Basic Techniques - PRAMS (Ch. 2)

1. Balanced Tree Computations

- Build balanced tree on top of input element, traverse up forward to root and then back down again.
- Internal nodes store info re data in leaves of subtree they root.
- Depends on fast method to determine info for parent from data at kids (or vice versa).

⇒ One of simplest, most useful parallel techniques
⇒ e.g. sum, OR (EREW, CREW), prefix sums

Example: Array Companion

- input: array \( A[1..n] \) w/ \( m \leq n \) labeled elts (label array \( L \))
- output: array \( B[1..m] \) containing \( m \) labeled elts in same order they occurred in \( A \).

\[
\begin{array}{cccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
A & a_1 & a_2 & a_3 & a_4 & a_5 & a_6 & a_7 & a_8 \\
L & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 0 \\
B & a_1 & a_3 & a_4 & a_5 & a_7 \\
\end{array}
\]

⇒ Compute indices for elts in \( B \) array by computing the prefix sums of the \( L \) array.

- Time ⇒ Time for prefix sums \( O(\log n) \)
- Work ⇒ \( O(n) \)
Example of Prefix Sums \((n = 8, \text{ non recursive})\)

```
Example of Prefix Sums \((n = 8, \text{ non recursive})\)

\[
\begin{array}{cccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
\downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \\
1-2 & 3-4 & 5-6 & 7-8 & \\
\downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \\
1-4 & 5-8 & \\
\downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \\
1-8 & (b) & (a) & \\
\downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \\
1-2 (b) & 3-4 (a) & 5-6 (c) & 7-8 (a) & \\
\downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \\
1-1 & 1-2 & 1-3 & 1-4 & 1-5 & 1-6 & 1-7 & 1-8 & \\
\end{array}
\]

\text{internal nodes are temporary variables}
```

- Use aux array \(B(h, j)\) for upward (1st) traversal, \(0 \leq h \leq \log n, 1 \leq j \leq n\).
- \(C(h, j)\) for downward (2nd) traversal, \(C(0, j)\) is prefix sum.

\[\text{1. for } 1 \leq j \leq n \text{ pardo}
\]
\[B(0, j) := A(j)\]

\[\text{1st traversal \(2. \text{ for } h := 1 \text{ to } \log n \text{ do}
\]
\[\text{for } 1 \leq j \leq n/2^h \text{ pardo}
\]
\[B(h, j) := B(h-1, 2j-1) + B(h-1, 2j)\]

\[\text{2nd traversal \(3. \text{ for } h := \log n \text{ downto } 0 \text{ do}
\]
\[\text{for } 1 \leq j \leq n/2^h \text{ pardo}
\]
\[(a) \ j \text{ even: } C(h, j) := C(h+1, j/2) \quad \text{previous (2nd)}
\]
\[(b) \ j = 1 \text{: } C(h, j) := B(h, j) \quad \text{2nd traversal (hi)}
\]
\[(c) \ j \text{ odd+1: } C(h, j) := C(h+1, \frac{j-1}{2}) + B(h, j) \quad \text{previous (2nd)}\]
parado ⇒ can do all iterations in parallel (they are independent).

Complexity

| EREW PRAM |

assuming "enough" processors \((p = n)\)

\[ T(n) = 1 + \log n + \log n = O(\log n) \]

The total number of useful operations performed is \(O(n)\)

if not enough processors \((p < n)\)

\[ T(n) = \begin{cases} 
\text{Step 1 } & O(\%p) \text{ time} \\
\text{Step 2 } & \text{at each level (value of } h) \text{ divide work evenly among } P \text{ procs} \\
& \sum_{h=1}^{\log n} \frac{1}{p} \cdot \frac{n}{2^h} = \frac{n}{p} \sum_{h=1}^{\log n} \frac{1}{2^h} = O\left(\frac{n}{p} + \log n\right) \\
\text{Step 3 } & \text{same} \\
& O\left(\frac{n}{p} + \log n\right) 
\end{cases} \]

\[ W(n) = O(n) \quad \text{(useful work)} \]

Total cost (including "wasted" processors)

\[ C(n) = W(n) + p \cdot T(n) = O(n + p \log n) \]

\(\#\) of time steps \(\#\) of processors

So if \(p = O\left(\frac{n}{\log n}\right)\) then \(C(n) = O(n)\) which is optimal
Recursive Version

- Note in non-recursive version that in 2nd to bottom level we have prefix sums for all elements w/ even indices.

