CSE 548: Analysis of Algorithms

Lectures 16, 17 & 18
( The \( \alpha \) Technique )

Inspiration Comes from Lectures Given by
Jeff Erickson, Seth Pettie, Vijaya Ramachandran and Raimund Seidel

Rezaul A. Chowdhury
Department of Computer Science
SUNY Stony Brook
Fall 2012
Iterated Functions

\[ f^*(n) = \begin{cases} 
0 & \text{if } n \leq 1 \\
1 + f^*(f(n)) & \text{if } n > 1 
\end{cases} \]

\[ = \min \left\{ i \geq 0 : f \left( f \left( f (\ldots f(n) \ldots) \right) \right) \leq 1 \right\} \]

\[ = \min \{ i \geq 0 : f^{(i)}(n) \leq 1 \}, \]

where \( f^{(i)}(n) = \begin{cases} 
n & \text{if } i = 0 \\
f (f^{(i-1)}(n)) & \text{if } i > 0 
\end{cases} \)

Example: If \( f = \log \), we have:

\[ \log^{(0)}(65536) = 65536 \quad \log^{(3)}(65536) = 2 \]
\[ \log^{(1)}(65536) = 16 \quad \log^{(4)}(65536) = 1 \]
\[ \log^{(2)}(65536) = 4 \quad \therefore \log^*(65536) = 4 \]
<table>
<thead>
<tr>
<th>$f(n)$</th>
<th>$f^*(n)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n - 1$</td>
<td>$n - 1$</td>
</tr>
<tr>
<td>$n - 2$</td>
<td>$n$</td>
</tr>
<tr>
<td>$n - c$</td>
<td>$\frac{n}{c}$</td>
</tr>
<tr>
<td>$\frac{n}{2}$</td>
<td>$\log_2 n$</td>
</tr>
<tr>
<td>$\frac{n}{c}$</td>
<td>$\log_c n$</td>
</tr>
<tr>
<td>$\sqrt{n}$</td>
<td>$\log \log n$</td>
</tr>
<tr>
<td>$\log n$</td>
<td>$\log^* n$</td>
</tr>
</tbody>
</table>
The Inverse Ackermann Function: $\alpha(n)$

$k = \alpha(n)$

\[
\begin{array}{ccc}
 f(n) & f^*(n) \\
 \log n & \log^* n & > 3 \\
 \log^* n & \log^{**} n & > 3 \\
 \log^{**} n & \log^{***} n & > 3 \\
 \end{array}
\]

\[
\begin{array}{ccc}
 \log^{**} n & \log^{**} n & > 3 \\
 \log^{***} n & \log^{****} n & \leq 3 \\
 \end{array}
\]

$$\alpha(n) = \min \left\{ k \geq 1 : \log^{\cdots \cdots \cdots} n \leq 3 \right\}$$
Union-Find:
A Disjoint-Set Data Structure
Disjoint Set Operations

A disjoint-set data structure maintains a collection of disjoint dynamic sets. Each set is identified by a representative which must be a member of the set.

The collection is maintained under the following operations:

**MAKE-SET( x ):** create a new set \{x\} containing only element x. Element x becomes the representative of the set.

**FIND( x ):** returns a pointer to the representative of the set containing x

**UNION( x, y ):** replace the dynamic sets \(S_x\) and \(S_y\) containing \(x\) and \(y\), respectively, with the set \(S_x \cup S_y\)
Union-Find Data Structure with Union by Rank and Find with Path Compression

**MAKE-SET** (x)
1. \( \pi(x) \leftarrow x \)
2. \( \text{rank}(x) \leftarrow 0 \)

**LINK** (x, y)
1. \( \text{if } \text{rank}(x) > \text{rank}(y) \text{ then } \pi(y) \leftarrow x \)
2. \( \text{else } \pi(x) \leftarrow y \)
3. \( \text{if } \text{rank}(x) = \text{rank}(y) \text{ then } \text{rank}(y) \leftarrow \text{rank}(y) + 1 \)

