$n = |V|$
$m = |E|$

## 1. Bootstrapping

- $t_A(n)$: maximum running time for inputs of size $n$
- $p_A(n)$: Algorithm A giving back right result
- $t(n) \leq \alpha n + \beta \cdot t(n/2)$ ($\beta$ important for runtime)

$$t(n) = \alpha n + \beta \cdot \alpha \frac{n}{2} + \beta^2 \cdot \alpha \frac{n}{4} + \dots + \beta^{\log_2(n)} \cdot \alpha \cdot t(1) \quad (t(1) \leftarrow 1)$$

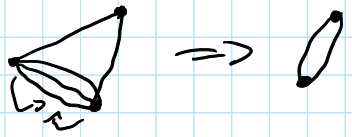
$$\hookrightarrow \sum_{i=0}^{\log_2(n)} \left(\frac{\beta}{2}\right)^i \leq \sum_{i=0}^{\infty} \left(\frac{\beta}{2}\right)^i = \frac{1}{1 - \beta/2}$$

$\beta < 2 \Rightarrow O(n)$

$\beta = 2 \Rightarrow O(n \cdot \log(n))$

$\beta > 2 \Rightarrow O(n^{\log_2(\beta)})$

- Edge contraction:



- Reduces vertices by exactly 1
- Reduces edges by at least 1

---

## 1.1. Computing minimum spanning Tree

- Prim's algorithm: $O(n \cdot \log(n+m))$
  1. Set set S to one vertex
  2. choose cheapest outgoing edge
  3. insert new vertex into set
  4. Do again for bigger set
- Kruskal's algorithm: $O(m \cdot \log(n))$
  1. Look for smallest edges
  2. If they create cycle, don't choose them
- Randomized MST algorithm: $O(m)$
- $e_{min}(v)$: edge of $v$ with minimum weight
- $e_{max}(c)$: edge in cycle $c$ with maximum weight
- T-heavy: edge with greatest weight in a closed cycle
- $T$ is a MST $\Leftrightarrow$ there are no edges that are $t$-heavy
- Boruvka's algorithm
  1. Compute for every vertex the minimum edge, insert that edge in the MST
  2. Contract that minimum edge of each vertex
  3. Recurse until only one vertex is left
  - One step done in $O(m)$
  - In every step we reduce number of vertices by 2
    $\hookrightarrow O(m \cdot \log(n))$ in total

- Problem Boruvka: · Reduce vertices by factor of 2
  · No control of reducement of edges

- Randomized Minimum Spanning Tree Algorithm
  1. Perform 3 iterations of Boruvka
  2. In new graph, select edges with probability 1/2
  3. Compute MSF $\bar{T}$ recursively of selected edges
  4. Use FindHeavy to find all unselected non-t-heavy edges
  5. Delete all t-heavy edges
  6. Recurse until only one vertex left

---

## 1.2. Computing minimum cuts

- Minimum number of edges such that graph is split
- $\mu(G)$ = size of minimum cut in $G$
- Basic algorithm in $O(n^2)$ (=with prob. amp. $O(n^4)$)
  1. while $G$ has more than 2 vertices do
     1.2. Pick a random edge $e$ in $G$
     1.3. Contract that edge $e$
  2. Return size of the only cut in $G$
  · Returns a number $\geq \mu(G)$
- $\Pr[\mu(G) = \mu(G \setminus e)] \geq 1 - \frac{\text{All edges in the minimum cut}}{\text{All edges in the graph}}$

  ($G \setminus e$: graph $G$ with edge $e$ contracted)

  $= 1 - \frac{\mu(G)}{|E(G)|}$

  $\geq 1 - \frac{\mu(G)}{\frac{n \cdot \mu(G)}{2}}$ ← Each vertex has at least degree $k$

  $= 1 - \frac{n}{2}$ ← $n$ vertices

Basic algorithm succeeds with $n$ vertices ↓

- $p(n) = \left(1 - \frac{n}{2}\right) \cdot p(n-1) \geq \frac{n-2}{n} \cdot \frac{n-3}{n-2} \cdot \frac{n-4}{n-2} \cdot \ldots \cdot \frac{2}{4} \cdot \frac{1}{3} \cdot 1$

$= \left(\frac{2}{n(n-1)}\right)^N$ ← number of independent runs (= probability amp.)

## 1.2.2. Bootstrapping

- Probability of contracting an edge in the minimum cut increases as the graph shrinks
  ↳ contract edges until some threshold $t$, then repeat algorithm two times to get smaller error probability: the smaller the number of vertices, the larger the number of recursive calls

- **MinCut(G):**
  1. If $n \leq 16$ → compute $\mu(G)$ by some det. method
  2. else
     - 2.2 $t = \lceil \alpha \cdot n \rceil + 1$
     - 2.3 $H_1$ = RandomContract$(G, t)$ ← stop until there are only $t$ vertices left
     - 2.4 $H_2$ = RandomContract$(G, t)$
     - 2.5 return min(MinCut$(H_1)$, MinCut$(H_2)$)

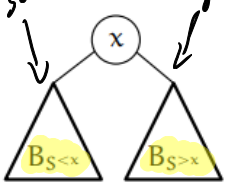
- $t(n) \leq \underbrace{O(n^2)}_{\text{RandContract 2x}} + \underbrace{2t(\lceil \alpha n \rceil + 1)}_{\text{Recursion}}$

- If we choose $\alpha \leq 1/\sqrt{2}$ → $O(n^2 \cdot \log(n))$

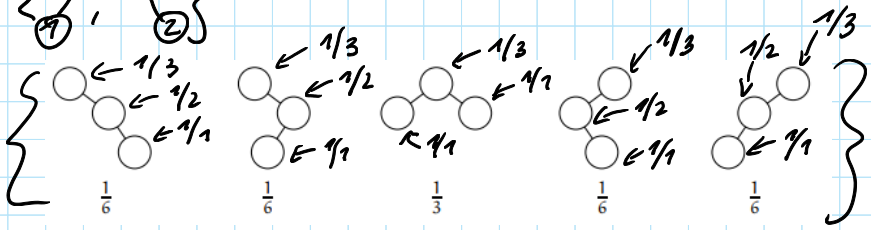
# 2. Random Search trees

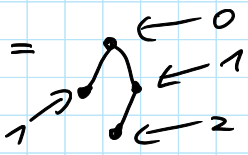
## 2.1. Definition

$B_S \rightarrow \begin{cases} \lambda, & \text{if } S = \emptyset, \text{ and} \\ \text{smaller } x \text{ greater} & \text{for } x \in S, \text{ otherwise.} \end{cases}$



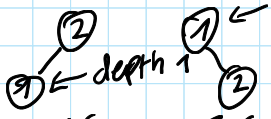

Random: for every root, we choose x u.a.r. from the set S of possible numbers

- $B_S$ = all possible variants of trees with $s$ nodes
- $B_0 = \{\lambda\}$
- $B_1 = \{(1)\}$
- $B_2 =$ 
- $B_3 =$ 


- Probability that a random-search-tree has a given structure: $\prod_{v \in V(t)} \frac{1}{w(v)}$ where $w(v) :=$ number of vertices in the subtree rooted at $v$ incl. $v$
- Height of a tree := maximum depth of all nodes in tree
- Depth of a tree := 

## 2.2 Overall (and average) depth of keys

- Rank of an element: How small is it?
  - $rk(1) = 1$
  - $rk(s)$ - smallest element in the set
- $D_n^{(i)}$ := depth of the key of rank $i$ in tree with $n$ keys
- Expected depth of smallest key in tree:
  - $E[D_n^{(1)}] = E[D_n]$
  - $E[D_1] = 0$
  - $E[D_2] = 1/2 \cdot 1 + 1/2 \cdot 0 = 1/2$




