a dynamic programming algorithm.

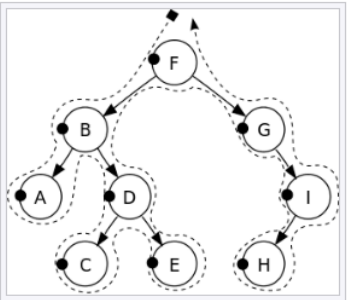
**Runtime:** $\Theta(n^2)$, because of the double for loop.

# 13.Depth-first search:

| | |
|---|---|
| **Class** | Search algorithm |
| **Data structure** | Graph |
| **Worst-case performance** | $O(\|V\| + \|E\|)$ for explicit graphs traversed without repetition, $O(b^d)$ for implicit graphs with branching factor *b* searched to depth *d* |
| **Worst-case space complexity** | $O(\|V\|)$ if entire graph is traversed without repetition, O(longest path length searched) = $O(bd)$ for implicit graphs without elimination of duplicate nodes |

**Pre-order (NLR)** [ edit ]

1. Check if the current node is empty or null.
2. Display the data part of the root (or current node).
3. Traverse the left subtree by recursively calling the pre-order function.
4. Traverse the right subtree by recursively calling the pre-order function.

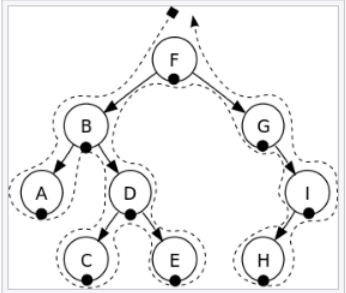


Pre-order: F, B, A, D, C, E, G, I, H.

**In-order (LNR)** [ edit ]

1. Check if the current node is empty or null.
2. Traverse the left subtree by recursively calling the in-order function.
3. Display the data part of the root (or current node).
4. Traverse the right subtree by recursively calling the in-order function.

In a binary search tree, in-order traversal retrieves data in sorted order.[4]



In-order: A, B, C, D, E, F, G, H, I.

**Out-order (RNL)** [ edit ]

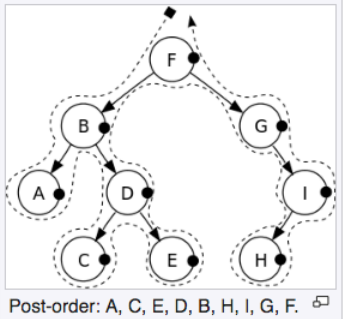
1. Check if the current node is empty or null.
2. Traverse the right subtree by recursively calling the out-order function.
3. Display the data part of the root (or current node).
4. Traverse the left subtree by recursively calling the out-order function.

In a binary search tree, out-order traversal retrieves data in reverse sorted order.

**Post-order (LRN)** [ edit ]

1. Check if the current node is empty or null.
2. Traverse the left subtree by recursively calling the post-order function.
3. Traverse the right subtree by recursively calling the post-order function.
4. Display the data part of the root (or current node).

The trace of a traversal is called a sequentialisation of the tree. The traversal trace is a list of each visited root. No one sequentialisation according to pre-, in- or post-order describes the underlying tree uniquely. Given a tree with distinct elements, either pre-order or post-order paired with in-order is sufficient to describe the tree uniquely. However, pre-order with post-order leaves some ambiguity in the tree structure.[5]

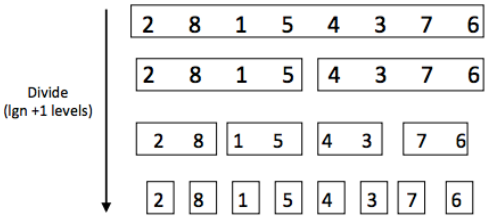


Post-order: A, C, E, D, B, H, I, G, F.