**UNION** (x, y)
1. \( \text{LINK} \left( \text{FIND} (x), \text{FIND} (y) \right) \)

**FIND** (x)
1. \( \text{if } x \neq \pi(x) \text{ then } \pi(x) \leftarrow \text{FIND} (\pi(x)) \)
2. \( \text{return } \pi(x) \)
Some Useful Properties of Rank

- If $x$ is not a root then $\text{rank}(x) < \text{rank}(\pi(x))$
- Node ranks strictly increase along any simple path towards a root
- Once a node becomes a non-root its rank never changes
- If $\pi(x)$ changes from $y$ to $z$ then $\text{rank}(z) > \text{rank}(y)$
- If the root of $x$'s tree changes from $y$ to $z$ then $\text{rank}(z) > \text{rank}(y)$
- If $x$ is the root of a tree then $\text{size}(x) \geq 2^{\text{rank}(x)}$
- If there are only $n$ nodes the highest possible rank is $\lfloor \log_2 n \rfloor$
- There are at most $\frac{n}{2^r}$ nodes with rank $r \geq 0$
Some Useful Properties of Rank

- We will analyze the total running time of \( m' \) \textsc{make-set}, \textsc{union} and \textsc{find} operations of which exactly \( n (\leq m') \) are \textsc{make-set}
- But each \textsc{union} can be replaced with two \textsc{find} and one \textsc{link}
- Hence, we can simply analyze the total running time of \( m \) \textsc{make-set}, \textsc{link} and \textsc{find} operations of which exactly \( n (\leq m) \) are \textsc{make-set} and where \( m' \leq m \leq 3m' \)
We will analyze the total running time of \( m \) \textsc{Make-Set}, \textsc{Union} and \textsc{Find} operations of which exactly \( n \ (\leq m) \) are \textsc{Make-Set}.

But \textsc{Find}(x) is nothing but \textsc{Compress}(x, y), where \( y \) is the root of the tree containing \( x \).

Hence, we can analyze the total running time of \( m \) \textsc{Make-Set}, \textsc{Link} and \textsc{Compress} operations of which exactly \( n \ (\leq m) \) are \textsc{Make-Set}.
**Compress**

\[
\text{Compress}(x, y) \quad \{ \text{y is an ancestor of } x \}\]

1. \( \text{if } x \neq y \text{ then } \pi(x) \leftarrow \text{Compress}(\pi(x), y) \)
2. \( \text{return } \pi(x) \)

We can reorder the sequence of \text{LINK} and \text{COMPRESS} operations so that all \text{LINK}'s are performed before all \text{COMPRESS} operations without changing the number of parent pointer reassignments!
**Shatter**

\[ \text{SHATTER}(x) \]

1. \( \text{if } x \neq \pi(x) \text{ then } \text{SHATTER}(\pi(x)) \)
2. \( \pi(x) \leftarrow x \)
Let $T(m, n, r) =$ worst-case number of parent pointer assignments

- during any sequence of at most $m$ COMPRESS operations
- on a forest of $n$ nodes
- with maximum rank $r$

**Bound 0**: $T(m, n, r) \leq nr$.

**Proof**: Since there are at most $r$ distinct ranks, and each new parent of a node has a higher rank than its previous parent, any node can change parents fewer than $r$ times.
Bound 1:

**Bound 1:** $T(m, n, r) \leq m + 2n \log^* r.$

**Proof:** Let $F$ be the forest, and $C$ be the sequence of `COMPRESS` operations performed on $F$.

Let $T(F, C)$ be the number of parent pointer assignments by $C$ in $F$.

Let $s$ be an arbitrary rank. We partition $F$ into two subforests:

- $F_b$ containing all nodes with rank $\leq s$, and
- $F_t$ containing all nodes with rank $> s$.
Bound 1: \( T(m, n, r) \leq m + 2n \log^* r. \)

Proof: Let \( s \) be an arbitrary rank. We partition \( F \) into two subforests:

- \( F_b \) containing all nodes with rank \( \leq s \), and
- \( F_t \) containing all nodes with rank \( > s \).

