Lecture 1: Amortized Analysis & Union Find

Rafael Oliveira

University of Waterloo
Cheriton School of Computer Science

rafael.oliveira.teaching@gmail.com

September 14, 2020

Overview

- Introduction
  - Why amortized analysis?
  - Types of amortized analyses
  - Union-Find
- Implementing Union-Find
  - Setup
  - First approach
  - Tree Representation & Path Compression
  - Analysis
- Acknowledgements

Why Amortized Analysis?

In your first data structures course, you learned how to devise data structures that had good *worst-case* or *average-case* behaviour *per query*.

Why Amortized Analysis?

In your first data structures course, you learned how to devise data structures that had good *worst-case* or *average-case* behaviour *per query*.

Worst or average-case complexity of data structures

Data Structure	search	insertion	deletion
Doubly-Linked List	$O(n)$	$O(1)$	$O(n)$
Ordered Array	$O(\log n)$	$O(n)$	$O(n)$
Hash Tables[a]	$O(1)$	$O(1)$	$O(1)$
Balanced Binary Search Trees[b]	$O(\log n)$	$O(\log n)$	$O(\log n)$

[a] Average-case, although worst-case search time is $\Theta(n)$

[b] Also average-case. Worst-case complexity is $O(\mathit{height})$ of the tree, which can be $\Theta(n)$.

Why Amortized Analysis?

In **amortized analysis**, one averages the *total time* required to perform a sequence of data-structure operations over *all operations performed*.

Upshot of amortized analysis: worst-case cost *per query* may be high for one particular query, so long as overall average cost per query is small in the end!

Remark

Amortized analysis is a *worst-case* analysis. That is, it measures the average performance of each operation in the worst case.

Types of amortized analyses

Three common types of amortized analyses:

Types of amortized analyses

Three common types of amortized analyses:

1. **Aggregate Analysis:** determine upper bound $T(n)$ on total cost of sequence of n operations. So amortized complexity is $T(n)/n$.

Types of amortized analyses

Three common types of amortized analyses:

1. **Aggregate Analysis:** determine upper bound $T(n)$ on total cost of sequence of n operations. So amortized complexity is $T(n)/n$.
2. **Accounting Method:** assign certain *charge* to each operation (independent of the actual cost of the operation). If operation is cheaper than the charge, then build up credit to use later.

Types of amortized analyses

Three common types of amortized analyses:

1. **Aggregate Analysis:** determine upper bound $T(n)$ on total cost of sequence of n operations. So amortized complexity is $T(n)/n$.
2. **Accounting Method:** assign certain *charge* to each operation (independent of the actual cost of the operation). If operation is cheaper than the charge, then build up credit to use later.
3. **Potential Method:** one comes up with *potential energy* of a data structure, which maps each state of entire data-structure to a real number (its "potential"). Differs from accounting method because we assign credit to the data structure as a whole, instead of assigning credit to each operation.

Why Union Find?

Certain problems/applications require one to maintain/group distinct elements into a collection of disjoint sets. For instance: maintaining connected components of a graph which keeps changing over time.

Why Union Find?

Certain problems/applications require one to maintain/group distinct elements into a collection of disjoint sets. For instance: maintaining connected components of a graph which keeps changing over time.

Uses: graph algorithms, social network graphs, etc.

These applications require data structure to perform two operations:

Why Union Find?

Certain problems/applications require one to maintain/group distinct elements into a collection of disjoint sets. For instance: maintaining connected components of a graph which keeps changing over time.

Uses: graph algorithms, social network graphs, etc.

These applications require data structure to perform two operations:

1. Find the unique set containing a particular element
    1. Input: element v from universe of elements
    2. Output: set containing v

Why Union Find?

Certain problems/applications require one to maintain/group distinct elements into a collection of disjoint sets. For instance: maintaining connected components of a graph which keeps changing over time.

Uses: graph algorithms, social network graphs, etc.