# Sorting

## 1.Merge-sort (Divide and Conquer algo):

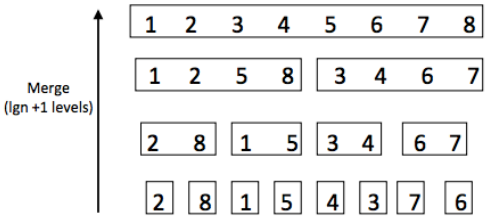
### Analyzing Merge-Sort




Why are there lgn + 1 levels? *Because lgn + 1 is the number of steps it takes to divide n by 2 until the size of the result is <= 1*

How long does it take to find the midpoint of an array? $D(n) = \theta(1)$

### Analyzing Merge-Sort




$C(n) = \theta(n)$

$$T(n) = \begin{cases} \theta(1) & \text{if } n = 1 \\ 2T(n/2) + \theta(n) & \text{otherwise} \end{cases}$$

Recurrence for worst-case running time for Merge-Sort

### Analyzing Merge-Sort

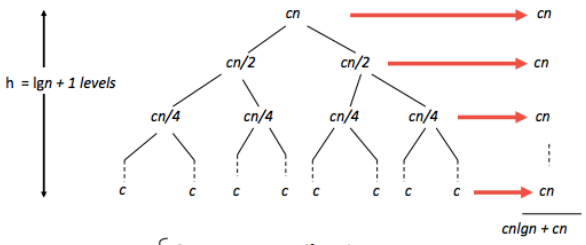
$$T(n) = \begin{cases} \theta(1) & \text{if } n = 1 \\ 2T(n/2) + \theta(n) & \text{otherwise} \end{cases}$$

Recurrence for worst-case running time for Merge-Sort

$$aT(n/b) + D(n) + C(n)$$

- $a$ = 2 (two subproblems)
- $n/b$ = n/2 (each subproblem has size approx $n/2$)
- $D(n)$ = $\theta$(1) (just compute midpoint of array)
- $C(n)$ = $\theta$(n) (merging can be done by scanning sorted subarrays)

### Recursion Tree for Merge-Sort




$$T(n) = \begin{cases} c & \text{if } n = 1 \\ 2T(n/2) + cn & \text{otherwise} \end{cases}$$

Recurrence tree for Merge-Sort

## Analysis [ edit ]

In sorting $n$ objects, merge sort has an average and worst-case performance of $O(n \log n)$. If the running time of merge sort for a list of length $n$ is $T(n)$, then the recurrence $T(n) = 2T(n/2) + n$ follows from the definition of the algorithm (apply the algorithm to two lists of half the size of the original list, and add the $n$ steps taken to merge the resulting two lists). The closed form follows from the master theorem for divide-and-conquer recurrences.

In the worst case, the number of comparisons merge sort makes is given by the sorting numbers. These numbers are equal to or slightly smaller than $(n \lceil \lg n \rceil - 2^{\lceil \lg n \rceil} + 1)$, which is between $(n \lg n - n + 1)$ and $(n \lg n + n + O(\lg n))$.[5]

For large $n$ and a randomly ordered input list, merge sort's expected (average) number of comparisons approaches $\alpha \cdot n$ fewer than the worst case where $\alpha = -1 + \sum_{k=0}^{\infty} \frac{1}{2^k + 1} \approx 0.2645$.

In the *worst* case, merge sort does about 39% fewer comparisons than quicksort does in the *average* case. In terms of moves, merge sort's worst case complexity is $O(n \log n)$—the same complexity as quicksort's best case, and merge sort's best case takes about half as many iterations as the worst case.[*citation needed*]

Merge sort is more efficient than quicksort for some types of lists if the data to be sorted can only be efficiently accessed sequentially, and is thus popular in languages such as Lisp, where sequentially accessed data structures are very common. Unlike some (efficient) implementations of quicksort, merge sort is a stable sort.

Merge sort's most common implementation does not sort in place;[6] therefore, the memory size of the input must be allocated for the sorted output to be stored in (see below for versions that need only $n/2$ extra spaces).