1. \( \text{if } n = 1 \) then \( S_1 = X_1 \)
2. for \( 1 \leq i \leq n/2 \) pardo
   \( y_i := X_{2i-1} + X_{2i} \)
3. recursively compute prefix sums of \( y_1, y_2, \ldots, y_{n/2} \) and store in \( z_1, z_2, \ldots, z_{n/2} \)
4. for \( 1 \leq i \leq n \) pardo
   if \( i \) even: \( S_i := z_i \)
   \( i = 1 \) : \( S_1 := X_1 \)
   \( i \text{ odd +1} \) : \( S_i := z_{(i-1)/2} + X_i \)

Complexity (assume \( p = n \))

\( T(n) = T(n/2) + 1 = O(\log n) \)
\( W(n) = W(n/2) + n = O(n) \)

(similar to previous to calculate w/ \( p < n \))

Lower bound on Time = \( \Omega(\log n) \) on \( \text{EREW} \oplus \text{CREW} \)
2. Pointer Jumping (Recursive Doubling)
- Used to traverse/search linked data structures
  - E.g., linked lists, rooted directed trees, graphs
- Assume start w/ one processor at every node

Ex: List ranking (also: parallel prefix)
Problem: Given a linked list, determine for each elt its distance from the end (tail) of the list.

\[ \text{HEAD} \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 7 \rightarrow 8 \rightarrow \text{TAIL} \]

- \( n \) = index of tail
- \( d(i) := 0 \) \( \forall i \leq n \)
- \( s(n) := n \)

For \( 1 \leq i < n \) par/do
- \( d(i) := 1 \)
- \( s(i) := \text{next}(i) \)
- While \( s(i) \neq s(s(i)) \) do
  - \( d(i) := d(i) + d(s(i)) \)
  - \( s(i) := s(s(i)) \)
- End while
- End for

Complexity
- \( T(n) = O(\log n) \) < every iter of while loop the "distance" to the tail is cut in half (w/ links)
- \( W(n) = O(n \log n) \) < all nodes keep updating \( s(i) \) ptrs every iteration until reach tail (Not work optimal since sequential alg takes \( O(n^2) \) time. Better ways are known)
Can do same thing w/ rooted directed in-trees

e.g. have a forest of trees & want each node to find root of its tree (parent of root is itself - self loop)

for $1 \leq i \leq n$ pardo
  Succ(i) := parent(i)
  while $S(i) \neq S(Sci)$ do
    $S(i) := S(Sci)$
  end for

Complexity:
- let $h$ be height of tallest tree
- $T(n) = O(\log h)$ -- every iter of while loop the "distance" to root is cut in half (# links)
- $W(n) = O(n \log h)$ -- all nodes keep updating Sci plus every iter until reach root

*if desired, can put wts on edges*
3. Divide-and-Conquer

1. Partition input into several subproblems of almost equal size (may want to do something smart here)
2. Solve subproblems recursively (in parallel)
3. Combine/merge solutions to the subproblems to get solution to original problem.

E.g. Convex Hull (in the plane)

![Convex Hull Diagram]

Fundamental problem in computational geometry.

Input: n points in the plane
Output: Points/vertices of convex hull in cyclic connection order

Sequential complexity: $O(n \log n)$ by simple divide-and-conquer matching lower bound (via sorting) of $\Omega(n \log n)$

Parallel Alg. based on same principle

- Let $p$ and $q$ be points of $S$ with min. $x$-max. $x$-coordinate

  - upperhull $\Rightarrow$ points of Convex Hull of $S$ from $p$ to $q$ in clockwise order ($UH(S)$, $CH(S)$)
  - lowerhull $\Rightarrow$ points of $CH(S)$ from $q$ to $p$ in clockwise order ($LH(S)$).

$\Rightarrow$ We can use similar algs to compute $UH(S)$ + $LH(S)$ so we only describe one to find $UH(S)$. 
Constant Time Convex Hull Algorithms

**Test 1** — only identifies convex hull vertices, not cyclic order
- point p e S is a convex hull vertex \( \iff p \) is external to the triangle formed by any 3 other points of S.
- Can test for p e S in \( O(1) \) time using \( \binom{n-1}{3} = O(n^3) \) procs
  - CRCW PRAM, have variable/flag for each p e S. Init to 1 for convex hull vertex & set to 0 if found internal.