These applications require data structure to perform two operations:

1. Find the unique set containing a particular element
    1. Input: element v from universe of elements
    2. Output: set containing v
2. Take union of two disjoint sets
    1. Input: two sets A, B from you current collection of sets
    2. Output: updated collection of sets, i.e. with $A \cup B$ and without A, B

Application: Kruskal's minimum spanning tree algorithm

Input: graph $G(V, E)$ and edge weights $w : E \to \mathbb{N}$
Output: spanning tree T of minimum weight among all spanning trees.

Application: Kruskal's minimum spanning tree algorithm

Input: graph $G(V, E)$ and edge weights $w : E \to \mathbb{N}$
Output: spanning tree T of minimum weight among all spanning trees.

1. Sort edges $e_1, \ldots, e_{|E|}$ by weight such that $w(e_i) \le w(e_{i+1})$

Application: Kruskal's minimum spanning tree algorithm

Input: graph $G(V, E)$ and edge weights $w : E \to \mathbb{N}$
Output: spanning tree T of minimum weight among all spanning trees.

1. Sort edges $e_1, \ldots, e_{|E|}$ by weight such that $w(e_i) \leq w(e_{i+1})$
2. Set $T \leftarrow \emptyset$ (each vertex is a component by itself)

Application: Kruskal's minimum spanning tree algorithm

Input: graph $G(V, E)$ and edge weights $w : E \to \mathbb{N}$
Output: spanning tree T of minimum weight among all spanning trees.

1. Sort edges $e_1, \ldots, e_{|E|}$ by weight such that $w(e_i) \leq w(e_{i+1})$
2. Set $T \leftarrow \emptyset$ (each vertex is a component by itself)
3. for $i = 1, \ldots, |E|$:

Application: Kruskal's minimum spanning tree algorithm

Input: graph $G(V, E)$ and edge weights $w : E \to \mathbb{N}$
Output: spanning tree T of minimum weight among all spanning trees.

1. Sort edges $e_1, \ldots, e_{|E|}$ by weight such that $w(e_i) \leq w(e_{i+1})$
2. Set $T \leftarrow \emptyset$ (each vertex is a component by itself)
3. for $i = 1, \ldots, |E|$:
    1. if endpoints of e_i in different connected components of T (use two find operations on endpoints of e_i to check this step)
        - $T \leftarrow T \cup \{e_i\}$
        - combine the connected components of endpoints of e_i (union operation)

Application: Kruskal's minimum spanning tree algorithm

Input: graph $G(V, E)$ and edge weights $w : E \to \mathbb{N}$
Output: spanning tree T of minimum weight among all spanning trees.

1. Sort edges $e_1, \ldots, e_{|E|}$ by weight such that $w(e_i) \leq w(e_{i+1})$
2. Set $T \leftarrow \emptyset$ (each vertex is a component by itself)
3. for $i = 1, \ldots, |E|$:
    1. if endpoints of e_i in different connected components of T (use two find operations on endpoints of e_i to check this step)
        - $T \leftarrow T \cup \{e_i\}$
        - combine the connected components of endpoints of e_i (union operation)
4. return T

Application: Kruskal's minimum spanning tree algorithm

Input: graph $G(V, E)$ and edge weights $w : E \to \mathbb{N}$
Output: spanning tree T of minimum weight among all spanning trees.

1. Sort edges $e_1, \ldots, e_{|E|}$ by weight such that $w(e_i) \leq w(e_{i+1})$
2. Set $T \leftarrow \emptyset$ (each vertex is a component by itself)
3. for $i = 1, \ldots, |E|$:
    1. if endpoints of e_i in different connected components of T (use two find operations on endpoints of e_i to check this step)
        - $T \leftarrow T \cup \{e_i\}$
        - combine the connected components of endpoints of e_i (union operation)
4. return T

Remark

In this application, we care about the *total cost* of all operations (unions and finds). Thus, amortized analysis is better than worst-case per query.