### Comparison with other sort algorithms [ edit ]

Although heapsort has the same time bounds as merge sort, it requires only $\Theta(1)$ auxiliary space instead of merge sort's $\Theta(n)$. On typical modern architectures, efficient quicksort implementations generally outperform mergesort for sorting RAM-based arrays.[*citation needed*] On the other hand, merge sort is a stable sort and is more efficient at handling slow-to-access sequential media. Merge sort is often the best choice for sorting a linked list: in this situation it is relatively easy to implement a merge sort in such a way that it requires only $\Theta(1)$ extra space, and the slow random-access performance of a linked list makes some other algorithms (such as quicksort) perform poorly, and others (such as heapsort) completely impossible.

# 2.Heap-sort:

## Max-Heapify: Running Time

*Running Time of Max-Heapify*

- every line is $\theta(1)$ time except the recursive call in line 10.
- in worst-case, last level of binary tree is half empty and the sub-tree rooted at left child of root has size at most (2/3)*n*. Note that in a complete binary tree (CBT) the subtrees to left and right would be equal size.

We get the recurrence $T(n) <= T(2n/3) + \theta(1)$

which, by case 2 of the Master Theorem, has the solution

$$T(n) = \theta(\lg n)$$

Max-Heapify takes O(h) time when node A[i] has height h in the heap. The height h of a tree is the longest root to leaf path in the tree. $h = O(\lg n)$ in the worst case

## Correctness of Build-Max-Heap

**Loop invariant:** At the start of each iteration *i* of the for loop, each node $i+1, i+2, \ldots, n$ is the root of a max-heap.

- Initialization: $i = \lfloor n/2 \rfloor$. Each node $\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, \ldots n$ is a leaf, trivially satisfying the max-heap property.
- Inductive hypothesis: At the start of iteration $k <= \lfloor n/2 \rfloor$ and $k >= 1$, the subtrees of *k* are the roots of max-heaps
- Inductive step (maintenance): During iteration k, Max-Heapify is called on node k. By the IH, the left and right subtrees of k are max-heaps. When Max-Heapify is called on node k, the value in node k is repeatedly swapped with larger valued child until that value is greater than either child. Therefore value that was in node k is correctly positioned in the max-heap rooted at k.

```
Build-Max-Heap(A)
1. A.heapsize = A.length
2. for i = ⌊A.length/2⌋ downto 1
3.     Max-Heapify(A, i)
```

## Variations [ edit ]

### Floyd's heap construction [ edit ]

The most important variation to the basic algorithm, which is included in all practical implementations, is a heap-construction algorithm by Floyd which runs in $O(n)$ time and uses siftdown rather than siftup, avoiding the need to implement siftup at all.

Rather than starting with a trivial heap and repeatedly adding leaves, Floyd's algorithm starts with the leaves, observing that they are trivial but valid heaps by themselves, and then adds parents. Starting with element $n/2$ and working backwards, each internal node is made the root of a valid heap by sifting down. The last step is sifting down the first element, after which the entire array obeys the heap property.

The worst-case number of comparisons during the Floyd's heap-construction phase of Heapsort is known to be equal to $2n - 2s_2(n) - e_2(n)$, where $s_2(n)$ is the number of 1 bits in the binary representation of $n$ and $e_2(n)$ is number of trailing 0 bits.[5]

The standard implementation of Floyd's heap-construction algorithm causes a large number of cache misses once the size of the data exceeds that of the CPU cache. Much better performance on large data sets can be obtained by merging in depth-first order, combining subheaps as soon as possible, rather than combining all subheaps on one level before proceeding to the one above.[6][7]

## Comparison with other sorts [ edit ]

Heapsort primarily competes with quicksort, another very efficient general purpose nearly-in-place comparison-based sort algorithm.

Quicksort is typically somewhat faster due to some factors, but the worst-case running time for quicksort is $\mathrm{O}(n^2)$, which is unacceptable for large data sets and can be deliberately triggered given enough knowledge of the implementation, creating a security risk. See quicksort for a detailed discussion of this problem and possible solutions.

