

Algorithms

Order of Complexity

$1, \log \log n, \sqrt{\log n}, \log \sqrt{n}, \log n, \sqrt{n}, n, n \log n, n^2, \binom{n}{3} \in n^3, n^c, 2^n, n!$

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0 \Rightarrow f \in \mathcal{O}(g), \mathcal{O}(f) \subsetneq \mathcal{O}(g); \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = C > 0 (C \text{ constant})$$

$$\Rightarrow f \in \Theta(g); \frac{f(n)}{g(n)} \xrightarrow{n \rightarrow \infty} \infty \Rightarrow g \in \mathcal{O}(f), \mathcal{O}(g) \subsetneq \mathcal{O}(f); \sum_{k=1}^n k = \frac{n(n+1)}{2}$$

Maximum Subarray Alorithm

Algorithm 1: Inductive Maximum Subarray

```
Input : (a1, a2, ..., an)
Output: max 0, max_{i,j} \sum_{k=i}^j a_k
1 for i = 1, ..., n do
2   R ← R + a_i
3   if R ≤ 0 then
4     R ← 0
5   end
6   if R > M then
7     M ← R
8   end
9 end
10 return M
```

Runtime : $\Theta(n)$

Searching

Linear Search

Best case: 1 comparison; Worst case: n comparisons
Expected: $E(x) = \frac{1}{n} \sum_{i=1}^n i = \frac{n+1}{2} \in \Theta(n)$

Binary Search

divide and conquer approach $\rightarrow \Theta(\log n)$ Works with two pointers l and r . If $l > r$ the search was without result.

Algorithm 2: Breadth-first search

```
Input : A graph G and a starting vertex root of G
Output: The parent links trace the shortest path back to root
1 let Q be a queue; label root as discovered; Q.enqueue(root)
2 while Q is not empty do
3   v := Q.dequeue() if v is the goal then
4     return v
5   end
6   for all edges from v to w in G.adjacentEdges(v) do
7     if w is not labeled as discovered then
8       label w as discovered
9       w.parent := v; Q.enqueue(w)
10    end
11  end
12 end
```

Selecting

Blum’s Algorithm

A good pivot can be selected using the median-of-medians-algorithm. $\mathcal{O}(n)$

- 1. Consider groups of 5 elements
- 2. Compute the median for each group (trivial)
- 3. Recursively compute medians

Pivot

Algorithm 3: Selection via Pivot

```
Input : Array A of length n with pivot p
Output: A partitioned around p with position of p
1 l ← 1
2 r ← n while l ≤ r do
3   while A[l] < p do
4     l ← l + 1
5   end
6   while A[r] > p do
7     r ← r - 1
8   end
9   swap(A[l],A[r]) if A[l] = A[r] then
10    l ← l + 1
11  end
12 end
13 return l - 1
```

Algorithm 4: Quickselect

```
Input : Array A of length n; 1 ≤ k ≤ n
1 x ← RandomPivot(A)
2 m ← Partition(A,x)
3 if k < m then
4   return Quickselect(A[0..m-1],k)
5 end
6 if k > m then
7   return Quickselect(A[m+1..n],k) else
8   return A[k]
9 end
10 end
```

Sorting

- **Bubblesort**: Always swap if $A[i - 1] > A[i]$. In each round, the max in the unsorted part will move to the right (like a bubble). $\Theta(n^2)$ stable
- **Selection sort**: swap the smallest element in the unsorted part with the most right element of the sorted part. $\Theta(n^2)$ unstable

```
arr[] = 64 25 12 22 11
// Place min at beginning
11 25 12 22 64
// Place min at beginning
11 12 25 22 64 ...
```

- **Insertion sort**: Determine the insertion position of element i. $\Theta(n^2)$ stable

- 1: Iterate over the array (curr).
- 2: Compare curr to predecessor (pre).
- 3: If curr < pre, compare it to the elements before. Larger elements are moved back 1 pos.

