Data Structures and Algorithms SS20

Algorithms

Notions of Growth

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

Tools Concerning Growth

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

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

Master Theorem

Let $a \geq 1$ and $b > 1$ be constants and $T(n) = aT(n/b) + f(n)$. Then $T(n)$ has the following asymptotic bounds:

1. If $f(n) = \mathcal{O}(n^{\log_b a - \epsilon})$ for $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \times \lg n)$
3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for $\epsilon > 0$, and $af(n/b) \leq cf(n)$ for $c < 1$, then $T(n) = \Theta(f(n))$

Logarithms and Important Sums

$\log_b x = \log_b a \times \log_a x,\ a^{log_b x} = x^{log_b a},\ \ln(n!) = \sum_{i=1}^{n} \ln i =\approx n\ln(n) - n,\ \sum_{i=0}^{n} i^k \in \Theta(n^{k+1}),\ \sum_{i=0}^{n} p^i = \frac{p^{n+1}-1}{p-1},\ \sum_{i=0}^{\infty} p^i = \frac{1}{1-p} \forall p \in [0, 1)$

Combinatorics

Binomial coefficient $\binom{n}{k} = \frac{n!}{k!(n-k)!}$, $\binom{n}{0} = \binom{n}{n} = 1,\ \binom{n+1}{k+1} = \binom{n}{k} + \binom{n}{k+1},\ \binom{n}{n-k} = \binom{n}{k}$

De l'Hôpital rule

Let $f, g : \mathbb{R} \to \mathbb{R}$ be differentiable functions with $f(x) \to \infty, g(x) \to \infty$ for $x \to \infty$. If $\lim_{x\to\infty} \frac{f'(x)}{g'(x)}$ exists, then $\lim_{x\to\infty} \frac{f(x)}{g(x)} = \lim_{x\to\infty} \frac{f'(x)}{g'(x)}$

Maximum Subarray Alogrithm $Runtime : \Theta(n)$

Algorithm 1: Inductive Maximum Subarray

```
Input  : $(a_1, a_2, ..-, a_n)$
Output: $\max 0, \max_{i,j} \sum_{k=i}^{j} a_k$
for $i = 1, ..., n$ do
    $R \leftarrow R + a_i$
    if $R < 0$ then
        $R \leftarrow 0$
    end
    if $R > M$ then
        $M \leftarrow R$
    end
end
return M
```

Searching

Linear Search

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

Binary Search

divide and conquer approach → $\Theta(\log n)$ 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
let Q be a queue
label root as discovered
Q.enqueue(root)
while Q is not empty do
    v := Q.dequeue() if v is the goal then
        return v
    end
    for all edges from v to w in G.adjacentEdges(v) do
        if w is not labeled as discovered then
            label w as discovered
            w.parent := v
            Q.enqueue(w)
        end
    end
end
```

Selecting

Pivot

Algorithm 3: Selection via Pivot

```
Input  : Array A of length n with pivot p
Output: A partitioned around p with position of p
$l \leftarrow 1$
$r \leftarrow n$ while $l \leq r$ do
    while $A[l] < p$ do
        $l \leftarrow l + 1$
    end
    while $A[r] > p$ do
        $r \leftarrow r - 1$
    end
    swap(A[l],A[r]) if $A[l] = A[r]$ then
        $l \leftarrow l + 1$
    end
end
return $l - 1$
```

Algorithm 4: Quickselect

```
Input  : Array A of length n; $1 \leq k \leq n$
$x \leftarrow$ RandomPivot(A)
$m \leftarrow$ Partition(A,x)
if $k < m$ then
    return Quickselect(A[0..m-1],k)
end
if k > m then
    return Quickselect(A[m+1..n],k) else
        return $A[k]$
    end
end
```

Sorting

3	8	5	4	1	2	7	6
3	5	4	1	2	7	6	8
3	4	1	2	5	6	7	8

Bubble Sort

3	8	5	4	1	2	7	6
3	5	8	4	1	2	7	6
3	4	5	8	1	2	7	6

Insertion Sort

3	8	5	4	1	2	7	6
3	8	4	5	1	2	6	7
3	4	5	8	1	2	6	7

Merge Sort

3	8	5	4	1	2	7	6
3	6	5	4	1	2	7	8
3	2	5	4	1	6	7	8

Selection Sort

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
if $n > 1$ then
    Choose Pivot $p \in A$ $k \leftarrow$ Partition(A,p)
    Quicksort(A[1,...,k-1])
    Quicksort(A[k+1,...,n])
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.
while $l < r$ do
    while $A[l] < p$ do
        $l = l + 1$
    end
    while $A[r] > p$ do
        $r = r - 1$
    end
    swap($A[l], A[r]$)
    if $A[l] = A[r]$ then
        $l = l + 1$
    end
end
return $l - 1$
```

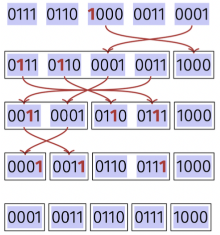
Runtime: in the mean $\mathcal{O}(n \cdot \log \cdot n)$, worst case $\Theta(n^2)$ if worst pivots are selected each time.