**Total Complexity** — CRCW PRAM
- Time \( O(1) \), Work (#procs) \( O(n^3), O(n^3) \) for each p e S

**Test 2** — gives cyclic connection order
- point p e S is a convex hull vertex \( \iff \exists p, p_r \in S \ s.t. \ all \ points \ of \ S \ lie \ on \ one \ side \ of \ \overrightarrow{pp} \ and \ \overrightarrow{p_p} \)
- Can test for p e S in \( O(1) \) using \( O(n^3) \) procs, i.e. for each p e S use n-2 procs to see if remaining points on one side of \( \overrightarrow{pp} \)
  (need CRCW PRAM)

**Total Complexity** — CRCW PRAM
- Time \( O(1) \), Work (#procs) \( O(n^3), O(n^3) \) for each p e S

**OPEN PROBLEM** : Compute Convex Hull of 3D point set
deterministically in \( O(n \log n) \) time \( + \) \( O(n \log n) \) work on EREW or CRCW PRAM
- random: \( O(n \log n) \) time, \( O(n \log n) \) work (Reif & Scu)
- determ: \( O(n \log^2 n) \) time, \( O(n \log n) \) work (Amato, Goodrich, Ramos)
- determ: \( O(n \log n) \) time, \( O(n^{1+\epsilon} \log n) \) work (Amato, Preparata)
Fact: Can sort $n$ numbers in $O(\log n)$ time on CREW PRAM by using $O(\log n)$ work. (Closely parallel merge sort)

Assume no two pts have same $x$-coordinate and $n = 2^k$, since $k$.

Preprocessing Step
Sort points by $x$-coordinate, and rename them as
$E, p_1, p_2, \ldots, p_n$. Let $p = p_1 + q = p_n$ (leftmost + rightmost).

Convex Hull $(S)$
input: set $S$ of $n$ pts w/ $x$-coord. in increasing order
output: UH($S$)
1. if $n \leq 4$
   then compute UH($S$) by brute force & return
2. let $S_1 = \{p_1, p_2, \ldots, p_{2^k}\}$,
   $S_2 = \{p_{2^k+1}, \ldots, p_n, p\}$
   recursively compute, in parallel,
   UH($S_1$) + UH($S_2$)
3. find upper common tangent between
   UH($S_1$) + UH($S_2$) and merge them
   along this to form UH($S$) (may need to compact in array)
end /

This computation, if done naively, may require concurrent read, CREW, since many procs may need to check same variable. Can be made EREW w/ little extra effort.
How to compute Upper Common Tangent?
• Sequential technique takes $O(\log n)$ by binary search

• Take midpoints of current upper hulls. Will be able to
discard eliminate at least half of one chain.
$\Rightarrow O(\log n)$ iterations.

Complexity:
Time $T(n) = T(\frac{n}{2}) + O(\log n) = O(\log^2 n) \leftarrow$ not time optimal

Work $W(n) = 2 W(\frac{n}{2}) + O(n) = O(n \log n) \leftarrow$ work optimal

Preprocessing Time $O(\log n)$, Work $O(n \log n)$ (setting)
Speeding up Divide-And-Conquer Algs

1. Break into more subproblems initially (versus sequential Alg)
   - implement/use parallel alg for merging phase
   e.g. for convex hull can do merge in O(n) time w/ n proc (CREW)

2. Pipeline merging operations (aka cascading divide-and-conquer)
   e.g. for sorting in O(log n) time w/ O(log n) work
   Coles parallel merge sort does this.
Partitioning
- break problem up into $p$ subproblems of (almost) equal size
- solve $p$ subproblems concurrently

(in contrast to divide-and-conquer, most work in partitioning not in combining stage)

Merging problem
input: two sorted arrays $A[a_1, \ldots, a_n]$ and $B[b_1, \ldots, b_n]$
output: sorted array $C = (c_1, \ldots, c_{2n})$ w/ $A + B$

idea: find rank of each $a_i$ (blue) in $B$ (red), i.e. the # elts in $B$ less than $a_i$ (do same for $b_i$ in $A$).
⇒ The position of $a_i$ in $C$ is $i + \text{rank}(a_i \text{ in } B)$.