- **Merge sort**: At least two parts of the Array are already sorted. Iterative merging of the already

sorted bits. - $\Theta(n \log n)$, $\Theta(n)$ storage, stable, needs intermediate storage for the merging step

Quicksort

Algorithm 5: Quicksort

```
Input : Array A of length n
Output: Array A sorted
1 if n > 1 then
2   Choose Pivot p ∈ A k ← Partition(A,p)
3   Quicksort(A[1,...,k-1])
4   Quicksort(A[k+1,...,n])
5 end
```

Algorithm 6: Partition

```
Input : Array A, that contains the pivot p in A[l, . . . , r] at least once.
Output: Array A partitioned in [l, . . . , r] around p. Returns position of p.
1 while l < r do
2   while A[l] < p do
3     l = l + 1
4   end
5   while A[r] > p do
6     r = r - 1
7   end
8   swap(A[l], A[r])
9   if A[l] = A[r] then
10    l = l + 1
11  end
12 end
13 return l - 1
```

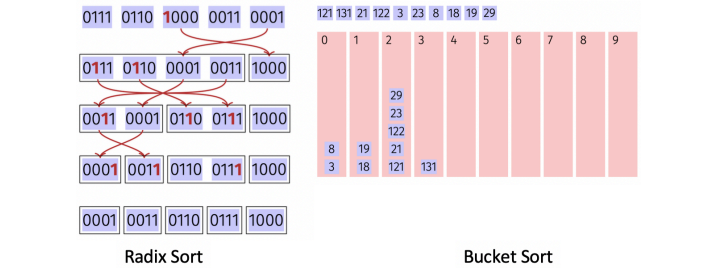
Runtime: in the mean $\mathcal{O}(n \cdot \log \cdot n)$, worst case $\Theta(n^2)$ if worst pivots are selected each time.

Radix Sort

n-locks for n-keys $\in \mathcal{O}(n)$. We have m-adic binary numbers, so two categories to sort the numbers into. Used for numbers (and strings via UTF-8/ASCII)

Bucket Sort

Create a number of buckets. Sort e.g. after decimality into buckets and sort those buckets then. Can be implemented via linked list or a dynamic list(heap?).



Hashing

Basics

Common: $h(k) = k \bmod m$
Often: $m = 2^k - 1$
Linear Probing: $S(k) = (h(k), h(k) + 1, \dots, h(k) + m - 1)$

mod m **Issue:** Primary clustering, long contiguous areas of used entries.

Quadratic Probing:

$S(k) = (h(k), h(k) + 1, h(k) - 1, h(k) + 4, \dots) \mod m$ **Issue:** Secondary clustering, traversal of the same probing sequence.

Double Hashing:

$S(k) = (h(k), h(k) + h'(k), h(k) + 2h'(k), \dots, h(k) + (m - 1)h'(k)) \mod m$

Trees

Trees are connected, directional and acyclic graphs.

Removing a child

- No children - Remove the node
- 1 child - Replace by the only child
- 2 children - Replace by the symmetric descendent

Ways of traversal

Preorder

v , then $T_{left}(v)$, next $T_{right}(v)$

Postorder

T_{left} , then T_{right} , next v

Inorder

T_{left} , then v , next $T_{right} \rightarrow$ ascending sequence.

Heaps

Keys are strictly larger/smaller depending on Max- or Minheap.

Insertion

Inserting a key into a heap can possibly violate the heap settings - Is reinstated by successive rising up.

Heap Sort

Every subtree is a heap - inductive sorting from below.
 $\rightarrow \mathcal{O}(n \cdot \log n)$

Quadtrees

Partitioning a subsection into 4 equal parts. If there are too many objects stored in one node, we split the node into four children. Objects that are falling on a border are stored in the parent node.

