Background

January 8, 2023 6:15 PM

Growth of functions - Asymptotics

- Notations: $O, \Omega, \theta, o, \omega$.
- O-nonation:
  - $O(g(n)) = \{f(n) : \exists c > 0, n_0, \text{s.t. } 0 \le f(n) \le cg(n), \forall n \ge n_0\}$.
  - $g(n)$ is an upper bound of $f(n)$, $g(n)$ bounds $f(n)$ from above.
  - E.g.
    - $13n + 7 \in O(n)$, since $13n + 7 \le 14n$ for $n \ge n_0 = 7$.
    - $\frac{1}{2}n^2 - 3n \in O(n^2)$, since $\frac{1}{2}n^2 - 3n \le cn^2$ holds for $c \ge \frac{1}{2}$.
    - $n! = 1 \cdot 2 \cdots n \le n \cdot n \cdots n = n^n \in O(n^n)$.
    - $\log n! \in O(n \log n)$, since $n! \in O(n^n)$.
    - $2^{n+1} \in O(2^n)$, since $2^{n+1} = 2 \cdot 2^n$.
    - $2^{2n} \notin O(2^n)$.
      - Assume $c, n_0 > 0$ exists, $2^{2n} \le c \times 2^n$, $c \ge 2^n$ for all $n \ge n_0$.
- Ω-notation:
  - $\Omega(g(n)) = \{f(n) : \exists c > 0, n_0, \text{s.t. } 0 \le cg(n) \le f(n), \forall n \ge n_0\}$.
  - e.g.
    - $f(n) = 1 + 2 + \cdots + n \ge \left\lceil \frac{n}{2} \right\rceil + \left\lceil \frac{n}{2} + 1 \right\rceil + \cdots + n$,
      $\ge \left\lceil \frac{n}{2} \right\rceil + \left\lceil \frac{n}{2} \right\rceil + \cdots + \left\lceil \frac{n}{2} \right\rceil = \left\lceil \frac{n}{2} \right\rceil \left(n - \left\lceil \frac{n}{2} \right\rceil + 1\right) \ge \left\lceil \frac{n}{2} \right\rceil \left\lceil \frac{n}{2} \right\rceil = \frac{n^2}{4} \in \Omega(n^2)$.
    - $\frac{1}{2}n^2 - 3n \in \Omega(n^2)$.
      - Take $n_0 = 7, c = \frac{1}{14}$.
- θ-notation
  - $\theta(g(n)) = \{f(n) : \exists c_1, c_2 > 0, n_0, \text{s.t. } 0 \le c_1 g(n) \le f(n) \le c_2 g(n), \forall n \ge n_0\}$.
  - Thm: $f(n) = \theta(n)$ iff $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.
  - e.g.
    - $f(n) = 1 + 2 + \cdots + n = \frac{n(n+1)}{2} \in \theta(n^2)$.
    - $f(n) = \sum_{i=1}^{n} i^k \in \theta(n^{k+1})$.
      - $f(n) \in O(n^{k+1})$ since $f(n) = \sum i^k \le \sum n^k = n \cdot n^k \in O(n^{k+1})$.
      - $f(n) \in \Omega(n^{k+1})$. Consider $2f(n) = \sum i^k + \sum (n - i + 1)^k = \sum i^k + (n - i + 1)^k$,
        $\ge \sum \left\lceil \frac{n}{2} \right\rceil^k = \frac{n^{k+1}}{2^k}$, so $f(n) \ge \frac{n^{k+1}}{2^{k+1}}$, $f(n) \in \Omega(n^{k+1})$.
    - $(n + a)^b = \theta(n^b)$.
      - Need to find $c_1, c_2, n_0 > 0$ such that $0 \le c_1 n^b \le (n + a)^b \le c_2 n^b$ for all $n \ge n_0$.
      - $n + a \le n + |a| \le 2n$ if $n \ge |a|$.
      - Also, $n + a \ge n - |a| \ge \frac{1}{2}n$, if $n \ge 2|a|$.
      - We get $0 \le \frac{1}{2}n \le n + a \le 2n$.
      - Raise to power of b, we get $0 \le \left(\frac{1}{2}\right)^b n^b \le (n + a)^b \le 2^b n^b$.
      - $c_1 = \left(\frac{1}{2}\right)^b, c_2 = 2^b, n_0 = 2|a|$.
- o-notation:
  - $o(g(n)) = \{f(n) : \forall c > 0, \exists n_0 > 0, \text{s.t. } 0 \le f(n) < cg(n), \forall n \ge n_0\}$.
  - Equivalently, $\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$.
  - $n^{1.9} \in o(n^2)$, $n^2 \notin o(n^2)$.
- ω-notation:
  - $\omega(g(n)) = \{f(n) : \forall c > 0, \exists n_0 > 0, \text{s.t. } 0 \le cg(n) < f(n), \forall n \ge n_0\}$.