  - $E[D_3] = 1/6 \cdot 0 + 1/6 \cdot 0 + 1/3 \cdot 1 + 1/6 \cdot 1 + 1/6 \cdot 2 = 5/6$
  - $E[D_n] = \sum_{i=1}^{n} \underbrace{E[D_n \mid rk(root) = i]}_{\begin{cases} 0, & \text{if } i = 1 \\ 1 + E[D_{i-1}], & \text{else} \end{cases}} \cdot \underbrace{Pr[rk(root) = i]}_{1/n}$

    We only care about left side of trees, because there is smallest element located, that tree is 1 element smaller
  - $E[D_n] = d_n = \begin{cases} 0, & \text{if } n = 1 \\ \frac{1}{n} \cdot \sum_{i=2}^{n} (1 + d_{i-1}), & \text{else} \end{cases}$
- Overall depth:
  - $x_n = \sum_{i=1}^{n} D_n^{(i)}$
  - $E[x_n] = \sum_{i=1}^{n} \underbrace{E[x_n \mid rk(root) = i]}_{\begin{matrix} = (n-1) + E[x_{i-1}] \text{ left} \\ + E[x_{n-i}] \text{ right} \end{matrix}} \cdot \underbrace{Pr[rk(root) = i]}_{1/n}$

$$= \sum_{i=1}^{n} \left[ (n-1) + 2E[x_{i-1}] \right] \cdot 1/n$$

$$= \left[ n(n-1) + 2 \cdot \sum_{i=1}^{n} E[x_{i-1}] \right] \cdot 1/n$$

$$= (n-1) + \frac{2}{n} \cdot \sum_{i=1}^{n} E[x_{i-1}]$$

  - $E[x_n] = x_n = \begin{cases} 0, & \text{if } n = 0 \\ (n-1) + \frac{2}{n} \cdot \sum_{i=0}^{n-1} x_i, & \text{else} \end{cases}$ *

* $\sum_{i=1}^{n} x_{i-1} = \sum_{i=0}^{n-1} x_i$

## 2.3. Expected height

- Want to get the expected maximum depth
- $X_n := \max_{i=1}^{n} D_n^{(i)}$
- Because of max-operator, we use:
- $E[X_n] \leq \log E[2^{X_n}] = \log E[2^{\max_{i=1}^{n} D_n^{(i)}}] \leq \log E[\underbrace{\sum_{i=1}^{n} 2^{D_n^{(i)}}}_{Z_n}]$
- $E[Z_n] = \sum_{i=1}^{n} \underbrace{E[Z_n \mid rk(root)=i]}_{2\cdot(E[Z_{i-1}]+E[Z_{n-i}])} \cdot \underbrace{Pr[rk(root)=i]}_{1/n}$

  $4 \cdot E[Z_{i-1}]$

  Without the trick, it would be $1 + 2 \cdot E[Z_{i-1}]$
- $E[Z_n] = z_n = \begin{cases} 0, & \text{if } n=0 \\ 1, & \text{if } n=1 \\ \frac{4}{n} \cdot \sum_{i=1}^{n} z_{i-1}, & \text{else} \end{cases}$

---

## 2.4. Expected depth of individual keys

- $d_{i,n} = E[d_n^{(i)}]$
- $A_i^j := [\text{node } j \text{ is ancestor of node } i]$


- $D_n^{(i)} = \sum_{j=0, j\neq i}^{n} A_i^j$
- $E[D_n^{(i)}] = \sum_{j=0, i\neq j}^{n} \underbrace{E[A_i^j]}_{Pr[A_i^j = 1]}$ (= indicator variable)

  Proof $i \leq j$, $j \leq i$ is symmetric

  $Pr[j \text{ is ancestor of } i] = \frac{1}{|i-j|+1}$

- 1. case: $i=j$: Trivially true
- 2. case: $i=1, j=n$: n must be the root, otherwise j would never be the ancestor of i, this happens with prob. $1/n$
- 3. case: root $= \{i, \dots, j\}$: We have to choose j as root, this happens with prob. $1/j-i+1$ ← choosing j from $\{i, \dots, j\}$
- 4. case: root $\notin \{i, \dots, j\}$: Both fall in same subtree ⇒ recurse

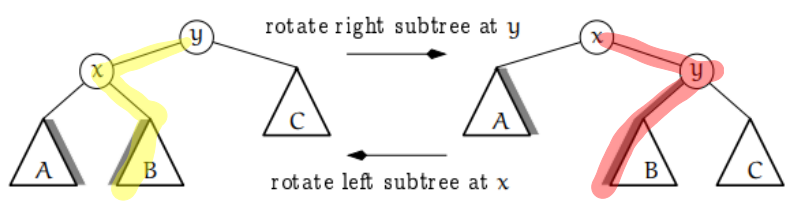
## 2.5. Quicksort $O(n \cdot \log(n))$ / $\Theta(n^2)$

- $t_n = \underbrace{(n-1)}_{\text{current sort}} + \underbrace{\sum_{i=1}^{n}}_{\text{all poss. pivots}} \underbrace{(t_{i-1}}_{\text{left}} + \underbrace{t_{n-i})}_{\text{right}} \cdot \underbrace{\frac{1}{n}}_{\text{average}} = (n-1) + \frac{2}{n} \cdot \sum_{i=1}^{n} t_{i-1}$
- Expected number of comparisons is equal to the overall depth of a random search tree

---

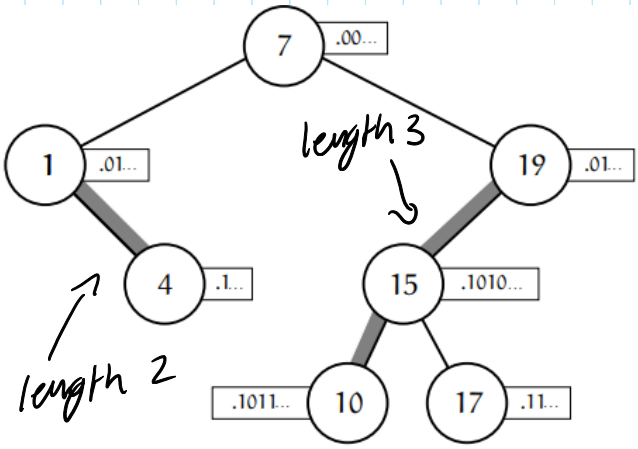
## 2.7. Randomized Search Trees

- Treap (= Binary search tree and a min-heap)
- Binary search tree with respect to key(x)
- Min-heap with respect to prio(x) $\in [0,1]$ (min is root)
- Insertion:
  1. Insert item x as a leaf as if in a binary-tree
  2. As long as x has smaller priority than its parent, push x up as in an AVL-tree
     ↳ $O(\log(n))$ (worst: down and then completely up again)
- Number of rotations needed:
  - Expected number of rotations is less than 2




- Sum of "left subtree right spline" and "right subtree left spline" increases/decreases by one for every rotation
- Length of "left subtree right spline" = $1 - \frac{1}{j}$
- Length of "right subtree left spline" = $1 - \frac{1}{n-j+1}$

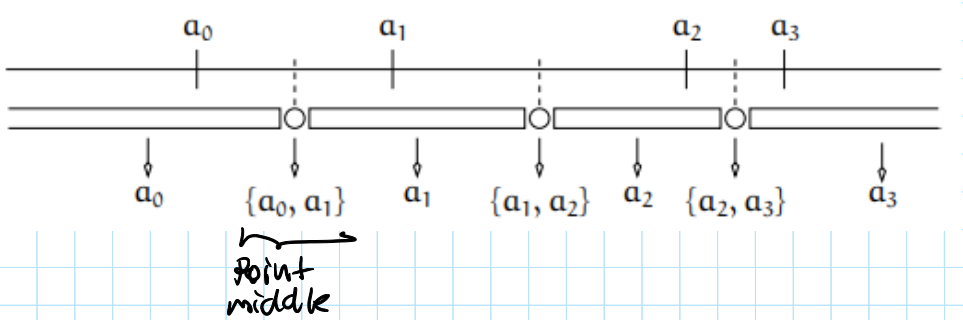
$\Big\} \oplus < 2$




- To insert 7, there were 5 rotations done

# 3. Point Locations

- Locate some query-point q in some partition
- partition is mostly fixed => preprocessing pays off
- Locus approach
  - Minimize distance q to next partition-point
    ↳ No unique answer: a | q • | b
  - Partition space into n-1 points and n open intervals, where the left/right one extend to ±∞