Let \( n_t = \#\text{nodes in } F_t \), and \( n_b = \#\text{nodes in } F_b \)

Let \( m_t = \#\text{COMPRESS operations with at least one node in } F_t \), and

\[ m_b = m - m_t \]
Bound 1

Bound 1: $T(m, n, r) \leq m + 2n \log^* r$.

Proof: The sequence $C$ on $F$ can be decomposed into

- a sequence of COMPRESS operations in $F_t$, and
- a sequence of COMPRESS and SHATTER operations in $F_b$

Suppose, this decomposition partitions $C$ into two subsequences

- $C_t$ in $F_t$, and
- $C_b$ in $F_b$
**Bound 1**

**Bound 1:** \( T(m, n, r) \leq m + 2n \log^* r. \)

**Proof:** We get the following recurrence:

\[
T(F, C) \leq T(F_t, C_t) + T(F_b, C_b) + m_t + n_b
\]

<table>
<thead>
<tr>
<th>Cost on Left Side</th>
<th>Corresponding Cost on Right Side</th>
</tr>
</thead>
<tbody>
<tr>
<td>node ( \in F_t ) gets new parent ( \in F_t )</td>
<td>( T(F_t, C_t) )</td>
</tr>
<tr>
<td>node ( \in F_b ) gets new parent ( \in F_b )</td>
<td>( T(F_b, C_b) )</td>
</tr>
<tr>
<td>node ( \in F_b ) gets new parent ( \in F_t ) (for the first time)</td>
<td>( n_b )</td>
</tr>
<tr>
<td>node ( \in F_b ) gets new parent ( \in F_t ) (again)</td>
<td>( m_t )</td>
</tr>
</tbody>
</table>
**Bound 1**

**Bound 1:** \( T(m, n, r) \leq m + 2n \log^* r \).

**Proof:** We get the following recurrence:

\[
T(F, C) \leq T(F_t, C_t) + T(F_b, C_b) + m_t + n_b
\]

Now \( n_t \leq \sum_{i>s} \frac{n}{2^i} = \frac{n}{2^s} \), and \( r_t = r - s < r \).

Hence, using bound 0: \( T(F_t, C_t) \leq n_t r_t < \frac{nr}{2^s} \).

Let \( s = \log r \). Then \( T(F_t, C_t) < n \).

Hence, \( T(F, C) \leq T(F_b, C_b) + m_t + 2n \)

\[\Rightarrow T(F, C) - m \leq T(F_b, C_b) - m_b + 2n\]
**Bound 1**

**Bound 1:** $T(m, n, r) \leq m + 2n \log^* r$.

**Proof:**

We got $T(F, C) - m \leq T(F_b, C_b) - m_b + 2n$

Let $T_1(m, n, r) = T(m, n, r) - m$

Then $T_1(m, n, r) \leq T_1(m_b, n_b, r_b) + 2n$

$\Rightarrow T_1(m, n, r) \leq T_1(m, n, \log r) + 2n$

Solving, $T_1(m, n, r) \leq 2n \log^* r$

Hence, $T(m, n, r) \leq m + 2n \log^* r$
Bound 2: $T(m, n, r) \leq 2m + 3n \log** r$.

Proof: Similar to the proof of bound 1.

But we solve $T(F_t, C_t)$ using bound 1, instead of bound 0!

We fix $s = \log^* r$ (instead of $\log r$ for bound 1)

Then using bound 1: $T(F_t, C_t) \leq m_t + 2n_t \log^* r_t$

$$\leq m_t + 2 \frac{n}{2 \log^* r} \log^* r$$

$$\leq m_t + 2n$$

Then from $T(F, C) \leq T(F_t, C_t) + T(F_b, C_b) + m_t + n_b$, we get

$$T(F, C) \leq T(F_b, C_b) + 2m_t + 3n_b$$
**Bound 2**

**Bound 2:** \( T(m, n, r) \leq 2m + 3n \log^* r \).

**Proof:** Our recurrence:

\[
T(F, C) \leq T(F_b, C_b) + 2m_t + 3n_b
\]

\[
\Rightarrow T(F, C) - 2m \leq T(F_b, C_b) - 2m_b + 3n_b
\]