Thus, because of the $\mathrm{O}(n \log n)$ upper bound on heapsort's running time and constant upper bound on its auxiliary storage, embedded systems with real-time constraints or systems concerned with security often use heapsort, such as the Linux kernel.[20]

Heapsort also competes with merge sort, which has the same time bounds. Merge sort requires $\Omega(n)$ auxiliary space, but heapsort requires only a constant amount. Heapsort typically runs faster in practice on machines with small or slow data caches, and does not require as much external memory. On the other hand, merge sort has several advantages over heapsort:

- Merge sort on arrays has considerably better data cache performance, often outperforming heapsort on modern desktop computers because merge sort frequently accesses contiguous memory locations (good locality of reference); heapsort references are spread throughout the heap.
- Heapsort is not a stable sort; merge sort is stable.
- Merge sort parallelizes well and can achieve close to linear speedup with a trivial implementation; heapsort is not an obvious candidate for a parallel algorithm.
- Merge sort can be adapted to operate on **singly** linked lists with $\mathrm{O}(1)$ extra space. Heapsort can be adapted to operate on **doubly** linked lists with only $\mathrm{O}(1)$ extra space overhead.[*citation needed*]
- Merge sort is used in external sorting; heapsort is not. Locality of reference is the issue.

Introsort is an alternative to heapsort that combines quicksort and heapsort to retain advantages of both: worst case speed of heapsort and average speed of quicksort.

# HeapSort loop invariant

$Build - Max - Heap(A)$

1 $heap\text{-}size[A] \leftarrow length[A]$
2 **for** $i \leftarrow \lfloor length[A]/2 \rfloor$ **downto** 1
3 **do** MAX-HEAPIFY$(A, i)$

**To show why BUILD-MAX-HEAP works correctly, we use the following loop invariant:**

**At the start of each iteration of the for loop of lines 2– 3, each node $i + 1, i + 2, \ldots, n$ is the root of a max-heap.**

# Loop Invariant Proof

**At the start of each iteration of the for loop of lines 2– 3, each node $i+1, i+2, \ldots, n$ is the root of a max-heap.**

**We need to show that this invariant is true prior to the first loop iteration, that each iteration of the loop maintains the invariant, and that the invariant provides a useful property to show correctness when the loop terminates.**

**Initialization:** **Prior to the first iteration of the loop, $i = \lfloor n/2 \rfloor$. Each node $\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, \ldots, n$ is a leaf and is thus the root of a trivial max-heap.**

**Maintenance:** **To see that each iteration maintains the loop invariant, observe that the children of node $i$ are numbered higher than $i$. By the loop invariant, therefore, they are both roots of max-heaps. This is precisely the condition required for the call MAX-HEAPIFY$(A, i)$ to make node $i$ a max-heap root. Moreover, the MAX-HEAPIFY call preserves the property that nodes $i+1, i+2, \ldots, n$ are all roots of max-heaps. Decrementing $i$ in the for loop update reestablishes the loop invariant for the next iteration.**

**Termination:** **At termination, $i = 0$. By the loop invariant, each node $1, 2, \ldots, n$ is the root of a max-heap. In particular, node $1$ is.**

# 3.Quick sort(Divide and Conquer algo):

## Quicksort Running Time

$T(n) = T(q - p) + T(r - q) + O(n)$

The value of T(n) depends on the location of q in the array A[p..r].

Since we don't know this in advance, we must look at worst-case, best-case, and average-case partitioning.

## Quicksort Running Time

*Worst-case partitioning*: Each partition results in a 0 : n-1 split $T(0) = \theta(1)$ and the partitioning costs $\theta(n)$, so the recurrence is

$$T(n) = T(n-1) + T(0) + \theta(n) = T(n-1) + \theta(n)$$

This is an arithmetic series which evaluates to $\theta(n^2)$. So worst-case for Quicksort is no better than Insertion sort!