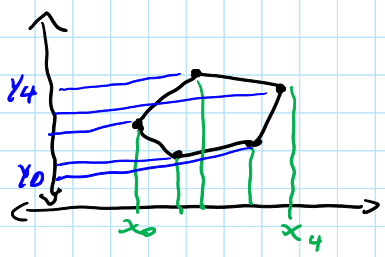



  - Space: O(n)
  - Preprocessing: O(n·log(n)) (=search for every point)
  - Query: O(log(n)) (=search)

---

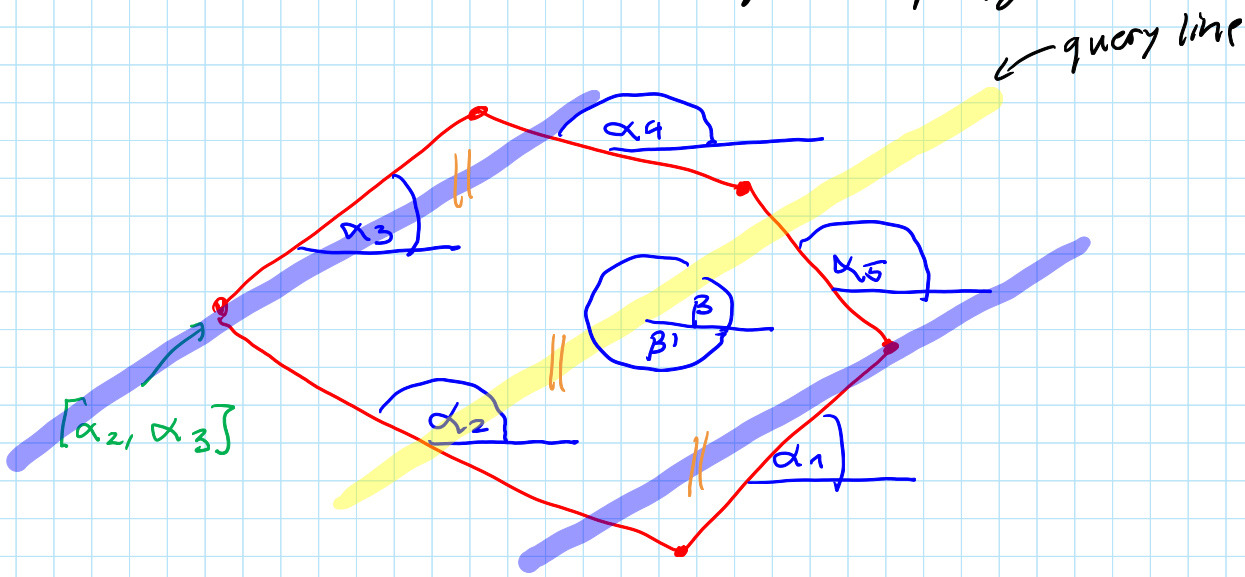
## 3.1. Point/Line Relative to convex polygon

- Inside/on/outside a convex polygon
  - Does a point lay inside a convex polygon?
  1. Do Locus-Approach on x-line
  2. If $q_x < x_0$ or $q_x > x_4$ => outside
  3. Otherwise check y-coordinates (Locus approach, again)
  - Space: O(n)
  - Preprocessing: O(2·n·log(n))
  - Query: O(2·log(n))



- A line hitting a convex polygon

· Does a line l intersect a given polygon C?




1. Compute for each edge e the ange $\alpha_e$ to x-axis
2. Store angles in sorted order
3. Each vertex gets an interval from the angles
4. We compute for our query line $\beta$ and $\beta'$
5. We search for the corresponding two vertices
6. From that two vertices we take the tangents parallel to the query line
7. Query line does not intersect, iff both tangent lines lie on same side

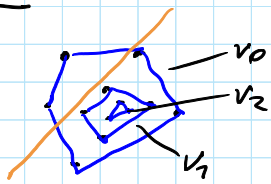
- Space: $O(n)$
- Preprocessing: $O(n \cdot \log(n))$
- Query: $O(\log(n))$

## 3.2. Line relative to point set

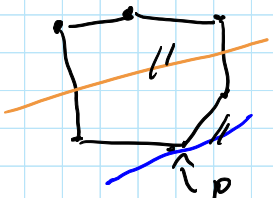
- Is line $q$ completely above/below points?
  ↳ Same as a line hitting a convex polygon

### Reporting the points below a query line

1. Use onion structure of points $P$
2. Start with $V_0$ and report all pts in $V_0$ that are below query line
3. Proceed to structure $V_1, V_2, \dots, V_n$ until either we have reached a $V_j$ that is completely above the query line or we have looked at all sets




- For finding points below $V_i$:
  1. Find vertex $p$ that has a parallel tangent to query line and has all other points of $V_i$ above

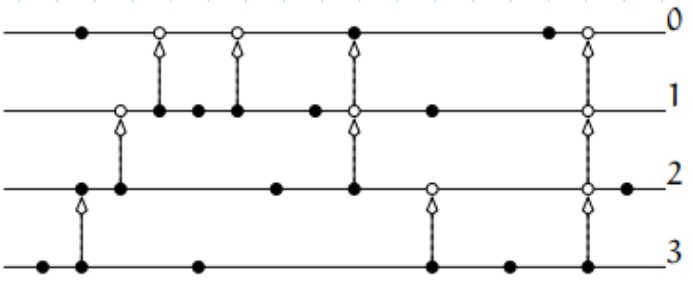



  2. If $p$ is above query-line $l$, we are done, because all other points are above $l_1$ too
  3. Else starting from $p$ we first move in clockwise order through $V_i$ and report them as "below $l$"
  4. If we end at $p$ again, we have to do it in counterclockwise fashion, again

- Space: $O(n)$
- Query: $O(k \cdot \log(n))$ ← Finding first point $p$ for each onion-ring
  ↑ (number of points reported below line
- Preprocessing: $O(n \cdot n \cdot \log(n))$
  ↑ At most $n$ convex polygons
  ↑ constructing one convex polygon

## Searching many lists

- We loose time by locating $\beta$ (=angle of query line) in an array with the angles of $V_i$ (=$S_i$) over and over again
- Fractional cascading
  - Obtain $\bar{S}_i$ by adding every other number from $\bar{S}_{i+1}$




↳ $S_3 = \bar{S}_3$

  - We can locate $\beta$ in $\bar{S}_{i+1}$ in constant time given we already located it in $\bar{S}_i$
- Query time: $O(k + \underbrace{\log(n)}_{\text{location in the first array } \bar{S}_1})$

---

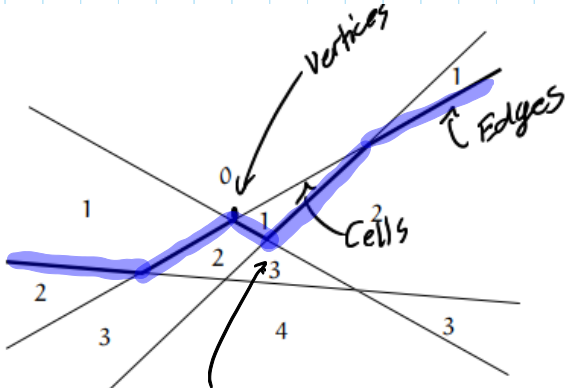
## Duality

- Lines and points in the plane are just different interpretations

---

## Counting the points below a query line

- Counting points below a query line
  ≙
  Counting lines above a query point } duality
- With use of fractional-tree-cascading:
  - leaves: numbers
  - nodes: x-coordinates
  - space: $O(n^2)$
  - Preprocessing: $O(n^2)$
  - Query: $O(\log(n))$