Radix Sort

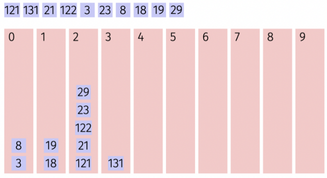
n-locks for n-keys $\in \mathcal{O}(n)$. We have m-adic binary numbers, so two categories to sort the numbers into. Used for numbers (and strings via UTF-8/ASCii)

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?).



Radix Sort



Bucket Sort

Hashing

Basics

Common: $h(k) = k \mod m$

Often: $m = 2^k - 1$

Linear Probing: $S(k) = (h(k), h(k)+1, ..., h(k)+m-1) \mod m$ *Issue*: Primary clustering, long contiguous areas of used entries.

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

Double Hashing: $S(k) = (h(k), h(k)+h'(k), h(k)+2h'(k), ..., h(k)+(m-1)h'(k)) \mod m$

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 $T_{right}(v)$ construct by: first element is root. first element larger than root is right child, remaining elements form left child. process both subrees recursively (first child is root)

Postorder

T_{left}, then T_{right}, next v construct by: last element is root. last element smaller than root is left child, remaining elements form right subtree. process the subtrees recursively (starting with right most node as parent)

Inorder

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

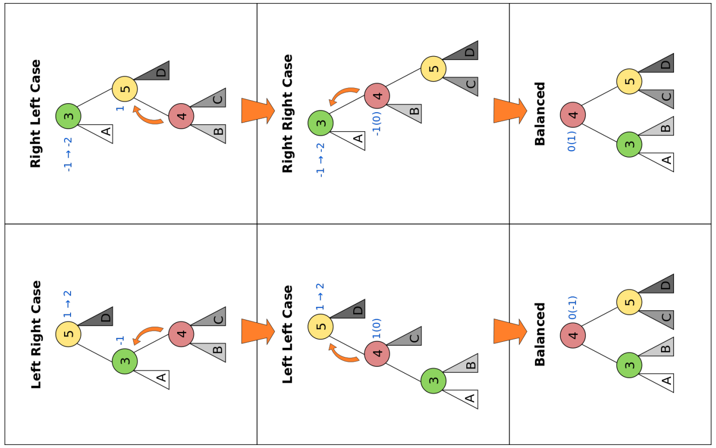
AVL trees

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**: $D[i] = 0, \forall i = 1, ..., n$ and $D[i] = \max\limits_{Slope(j,i)} \{D[j] + l(j,i)\}$
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 $A_{i,j} = \min\limits_{i \leq k \leq j} \{A_{i,k-1} \langle s_{k-1} \rangle A_{k,j}\}; 1 \leq i \leq j \leq n$
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.

Algorithms

Algorithm 7: Depth First Visit

```
Input : $G = (V, E)$
for $v \in V$ do
    $v.color \leftarrow$ white;
end
for $v \in V$ do
    if $v.color = white$ then
        DFS-Visit($G, v$)
    end
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

```
$A[v]$ contains number of entry edges of vertex v (calculate by setting
  $A[w] = 0$ and then loop through $(v, w) \in E$ and set $A[w] += 1$
for $v \in V$ where $A[v] == 0$ do
    Push($S$, $v$)
end
i=0;
while $S! = \{\}$ do
    $v \leftarrow$ pop($S$); ord[$v$] $\leftarrow i$
    $i$++;
    for (v, w) in E do
        $A[w] \leftarrow A[w] - 1$ ; //decrease incoming for all successors
        if A[$w$] == 0 then
            push($S$,$w$)
        end
    end
end
if i = |V| then
    return SUCCESS
end
else
    return “Cycle detected”
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  : $G = (V, E, source)$
create vertex set Q //as a queue / min heap;
for $u \in V$ do
    $dist[u] \leftarrow$ INFINITY;
    $prev[u] \leftarrow$ UNDEFINED;
    $Q.insert(u)$;
end
$dist[source] = $ 0;
while $Q$ not empty do
    $u = Q.ExtractMin()$;
    for $v$ in Neighbors of $u$ still in $Q$ do
        $alt = dist[u] + length(u, v)$;
        if $alt < dist[v]$ then
            $dist[v] = alt$;
            $prev[v] = u$;
            $Q$.DecreasePriority($v$, $alt$);
        end
    end
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

Dijkstra with a heuristic to visit nodes closer to the goal first (i.e. use Euclidean distance has an underestimation of which could be the closest points). A* minimizes f(n) = g(n)+h(n), where g(n) is distance from the start, h(n) estimation to goal.