What does the input look like in Quicksort's worst-case?

## Quicksort Best-case

$T(n) = T(q - p) + T(r - q) + O(n)$

Best-case partitioning: Each partition results in a $\lfloor n/2 \rfloor : \lceil n/2 \rceil - 1$ split (i.e., close to balanced split each time), so recurrence is

$$T(n) = 2T(n/2) + \theta(n)$$

By the master theorem, this recurrence evaluates to

$\theta(n \lg n)$

## Quicksort Average-case

Intuition: Some splits will be close to balanced and others close to unbalanced ⇒ good and bad splits will be randomly distributed in recursion tree.

The running time will be (asymptotically) bad only if there are many bad splits *in a row*.

- A bad split followed by a good split results in a good partitioning after one extra step.
- Implies a $\theta(n \lg n)$ running time (with a larger constant factor to ignore than if all splits were good).

## Correctness of Quicksort

**Claim**: Partition satisfies the specifications of the Divide step.

**Loop invariant**: At the beginning of each iteration of the for loop (lines 3-6), for any array index k,

1. If $p \le k \le i$, then $A[k] \le x$.
2. If $i+1 \le k \le j-1$, then $A[k] > x$.
3. If $k=r$, then $A[k] = x$.

```
Partition(A, p, r)
1. x = A[r]  // choose pivot
2. i = p - 1
3. for j = p to r - 1
4.     if A[ j ] ≤ x
5.         i = i + 1
6.         swap A[ i ] and A[ j ]
7. swap A[ i+1 ] and A[ r ]
8. return i + 1
```

**Loop invariant**: At the beginning of each iteration of the for loop (lines 3-6), for any array index k,

1. If $p \le k \le i$, then $A[k] \le x$.
2. If $i+1 \le k \le j-1$, then $A[k] > x$.
3. If $k=r$, then $A[k] = x$.

```
Partition(A, p, r)
1. x = A[r]  // choose pivot
2. i = p - 1
3. for j = p to r - 1
4.     if A[ j ] ≤ x
5.         i = i + 1
6.         swap A[ i ] and A[ j ]
7. swap A[ i+1 ] and A[ r ]
8. return i + 1
```

**Initialization**: $i = p-1 = 0$ and $j = p = 1$. k cannot be between 0 and 1 (cond 1), nor can k be between $i+1 = 1$ and $j - 1 = 0$ (cond 2). Partition satisfies condition 3 in this case, and conditions 1 and 2 are vacuously true.

**Inductive Hypothesis:** Assume the invariant holds through iteration $j = k < n - 1$.

**Loop invariant**: At the beginning of each iteration of the for loop (lines 3-6), for any array index k,

1. If $p \le k \le i$, then $A[k] \le x$.
2. If $i+1 \le k \le j-1$, then $A[k] > x$.
3. If $k=r$, then $A[k] = x$.

```
Partition(A, p, r)
1. x = A[r]  // choose pivot
2. i = p - 1
3. for j = p to r - 1
4.     if A[ j ] ≤ x
5.         i = i + 1
6.         swap A[ i ] and A[ j ]
7. swap A[ i+1 ] and A[ r ]
8. return i + 1
```

**Ind. Step**: At the start of iteration k+1, either $A[ j ] > x$ or $A[ j ] \le x$. If $A[ j ] > x$, j is incremented and cond 2 holds for A[ j-1 ] with no other changes. If $A[ j ] \le x$, A[ i+1 ] and A[ j ] are swapped, and then j is incremented. Cond 1 holds for A[ i ] after swap. By the IH, the item in A[ j ] was in A[ i+1 ] during the last iteration, and was > x then, so cond. 2 holds at the end of iteration k+1.