- $\circ$ Equivalently, $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$.
- $\circ$ $n^{2.1} \in \omega(n^2)$, $n^2 \notin \omega(n^2)$.
- Properties
  - Transitivity: $f(n) = \theta(g(n))$, $g(n) = \theta(h(n))$, then $f(n) = \theta(g(n))$.
    - True for O, Ω, ω, o.
  - Reflexivity: $f(n) = \theta(f(n))$.
    - True for O, Ω.
  - Symmetry: $f(n) = \theta(g(n))$ iff $g(n) = \theta(f(n))$.
  - Transpose symmetry:
    - $f(n) = O(g(n))$ iff $g(n) = \Omega(f(n))$.
    - $f(n) = o(g(n))$ iff $g(n) = \omega(f(n))$.
- Theorem: if $f(n) \in O(f'(n))$, $g(n) \in O(g'(n))$, then
  - $f(n)g(n) \in O(f'(n)g'(n))$.
  - $f(n) + g(n) \in O(\max\{f'(n), g'(n)\})$.

Polynomial-bounded functions

- A function $f(n)$ is polynomial bouned if $f(n) = O(n^k)$.
- $f(n) = O(n^k)$ iff $\log(f(n)) = O(\log n)$.
  - Proof: (⇒) Assume $f(n) = O(n^k)$.
    Then $f(n) \le c_1 n^k$, for $n \ge n_0$.
    $\log(f(n)) \le \log(c_1 n^k) = \log c_1 + k \log n \le c_2 \log n$ for constant c_2.
    (⇐) assume $\log(f(n)) = O(\log n)$.
    Then $\log(f(n)) \le c_3 \log n$.
    $\log(f(n)) \le \log(n^{c_3})$.
    $f(n) \le n^{c_3}$.

Limit method

- $\lim \frac{f(n)}{g(n)} = 0 \Rightarrow f(n) = O(g(n))$.
- $\lim \frac{f(n)}{g(n)} = c \in (0, \infty) \Rightarrow f(n) = \theta(g(n))$.
- $\lim \frac{f(n)}{g(n)} = \infty \Rightarrow f(n) = \Omega(g(n))$.
- More precisely
  - $\lim \frac{f(n)}{g(n)} = 0 \Rightarrow f(n) = o(g(n))$.
  - $\lim \frac{f(n)}{g(n)} = c \in [0, \infty) \Rightarrow f(n) = O(g(n))$.
  - $\lim \frac{f(n)}{g(n)} = c \in (0, \infty) \Rightarrow f(n) = \theta(g(n))$.
  - $\lim \frac{f(n)}{g(n)} = c \in (0, \infty] \Rightarrow f(n) = \Omega(g(n))$.
  - $\lim \frac{f(n)}{g(n)} = \infty \Rightarrow f(n) = \omega(g(n))$.
- L'Hopital's rule
  - If $\lim_{x\to c} f(x) = \lim_{x\to c} g(x) = 0$ or $\pm\infty$.
  - $\lim_{x\to c} \frac{f(x)}{g(x)} = \lim_{x\to c} \frac{f'(x)}{g'(x)}$.

Log of limits and limits of logs

- $\log(\lim_{x\to c} g(x)) = \lim_{x\to c} \log(g(x))$.
- e.g. $f(n) = 2^{n^2}$, $g(n) = 3^n$.
  - $\log\left(\lim_{n\to\infty} \frac{f(n)}{g(n)}\right) = \lim_{n\to\infty} \log\left(\frac{f(n)}{g(n)}\right) = \lim_{n\to\infty}\left(\log 2^{n^2} - \log 3^n\right)$.
  - $= \lim_{n\to\infty}(n^2 - n\log 3) = \infty$.
  - Then $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$, so $f(n) = \Omega(g(n))$.
- e.g. $f(n) = 2^{n+1}$, $g(n) = 4^n$.

- $\log\left(\lim_{n\to\infty}\frac{f(n)}{g(n)}\right) = \lim_{n\to\infty}\log\left(\frac{f(n)}{g(n)}\right) = \lim_{n\to\infty}(\log 2^{n+1} - \log 4^n)$.
  - $= \lim_{n\to\infty} n + 1 - 2n = -\infty$.
  - Then $\lim_{n\to\infty}\frac{f(n)}{g(n)} = 0$, so $f(n) = O(g(n))$.

$1 \ll \log^* n \ll \log^{(k)} n \ll \log^{\mathrm{k}} n \ll n^{\frac{1}{2}} \ll a^{\log n} \ll n \ll n\log n \ll n^{1+c} \ll n^2 \ll n^k \ll c^n \ll n!$.

- $2^n \ll 10^n$.

e.g.