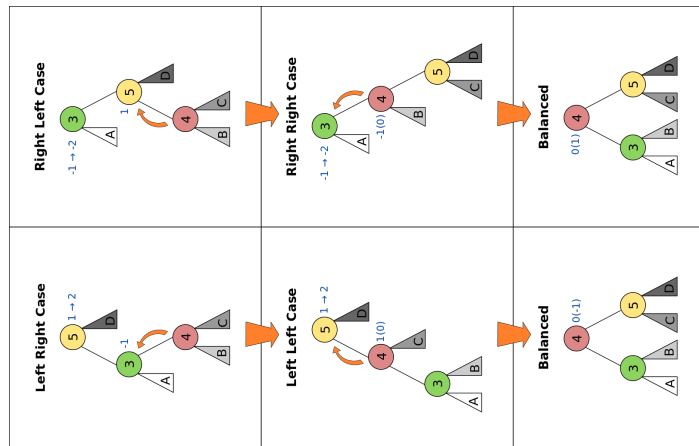
AVL trees

AVL trees guarantee a runtime of $\mathcal{O}(\log n)$

$bal(v) := h(T_r(v)) - h(T_l(v))$

AVL condition: $\forall v \in V : bal(v) \in \{-1, 0, 1\}$

Rebalancing AVL trees



Dynamic Programming

Samples

One-dimensional

Problem: Finding the longest possible combination of downwards ski slopes with lengths l_i . The slopes connect the stations with heights h_i .

1. **Table:** $n \times 1$
2. **Entry:** $[i]$: longest descent that ends in i .
3. **Calculation:** $D[i] = 0, \forall i = 1, \dots, n$ and

$$D[i] = \max_{Slope(j,i)} \{D[j] + l(j,i)\}$$
4. **Order:** for i in $(1, n)$; $D[i]$
5. **Result:** $\max(D)$
6. **Reconstruction:** Recursively walk back from result and check $D[i] = D[j] + l(j,i)$ for all slopes (j,i)

Two-dimensional

Problem: Finding the smallest possible value of an expression (n values a_i and $n - 1$ operators s_i) using optimal bracket placement.

1. **Table:** $n \times n$: Only upper right triangular matrix is used.
2. **Entry:** $[i, j]$: smallest possible value of sub-expression from value a_i to a_j .
3. **Calculation:** $A_{i,i} = a_i; 1 \leq i \leq n$ and

$$A_{i,j} = \min_{i \leq k \leq j} \{A_{i,k-1} \langle s_{k-1} \rangle A_{k,j}\}; 1 \leq i \leq j \leq n$$
4. **Order:** for s in $(0, n-1)$; for i in $(1, n-s)$; $A[i, i+s]$
5. **Result:** $A[1, n]$
6. **Reconstruction:** Recursively walk back and check
 $A_{i,j} = A_{i,k-1} \langle s_{k-1} \rangle A_{k,j}$

Graphs

Basics

Connected: Graph where there is a connecting path (not edge) between each pair of nodes.

Complete: Graph where there is an edge between each pair of nodes.

Algorithm 7: Depth First Search

Input : A graph G and a vertex v of G
Output: All vertices reachable from v labeled as discovered

```

1 label v as discovered
2 for  $E \in G.adjacentEdges(v)$  (lexicographic) do
3   if vertex  $w$  is not labeled as discovered then
4     DFS( $G, w$ )
5   end
6 end
```

Topological Sorting

A directed graph has a topological sorting if it is acyclic.

Idea We successively prune our graph by removing elements that have 0 entry edges (and then update the entry edges of the successors to find the next one.

Algorithm 8: Topological Sorting

```

1  $A[v]$  contains number of entry edges of vertex  $v$  (calculate by setting  

 $A[w] = 0$  and then loop through  $(v, w) \in E$  and set  $A[w] += 1$ )
2 for  $v \in V$  where  $A[v] == 0$  do
3   Push( $S, v$ )
4 end
5  $i = 0$ ;
6 while  $S \neq \{\}$  do
7    $v \leftarrow pop(S)$ ;  $ord[v] \leftarrow i$ 
8    $i++$ ;
9   for  $(v, w) \in E$  do
10     $A[w] \leftarrow A[w] - 1$ ; //decrease incoming for all successors
11    if  $A[w] == 0$  then
12      push( $S, w$ )
13    end
14  end
15 end
16 if  $i = |V|$  then
17   return SUCCESS
18 end
19 else
20   return "Cycle detected"
21 end
```

Shortest Path

On either directed or non-directed, weighted graph, find the shortest distance between a point A and all the other points in the graph.