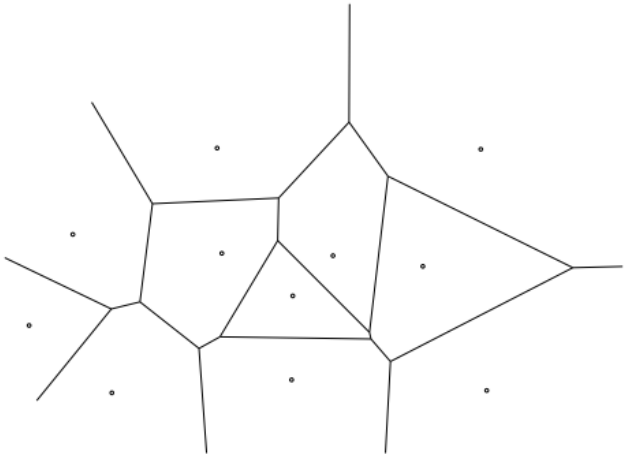

- Since each line has at most $n-1$ intersections with other lines, there are at most $n$ cages per line, in total $n^2$ edges
- There are at most $\binom{n}{2}$ vertices

Tells us there are 3 lines above
≙
3 points are below line

## 3.3. Planar Point Location

### Closest point in cell

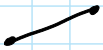
- Use locus approach to construct voronoi cells

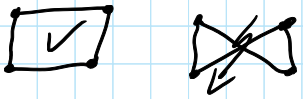


---

## 3.4. Trapezoidal Decomposition

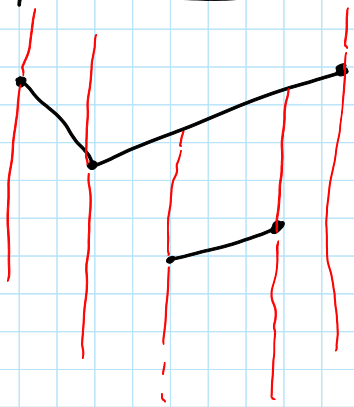
### The segments above setup

- Segment: 



- S := set of segments, they do not cross each other
- for every query point return segment that lays above that query point q => above(q)

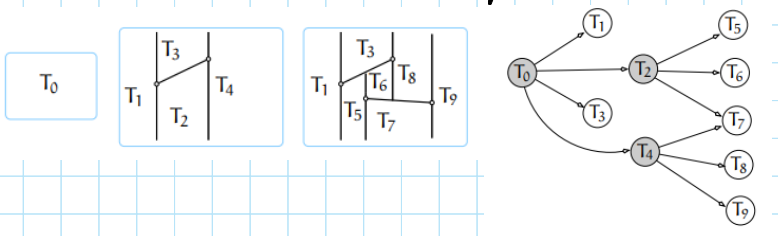
### Trapezoidal Decomposition

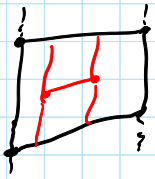


- I know in which trapezoid my query point lays, so I know which segment is above

- If S is empty, we have an infinity trapezoid

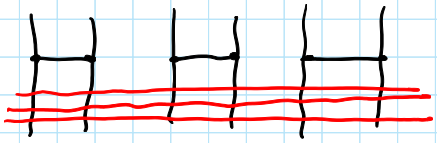
- History graph
  - When inserting the segments a history tree forms
  - Every time we add a new segment, we destroy at least one trapezoid and create at most 4 new trapezoids



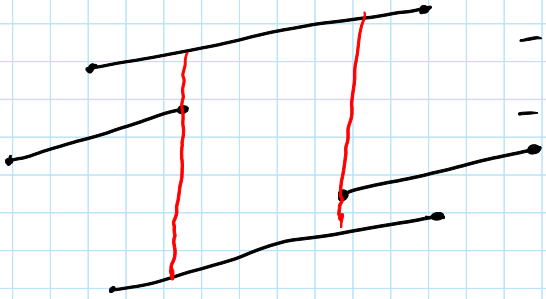
  - With a query point I can go down the tree, the leafes tells us, which segment is above query-point

- With n segments we have at most $n^2$ trapezoids



6 segments

- A trapezoid ist determined by at most 4 segments
  ↳ Each trapezoid disappears with prob. $\leq \frac{4}{i}$



  - 2 segments limit upper/lower boundry
  - 2 segments limit left/right boundry

- Space: $O(n)$ (= History graph has size $O(n)$)
- Preprocessing: $O(n \cdot \log(n))$
- Query: $O(\log(n))$ (= History graph has depth $\log(n)$)

# 4. Linear Programming

## 4.1. Basic setting

- Maximize [objective function]
- Subject to [constraints]
- Each inequality form a half plane

- All inequalities together form a convex polyhedron
- Objective function: $c^T x = c_1 x_1 + c_2 x_2 + \dots$
- Feasable solution: Every vector x that satisfy all of the constraints
- Basic feasable solution: Feasable solution x for which some inequalities hold with equality
  - ↳ All the corners $\binom{m}{n}$
- Optimal solution: Solution with maximum value
  - 1 optimal solution
  - Infinitely many optimal solutions
  - No optimal solution
    - No feasable solution at all
    - Objective function is unbounded from above
- Equational form: maximize $c^T x$
  subject to $Ax = b,\ x \geq 0$
- Maximize $-c^T x \Leftrightarrow$ Minimize $c^T x$
- Since there are only a finite number of basic feasable solutions, one of them gives the maximum value
- For every feasable solution there is a basic feasable solution with the same or larger objective value

- Least squares

  - $\sum_{i=1}^{n} \underbrace{|ax_i + \delta - y_i|}_{=e_i}$

  ↳ minimize $e_1 + e_2 + e_3 + \dots + e_n$

  subject to $e_i \geq ax_i + \delta - y_i$

  $e_i \geq -(ax_i + \delta - y_i)$ } for $i = 1, 2, \dots, n$

---

## 4.4. Bounds on solutions

- If linear program with rational coefficients has a feasable solution, then it also has a rational feasable solution

---

## 4.5. Duality and Farkas lemma

- The Farkas lemma provides easy certificates for unsolvability of systems of linear inequality

- Farkas Lemma I:

  - $Ax \leq b$ unsolvable $\Leftrightarrow$ $A^T y = 0$, $b^T y < 0$, $y \geq 0$

$y^T A x \leq y^T b \quad / \cdot y^T$

$0$ ↯ $< 0$

- Farkas Lemma II:

  - $Ax = b$, $x \geq 0$ unsolvable $\Leftrightarrow$ $A^T y \geq 0$, $b^T y < 0$

- Farkas Lemma III:

  - $Ax \leq b$, $x \geq 0$ unsolvable $\Leftrightarrow$ $A^T y \geq 0$, $b^T y < 0$, $y \geq 0$

- Equivalent form:

  - $Ax = b$, $x \geq 0$ unsolvable $\Leftrightarrow$ $A^T y \geq 0$, $b^T y = -N$

* $(y^T A)^T = A^T y$

- Geometrically this means that all the $a_j$ lie on one side of the hyperplane and $b$ lies on the other side
  ↳ we can't transform $Ax$ in such a way that $Ax$ represents $b$

- Transformations:

  · $Ax = b$, $x \geq 0$ unsolvable $\Rightarrow$ $A'x \leq b'$ unsolvable

  ↳ $\begin{bmatrix} A \\ -A \\ -I \end{bmatrix} \cdot x \leq \begin{bmatrix} b \\ -b \\ 0 \end{bmatrix}$ ($A'$, $x$, $b'$)