Example

Example (continued)

Setup

Notation:

[1] Number of unions is $\leq n - 1$. We will assume that $m \geq n$

Setup

Notation:

- $n \leftarrow$ number of elements (we denote the elements by $1, 2, \ldots, n$)

[1] Number of unions is $\leq n - 1$. We will assume that $m \geq n$

Setup

Notation:

- $n \leftarrow$ number of elements (we denote the elements by $1, 2, \ldots, n$)
- $m \leftarrow$ number of operations. That is

$$m = (\text{number of finds}) + (\text{number of unions})^1$$

[1] Number of unions is $\leq n - 1$. We will assume that $m \geq n$

Setup

Notation:

- $n \leftarrow$ number of elements (we denote the elements by $1, 2, \ldots, n$)
- $m \leftarrow$ number of operations. That is

$$m = (\text{number of finds}) + (\text{number of unions})^1$$

- $FIND(k) \leftarrow$ find the set containing element k

[1] Number of unions is $\leq n - 1$. We will assume that $m \geq n$

Setup

Notation:

- $n \leftarrow$ number of elements (we denote the elements by $1, 2, \ldots, n$)
- $m \leftarrow$ number of operations. That is

$$m = (\text{number of finds}) + (\text{number of unions})^1$$

- $FIND(k) \leftarrow$ find the set containing element k
- $UNION(A, B) \leftarrow$ updates data structure by deleting sets A, B and constructing $A \cup B$

[1] Number of unions is $\leq n - 1$. We will assume that $m \geq n$

Naive approach

Keep an array S of size n where

$S[i]$ contains the name of set containing element i.

Naive approach

Keep an array S of size n where

$$S[i] \text{ contains the name of set containing element } i.$$

In this case, we have

- $FIND(k)$ takes time $O(1)$ (per operation)
- $UNION(A, B)$ takes time $O(|A| + |B|)$. Thus, $\Theta(n)$ worst case (per operation)

Naive approach

Keep an array S of size n where

$$S[i] \text{ contains the name of set containing element } i.$$

In this case, we have

- $FIND(k)$ takes time $O(1)$ (per operation)
- $UNION(A, B)$ takes time $O(|A| + |B|)$. Thus, $\Theta(n)$ worst case (per operation)

No amortized analysis yet.

Naive approach

Keep an array S of size n where

$$S[i] \text{ contains the name of set containing element } i.$$

In this case, we have

- $FIND(k)$ takes time $O(1)$ (per operation)
- $UNION(A, B)$ takes time $O(|A| + |B|)$. Thus, $\Theta(n)$ worst case (per operation)

No amortized analysis yet.

What if when taking the union of A and B, we only change name of the set of least size?

Naive Approach

What if when taking *UNION*, we only change name of the set of least size?

Naive Approach

What if when taking *UNION*, we only change name of the set of least size?

We will use *aggregate analysis* for this case: that is, determine upper bound on total cost of all operations.

Naive Approach

What if when taking *UNION*, we only change name of the set of least size?

We will use *aggregate analysis* for this case: that is, determine upper bound on total cost of all operations.

Cost of all unions $= O(n \log n)$, as for each element $i \in \{1, \ldots, n\}$, we have that the *UNION* operation will change $S[i]$ at most $\log n$ times.

Proof.

Every time we change $S[i]$, the size of the set containing element i doubles. □

Naive Approach

What if when taking *UNION*, we only change name of the set of least size?

We will use *aggregate analysis* for this case: that is, determine upper bound on total cost of all operations.

Cost of all unions $= O(n \log n)$, as for each element $i \in \{1, \ldots, n\}$, we have that the *UNION* operation will change $S[i]$ at most $\log n$ times.

Proof.

Every time we change $S[i]$, the size of the set containing element i doubles. □

Thus, cost of m operations is $O(m + n \log n)$ and we get that amortized cost is $O\left(1 + \frac{n \log n}{m}\right)$. If $m = \Omega(n \log n)$ this is best possible.

Naive Approach

What if when taking *UNION*, we only change name of the set of least size?

We will use *aggregate analysis* for this case: that is, determine upper bound on total cost of all operations.

> Cost of all unions $= O(n \log n)$, as for each element $i \in \{1, \ldots, n\}$, we have that the *UNION* operation will change $S[i]$ at most $\log n$ times.