Dijkstra

Algorithm 9: Dijkstra

```
Input :  $G = (V, E, source)$ 
1 create vertex set  $Q$  // as a queue / min heap;
2 for  $u \in V$  do
3    $dist[u] \leftarrow \text{INFINITY}$ ;
4    $prev[u] \leftarrow \text{UNDEFINED}$ ;
5    $Q.insert(u)$ ;
6 end
7  $dist[source] = 0$ ;
8 while  $Q$  not empty do
9    $u = Q.ExtractMin()$ ;
10  for  $v$  in Neighbors of  $u$  still in  $Q$  do
11     $alt = dist[u] + length(u, v)$ ;
12    if  $alt < dist[v]$  then
13       $dist[v] = alt$ ;
14       $prev[v] = u$ ;
15       $Q.DecreasePriority(v, alt)$ ;
16    end
17  end
18 end
```

Runtime of Dijkstra

- any data structure: $\mathcal{O}(|V| \cdot T_{em} + |E| \cdot T_{dp})$
- with an array or linked list $\mathcal{O}(|V|^2 + |E|) = \mathcal{O}(|V|^2)$
- dense graph in adjacency list $\mathcal{O}(|V|^2 \log |V|)$ since $|E| = |V|^2$ and DecreaseKey $\log(|V|)$
- sparse connected graph in adjacency list/ stored in binary tree $\mathcal{O}(|E| \log |V|)$

A-Star

A* is an extension of Dijkstra's algorithm using an additional heuristic to guide the direction of the search. Like Dijkstra, A* works by making a lowest-cost path tree from the start node to the target node but it uses an estimate $f(n)$ as an estimate of the total cost using this path.

$$f(n) = g(n) + h(n)$$

$f(n)$: total estimated cost of path through node n , $g(n)$: cost so far to reach node n , $h(n)$: estimated cost from n to goal.

A good heuristic $h(n)$ is the Manhattan distance.

Bellman-Ford

Instead of optimizing the order in which vertices are processed, Bellman-Ford simply relaxes all the edges $|V| - 1$ times and hence runs in $\mathcal{O}(|V||E|)$ time.

If we want to limit the number of edges that we pass through for our solution we can limit the outer loop to only run k times.

Algorithm 10: Bellman-Ford

```
Input :  $G = (V, E, source)$ 
1 for  $u \in V$  do
2    $dist[u] \leftarrow \text{INFINITY}$ ;
3    $prev[u] \leftarrow \text{UNDEFINED}$ ;
4 end
5  $dist[source] = 0$ ;
6 for  $u$  in  $|V|$  do
7   for  $v$  in Neighbors of  $u$  do
8      $alt = dist[u] + length(u, v)$ ;
9     if  $alt < dist[v]$  then
10        $dist[v] = alt$ ;
11        $prev[v] = u$ ;
12     end
13   end
14 end
15 for each edge  $(u, v)$  with weight  $w$  in  $|E|$  do
16   if  $dist[u] + w < dist[v]$  then
17     error "Graph contains a negative-weight cycle"
18   end
19 end
```

Runtime of Bellman Ford

- $\mathcal{O}(|E| \cdot |V|)$

Floyd-Warshall

Goal is to find the shortest path between all pairwise edges in a Graph G .

Algorithm 11: Floyd-Warshall

```
Input :  $G = (V, E)$ 
1 let  $G$  dist be a  $|V| \times |V|$  array of minimum distances initialized to  $\infty$ 
2 for each edge  $(u, v)$  do
3    $dist[u][v] \leftarrow w(u, v)$  // The weight of the edge  $(u, v)$ 
4 end
5 for each vertex  $v$  do
6    $dist[v][v] \leftarrow 0$ 
7 end
8 for  $k$  from 1 to  $|V|$  do
9   for  $i$  from 1 to  $|V|$  do
10    for  $j$  from 1 to  $|V|$  do
11      if  $dist[i][k] + dist[k][j] < dist[i][j]$  then
12         $dist[i][j] \leftarrow dist[i][k] + dist[k][j]$ ;
13      end
14    end
15  end
16 end
```

Runtime of Floyd-Warshall

- $\mathcal{O}(|V|^3)$

Johnson's Algorithm

Find the shortest paths between all pairs of vertices in an edge-weighted (negative), directed graph. Negative cycles are not allowed. It uses Bellman-Ford to remove all negative weights and then applies Dijkstra on the graph. The runtime is given by $\mathcal{O}(|V|^2 \log |V| + |V||E|)$. Thus when the graph is sparse the algorithm is faster than Floyd-Warshall which solves the same problem in $\mathcal{O}(|V|^3)$.