  $Ax \leq b \Rightarrow Ax \leq b$
  $-Ax \leq -b \Rightarrow Ax \geq b$ $\Big\} \Rightarrow Ax = b$
  $-Ix \leq 0 \Rightarrow x \geq 0 \Rightarrow x \geq 0$

  · Introduce $x \geq 0$:
    - Replace $x = (x^+ - x^-)$ with $x^+, x^- \geq 0$

  · Replace $Ax = b$ with $A'x \geq b'$:
    - $\begin{bmatrix} A \\ -A \end{bmatrix} x \geq \begin{bmatrix} b \\ -b \end{bmatrix}$ $\Rightarrow Ax \geq b \Rightarrow Ax \geq b$
      $\Rightarrow -Ax \geq -b \Rightarrow Ax \leq b$ $\Big\} \Rightarrow Ax = b$

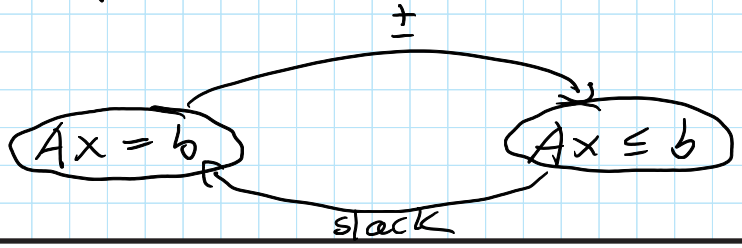
  · Replace $Ax \leq b$ with $Ax = b$:
    - Add slack variable $\varepsilon$
    - $Ax + \varepsilon = b$, $\varepsilon \geq 0$

  · Change min $\leftrightarrow$ max:

    $\min c^T x \iff \max -c^T x$

  · Remove $x \geq 0$

$$\begin{bmatrix} A \\ -I \end{bmatrix} \begin{bmatrix} x \end{bmatrix} \leq \begin{bmatrix} b \\ 0 \end{bmatrix}$$



## 4.5.2. The strong duality theorem

- Upper bound on best possible optimal value

| Maximize $c^T x$ | $\Leftrightarrow$ | Minimize $b^T y$ |
|---|---|---|
| subject to $Ax \overset{*}{\leq} b$ | | Subject to $A^T y \overset{*}{\geq} c$ |
| $x \geq 0$ | | $y \geq 0$ |
| Primal P | | Dual D |

- Weak duality theorem: $c^T x^* \leq b^T y^*$
- Strong duality theorem: $c^T x^* = b^T y^*$
  - ↳ Assuming P is feasable and bounded
- $c^T x \overset{*}{\leq} (A^T y)^T \cdot x = y^T (Ax) \overset{*}{\leq} y^T b = b^T y$

- Both (P) and (D) are infeasable
- Both (P) and (D) are feasable and bounded
- (P) is unbounded $\Rightarrow$ (D) is infeasable
- (D) is unbounded $\Rightarrow$ (P) is infeasable

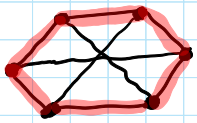
## 4.6. The ellipsoid method

- Relaxed feasability problem:
  - We use rational numbers instead of integer numbers
- Oracle
  - Black Box
  - Accepts a point $x$ as input
  - If $x$ is a solution of the system it returns yes
  - Else it returns one inequality that $x$ violates
- The ellipsoid method calls the separation oracle with the center $s_k = x_i$ of the generated ellipsoid, and it always uses the violated inequality returned by the oracle for determining the next ellipsoid.

---

## 4.7. Travelling Salesmen

- Tour $\hat{=}$ spanning cycle $\hat{=}$ Hamiltonian cycle
  - ↳ Connected graph
  - ↳ Every vertex is visited once



- Optimal tour $\hat{=}$ minimal cost tour $\sum_{e \in E_H} c_e$
- $\delta(v) :=$ edges that are connected to $v$
- Integer Constraints: $\sum_{e \in \delta(v)} x_e = 2$, for all $v \in V$
  - ↳ $x \in \{0,1\}^E$
  - ↳ Each node has only two edges —
  - $\sum_{e \in \delta(S)} x_e \geq 2^*$, for all $S \subseteq V$, $\emptyset \neq S \neq V$
    - ↳ Connected graph — {S}
- Relaxed / subtour constraints:
  - $\sum_{e \in \delta(v)} x_e = 2$, $v \in V$
  - $\sum_{e \in \delta(S)} x_e \geq 2$, $S \subseteq V$, $\emptyset \neq S \neq V$
  - $0 \leq x_e \leq 1$ $e \in E$

* Because it is a cycle

- $\underbrace{c^T \tilde{x}}_{\text{Relaxed}} \leq \underbrace{c^T x^*}_{\text{Integer}} = \underbrace{OPT_{Tour}}_{\text{problem}} \leq \underbrace{\frac{3}{2} \cdot c^T \tilde{x}}_{3/2 \text{ approximation}}$
  - Because we can use rational numbers $\Rightarrow$ more freedom
- Exponential number of constraints

---

## 4.8. Minimum Spanning Tree

- Spanning tree: · has exactly $(n-1)$ edges
  · it is connected
- Integer constraints: · $x \in \{0,1\}^E$
  · $\sum_{e \in E} x_e = n-1$
  · $\sum_{e \in \delta(S)} x_e \geq 1$, $S \subseteq V$, $\emptyset \neq S \neq V$ (#)

(#) Because it is a tree

- Relaxed/Loose Spanning Tree Constraints
  · $\sum_{e \in E} x_e = n-1$
  · $\sum_{e \in \delta(S)} x_e \geq 1$, $S \subseteq V$, $\emptyset \neq S \neq V$
  · $0 \leq x_e \leq 1$, $e \in E$
- $\underbrace{c^T \tilde{x}}_{\text{Loose}} \leq \underbrace{c^T x}_{\text{Integer}} \cdot 1/2$
- Spanning Tree: · has exactly $(n-1)$ edges
  · contains no cycle
  ↳ There is no set of $n$ vertices with more than $(n-1)$ edges
- Tight Spanning Tree Constraint: (with no cycle)
  · $\sum_{e \in E} x_e = n-1$
  · $\sum_{e \in E \cap \binom{S}{2}} x_e \leq |S|-1$, $S \subseteq V$, $\emptyset \neq S \neq V$ (*)
  · $0 \leq x_e \leq 1$, $e \in E$

(*) convex hull of integral solution

- Every basic feasible solution on tight-spanning-tree is integral $\Rightarrow$ Tight spanning tree cost $=$ Minimum spanning tree cost
  ↳ But we have exponentially many constr.

#-E: all edges in the graph
-$\binom{S}{2}$: all edges that can be formed inside the set S $\{\cap$: all edges inside the set S

## 4.9. Back to Subtour LP

- Does Hamiltonian cycle has a "perfect" relaxed LP that gives out the optimal integer solution?