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.

Algorithm 10: Bellman-Ford

```
Input  : $G = (V, E, source)$
for $u \in V$ do
    $dist[u] \leftarrow$ INFINITY;
    $prev[u] \leftarrow$ UNDEFINED;
end
$dist[source] = $ 0;
for $i$ in $|V| - 1$ do
    //i is never used, just a counter
    for $u$ in $|V|$ do
        for $v$ in Neighbors of $u$ do
            $alt = dist[u] + length(u, v)$;
            if $alt < dist[v]$ then
                $dist[v] = alt$;
                $prev[v] = u$;
            end
        end
    end
end
for each edge ($u$, $v$) with weight $w$ in $|E|$ do
    if $dist[u] + w < dist[v]$ then
        error "Graph contains a negative-weight cycle"
    end
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  : $G = (V, E)$
let $G$ dist be a $|V||V|$ array of minimum distances initialized to $\infty$
for each edge ($u$, $v$) do
    dist[$u$][$v$] $\leftarrow w(u, v)$ // The weight of the edge (u, v)
end
for each vertex $v$ do
    dist[v][v] $\leftarrow$ 0
end
for k from 1 to $|V|$ do
    for i from 1 to $|V|$ do
        for j from 1 to $|V|$ do
            if dist[i][j] > dist[i][k] + dist[k][j] then
                dist[i][j] $\leftarrow$ dist[i][k] + dist[k][j];
            end
        end
    end
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 $\mathbf{O}(|V|^2 \log |V| + |V||E|)$. Thus when the graph is sparse the algorithm is faster than Floyd-Warshall which solves the same problem in $\mathbf{O}(|V|^3)$.

1. New node q is added to the graph connected by zero-weight edges to each of the other nodes.
2. 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.
3. The original edges are reweighted using the values computed in the Bellman-Ford step. $w'(u, v) = w(u, v) + h(u) - h(v)$
4. 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 → BFS ($\Theta(|V| + |E|)$)
- Only positive weights → Dijkstra with Fibonacci Heap ($\mathcal{O}(|V| \cdot \log(|V|) + |E|)$)
- Some negative weights → 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. $\mathbf{O}(|V|^2 \log |V| + |V||E|)$

Minimum Spanning Tree

Given is a 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

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

Starts with the smallest edge! Edges that would create a cycle are subsequently discarded in the process → exam question.
Runtime: $\mathcal{O}(E \log E)$

Jarnik (Prims) Algorithm

Algorithm 13: Jarnik Algorithm

```
start with $v \in V$ $A \leftarrow \emptyset$
$S \leftarrow v_0$ for i = 1 to |V| do
    choose cheapest $(u, v)$ with $u \in S$ and $v \notin S$
    $A \leftarrow A \cup (u, v)$
    $S \leftarrow S \cup v$
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 capicity as the maximal flow.

Ford-Fulkerson

Algorithm 14: Ford-Fulkerson

```
for $(u, v) \in E$ do
    $f(u, v) = 0;$
end
//$G_f$ describes network capacities minus the existing flows
while Path $p$ exists from s to t in residual network $G_f$ do
    $c_f(p) \leftarrow min(c_f(u, v) \in p);$
    //increase the flow along this path
    for edge $e(u, v) \in p$ do
        $f(e) \leftarrow f(e) + c_f(p);$
        $c_f(e) \leftarrow c_f(e) - c_f(p);$
    end
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, $\mathcal{O}(|E| \cdot f*)$, because the flow needs to increase by at least 1 in each iteration and each can be done in $\mathcal{O}(|E|)$ time.
Runtime of Edmonds-Karp $\mathcal{O}(|V||E|)$ iterations, each of which can be done in $\mathcal{O}(|E|)$ times, so $\mathcal{O}(|V||E|^2)$

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

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

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.

$$\text{Parallelism} = T_1 / T_\infty$$

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 Best	 Average	 Worst	Space Complexity 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	
Heap	Acc min: $\mathcal{O}(1)$	N/A	$\mathcal{O}(1)$	$\mathcal{O}(\log(n))$	$\mathcal{O}(n)$
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	Worst
Heap	Acc min: $\mathcal{O}(1)$	N/A	$\mathcal{O}(\log(n))$	$\mathcal{O}(\log(n))$	$\mathcal{O}(n)$
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	$\mathcal{O}(log(n))$	$\mathcal{O}(log(n))$	$\mathcal{O}(log(n))$	$\mathcal{O}(log(n))$	$\mathcal{O}(n)$