e.g. $A = \begin{bmatrix} 1 & 3 \end{bmatrix}$ $B = \begin{bmatrix} 2 & 4 & 6 \end{bmatrix}$
\[
\begin{align*}
\text{rank } (a_1 \text{ in } B) &= 0 & \text{rank } (b_1 \text{ in } A) &= 1 \\
\text{rank } (a_2 \text{ in } B) &= 1 & \text{rank } (b_2 \text{ in } A) &= 2 \\
\text{rank } (a_3 \text{ in } B) &= 2 & \text{rank } (b_3 \text{ in } A) &= 3
\end{align*}
\]
\[
\begin{align*}
a_1 \text{ at position } 0 & \quad \text{in } C \\
(1+0) &= a_1 \\
C(1+0) &= a_1 \\
C(2+1) &= a_3 \\
C(3+2) &= a_4 \\
C &= [a_1, b_1, a_2, b_2, a_3, b_3]
\end{align*}
\]
Simple, non-optimal parallel algorithm

- give each elt of $A + B$ a processor + determine its rank in appropriate array by binary search (CREW PRAM)

$\text{Time } O(n \log n)$

$\text{Work } O(n \log n)$ \leftarrow \text{not optimal, seq time is } O(n)$

Work-optimal parallel algorithm

- select every $\log n$ th elt of $A + B$ and rank in $B$

```
   1   2logn   3logn   ...   n
B   A
```

- NOTE: None of "arrows" cross since $A + B$ sorted

$\Rightarrow$ Now have $O(\log n)$ lists, each of size $O(\log n)$, whose els need to be mutually ranked (merged pairwise)

2) Rank the remaining els in appropriate list using seq alg.

$\Rightarrow$ all els can be ranked in $O(m)$ time using

1 processor for list of size $m$.

- for us $m = O(\log n)$

- use total of $O(\log n)$ proc (one for each list)

$\text{Complexity }$ $\text{Time } O(n \log n)$, $\text{Work } = O(\log n \cdot \frac{n}{\log n}) = O(n)$ (CREW PRAM)
5. Pipelining

Text does ex w/ 2-3 trees
- Later we'll do sorting w/ pipelining

Basic Idea
Computation on some tree structure
- Each computation moves up tree level by level from leaf to root
- As one computation leaves a level, start next comp. at that level
- So start computations serially, but process multiple computations simultaneously on different levels.

So Time w/ pipelining 8
Time w/o pipelining $5 \times 4 = 20$
**Accelerated Cascading** (to get very fast algo)

Method to combine:
(i) A slow, work-optimal algorithm
(ii) A fast(er), work sub-optimal algorithm

Both are parallel algo to get a
(iii) A fast work-optimal algorithm.

**Basic Idea**
- Start w/ the slow, optimal alg & use it to reduce the problem size to some threshold value.
- Then use fast, sub-optimal alg on the smaller sized problem (it will be work-sub-optimal for smaller size, but hopefully asymptotically work optimal for original size).

Ex. Finding maximum of n elts.

**Alg 1 - Slow, work-optimal**
B. Time $O(\log n)$
Work $O(n)$

Basic Binary tree alg
- build binary tree on inputs

```
  o
 / \
/   \
/     \
/       \
/         \
/           \
/             \
/               \
/                 \
/                   \
/                     \
/                       \
/                          \
/                             \
/                                \
/                                   \
/                                      \
/                                          \
/                                             \
/                                               \
/                                                   \
/                                                       \
/                                                           \
/                                                               \
/                                                                     \
/                                                                                     \
/                                                                                           \
/                                                                                                             \
/                                                                                                                   \
/                                                                                                                     \
/                                                                                                                               \
/                                                                                                                                     \
/                                                                                     \
/                                                                                           
/                                      x_1
/                                     / \
/                                    /   \n/                                  x_2  x_3
/                                /     \
/                            /       \
/                        x_4 \
/                     / \
/                  x_n-1
/               / \
/           x_n
```

Log n
Alg 2 (very) fast, (very) work-suboptimal

Constant time algorithm, uses $O(n^2)$ work.
- basic idea, perform all possible comparisons

input: array $A$ of $n$ distinct elts
output: Boolean array $M$ s.t. $M(i) = 1 \iff A(i)$ is max of $A$