**Termination**: At termination, j = r and A has been partitioned into 3 sets: items $\le x$, items > x, and A[ j ] = x. The item in $A[i+1] > x$ in the last iteration, so it remains > x at termination, when it is swapped with the element in A[r] = x. QED

**Loop invariant**: At the beginning of each iteration of the for loop (lines 3-6), for any array index k,

1. If $p \le k \le i$, then $A[k] \le x$.
2. If $i+1 \le k \le j-1$, then $A[k] > x$.
3. If $k=r$, then $A[k] = x$.

```
Partition(A, p, r)
1. x = A[r]  // choose pivot
2. i = p - 1
3. for j = p to r - 1
4.     if A[ j ] ≤ x
5.         i = i + 1
6.         swap A[ i ] and A[ j ]
7. swap A[ i+1 ] and A[ r ]
8. return i + 1
```

So at the end of the algorithm, A is split into 3 parts, from left to right: items in $A[1...i]$ that are $\le A[r]$, $A[i+1] = A[r]$, and the items in $A[i+2...n]$ that are $> A[r]$.

Therefore, the item in position $A[i+1] = A[q]$ is in its proper sorted position after the first iteration, Partition(A, 1, n). The Quick-Sort algorithm then makes recursive calls on $A[1...q-1]$ and $A[q+1...n]$.

# 4.Insertion Sort:

**Insertion sort** is a simple sorting algorithm that builds the final sorted array (or list) one item at a time. It is much less efficient on large lists than more advanced algorithms such as quicksort, heapsort, or merge sort. However, insertion sort provides several advantages:

- Simple implementation: Jon Bentley shows a three-line C version, and a five-line optimized version[2]
- Efficient for (quite) small data sets, much like other quadratic sorting algorithms
- More efficient in practice than most other simple quadratic (i.e., $O(n^2)$) algorithms such as selection sort or bubble sort
- Adaptive, i.e., efficient for data sets that are already substantially sorted: the time complexity is $O(kn)$ when each element in the input is no more than $k$ places away from its sorted position
- Stable; i.e., does not change the relative order of elements with equal keys
- In-place; i.e., only requires a constant amount O(1) of additional memory space
- Online; i.e., can sort a list as it receives it

To avoid having to make a series of swaps for each insertion, the input could be stored in a linked list, which allows elements to be spliced into or out of the list in constant-time when the position in the list is known. However, searching a linked list requires sequentially following the links to the desired position: a linked list does not have random access, so it cannot use a faster method such as binary search. Therefore, the running time required for searching is $O(n)$ and the time for sorting is $O(n^2)$. If a more sophisticated data structure (e.g., heap or binary tree) is used, the time required for searching and insertion can be reduced significantly; this is the essence of heap sort and binary tree sort.

| **Data structure** | Array |
|---|---|
| **Worst-case performance** | $O(n^2)$ comparisons and swaps |
| **Best-case performance** | $O(n)$ comparisons, $O(1)$ swaps |
| **Average performance** | $O(n^2)$ comparisons and swaps |
| **Worst-case space complexity** | $O(n)$ total, $O(1)$ auxiliary |

# 5.Counting Sort(non-comparison):

## Summary NCB Sorts

### Non-Comparison-Based Sorts

| | Running Time | | | |
|---|---|---|---|---|
| | worst-case | average-case | best-case | in place |
| Counting Sort | $O(n + k)$ | $O(n + k)$ | $O(n + k)$ | no |
| Radix Sort | $O(d(n + k'))$ | $O(d(n + k'))$ | $O(d(n + k'))$ | no |
| Bucket Sort | | $O(n)$ | | no |

Counting sort requires known range of data [0,1,2,..,k] and uses array indexing to count the number of occurrences of each value.

Radix sort requires that each integer consists of d digits, and each digit is in range [1,2,..,k'].

Bucket sort requires advance knowledge of input distribution (sorts n numbers uniformly distributed in range in O(n) time).

## Running Time of Counting Sort

**for loop in lines 1-2 takes $\theta(n)$ time.**
**for loop in lines 3-4 takes $\theta(k)$ time.**
**for loop in lines 5-7 takes $\theta(n)$ time.**

**Overall time is $\theta(k + n)$.**

In practice, use counting sort when we have $k = \theta(n)$, so running time is $\theta(n)$.