- New node q is added to the graph connected by zero-weight edges to each of the other nodes.
- Bellman-Ford is used starting from the new vertex q to find the minimum weight from q to each vertex v . If a negative cycle is detected the algorithm

terminates.

- The original edges are reweighted using the values computed in the Bellman-Ford step.
 $w'(u, v) = w(u, v) + h(u) - h(v)$
- q is removed and Dijkstra is used to find the shortest paths from each node s to every other vertex in the reweighted graph. The original distance is computed by adding $h(v) - h(u)$.

Choice of algorithm

- No weights or all equal weights \rightarrow BFS ($\mathcal{O}(|V| + |E|)$)
- Only positive weights \rightarrow Dijkstra with Fibonacci Heap ($\mathcal{O}(|V| \cdot \log(|V|) + |E|)$)
- Some negative weights \rightarrow Bellman Ford ($\mathcal{O}(|E| \cdot |V|^2)$)
- All pairs of shortest paths.
 - V times Dijkstra. If negative edges, recreate graph with Johnson first $\mathcal{O}(|E| \cdot |V| \log |V|)$
 - Floyd-Warshall. $\mathcal{O}(|V|^3)$
 - Johnsons in a sparse graph.
 $\mathcal{O}(|V|^2 \log |V| + |V||E|)$

Minimum Spanning Tree

Given is an undirected weighted connected graph $G(V, E)$.

Searched is a minimum spanning tree:

- Tree: connected and acyclic
- Spanning tree: All vertices $v \in V$ are connected.
- minimal: $c(T) = \min \sum_{e \in E} c(e)$

Kruskal algorithm

Algorithm 12: Kruskal

```
1 Sort edges increasingly after their weight:  $c(e_1) \leq c(e_2) \leq \dots \leq c(e_m)$ 
2  $A \leftarrow \emptyset$  for  $k = 1$  to  $m$  do
3   if  $A \cup e_k$  then
4      $A \leftarrow A \cup e_k$ 
5   end
6 end
```

Starts with the smallest edge! Edges that would create a cycle are subsequently discarded in the process \rightarrow exam question.

Runtime: $\mathcal{O}(E \log E)$

Jarnik (Prims) Algorithm

Algorithm 13: Jarnik Algorithm

```
1 start with  $v \in V$   $A \leftarrow \emptyset$ 
2  $S \leftarrow v_0$  for  $i = 1$  to  $|V|$  do
3   choose cheapest  $(u, v)$  with  $u \in S$  and  $v \notin S$ 
4    $A \leftarrow A \cup (u, v)$ 
5    $S \leftarrow S \cup v$ 
6 end
```

Main difference to Kruskal is, that it starts at $v \in V$ and chooses the cheapest edge from there.

Runtime: $\mathcal{O}(E + V \log V)$ with fibonacci heaps.

UnionFind

Find(x): Find the node x, go to the root of this subtree and return it. Union: Add the smaller subtree as a child to the larger subtree.

Max Flow / Min Cut

Given a flow network, determine the maximal flow allowed. The cut of the Graph G(S,T) into a source graph S and a sink graph T with the smallest capacity (min cut) will have the same capacity as the maximal flow.

Ford-Fulkerson

Algorithm 14: Ford-Fulkerson

```
1 for (u, v) ∈ E do
2   | f(u, v) = 0;
3 end
4 //G_f describes network capacities minus the existing flows
5 while Path p exists from s to t in residual network G_f do
6   | c_f(p) ← min{c_f(u, v) ∈ p};
7   | //increase the flow along this path
8   | for edge e(u, v) ∈ p do
9     |   f(e) ← f(e) + c_f(p);
10    |   c_f(e) ← c_f(e) - c_f(p);
11   end
12 end
```

Edmonds-Karp

Edmonds-Karp implements the Ford-Fulkerson algorithm by using a BFS search on the residual network.

Runtime of Ford-Fulkerson with Integers If f* is the maximum flow in the graph then, O(|E| · f*), because the flow needs to increase by at least 1 in each iteration and each can be done in O(|E|) time.