↳ No, otherwise P=NP

---

# 5. Randomized Algorithms

## 5.1. Checking Matrix Multiplication

- $A \cdot B \overset{?}{=} C$

### 1. Solution: Just checking

1. choose $l$ entries $c_{ij}$
2. Verify $c_{ij} = \sum_{i=1}^{n} a_{ik} b_{kj}$ } $O(l \cdot n)$
- Probability that we capture error: $\frac{l}{n^2}$
- If we set $l = n \Rightarrow O(n^2)$ with $Pr = \frac{1}{n}$ ☹

### 2. Solution

1. Generate random n length vector $\in \{0,1\}^N$
2. Compute Cx $O(n^2)$
3. Compute $(A \cdot (B \cdot x))$ $O(n^2)$
4. Check, if both results match

- If $AB = C \Rightarrow$ Algorithm always answers yes
- If $AB \neq C \Rightarrow$ Algorithm still says yes with $Pr > 1/2$
- $D = C - AB$ (= has a nonzero element because of wrong matrix multiplication)
- $y = D \cdot x$ has with probability $1/2$ a nonzero element, because $x_i = 1$ with prob. $1/2$

↳ probability amplification $\left(\frac{1}{2}\right)^N$

## 5.2. Is a polynomial identically zero?

- Is a polynomial everywhere zero?
- Polynomial is given as black-box $f(x)$
- Univariante polynomials: only one variable (e.g. $x$)
  - Polynomial with degree $d$ has at most $d$ zeros
  - Evaluate polynomial at $d+1$ points
    - $(d+1)$ x $f(d_i)=0 \Rightarrow$ everywhere zero
- Multivariante polynomials: Many variables (e.g. $x+y^2$)
  - $3x^6y^2z$ has degree 9
- Schwartz-Zippel theorem
  - If I take a set $S$ of numbers and insert combinations of them into the polynomial with degree $d$ I don't get not many zeros.
  - A polynom $f(x_1,\dots,x_n)$ has at most $d\cdot|S|^{n-1}$ zeros
  - Error: $\frac{d}{|S|}$ ← Unluckily we spot all $d$ zero-points; Saying polynomial is everywhere zero, but it isn't
  - Proof: $n=1$: · Univariant polynom: $d\cdot|S|^0 = d\cdot 1 = d$ ✓
  - Proof: $n>1$: · Extract $x_1$ out of the polynomial

$$\cdot f(x_1,x_2,x_3,\dots x_n) = x_1^k(\dots) + x_1(\dots) + x_1^0(\dots)$$

($p_k$, $p_1$, $p_0$)

$$= \sum_{i=0}^{k} x_1^i \cdot p_i(x_2,\dots,x_n)$$

(degree $i$; degree $d-i$)

· If we fix $x_i$, we have with $x_1$ at most $k$ zeros. We don't care about the rest of the polynom $0\cdot p_i(\dots)$
↳ $\le k$ zeros in $|S|^{n-1}$ different ways

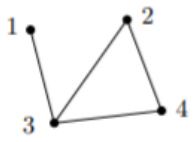
## 5.3. Testing for bipartite matching

- Perfect matching: All vertices are covered
- Permutation matrix: $b_{i,j} = 1$, if there is an edge
- $\pi$ := Permutation of selected edges
  $= \{\{u_1, v_{\pi(1)}\}, \{u_2, v_{\pi(2)}\}, \ldots\}$
- $S_n$ = Set of all permutations with $n$ elements
- $per(B) = \sum_{\pi \in S_n} b_{1,\pi(1)} \cdot b_{2,\pi(2)} \cdot \ldots$
  - ↳ Tells us the exact number of perfect matchings, since each $b_{i,\pi[i]}$ must be 1
- $per(B)$ is the same as the determinant, but the determinant has $\pm$ signs in front, that can cancel each other out $\Rightarrow$ det is $\in P$
- $per(B)$ is #P-hard
- $det(B)$ can be solved in polynomial time
- $det(B) = 0$ even if $per(B) \neq 0$, if $\pm$ signs cancel each other out
  - ↳ Replace 1s with $x_{ij}$ variable to avoid cancellation
- Now we have as $det(A)$ a polynomial with degree $n$
- $G$ has a perfect matching, if that polynomial isn't zero everywhere
  - ↳ Each monomial of $det(A)$ with nonzero coefficients represent a perfect matching
- That polynomial can't be computed, since it can have exponentially many terms

- We can use Schwartz-Zippel
  - Set $|S| = 2n$ $\Rightarrow$ Detecting $\det(A)$ is nonzero is $\frac{n}{2n} = \frac{1}{2}$
  - Choose prime number $p$, $2n \leq p \leq 4n$
  - Regard $\det(A)$ as a finite field $GF(p)$ (mod $p$)

## 5.4. Perfect matchings in general graphs

- By the permutation we get a directed graph
- Therefore we get cycles as well
- The permutation has the sign $(-1)^{\#\text{even cycles}}$
- Tutte Matrix:




$$A = \begin{pmatrix} 0 & 0 & x_{13} & 0 \\ 0 & 0 & x_{23} & x_{24} \\ -x_{13} & -x_{23} & 0 & x_{34} \\ 0 & -x_{24} & -x_{34} & 0 \end{pmatrix}$$

$-x_{ij}$, $+x_{ij}$ } cancells odd cycles out

- The $\det(A)$-polynomial is non-zero, if there exists one perfect matching
- Permutation $\pi$ is important, if $a(\pi) \neq 0$

# 5.6. Counting perfect matchings in planar graphs

- $a_{ij} = \begin{cases} +1 & \text{if } \{i,j\} \in E \\ -1 & \text{if } \{j,i\} \in E \\ 0 & \text{otherwise} \end{cases}$ $\Big\}$ $a_{ij} = -a_{ji}$ $\Rightarrow$ Like Tutte-matrix

- **A lower Bound for the number of perfect Matchings**
  - $\det(A(G)) \leq pm(G)^2$ $/\sqrt{\ }$
    $\sqrt{\det(A(G))} \leq pm(G)$

- **Nice cycles in $G$ and oddly oriented cycles in $\vec{G}$**
  - Oddly oriented: If we traverse a cycle in $\vec{G}$ in some orientation and the number of arcs going towards is odd
  - Nice cycle: Remove vertices of cycle $\Rightarrow$ still PM.

- If every nice cycle is oddly oriented: $\det(A(\vec{G})) = pm(G)^2$
  (every nice cycle is oddly oriented: Pfaffian Orientation; $\vec{G}$)

- Every planar graph has a pfaffian orientation that can be calculated in linear time
- Number of perfect matchings in a planar graph can be determined in polynomial time
- Number of perfect matchings: $2^{\#\text{cycles}} \geq 4$
- The pfaffian orientation let us run for any cycle $C$ along it in the graph $\vec{G}$ and the number of times we move against an edge of $\vec{G}$ is odd therefore that cycle gives us a $-1$
- Since in the perfect matching $\Pi$, all cycles are even, we always get $-1$ per cycle
  $\hookrightarrow \operatorname{sign}(\Pi) = (-1)^{\#\text{ of even cycles in } \Pi}$

- If we add a $-$ sign in front of every cycle, we only get $+1$s $\longrightarrow pm(G) = \sqrt{\det(\vec{G})}$

## 6.1. Adding two numbers

### Carry-Ripple

- $a = a_n, a_{n-1}, \dots, a_1$
- $b = b_n, b_{n-1}, \dots, b_1$

} add a+b in binary

↳ $s_1 = a_1 + b_2 \bmod 2$

$c_1 = 1$ iff $a_1 + b_2 \geq 2$

$s_2 = a_2 + b_2 + s_1 \bmod 2$

$c_2 = 1$ iff $a_2 + b_2 + c_1 \geq 2$

- There is a dependency to the previous computation

### Carry-look-ahead

- $a_i = b_i = 1 \Rightarrow c_i = 1$ (carry bit is generated)
- $a_i = b_i = 0 \Rightarrow c_i = 0$ (carry bit is killed)
- $a_i \neq b_i \Longrightarrow c_i = c_{i-1}$ (carry bit is propagated)

| $x_i$ \ $x_{i-1}$ | k | g | p |
|---|---|---|---|
| k | k | k | k |
| g | g | g | g |
| p | k | g | p |

$x_i$ = determined by $a_i$ and $b_i$

$y_i$ = determined by $a_i$, $b_i$ and $c_i$

- $y_i = k \Rightarrow c_i = 0$
- $y_i = g \Rightarrow c_i = 1$

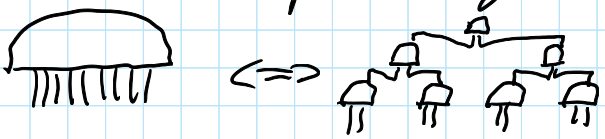
} we never can have $y_i = p$

- $x_i$ can be calculated in parallel
- $y_i$ can be calculated using a binary tree
- $O(\log(n))$ computation with $O(n)$ processors
- $O(n)$ computation in total

