Data Structures and Algorithms SS20
Flavio Rump, Martin Emons and Gian Hiltbrunner

Algorithms

Function Growth

$$\mathbb{O}(g) = \{f : \mathbb{N} \to \mathbb{R} \mid \exists c > 0, \exists n_0 \in \mathbb{N} : \forall n \geq n_0 : 0 \leq f(n) \leq c \cdot g(n)\} \quad (g : \mathbb{N} \to \mathbb{R})$$

$$\Omega(g) = \{f : \mathbb{N} \to \mathbb{R} \mid \exists c > 0, \exists n_0 \in \mathbb{N} : \forall n \geq n_0 : 0 \leq c \cdot g(n) \leq f(n)\} \quad (g : \mathbb{N} \to \mathbb{R})$$

$$g : \mathbb{N} \to \mathbb{R} : \qquad \Theta(g) := \nleq(g) \cap \mathbb{O}(g)$$

Notions of Growth

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

Tools Concerning Growth

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

$$\Rightarrow f \in \Theta(g); \frac{f(n)}{g(n)} \underset{n\to\infty}{\to} \infty g \in \mathcal{O}(f), \mathcal{O}(g) \subsetneq \mathcal{O}(f)$$

Complexity

Def: Complexity of problem P: Minimal costs over all algorithms A that solve P.

Correctness

Mathematical Proof

$$f(a,b) = \begin{cases} a & b = 1 \\ f\left(2a, \frac{b}{2}\right) & b \text{ even} \\ a + f(2a, \frac{b-1}{2}) & b \text{ odd} \end{cases}$$

Let $a \in \mathbb{Z}$, to show $f(a,b) = a \cdot b \forall b \in \mathbb{N}^+$
Base: $f(a,1) = a = a \cdot 1$
Hypothesis: $f(a,b') = a \cdot b' \ \forall 0 < b' \leq b$
Step: $f(a,b) = a \cdot b' \ \forall 0 < b' \leq b \to f(a, b+1) = a \cdot (b+1)$

$$f(a, b+1) = \begin{cases} f(2a, \overbrace{\frac{b+1}{2}}^{0<\cdot\leq b}) = a \cdot a(b+1) & b > 0 \text{ odd} \\ a + f(2a, \underbrace{\frac{b}{2}}_{0<\cdot<b}) = a + a \cdot b & b > 0 \text{ even} \end{cases}$$

Karatsuba-Ofman

Fast multiplication algorithm using at most $n^{\log_2 3} \approx n^{1.58}$ single digit multiplications.

Algorithm 1: Karatsuba-Ofman

Input : Two positive integers x and y with n decimal digits each.
Output: Product $x \cdot y$
if *n = 1* **then**
  return $x_1 \cdot y_1$
end
else
  $m := \lfloor\frac{n}{2}\rfloor;\ a := (x_1, \ldots, x_m);\ b := (x_{m+1}, \ldots, x_n);\ c := (y_1, \ldots, y_m); d := (y_{m+1}, \ldots, y_n)$
  $A :=$ karaof(a, c); $B :=$ karaof(b, d); $C :=$ karaof$(a - b, d - c)$;
  return $10^n \cdot A + 10^m \cdot A + 10^m \cdot B + B + 10^m \cdot C$
end

Iterative Substitution

Used to solve recurrence relations in two steps: ① Guess the form of the solution. ② Use induction to show that the guess is valid.
Ex: Analysis of Karatsuba-Ofman; recursive application of the algorithm.

$$M(2^k) = \begin{cases} 1 & k = 0 \\ 3 \cdot M(2^{k-1}) & k > 0 \end{cases} \qquad M(2^k) = 3 \cdot M(2^{k-1}) = 3 \cdot 3 \cdot M(2^{k-2}) = 3^2 \cdot M(2^{k-1}) = \ldots = 3^k \cdot M(2^0) = 3^k$$

Maximum Subarray Alogrithm

Algorithm 2: Inductive Maximum Subarray

Input : $(a_1, a_2, ..-, a_n)$
Output: $\max 0, \max_{i,j} \sum_{k=i}^{j} a_k$
for $i = 1, \ldots, n$ **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*

Runtime : $\Theta(n)$

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

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

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

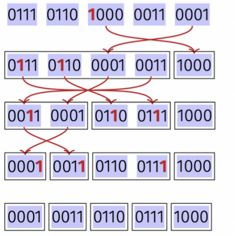
Runtime: in the mean $\mathcal{O}(n \cdot \log \cdot n)$, worst case $\Theta(n^2)$ if worst pivots are selected each time.

Radix Sort

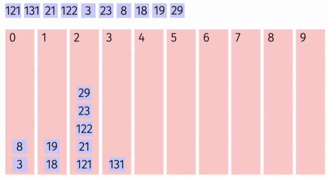
n-locks for n-keys $\in \mathcal{O}(n)$. We have m-adic binary numbers, so two categories to sort the numbers into. Used for numbers (and strings via UTF-8/ASCii)

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 - just 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.

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. $\rightarrow \mathcal{O}(n \cdot \log n)$

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 $D[i] = \max_{Slope(j,i)} \{D[j] + l(j,i)\}$
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.

Algorithms

Algorithm 6: 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 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

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

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 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 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
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 10: Floyd-Warshall

Input : $G = (V, E)$
let G dist be a $|V| \times |V|$ array of minimum distances initialized to ∞
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

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

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

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 11: 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 $\rightarrow$ exam question.

Jarnik Algorithm

Algorithm 12: 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.

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 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 $\mathcal{O}(|E| \cdot f*)$, 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

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

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

$$T_p \geq T_1/p \quad \text{Work law}$$
$$T_p \geq T_\infty \quad \text{Span law}$$

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.

Section	**Who**	**Status**
Searching / Selecting	Martin	1st draft
Sorting	Martin	1st draft
Data structures (?)	?	
Hashing	Gian	
Trees ?	Gian (Martin)	
DP	Gian	
Graphs	Flavio	what are we missing?
Shortest Paths	Flavio	1st draft
MST	Martin	1st draft
MaxFlow	Flavio	1st draft
PP	Gian	