Runtime of Edmonds-Karp O(|V||E|) iterations, each of which can be done in O(|E|) times, so O(|V||E|^2)

Classes of Problems

Shortest-Path Problem

- Representation of simple graph with nodes representing the actual states (e.g. city).
- Representation of state space with nodes representing the current state of the system. (e.g. city and money left) → City is connected to neighbouring cities with the states that the system can take from here. (A5\$ → B3\$ -2\$)
- Cycle detection problem - e.g. figuring out if we can generate ∞ revenue. The Bellman–Ford algorithm will converge after iterating through the edges at most |V| – 1 times (as there cannot be more edges in a shortest path) if and only if there is no such negative cycle.

Bipartite Matching Problem

- Two classes of nodes that need to be matched in an

optimal fashion. Use either Ford-Fulkerson or Edmonds-Karp depending on the runtime.

Minimum Spanning Tree Problem

- Finding the minimal tree to connect all nodes of a tree/subtree. Use Kruskal or Prim for a dense graph.

Failure Resilience Problem

- Edge failure resilience - Maximal flow –1 between A and B gives the number of edges that can be removed before the connection fails.
- Node failure resilience - Replace each node with an edge - Model failure state by this edge weight.

Parallel Programming

Amdahl assumes a fixed relative sequential portion (λ), Gustafson assumes a fixed absolute sequential part.

Amdahl: $S_A = \frac{1}{\lambda + \frac{1-\lambda}{p}}$ Gustafson: $S_G = p - \lambda(p - 1)$

Speedup calculation

$T_p \leq \frac{T_1}{p} + T_\infty \mid S_p \geq \frac{T_1}{T_p}$
 $T_\infty = \text{longest single path} \mid S_\infty = \frac{T_1}{T_\infty}$

Performance Model

We have p processors and the corresponding execution time T_p . T_∞ : The span of the execution network or longest path. Thus the time needed if we have an infinite number of processors.

Parallelism = T_1/T_∞

Lower Bound Laws

$T_p \geq T_1/p$ Work law
 $T_p \geq T_\infty$ Span law

Parallel Programming in C++

std::mutex

- Owned when lock was called until unlock is called.
- When owned all other threads block (halt) when lock is called.

std::unique_lock

```
std::unique_lock<std::mutex> lck (mtx); //Locked
lck.unlock();
```

- In locked state upon construction unless deferred using std::defer_lock.
- Will handle unlocking upon destruction like std::lock_guard but additionally provided locking and unlocking capabilities.

std::condition_variable

```
std::condition_variable cv;
std::unique_lock<std::mutex> lk(m);
cv.wait(lk, []{return x == 1;});
```

```
lk.unlock();
cv.notify_one();
cv.notify_all();
```

- std::condition_variable takes a std::unique_lock<std::mutex> which protects the shared variable.
- Releases the std::mutex and executes a wait operation on the current thread if the condition does not hold.
- Upon notify_all or notify_one wakeup it will reacquire the mutex atomically and check the condition.

Race Conditions

Data Race

Bad synchronisation of a shared resource, e.g. two writing processes at the same time.

Bad Interleaving

Unlucky order of execution of e.g. two threads even though the shared resource is otherwise well synchronised.