Proof.

Every time we change $S[i]$, the size of the set containing element i doubles. □

Thus, cost of m operations is $O(m + n \log n)$ and we get that amortized cost is $O\left(1 + \frac{n \log n}{m}\right)$. If $m = \Omega(n \log n)$ this is best possible.
Are we done? What if $m = o(n \log n)$, can we do better?

Tree Representation

Represent each set as a tree of parent pointers. Each set will have its root as its representative element.

Tree Representation

Represent each set as a tree of parent pointers. Each set will have its root as its representative element.

- $FIND(k) \leftarrow$ walk up the tree from k and output name of the root

Tree Representation

Represent each set as a tree of parent pointers. Each set will have its root as its representative element.

- $FIND(k) \leftarrow$ walk up the tree from k and output name of the root
- $UNION(A, B) \leftarrow$ link both trees by making "smaller" tree's root point to "larger" tree's root.

Tree Representation

Represent each set as a tree of parent pointers. Each set will have its root as its representative element.

- $FIND(k) \leftarrow$ walk up the tree from k and output name of the root
- $UNION(A, B) \leftarrow$ link both trees by making “smaller” tree’s root point to “larger” tree’s root.

Question

How to define “smaller” (i.e., the “size” of a tree)?

Tree Representation

Represent each set as a tree of parent pointers. Each set will have its root as its representative element.

- $FIND(k) \leftarrow$ walk up the tree from k and output name of the root
- $UNION(A, B) \leftarrow$ link both trees by making "smaller" tree's root point to "larger" tree's root.

Question

How to define "smaller" (i.e., the "size" of a tree)?

- What if we define the size of a tree to be number of elements?
- What if we define the size of a tree to be it's height (longest path from leaf to root)?

Bad instances

- What if we define the size of a tree to be number of elements?

Bad instances

- What if we define the size of a tree to be it's height (longest path from leaf to root)?

Path Compression

To fix problems above, need path compression (i.e. make all trees "flat").

Definition (Path compression)

After each $FIND(k)$, for every node j on path $k \to \cdots \to$ root, set

$$PARENT(j) \leftarrow \text{root}.$$

Path Compression

To fix problems above, need path compression (i.e. make all trees "flat").

Definition (Path compression)

After each $FIND(k)$, for every node j on path $k \to \cdots \to$ root, set

$$PARENT(j) \leftarrow \text{root}.$$

This doubles the work of $FIND$, but that is fine, since it has same $O(\cdot)$ complexity. (no effect on asymptotics)

Path Compression

To fix problems above, need path compression (i.e. make all trees "flat").

Definition (Path compression)

After each $FIND(k)$, for every node j on path $k \to \cdots \to$ root, set

$$PARENT(j) \leftarrow \text{root}.$$

This doubles the work of $FIND$, but that is fine, since it has same $O(\cdot)$ complexity. (no effect on asymptotics)

This messes up the height of the tree, as path compression may change it.

Rank of a tree

Definition (Rank of tree)

For each tree with root r, define $\text{rank}(r)$ as follows:

- if the tree is a single element (r in this case) $\text{rank}(r) = 0$
- when performing union of two trees with roots r_1, r_2, if $\text{rank}(r_1) \geq \text{rank}(r_2)$, then
  - make r_1 the new root
  - set $\text{rank}(r_1) \leftarrow \max(r_1, r_2 + 1)$.

Rank of a tree

Definition (Rank of tree)

For each tree with root r, define $\text{rank}(r)$ as follows:

- if the tree is a single element (r in this case) $\text{rank}(r) = 0$
- when performing union of two trees with roots r_1, r_2, if $\text{rank}(r_1) \geq \text{rank}(r_2)$, then
  - make r_1 the new root
  - set $\text{rank}(r_1) \leftarrow \max(r_1, r_2 + 1)$.

Intuition: rank of a tree is the height if *no path compressions* had been done.

