**Data Structures and Algorithms** SS20 // Gian Silvan Hiltbrunner

# Algorithms

## Order of Complexity

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

$$\Theta(\sum_{i=0}^{n} i^2) = \Theta(n^3), \sum_{k=1}^{n} k = \frac{n(n+1)}{2}, \log_2 n! = n\log_2 n - n\log_2 e + O(\log_2 n).$$

## Asymptotics of Recursions

```
void g(int n){
    for (int i = 0; i<n ; ++i ){
        g(i)
    } f();
}
void w(int n){
    if (n > 1){ w(n/2); w(n/2); }
    while (--n > 0){ f(); }
}
void s(int n){ f();
if (n>1) { s(n/2); s(n/2); }}
```

$$g : \Theta(2^n),\ w : \Theta(n\log(n)),\ s : \Theta(n)$$

## Logarithms and Important Sums

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

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

**Algorithm 1:** Breadth First Search $\mathcal{O}(|V| + |E|) = \mathcal{O}(b^d)$

**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

## 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

## Quickselect

**Algorithm 2:** Selection via Pivot

**Input** : Array A of length n with pivot p
**Output:** A partitioned around p with position of p

```
l ← 1
r ← n 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
```

**Algorithm 3:** Quickselect $\mathcal{O}(n)$

**Input** : Array A of length n; $1 \le k \le n$

```
x ← RandomPivot(A)
m ← 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

- **Bubblesort**: Swap if $A[i-1] > A[i]$. In each round, the max in the unsorted part will move to the right. $\Theta(n^2)$ 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** 1. Pick a pivot 2. Partitioning: reorder the array so that all elements with values less than the pivot come before the pivot, while all elements with values greater than the pivot come after it (equal values can go either way). After this partitioning, the pivot is in its final position. 3. Recursively apply the above steps to the sub-array of elements with smaller values and separately to the sub-array of elements with greater values.

## Quicksort

**Algorithm 4:** Quicksort $\mathcal{O}(n \cdot \log n)$

**Input** : Array A of length n
**Output:** Array A sorted

```
if n > 1 then
    Choose Pivot p ∈ A k ← Partition(A,p)
    Quicksort(A[1,...,k-1])
    Quicksort(A[k+1,...,n])
end
```

**Algorithm 5:** 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
```

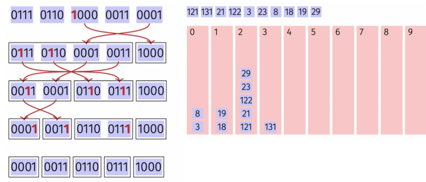
Runtime: $\Theta(n \cdot \log n)$, worst case $\Theta(n^2)$ if worst pivots are selected each time.

## Radix Sort $\mathcal{O}(w_{\text{bits}} n)$

$n$-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 $\Theta(n + n^2/k + k)$

Create a number of buckets. Sort e.g. after decimality into buckets and sort those buckets then. Can be implemented via linked list.



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 $T_{right}(v)$

#### Postorder

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

#### Inorder

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

### Heaps

Keys are strictly larger/smaller depending on Max- or Minheap. If the root is at index $0$: Children at indices $2i+1$ and $2i+2$

#### Build

In order to build a heap we run `heapify` on each element letting it trickle down. This takes $n \cdot \log n$ steps. Thus we get a runtime of $\mathcal{O}(n \log n)$

#### Insertion

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

#### Heap Sort

Non-stable sort using inductive sorting from below. With a running time of $\mathcal{O}(n \cdot \log n)$.

1. Call the buildMaxHeap() function on the list. Also referred to as heapify(), this builds a heap from a list in $\mathcal{O}(n)$ operations.
2. Swap the first element of the list with the final element. Decrease the considered range of the list by one.
3. Call the siftDown() function on the list to sift the new first element to its appropriate index in the heap.
4. Go to step (2) unless the considered range of the list is one element.

### 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.

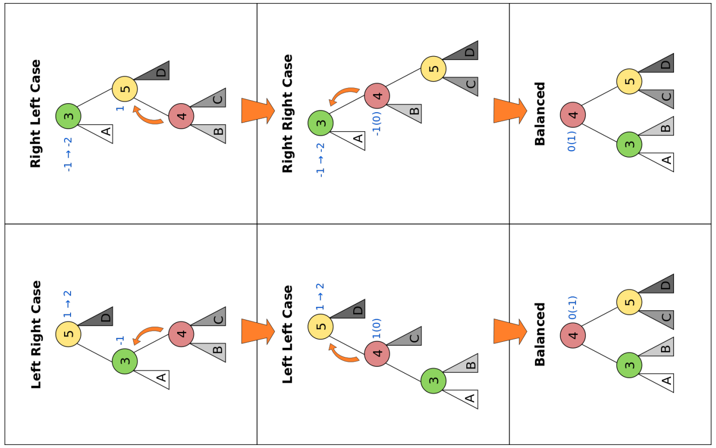
### AVL trees

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

### Examples

**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 $D[i] = \max_{Slope(j,i)} \{D[j] + l(j,i)\}$
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 6:** Depth First Search $\mathcal{O}(|V| + |E|) = \mathcal{O}(b^d)$

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

label v as discovered
**for** $E \in$ *G.adjacentEdges(v) (lexicographic)* **do**
  **if** *vertex w is not labeled as discovered* **then**
    DFS(G, w)
  **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 7:** 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's Algorithm

**Algorithm 8:** Dijkstra $\mathcal{O}(|E| log |V|)$

**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**

- 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 a 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 a function $f(n)$ 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 trough for our solution we can limit the outer loop to only run $k - 1$ times.