This version of counting sort has the important property of ***stability***.

A sorting algorithm is ***stable*** when numbers with equal values appear in the output array in the same order as they do in the input array.

Important when satellite data is stored with elements being sorted and when counting sort is used as a subroutine for radix sort, the next NCB algorithm we'll look at.

## Complexity analysis [ edit ]

Because the algorithm uses only simple for loops, without recursion or subroutine calls, it is straightforward to analyze. The initialization of the count array, and the second for loop which performs a prefix sum on the count array, each iterate at most $k + 1$ times and therefore take $O(k)$ time. The other two for loops, and the initialization of the output array, each take $O(n)$ time. Therefore, the time for the whole algorithm is the sum of the times for these steps, $O(n + k)$.[1][2]

Because it uses arrays of length $k + 1$ and $n$, the total space usage of the algorithm is also $O(n + k)$.[1] For problem instances in which the maximum key value is significantly smaller than the number of items, counting sort can be highly space-efficient, as the only storage it uses other than its input and output arrays is the Count array which uses space $O(k)$.[5]

## Variant algorithms [ edit ]

If each item to be sorted is itself an integer, and used as key as well, then the second and third loops of counting sort can be combined; in the second loop, instead of computing the position where items with key `i` should be placed in the output, simply append `Count[i]` copies of the number `i` to the output.

This algorithm may also be used to eliminate duplicate keys, by replacing the `Count` array with a bit vector that stores a `one` for a key that is present in the input and a `zero` for a key that is not present. If additionally the items are the integer keys themselves, both second and third loops can be omitted entirely and the bit vector will itself serve as output, representing the values as offsets of the non-`zero` entries, added to the range's lowest value. Thus the keys are sorted and the duplicates are eliminated in this variant just by being placed into the bit array.

# 6. Radix Sort (non-comparison):

## Radix Sort

Let *d* be the number of digits in each input number.

```
Radix-Sort(A, d)
1. for i = 1 to d
2.     use stable sort to sort array A on digit i
```

Running time of radix sort: $O(dT_{cs}(n))$

- $T_{cs}$ is the time for the internal sort. Counting sort gives $T_{cs}(n) = O(k + n)$, so $O(dT_{cs}(n)) = O(d(k + n))$, which is $O(n)$ if $d = O(1)$ and $k = O(n)$.

**Radix sort**

| | |
|---|---|
| **Class** | Sorting algorithm |
| **Data structure** | Array |
| **Worst-case performance** | $O(w \cdot n)$, where w is the number of bits required to store each key. |
| **Worst-case space complexity** | $O(w + n)$ |

# 7. Bucket Sort (non-comparison):

## Bucket Sort

Bucket-Sort(A, *x*, *y*)
1. divide interval *[x, y)* into *n* equal-sized subintervals (buckets)
2. distribute the *n* input keys into the buckets
3. sort the numbers in each bucket (e.g., with insertion sort) as they are inserted in the bucket
4. scan the (sorted) buckets in order and produce output array

Running time of bucket sort: *O(n)* expected time
*Step 1:* *O(1)* for each interval = *O(n)* time total.
*Step 2:* *O(n)* time.
*Step 3:* The expected number of elements in each bucket is *O(1)* *(see book for formal argument, section 8.4)*, so total is *O(n)*
*Step 4:* *O(n)* time to scan the *n* buckets containing a total of *n* input elements

A bucket is really a linked list.

**Bucket sort**

| | |
|---|---|
| **Class** | Sorting algorithm |
| **Data structure** | Array |
| **Worst-case performance** | $O(n^2)$ |
| **Average performance** | $O(n + \frac{n^2}{k} + k)$, where k is the number of buckets. $O(n)$, when $k \approx n$. |
| **Worst-case space complexity** | $O(n \cdot k)$ |