Final Algorithm

Input: set of elements $\{1, 2, \ldots, n\}$
Output: at each step, a union-find data structure comprised of disjoint union of sets whose union is $\{1, 2, \ldots, n\}$

1. Start with each set being $\{k\}$, where $k \in \{1, \ldots, n\}$. Set $\text{rank}(k) = 0$.
2. $UNION(S_1, S_2)$: where r_1, r_2 are the roots of S_1, S_2
   if $\text{rank}(r_1) \geq \text{rank}(r_2)$:
   1. make $\text{root}(S_1 \cup S_2) = r_1$, by creating pointer $r_2 \to r_1$.
   2. $\text{rank}(r_1) = \max(\text{rank}(r_1), \text{rank}(r_2) + 1)$

   else:
   1. make $\text{root}(S_1 \cup S_2) = r_2$, by creating pointer $r_1 \to r_2$.
   2. $\text{rank}(r_2) = \max(\text{rank}(r_2), \text{rank}(r_1) + 1)$
3. $FIND(k)$: walk up the tree from k to the root of its tree. Return name of root, and perform path compression.

Analysis

Theorem ([Tarjan 1975])

The amortized cost per operation of union-find is $\Theta(\alpha(m, n))$*, where* $\alpha(m, n)$ *is the inverse Ackermann function. That is, the (worst-case) cost of* m *operations is* $\Theta(m \cdot \alpha(m, n))$.

Analysis

Theorem ([Tarjan 1975])

The amortized cost per operation of union-find is $\Theta(\alpha(m, n))$*, where* $\alpha(m, n)$ *is the inverse Ackermann function. That is, the (worst-case) cost of* m *operations is* $\Theta(m \cdot \alpha(m, n))$.

Remark

Note the Θ in the statement. This means that the bound above is tight. Many tight examples exist.

Analysis

Theorem ([Tarjan 1975])

The amortized cost per operation of union-find is $\Theta(\alpha(m,n))$*, where* $\alpha(m,n)$ *is the inverse Ackermann function. That is, the (worst-case) cost of* m *operations is* $\Theta(m \cdot \alpha(m,n))$.

Remark

Note the Θ in the statement. This means that the bound above is tight. Many tight examples exist.

Remark

Inverse Ackermann function is mega-hyper-super slow growing. For more about the Ackermann function and its inverse, see `https://en.wikipedia.org/wiki/Ackermann_function`.

Analysis

In this class, we will see a weaker amortized bound of $O(\log^*(n))$ per operation. For another analysis, see [Seidel, Sharir 2005]. We will use the *accounting method*.

Analysis

In this class, we will see a weaker amortized bound of $O(\log^*(n))$ per operation. For another analysis, see [Seidel, Sharir 2005]. We will use the *accounting method*.

Definition

$$\log^*(n) := \min\{i \mid \log^{(i)}(n) \leq 1\},$$

where $\log^{(i)}$ means that we apply the log function i times.

n	1	2	$3, 4 = 2^2$	$5, \ldots, 16 = 2^{2^2}$	$17, \ldots, 65536 = 2^{16}$
$\log^*(n)$	0	1	2	3	4

Analysis

In this class, we will see a weaker amortized bound of $O(\log^*(n))$ per operation. For another analysis, see [Seidel, Sharir 2005]. We will use the *accounting method*.

Definition

$$\log^*(n) := \min\{i \mid \log^{(i)}(n) \leq 1\},$$

where $\log^{(i)}$ means that we apply the log function i times.

n	1	2	$3, 4 = 2^2$	$5, \ldots, 16 = 2^{2^2}$	$17, \ldots, 65536 = 2^{16}$
$\log^*(n)$	0	1	2	3	4

In the accounting method, we need to choose a charge to each operation $\hat{c}_i$ such that

$$\sum_{i=1}^{\ell} \hat{c}_i \geq \sum_{i=1}^{\ell} c_i$$

for all $\ell \leq m$, where c_i is the actual cost of the i^{th} operation.