**Algorithm 9:** 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**
  **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**

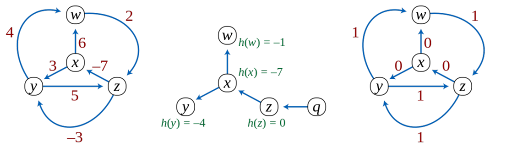
## Floyd-Warshall

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

**Algorithm 10:** Floyd-Warshall $\mathcal{O}(|V|^3)$

**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**

## Johnson's Algorithm




original graph with negative edges — shortest path tree found by Bellman-Ford — reweighted graph with no negative edges

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

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 $\rightarrow$ BFS ($\Theta(|V| + |E|)$)
- 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 on a sparse graph. $\mathcal{O}(|V|^2 \log |V| + |V||E|)$

## 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's Algorithm

**Algorithm 11:** Kruskal $\mathcal{O}(E \log E)$

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 $\rightarrow$ exam question.

## Jarnik (Prims) Algorithm

**Algorithm 12:** Jarnik Algorithm $\mathcal{O}(E + V \log V)$

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 capacity as the maximal flow.

## Ford-Fulkerson $\mathcal{O}(|E| \cdot f*)$

**Algorithm 13:** 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 $\mathcal{O}(|V||E|^2)$

## 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) $\rightarrow$ City is connected to neighbouring cities with the states that the system can take from here. ($A_{5\$} \xrightarrow[-2\$]{} B_{3\$}$)
- Finding a shortest path of length exactly $l$. We use a layered graph with $l$ layers. Then we run Dijkstra on this graph.

(Finding the shortest path when visiting $l$ cities.)

- Cycle detection problem - e.g. figuring out if we can generate $\infty$ revenue. The check includes if the shortest path to node 1 has decreased after one more iteration of the outermost loop of Bellman-Ford. 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 disjoint graph - Maximal flow $k-1$ between A and B gives the number of edges that can be removed before the connection fails.
- Node disjoint graph - Replace each node with an edge - Model failure state by this edge weight.

## Parallel Programming

Amdahl assumes a fixed relative sequential portion ($\lambda$), 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

$T_p \leq \frac{T_1}{p} + T_\infty$ * | $S_p = \frac{T_1}{T_p}$

$T_\infty =$ longest single path | $S_\infty = \frac{T_1}{T_\infty}$

In order to comply with the estimates for a greedy scheduler, it must hold that * and the **Lower Bound Laws** hold.

### Performance Model

We have $p$ processors and the corresponding execution time $T_p$. $T_\infty$: 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 $\qquad T_p \geq T_\infty$ Span law

**Work and Span for Recursions**

$$T_1(n) = r \times T_1(n_{new}) + \Theta,\ T_\infty(n) = T_\infty(n_{new}) + \Theta$$

If the setup is not fully concurrent then $T_\infty$ will depend on the maximal concurrency. For recursive calls $T(n/4)$ which are joined as groups of two we have: $\Theta(2^{\log_4(n)}) = \Theta(n^{\log_4(2)}) = \Theta(n^{\frac{1}{2}})$

### 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.

### Examples

**Readers-Writers Lock**

```
void acquire_read(){
    guard lock(m);
    c.wait(lock,[&]{return number_writers == 0 ;});
    ++number_readers;
}

void release_read(){
    guard lock(m);
    assert(number_readers > 0);
    --number_readers;
    if (number_readers == 0){
      c.notify_all();
    }
}
```

**Reentrant Lock**

```
void lock(){
    std::thread::id current = std::this_thread::get_id();
    guard lck(m);
    cv.wait(lck, [&]{return count == 0 or id == current;});
    if (id != current) id = current;
    count++;
}

void unlock(){
    guard lck(m);
    count--;
    if (count == 0) cv.notify_one();
}
```

### 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.

## Addendum

| **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 | |
| 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 |
| 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 | $\mathcal{O}(log(n))$ | $\mathcal{O}(log(n))$ | $\mathcal{O}(log(n))$ | $\mathcal{O}(log(n))$ | $\mathcal{O}(n)$ |

### Recursion Proofs

$T(n) = 3T(n/3) + n$ for $n > 1$; $0$ else

**Hypothesis:** $T(n) = f(n) =: n \log_3 n$

**Base Case:** $T(1) = 01 \log_3 1 = 0$

**Step:** $T(3n) = f(3n)$

$$T(3n) = 3T(n) + 3n = 3f(n) + 3n$$

$$3n(1 + \log_3 n) = (3n)\log_3(3n) = f(3n) \text{ qed.}$$

---

$T(n) = 3T(n-1) + 2^n$ for $n > 0$; $1$ else

$$= \sum_{k=0}^{n} 3^k 2^{n-k} = 2^n \sum_{k=0}^{n} 3^k 2^{-k} = 2^n \frac{(3/2)^{n+1} - 1}{3/2 - 1} = 3^{n+1} - 2^{n+1}$$

**Hypothesis:** $T(n) = 3^{n+1} - 2^{n+1}$

**Base Case:** $T(0) = 1 = 3 - 2 = 1$

**Step:** $T(n) = 3T(n-1) + 2^n = 3(3^n - 2^n) + 2^n$

$$= 3^{n+1} - 3 = 3^{n+1} - 2^{n+1} \text{ qed.}$$