Let \( T_2(m, n, r) = T(m, n, r) - 2m \)

Then \( T_2(m, n, r) \leq T_2(m_b, n_b, r_b) + 3n \)

\[
\Rightarrow T_2(m, n, r) \leq T_2(m, n, \log^* r) + 3n
\]

Solving, \( T_2(m, n, r) \leq 3n \log^* r \)

Hence, \( T(m, n, r) \leq 2m + 3n \log^* r \)
Bound $k$: $T(m, n, r) \leq km + (k + 1)n \log^k r$.

Observation: As we increase $k$:
- the dependency on $m$ increases
- the dependency on $r$ decreases

When $k = \alpha(r)$, we have $\log^k r \leq 3!$

Bound $\alpha$: $T(m, n, r) \leq m\alpha(r) + 3(\alpha(r) + 1)n$. 
The $\alpha$ Bound

**Bound $\alpha$:** $T(m, n, r) \leq m\alpha(r) + 3(\alpha(r) + 1)n$.

Observing that $r < n$, we have:

**Bound $\alpha$:** $T(m, n, r) \leq (m + 3n)\alpha(n) + 3n = O((m + n)\alpha(n))$.

Assuming $m \geq n$, we have:

**Bound $\alpha$:** $T(m, n, r) = O(m\alpha(n))$.

So, amortized complexity of each operation is only $O(\alpha(n))$!
The Partial Sums Data Structure
Semigroups

Semigroup \((\Pi, \oplus)\): A set \(\Pi\) together with an associative binary operation \(\oplus: \Pi \times \Pi \rightarrow \Pi\).

Examples:

\((\mathbb{R}, \max)\)

\((\{true, false\}, \text{logical OR})\)

\((k \times k \text{ matrices, matrix multiplication})\)
Partial Semigroup Sums

Given (i) a semigroup \((\Pi, \oplus)\), and
(ii) an array \(A[1 \ldots n]\) with each entry \(A[i] \in \Pi\)

Goal: Preprocess \(A\) using as little space as possible so that for all \(1 \leq i \leq j \leq n\), queries of the form \(A[i] \oplus A[i + 1] \oplus \ldots \oplus A[j]\) can be answered efficiently.

Space Complexity: \#values from \(\Pi\) stored in the data structure

Query Complexity: \#times the \(\oplus\) operation is applied

\(S_k(n)\): \#values from \(\Pi\) to be stored so that every partial sum query can be answered using at most \(k\) applications of the \(\oplus\) operation

\(k\)-op structure: A data structure with query complexity \(k\)
Bound 0

Bound 0: \( S_1(n) \leq n \log n \).

Construction of a 1-op structure:

Input array \( A \) of size \( n \)

Split \( A \) into \( A_l \) and \( A_r \) of size \( \frac{n}{2} \) each

Compute: all suffix-sums of \( A_l \), and
all prefix-sums of \( A_r \).

Recurse: 1-op structure for \( A_l \), and
1-op structure for \( A_r \).

Query: Either crosses \( A \)'s midpoint ( return suffix-sum \( \oplus \) prefix-sum ),
or lies completely inside \( A_l \) ( recurse ) or \( A_r \) ( recurse )
Bound 0

**Bound 0:** $S_1(n) \leq n \log n$.

**Construction of a 1-op structure:**

Input array $A$ of size $n$

Split $A$ into $A_l$ and $A_r$ of size $\frac{n}{2}$ each

Compute: all suffix-sums of $A_l$, and
all prefix-sums of $A_r$

Recurse: 1-op structure for $A_l$, and
1-op structure for $A_r$

**Space:** $S_1(n) \leq n + 2S_1\left(\frac{n}{2}\right) \\ \leq n \log n$
Bound 1: $S_3(n) \leq 3n \log^* n$.

Construction of a 3-op structure:

Split $A$ into $\frac{n}{\log n}$ subarrays of size $\leq \log n$ each.