Final Algorithm - recap

Input: set of elements $\{1, 2, \ldots, n\}$
Output: at each step, a union-find data structure comprised of disjoint union of sets whose union is $\{1, 2, \ldots, n\}$

1. Start with each set being $\{k\}$, where $k \in \{1, \ldots, n\}$. Set $\text{rank}(k) = 0$.
2. $UNION(S_1, S_2)$: where r_1, r_2 are the roots of S_1, S_2
   if $\text{rank}(r_1) \geq \text{rank}(r_2)$:
   1. make $\text{root}(S_1 \cup S_2) = r_1$, by creating pointer $r_2 \to r_1$.
   2. $\text{rank}(r_1) = \max(\text{rank}(r_1), \text{rank}(r_2) + 1)$

   else:
   1. make $\text{root}(S_1 \cup S_2) = r_2$, by creating pointer $r_1 \to r_2$.
   2. $\text{rank}(r_2) = \max(\text{rank}(r_2), \text{rank}(r_1) + 1)$
3. $FIND(k)$: walk up the tree from k to the root of its tree. Return name of root, and perform path compression.

Analysis

The complex operation is *FIND*, since we will perform *path compression*.

Analysis

The complex operation is *FIND*, since we will perform *path compression*.

Claim

When an element k is assigned $\text{rank}(k) = r$ *then k has* $\geq 2^r$ *descendants.*

Analysis

The complex operation is *FIND*, since we will perform *path compression*.

Claim

When an element k is assigned $\text{rank}(k) = r$ *then k has $\geq 2^r$ descendants.*

Claim

$\text{rank}(k) < \text{rank}(\mathit{parent}(k))$

Analysis

The complex operation is *FIND*, since we will perform *path compression*.

Claim

When an element k is assigned $\text{rank}(k) = r$ *then k has* $\geq 2^r$ *descendants.*

Claim

$\text{rank}(k) < \text{rank}(parent(k))$

Claim

Number of vertices of rank r is $\leq n/2^r$.

Grouping Elements Based on Rank

Idea: divide vertices into groups based on rank.

Element of rank r goes into group $\log^*(r)$. In particular, for element k, we have:

$$group(k) := \log^*(\text{rank}(k))$$

Grouping Elements Based on Rank

Idea: divide vertices into groups based on rank.

Element of rank r goes into group $\log^*(r)$. In particular, for element k, we have:

$$\mathit{group}(k) := \log^*(\text{rank}(k))$$

Remark

Number of groups: $\log^*(n)$.

Analysis

Actual cost of $FIND(k)$: distance from k to root.
Idea: charge some of this cost to *FIND* and some to nodes along path.

Analysis

Actual cost of $FIND(k)$: distance from k to root.

Idea: charge some of this cost to *FIND* and some to nodes along path.

Charging scheme:

1. $FIND(k)$
   - For each element u in the path $k \to$ root:
     - if u has parent and grandparent in path and group(u) = group(parent(u)), then charge 1 to u
     - else charge 1 to $FIND(k)$.
2. $UNION(A, B)$: just charge 1 to this operation

Analysis

Actual cost of $FIND(k)$: distance from k to root.

Idea: charge some of this cost to *FIND* and some to nodes along path.

Charging scheme:

1. $FIND(k)$
    - For each element u in the path $k \to$ root:
        - if u has parent and grandparent in path and group(u) = group(parent(u)), then charge 1 to u
        - else charge 1 to $FIND(k)$.
2. $UNION(A, B)$: just charge 1 to this operation

Remark

Note that charging scheme for $FIND(k)$ and nodes covers the actual cost of $FIND(k)$, since we are charging either the node on the path or the operation $FIND(k)$.

Since charging for *UNION* also covers the cost of the union operation, we have a valid charging scheme.

Charging Scheme Formally

So, how do we define the charges to $FIND(k)$?

$$\hat{c}_i(FIND(k)) = \tilde{c}_i(FIND(k)) + \sum_{u \in \text{ path } k \to u} (\text{charge to } u)$$

Analysis

Now we need to analyse the total amortized cost of this charging scheme.

Analysis

Now we need to analyse the total amortized cost of this charging scheme.