1. for $1 \leq i \leq n$ pardo
   \[ M(i) := 1 \]  \* init all els to max +/
   end pardo

2. for $1 \leq i, j \leq n$ pardo  \* $\Theta(n^2)$ iterations +/
   if $A(i) < A(j)$
   then $M(i) := 0$  \* $A(i)$ is not max +/
   end pardo

3. for $1 \leq i \leq n$ pardo
   if $M(i) = 1$
   then $\max := i$  \* $A(i)$ is max +/
   end pardo

Complexity
\[ T(n) = O(1), \quad W(n) = O(n^2) \]

CRCW $\Rightarrow$ if statement in Step 2 needs both CR + CW

- Unfortunately, this alg is too work suboptimal to obtain good work optimal algorithm this is fast...
**Alg 3** fast, work suboptimal alg

Time = $O(\log \log n)$, Work = $O(n \log \log n)$

Based on Doubly logarithmic-depth tree */ Useful Technique */
Root has $\sqrt{n}$ children
children of root have $\sqrt{n}$ children
etc.

Wlog, assume $n = 2^k$, integer $k$
root has $2^k$ children
children of root have $2^{k-2}$ children
nodes at level $i$ have $2^{k-i-1}$ children (root level 0)
nodes at level $k$ have 2 children

e.g. $n=16$

Max Alg
do just like balanced binary tree max alg
except use constant time alg at internal nodes

Time = $O(\log \log n)$ Work at level $i$ nodes is $O((2^{2(k-1)}-1)^2)$ ($# \text{kids}^2$)
	$\times$ # level i nodes = $O(n) = O(2^k)$

Total Work = #levels $\times O(n) = O(n \log \log n)$
Applying Accelerating Cascading

1. Run binary tree algorithm \((A_2)\) for \(\log \log n\) levels
   - at each level reduce \# candidates by factor of \(2\).
   - after \(\log \log n\) levels we've reduced \# candidates by a factor of \(2^{\log \log n} = \log n\)

   i.e. we now have
   \[ n' = O\left(\frac{n}{\log \log n}\right) \text{ candidates} \]

   \[ \text{Time} = O(\log \log n) \quad \text{Work} = O(n) \]

2. Run Doubly logarithmic algorithm \((A_3)\) on the remaining \(n'\) candidates

   \[ \text{Time} = O(\log \log n') = O(\log \log n) \]

   \[ \text{Work} = O\left(\frac{n'}{\log \log n'}\right) = O\left(\frac{n}{\log \log n \cdot \log \log n}\right) = O(n) \]

   Still CRLW since \(2\) needs it.

\textbf{Lower bounds}

\[ \text{Work} = \Omega(n) \quad \text{(need to look at all)} \]

\[ \text{Time} = \Omega(\log \log n) \quad \text{on GRCW PRAM (any model, Ch.4)} \]

\[ \Omega(\log n) \quad \text{on CREW or EREW (binary tree optimal)} \]

\( \leq \) from or lower bound.

\( \Rightarrow \) \text{so ALGS are Work + Time Optimal}
Remark

For Max problem, can also use optimal sequential algorithm in phase D.

$\rightarrow$ Partition input into $\frac{1}{\log\log n}$ blocks of $\log\log n$ elts.
  * Sequentially find max of each block (in parallel)

  Time $O(\log\log n)$
  Work $O(n)$

Then continue as before & use doubly logarithmic alg for the remaining $O(\frac{1}{\log\log n})$ candidates.
Symmetry Breaking

**Graph**

Ex. Coloring a directed cycle $G = (V, E)$

**def** A *k*-coloring of $G$ is a mapping $c : V \rightarrow \{0, 1, ..., k-1\}$

**s.t.** $c(i) \neq c(j)$ if $(i, j) \in E$.

**note:** A cycle can be 3-colored.

**Sequential Alg. to 3-color $G$, a directed cycle**

- traverse cycle and assign vertices colors alternating between 0 and 1
- we may need third color(2) for last vertex

$\Rightarrow$ optimal $O(n)$ time sequential alg., hard to parallelize

**problem w/ parallelization:** Want to process many vertices at same time, but it's hard to partition them into sets that can be dealt w/ independently (Symmetry breaking)

- one technique we'll see here (play w/ bits)
- another technique we'll see later uses randomization.