Compute: all suffix- and prefix- sums within each subarray.

Build: 1-op structure for $\frac{n}{\log n}$ subarray sums.

Recurse: 3-op structure for each subarray.

Query: Either completely inside a subarray (recurse), or crosses subarray boundaries (return suffix-sum $\oplus$ answer from 1-op structure $\oplus$ prefix-sum).
Bound 1

Bound 1: $S_3(n) \leq 3n \log^* n$.

Construction of a 3-op structure:

Split $A$ into $\frac{n}{\log n}$ subarrays of size $\leq \log n$ each.

Compute: all suffix- and prefix- sums within each subarray.

Build: 1-op structure for $\frac{n}{\log n}$ subarray sums.

Recurse: 3-op structure for each subarray.

Space: $S_3(n) \leq 2n + S_1\left(\frac{n}{\log n}\right) + \frac{n}{\log n} S_3(\log n) \\
\leq 3n + \frac{n}{\log n} S_3(\log n) \leq 3n \log^* n$
**Bound 1**

**Bound 1:** $S_3(n) \leq 3n \log^* n$.

**Construction of a 3-op structure:**

Split $A$ into $\frac{n}{\log n}$ subarrays of size $\leq \log n$ each.

Compute: all suffix- and prefix-sums within each subarray.

Build: 1-op structure for $\frac{n}{\log n}$ subarray sums.

Recurse: 3-op structure for each subarray.

**Space:**

$S_3(n) \leq 2n + S_1 \left( \frac{n}{\log n} \right) + \frac{n}{\log n} S_3(\log n)$

$\leq 3n + \frac{n}{\log n} S_3(\log n) \leq 3n \log^* n$
Bound $k$

Bound $k$: $S_{2k+1}(n) \leq (2k + 1)n \log^{k} n = (2k + 1)n \log^{[*(k)]} n$.

Construction of a $(2k + 1)$-op structure:

Split $A$ into $n/\log^{[*(k-1)]} n$ subarrays of size $\leq \log^{[*(k-1)]} n$ each.

Compute: all suffix- and prefix- sums within each subarray.

Build: $(2k - 1)$-op structure for $n/\log^{[*(k-1)]} n$ subarray sums.

Recurse: $(2k + 1)$-op structure for each subarray.

Query: Either completely inside a subarray (recurse), or crosses subarray boundaries (return suffix-sum $\oplus$ answer from $(2k - 1)$-op structure $\oplus$ prefix-sum).
**Bound $k$**

**Bound $k$:** $S_{2k+1}(n) \leq (2k + 1)n \log^{\ast \ldots \ast} n = (2k + 1)n \log[\ast(\#)] n$.

**Construction of a $(2k + 1)$-op structure:**

Split $A$ into $n/\log[\ast(\#)] n$ subarrays of size $\leq \log[\ast(\#)] n$ each.

Compute: all suffix- and prefix- sums within each subarray.

Build: $(2k - 1)$-op structure for $n/\log[\ast(\#)] n$ subarray sums.

Recurse: $(2k + 1)$-op structure for each subarray.

**Space:** $S_{2k+1}(n) \leq 2n + S_{2k-1} \left( \frac{n}{\log[\ast(\#)] n} \right) + \frac{n}{\log[\ast(\#)] n} S_{2k+1} \left( \log[\ast(\#)] n \right) \leq (2k + 1)n + \frac{n}{\log[\ast(\#)] n} S_{2k+1} \left( \log[\ast(\#)] n \right) \leq (2k + 1)n \log[\ast(\#)] n$.
The $\alpha$ Bound

Bound $k$: $S_{2k+1}(n) \leq (2k + 1)n \log^k n$.

Putting $k = \alpha(n)$, we have:

Bound $\alpha$: $S_{2\alpha(n)+1}(n) \leq 3(2\alpha(n) + 1)n = O(n\alpha(n))$.

Linear Space: Use the $\alpha$-bound to show that the space complexity of the data structure can be reduced to $O(n)$ while still supporting range queries in $O(\alpha(n))$ time.