## 6.2.1 Circuits

- $NC(i)$ set of all decision problems
  - $poly(n)$ gates
  - $O(\log^i(n))$ depth ($=n$ is input size)
- $AC(i)$
  - $poly(n)$ gates
  - $O(\log^i(n))$ depth with unbounded fan-in (inputs in circuit)
- $NC(i) \subseteq AC(i) \subseteq NC(i+1)$

  Use $O(\log(n))$ depth binary trees of bounded fan-in gates to replace unbounded fan-in gates



---

## 6.2.2 Parallel Random Access Machines (=PRAM)

- PRAM processors with shared memory
  - Can write/read locally or globally
- Exclusive / Concurrent Read / Write
- $CRCW(k)$:
  - $poly(n)$ processors
  - $O(\log^k n)$ time
- $CRCW(k) \subseteq EREW(k+1)$

## 6.2.3. Some basic Problems

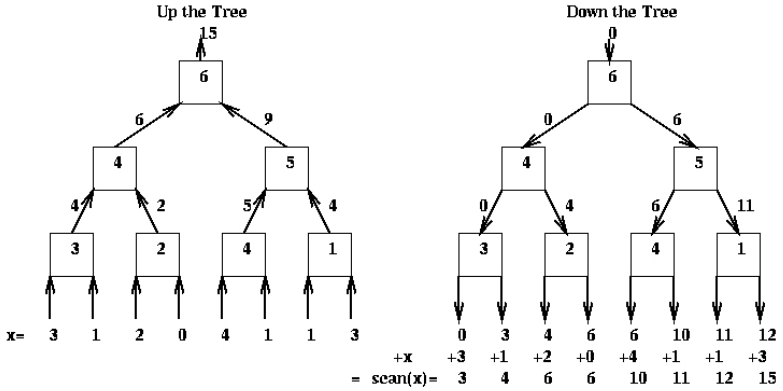
### - Maximum of n numbers

- Set register with n elements = 0
- Use $\binom{n}{2}$ processors, each processor responsible to compare two elements in the array
- If $i < j$, write to the $i$th register = 1
- Use n processors to check, which element ≠ 1

↳ O(1) time steps in the CRCW-model

### - Parallel Prefix

- $B[j] = \sum_{i=1}^{j} A[i]$
- O(log(n)) time
- O(n) processors
- O(n) time in total




### - List Ranking

- Linked list as input:
  - content array c
    ↳ c(i) = value of element i
  - successor pointer array s
    ↳ s(i) = successor of i
- Pointer Jumping:
  - In parallel, set $c(i) = c(i) + c(s(i))$
  - In parallel, set $s(i) = s(s(i))$

} log(n) times

- At end, c(i) is the sum from i to end of list
- O(log(n)) time
- O(n·log(n)) work (for each of the element n we do log(n) times pointer jumping)

## 6.2.4. Work efficient parallel algorithms

- Depth: · Total amount of rounds
  · As small as possible => more parallelism
- Work: · Summation of all computations performed over all the rounds
- Brent's Principle: · Algorithm can be run in time $\frac{x}{p} + t$ (work / #CPUs + depth)

---

## 6.3. Lists and Trees

### 6.3.1. List ranking

- Improve $O(n \cdot \log(n))$ work => $O(n)$ work
- An element of a linked list doesn't know if its position is even/odd

1. Get a set S from the list, such that no two elements of its are consecutive:
   1. Flip coin for each element (=Head/Tail)
   2. Element in S, if element itself holds a head coin and its successor a tail coin
      ↳ $E[|S|] \geq n/8$ ($\frac{1}{2} \cdot \frac{1}{2} \cdot \frac{1}{2}$) (← Head, Tail, Amounts of elements left)
2. Create a new list out of S to get a new level in the binary tree
   1. Number the items in S, set all items in S = 1 and all others to 0
   2. Compute parallel prefix-sum
      ↳ Ignoring order in linked list
   3. Build new linked list by finding successor in S
3. Repeat recursively until we have less than $n/\log(n)$ elements in S

- For one level: $O(n)$ work
  $O(\log(n))$ depth } parallel prefix for creating list
- Tree depth: $O(\log(\log(n)))$ levels to reach size $n/\log(n)$
- Total: $O(n)$ work
  $O(\log(n) \cdot \log(\log(n)))$ depth
  ($\log(n)$ per level, $\log(\log(n))$ # levels)

---

## 6.3.2. Euler Tour Technique (=Rooting tree / preorder)

- Given a tree $t$

### Rooting the tree and determining the parents

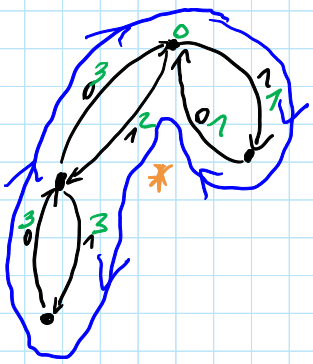
- Additional input: root node $r$
- Compute for each $v \in V$: $parent(v_i) = v_j$

1. $T \to T'$ where each edge is — $\Leftrightarrow$ 
2. Compute Eulerian Tour (cycle with each vertex once visited)
2. Break the Tour to a path by removing last incoming arc to root $r$
3. Generate numbering $\{1, \dots, 2n-2\}$ by list-ranking
4. For each vertex $v$: number$\{u,v\} <$ number$\{v,u\}$
   ↳ parent$(v) = u$, because we first traverse $u$

### Computing Pre-Order Numbering

$v$, left$(v)$, right$(v)$

1. Identify parents
2. Set weights
3. Compute parallel prefix
4. Set pre(root) = 0
5. Set pre$(v)$ = prefix sum of incoming arc

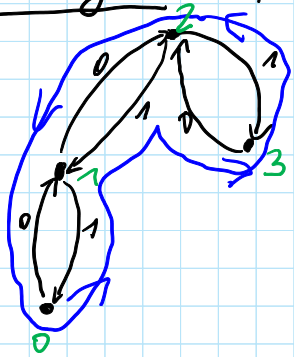



* Note, we don't care about left/right here

## Computing In-order Numbering

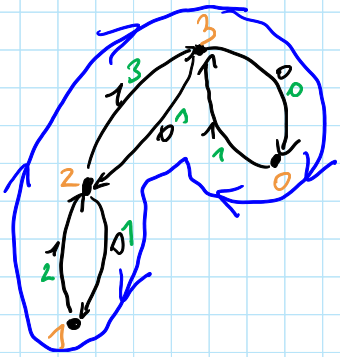
left(v), v, right(v)

- Go against eulerian path



## Computing Post-order Numbering

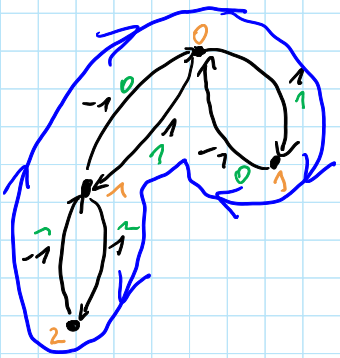
left(v), right(v), v



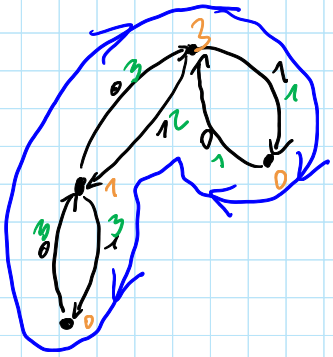
1. Compute parallel-prefix
2. post(v) = <v, parent(v)> − 1
   post(r) = n − 1