**Input Assumption:**

- $G = (V, E)$, $|V| = |E| = n$
- $E$ represented by array $P[1..n]$ s.t. $P(i) = j$ if $(i, j) \in E$

$\Rightarrow$ $P$ gives predecessor relation
def: given integer x in binary, $x_k$ is k-least sig digit of x.

Basic Coloring Algorithm

Input: G = (V,E), predecessor array $\pi$(1..n)
Output: A coloring c' of V

\[ \text{Init color} \]

1. for $1 \leq i \leq n$ pardo
   \[ c(i) := i \quad \text{\# init color of vertex to be its number} \]
   end pardo

2. for $1 \leq i \leq n$ pardo
   \[ k := \text{least sig digit where } c(i) + c(\pi(i)) \text{ differ} \]
   \[ c'(i) := 2k + c(i)_k \]
   end pardo

end /* Coloring */

<table>
<thead>
<tr>
<th>V</th>
<th>c</th>
<th>K</th>
<th>c'</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>001</td>
<td>1</td>
<td>2+0 = 2</td>
</tr>
<tr>
<td>2</td>
<td>010</td>
<td>0</td>
<td>0+0 = 0</td>
</tr>
<tr>
<td>3</td>
<td>011</td>
<td>0</td>
<td>0+1 = 1</td>
</tr>
<tr>
<td>4</td>
<td>100</td>
<td>0</td>
<td>0+0 = 0</td>
</tr>
<tr>
<td>5</td>
<td>101</td>
<td>0</td>
<td>0+1 = 1</td>
</tr>
<tr>
<td>6</td>
<td>110</td>
<td>0</td>
<td>0+0 = 0</td>
</tr>
<tr>
<td>7</td>
<td>111</td>
<td>0</td>
<td>0+1 = 1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>V</th>
<th>c</th>
<th>K</th>
<th>c'</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>001</td>
<td>0</td>
<td>0+1 = 1</td>
</tr>
<tr>
<td>2</td>
<td>011</td>
<td>1</td>
<td>2+1 = 3</td>
</tr>
<tr>
<td>3</td>
<td>111</td>
<td>2</td>
<td>4+1 = 5</td>
</tr>
<tr>
<td>4</td>
<td>101</td>
<td>1</td>
<td>2+0 = 2</td>
</tr>
<tr>
<td>5</td>
<td>110</td>
<td>0</td>
<td>0+0 = 0</td>
</tr>
<tr>
<td>6</td>
<td>101</td>
<td>1</td>
<td>4+1 = 5</td>
</tr>
<tr>
<td>7</td>
<td>011</td>
<td>2</td>
<td>4+0 = 4</td>
</tr>
</tbody>
</table>
**Complexity**

EREW PRAM

\[ T(n) = O(1) \quad \text{for assume can determine bit positions in constant time} \]
\[ W(n) = O(n) \]

**Lemma** Basic Coloring Alg. produces a valid coloring.

**Proof**

Suppose \( c'(i) = c'(j) \) for some \( (i,j) \in E \).

\[
\begin{align*}
  c'(i) &= 2k + c(i)_k \\
  c'(j) &= 2l + c(j)_l
\end{align*}
\]

where \( k \) and \( l \) come from alg.

For \( c'(i) = c'(j) \) we need \( K = l \) and thus \( c(i)_k = c(j)_l \)

But this is not possible since contradicts definition of \( K \), i.e. \( K \) is lsb that differs in \( c(i) + c(j) \)

\[ \therefore \quad (c(i)_k + c(j)_k = 0) \]

No colors resulting from BCA: (rough estimate)

* start with \( 2^t \) colors \((t = \# \text{bits} \oplus \text{for } n, \text{i.e. } n/\log n)\)

* use \( \leq 3 \times 2t + 1 \) colors, i.e. \( O(\log n) \).

\( \Rightarrow \) exponential decrease.

**Fact:** Can apply BCA repeatedly

* after \( O(\log^* n) \) iterations reach at most 6 colors

\( T(n) = O(\log^* n) \)

\( \Rightarrow \) To finish off can (sequentially) process all vertices w/ colors 4, 5, 6 and resolve w/ lowest valid 0, 1, 2

\( \Rightarrow \) result w/ 3 coloring in \( O(\log^* n) \) time, \( W(n) = O(\log^* n) \)