Complexities

| Algorithm | Time Complexity | | | Space Complexity |
|----------------|---------------------------|-----------------------------|----------------------------------|------------------------|
| | Best | Average | Worst | Worst |
| Quicksort | $\Omega(n \cdot \log(n))$ | $\Theta(n \cdot \log(n))$ | $\mathcal{O}(n^2)$ | $\mathcal{O}(\log(n))$ |
| Mergesort | $\Omega(n \cdot \log(n))$ | $\Theta(n \cdot \log(n))$ | $\mathcal{O}(n \cdot \log(n))$ | $\mathcal{O}(n)$ |
| Heapsort | $\Omega(n \cdot \log(n))$ | $\Theta(n \cdot \log(n))$ | $\mathcal{O}(n \cdot \log(n))$ | $\mathcal{O}(1)$ |
| Bubble Sort | $\Omega(n)$ | $\Theta(n^2)$ | $\mathcal{O}(n^2)$ | $\mathcal{O}(1)$ |
| Insertion Sort | $\Omega(n)$ | $\Theta(n^2)$ | $\mathcal{O}(n^2)$ | $\mathcal{O}(1)$ |
| Selection Sort | $\Omega(n^2)$ | $\Theta(n^2)$ | $\mathcal{O}(n^2)$ | $\mathcal{O}(1)$ |
| Shell Sort | $\Omega(n \cdot \log(n))$ | $\Theta(n \cdot \log(n)^2)$ | $\mathcal{O}(n \cdot \log(n)^2)$ | $\mathcal{O}(1)$ |
| Bucket Sort | $\Omega(n + k)$ | $\Theta(n + k)$ | $\mathcal{O}(n^2)$ | $\mathcal{O}(n)$ |
| Radix Sort | $\Omega(n \cdot k)$ | $\Theta(n \cdot k)$ | $\mathcal{O}(n \cdot k)$ | $\mathcal{O}(n + k)$ |

| Data Structure | Time Complexity | | | | |
|--------------------|------------------------|------------------------|------------------------|------------------------|--------------------------------|
| | Average | | | | |
| Average | Access | Search | Insertion | Deletion | |
| Array | $\Theta(1)$ | $\Theta(n)$ | $\Theta(n)$ | $\Theta(n)$ | |
| Stack | $\Theta(n)$ | $\Theta(n)$ | $\Theta(1)$ | $\Theta(1)$ | |
| Queue | $\Theta(n)$ | $\Theta(n)$ | $\Theta(1)$ | $\Theta(1)$ | |
| Linked-List | $\Theta(n)$ | $\Theta(n)$ | $\Theta(1)$ | $\Theta(1)$ | |
| Skip-List | $\Theta(\log(n))$ | $\Theta(\log(n))$ | $\Theta(\log(n))$ | $\Theta(\log(n))$ | |
| Hash-Table | N/A | $\Theta(1)$ | $\Theta(1)$ | $\Theta(1)$ | |
| Binary Search Tree | $\Theta(\log(n))$ | $\Theta(\log(n))$ | $\Theta(\log(n))$ | $\Theta(\log(n))$ | |
| AVL Tree | $\Theta(\log(n))$ | $\Theta(\log(n))$ | $\Theta(\log(n))$ | $\Theta(\log(n))$ | |
| | Worst | | | | Space Complexity |
| | Access | Search | Insertion | Deletion | |
| Array | $\mathcal{O}(1)$ | $\mathcal{O}(n)$ | $\mathcal{O}(n)$ | $\mathcal{O}(n)$ | $\mathcal{O}(n)$ |
| Stack | $\mathcal{O}(n)$ | $\mathcal{O}(n)$ | $\mathcal{O}(1)$ | $\mathcal{O}(1)$ | $\mathcal{O}(n)$ |
| Queue | $\mathcal{O}(n)$ | $\mathcal{O}(n)$ | $\mathcal{O}(1)$ | $\mathcal{O}(1)$ | $\mathcal{O}(n)$ |
| Linked-List | $\mathcal{O}(n)$ | $\mathcal{O}(n)$ | $\mathcal{O}(1)$ | $\mathcal{O}(1)$ | $\mathcal{O}(n)$ |
| Skip-List | $\mathcal{O}(n)$ | $\mathcal{O}(n)$ | $\mathcal{O}(n)$ | $\mathcal{O}(n)$ | $\mathcal{O}(n \cdot \log(n))$ |
| Hash-Table | N/A | $\mathcal{O}(n)$ | $\mathcal{O}(n)$ | $\mathcal{O}(n)$ | $\mathcal{O}(n)$ |
| Binary Search Tree | $\mathcal{O}(n)$ | $\mathcal{O}(n)$ | $\mathcal{O}(n)$ | $\mathcal{O}(n)$ | $\mathcal{O}(n)$ |
| AVL Tree | $\mathcal{O}(\log(n))$ | $\mathcal{O}(\log(n))$ | $\mathcal{O}(\log(n))$ | $\mathcal{O}(\log(n))$ | $\mathcal{O}(n)$ |