## Depth of Graph

- depth(r) = 0
- depth(v) = <parent(v), v>



## Number of descendants

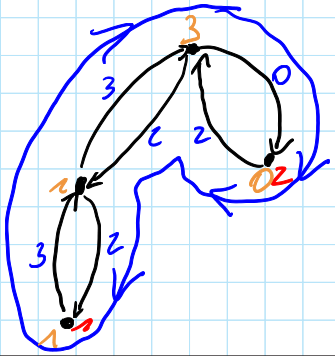


des(r) = n − 1

des(v) = <v, parent(v)> − <parent(v), v>

## Number of leaves

1. Give each leaf a number
2. Do prefix sum
3. sl(v) = <v, parent(v)> − <parent(v), v>



# 6.4. Merging and Sorting

## 6.4.1. Merge Sort

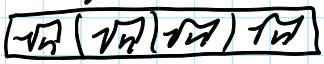
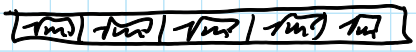
- Sorted arrays $A[1,\dots,n/2]$, $B[1,\dots,n/2]$ $\rightarrow C[1,\dots,n]$

### Basic Merging algorithm

1. $C[i+j] = A[i]$
   ↳ $j$ items in $B$ are smaller than $A[i]$
2. $C[i+j] = B[i]$

- Depth: $O(\log(n))$ (= Binary search)
- Work: $O(n \cdot \log(n))$ (= Binary search for each element)

### Improved Merging algorithm

- $A[1,\dots,n] \Rightarrow$ 
- $B[1,\dots,m] \Rightarrow$ 
- Fenceposts: $\cdot A[\alpha_i] \Rightarrow \alpha_i = (i-1)\cdot\sqrt{n}$
  $\cdot B[\beta_i] \Rightarrow \beta_i = (i-1)\cdot\sqrt{m}$
- compare fenceposts such that $B[\beta_j] \le A[\alpha_i] \le B[\beta_{j+1}]$
- Now we have the exact spot where $A[\alpha_i]$ lays in $B$
- We can use these $\alpha_i$ fenceposts to break the problem into $\sqrt{n}$ many subproblems

## 6.4.2. Quicksort $O(n \cdot \log(n))$

### Basic Quick sort

1. Each processor compares one element with the pivot and writes 1, if it is smaller than the pivot
2. Use prefix sum to write smaller elements in a new array

- Level $k$ is sucess, if $i \in [\frac{1}{4} \cdot n, \frac{3}{4} n]$
  - ↳ $Pr[x_i = \text{success}] = 1/2$
- Once we have $\log_{4/3}(n)$ success in one branch, size is 1
  - ↳ $O(\log(n))$ depth with probability $1 - 1/n^3$

- Depth: $O(\log(n) \cdot \log(n))$ (prefix sum, Levels)
- Work: $O(n \cdot \log(n))$ (prefix sum, Levels)

### Improved Quick Sort

1. Pick $\sqrt{n}$ pivots at random
2. Sort the pivots
   - For each pivot sum up together, how many pivots are smaller => Index
3. Put each element between two pivots
4. Recurse on each subproblem between two pivots

- Depth: $O(\log(n))$
- Work: $O(n \cdot \log(n))$

## 6.5. Connected components

- Component identifier: $D(v) = D(u)$ if they are connected

### 6.5.1. Basic deterministic algorithm

1. Each vertex is its own fragment
2. Merge fragments together

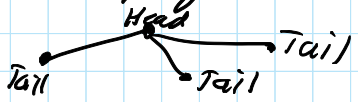
- Each fragment is maintained as a star
  - ↳ Easy to check, if $d(u)=d(v)$
- $D(root) = root$
- Merging two fragments
  - Identify minimum root node $r_k$ that is in same component as $r_j$
    - ↳ $F_j$ proposes to merge with $F_k$
  - If there is no more edge: fragment $\hat{=}$ component
    - ↳ self-loop as proposal
  - Transforming back to stars
    - For each root remove self loop in its fragment and set $D(r_j) = r_k$
    - Do pointer jumping for every node in fragment $r_j$

- $O(\underbrace{\log(n)}_{\text{depth}} \cdot \underbrace{\log(n)}_{\text{pointer jumping}})$ depth → if they form long chains

- $O(\underbrace{m}_{\#\text{fragments}} \cdot \log(n) \cdot \underbrace{\log(n)}_{\text{depth}})$ work

## 6.8.2. Randomized algorithm

1. Each CPU tries to propose an edge, because we are in CRCW-model, one fragment will be proposed
2. Overcoming long chains
   - 2.1. Toss Head/Tail for each fragment
   - 2.2. Only accept proposal




↳ only head fragments can have other merge it ⇒ only one pointer-jump

- $O(\log(n))$ depth (= No pointer jumping)
- $O(m \cdot \log(n))$ work
- $O(\log(n))$ of merging with high prob. still enough:
  1. Each fragment has a proposed edge
  2. Proposed edge gets merged with Pr $1/4$ $(= 1/2 \cdot 1/2)$
  3. At least $n_i/4$ fragments merge
  4. In expectation fragments are reduced by at least $\frac{1}{8} \cdot n_i$, because two fragments can propose to each other
  5. $E[n_{i+1}] \leq 7/8 \cdot n_i$
  6. If we do that $\log(n)$ times: $(7/8)^{\log(n)} \cdot n \leq O(1/n^3)$

## 6.6. (Bipartite) Perfect Matching

- What is the perfect matching
  - If there is only one matching, do checking in parallel by removing each edge and checking, if there is still a perfect matching
- If there are many matchings in graph?
  - Find edge weights
  - Use minimum-edge-weight perfect matching
    ↳ It is unique
- Isolation Lemma
  - $E$ := $m$ elements (= edges)
  - $F$ := $N$ subsets of $E$ (= perfect matchings)
  - Assign each element a random weight $w(e) \in \{1, 2, \dots, 2m\}$
  - Weight of subset $\delta_i = \sum_{e \in \delta_i} w(e)$
  - With probability $1/2$ there is a unique minimum set
  - **Proof:**
    - 2 different minimum-weight sets
    - $e$ is ambiguous, if $e$ is in one of the two minimum sets
      ↳ This only happens with probability $1/2$
    1. Fix all weights only the one of $e$
    2. $W$ := minimum weight set including $e$
    3. $\bar{W}$ := minimum weight set not including $e$
    4. $d = |W - \bar{W}|$
    5. If $w(e) = d$, there are two minimum sets

6. $P_r[w(e) = d] = 1/em$

7. All elements $m/em = 1/2$

---

## 6.7. Modern/Massively parallel computation (=MPC)

- PRAM: fine grained
  - ↳ If one processor is slow => whole system is slow
  - ↳ communication overhead
- MPC: · Coarse grained
  - · M machines with memory sizes S
  - · Each machine gets some chunk of data
  - · In one round, each machine can do some computation and communicating with others
- If the chunk of data is 1 => PRAM model

---

## 6.8. MPC: Sorting

- Each machine gets a chunk of the array
- At the end the machine should know the rank of each element of the chunk

1. Each machine marks some elements as pivots
2. All pivots are gathered in one machine and then sorted
3. Machine broadcasts sorted list to all other machines
4. Now the leader tells each machine on what subproblem it should work

## 6.9. Connected Components

- Each edge is stored in some machine

1. Pick some machines
2. Each machine sends its edges randomly to the other machines
3. Each machine computes the maximal forest
4. Remove the edges that are not in the forest
5. Do this recursively
6. Until size is good for one machine, send all maximum forests to that machine.

---

## 6.10. MPC: Maximal Matching

- Maximal matching: I can't add any additional edges
- Maximum matching: Matching with maximum cardinality

1. Mark some edges
2. Move marked edges to one machine and compute maximal matching among them
3. Remove vertices from matched edges
4. Do it again