- Total charge to each $FIND(k)$ is $\leq \log^*(n) + 1$
  - Group changes $\leq \log^*(n) - 1$ times
  - $+2$ for root of tree and child of root of tree
- Total charge to each element of $\{1, \ldots, n\}$:
  - if k is charged in a path compression, then k is not root and path compression will give it a parent of higher rank than old parent.
  - if k has a parent in a higher group, then k will no longer be charged.
  - thus, if $\text{group}(k) = g$ then k can be charged at most

$$(\text{number of ranks in group } g) - 1 \leq 2 \uparrow g$$

Analysis

Now we need to analyse the total amortized cost of this charging scheme.

- Total charge to each $FIND(k)$ is $\leq \log^*(n) + 1$
  - Group changes $\leq \log^*(n) - 1$ times
  - $+2$ for root of tree and child of root of tree
- Total charge to each element of $\{1, \dots, n\}$:
  - if k is charged in a path compression, then k is not root and path compression will give it a parent of higher rank than old parent.
  - if k has a parent in a higher group, then k will no longer be charged.
  - thus, if group$(k) = g$ then k can be charged at most

$$(\text{number of ranks in group } g) - 1 \leq 2 \uparrow g$$

Let $N(g)$ be number of elements in group g. Then

$$N(g) \leq \sum_{r=2\uparrow(g-1)+1}^{2\uparrow g} \frac{n}{2^r} \leq \frac{n}{2^{2\uparrow(g-1)+1}} \cdot \sum_{0}^{\infty} 1/2^i = \frac{n}{2 \uparrow g}$$

Analysis

1. Thus, total charge to all elements in group g:

$$(\text{total charge per element in group } g) \cdot N(g) \leq (2 \uparrow g) \cdot \frac{n}{2 \uparrow g} = n$$

Analysis

1. Thus, total charge to all elements in group g:

$$(\text{total charge per element in group } g) \cdot N(g) \leq (2 \uparrow g) \cdot \frac{n}{2 \uparrow g} = n$$

2. Total charge to all elements of $\{1, \ldots, n\}$:

$$(\text{charge to all elements in group } g) \cdot (\text{number of groups}) \leq n \cdot \log^*(n)$$

Analysis

1. Thus, total charge to all elements in group g:

$$(\text{total charge per element in group } g) \cdot N(g) \leq (2 \uparrow g) \cdot \frac{n}{2 \uparrow g} = n$$

2. Total charge to all elements of $\{1, \ldots, n\}$:

$$(\text{charge to all elements in group } g) \cdot (\text{number of groups}) \leq n \cdot \log^*(n)$$

3. Total charge to all *FIND* operations:

$$(\text{number of } \mathit{FIND} \text{ operations}) \cdot (\text{charge per } \mathit{FIND}) \leq m \cdot (\log^*(n) + 1)$$

Analysis

1. Thus, total charge to all elements in group g:

$$(\text{total charge per element in group } g) \cdot N(g) \leq (2 \uparrow g) \cdot \frac{n}{2 \uparrow g} = n$$

2. Total charge to all elements of $\{1, \ldots, n\}$:

$$(\text{charge to all elements in group } g) \cdot (\text{number of groups}) \leq n \cdot \log^*(n)$$

3. Total charge to all *FIND* operations:

$$(\text{number of } \mathit{FIND} \text{ operations}) \cdot (\text{charge per } \mathit{FIND}) \leq m \cdot (\log^*(n) + 1)$$

4. Total charge overall: sum of 2 + 3.

$$O((m+n)\log^* n) = O(m \log^* n), \text{ as we assumed} n \leq m$$

Acknowledgement

Lecture based largely on Anna Lubiw's notes. See her notes at https://www.student.cs.uwaterloo.ca/~cs466/Lectures/Lecture5.pdf

References I

Tarjan, Robert (1975)
Efficiency of a good but not linear set union algorithm.
J. Assoc. Comput. Mach. 22, 215 – 225

Seidel, Raimund and Sharir, Micha. (2005)
Top-down analysis of path compression.
SIAM J. Computing 34(3), 515 – 525.