- $\log(n!) \ll n\left(\log n\right)^2$.
  - $\log n! = O(n\log n)$.
- $n^3 \ll n^{\log\log n}$.
  - Log both sides and take the limit.
- $n^{\log\log n} \equiv \left(\log n\right)^{\log n}$ since $x^{\log y} = y^{\log x}$.
- $\log x \ll \log y \Leftrightarrow x \ll y$.
- $\log_{\log n} n \ll \log(n\log n) \ll \left(\log\log n\right)^{\log\log n} \ll 2^{\log n} \ll \left(\sqrt{2}\right)^{\log n} \ll n$.
- $2^{\log n} \ll \left(\log n\right)^{\log n}$.

Summations

- $\sum_{k=1}^n (ca_k + b_k) = c\ \sum_{k=1}^n a_k + \sum_{k=1}^n b_k$.
- $\sum_{k=1}^n \theta(f(k)) = \theta(\sum_{k=1}^n f(k))$.
- $\sum_{k=1}^n k = \frac{n(n+1)}{2} = \theta(n^2)$.
  - $\sum_{k=1}^n (a + bk) = \theta(n^2)$.
- $\sum_{k=0}^n k^2 = \frac{n(n+1)(2n+1)}{6}$.
- $\sum_{k=0}^n k^3 = \frac{n^2(n+1)^2}{4}$.
- For $x \neq 1$, $\sum_{k=0}^n x^k = \frac{x^{n+1}-1}{x-1}$.
  - For $|x| < 1$, $\sum x^k = \frac{1}{1-x}$.
  - Differentiation gives $\sum kx^k = \frac{x}{(1-x)^2}$.
- $\sum_{k=1}^n \frac{1}{k} = \ln n + O(1)$.
  - $\ln(n+1) \leq \sum_{k=1}^n \frac{1}{k} \leq \ln n + 1$.
- Telescoping
  - $\sum_{k=1}^n (a_k - a_{k-1}) = a_n - a_0$.
  - $\sum_{k=1}^{n-1} \frac{1}{k(k+1)} = 1 - \frac{1}{n}$.
- $\log(\prod_{k=1}^n a_k) = \sum_{k=1}^n \log a_k$.

Logarithm

- $\log^{\mathrm{k}} n = \left(\log n\right)^k$.
- $\log^{(k)} n = \log\log \dots \log n$.
- $\log^* n = \min\left\{i \geq 0 : \log^{(i)} n \leq 1\right\}$.
- e.g.
  - $\log^* 2 = 1$.
  - $\log^* 4 = 2$.
  - $\log^* 256 = \log^* 8 + 1 = \log^* 3 + 2 = 4$.
  - $\log^* 2^{256} = 5$.
  - $\log^* n$ is the slowest besides constant.

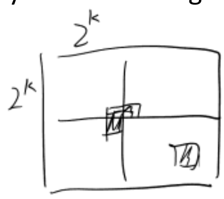
Stirling approximation: $\sqrt{2\pi n}\left(\frac{n}{e}\right)^n \le n! \le \sqrt{2\pi n}\left(\frac{n}{e}\right)^{n+\frac{1}{12n}} \Rightarrow \log n! = \theta(n \log n)$.

Proofs

Induction

- Predicates $P(n)$ (proposition).
  - e.g. $P(n) : n < 2^n, \forall n$.
- $[P(base) \wedge \forall n(P(n) \Rightarrow P(n+1))] \Rightarrow \forall n\, P(n)$.
- e.g. prove the sum of first n odd positive integers is n^2.
  - Base: for $n = 1$, sum is $1 = 1^2$.
  - Induction Hypothesis: assume $1 + 3 + 5 + \cdots + 2n - 1 = n^2$.
  - Induction step: $1 + 3 + \cdots + 2n - 1 + 2n + 1 = n^2 + 2n + 1 = (n+1)^2$.
- e.g. Show that every $2^n \times 2^n$ board with single tile removed can be tiled with L-shaped 3 piece segment of tiles.
  - Base: 2×2

  - IH: suppose for some $n = k \ge 1$, $2^k \times 2^k$ board with single tile removed can be tiled with L-shape segment of tiles.
  - Induction: when $n = k + 1$, split into 4 $2^k \times 2^k$ boards. For each of them, can be tiled by I.H. Center segment can be tiled by single L-shaped piece.



Strong induction

- $[P(1) \wedge \forall n(P(1) \wedge \cdots P(n)) \Rightarrow P(n+1)] \Rightarrow \forall n, P(n)$.
- e.g. every integer $n \ge 2$ can be written as a product of primes
  - Base: $P(2)$ is true since 2 is a product of itself.
  - IH: Assume $P(2), P(3), \ldots, P(n)$ true for some $n > 2$.
  - IS: for $n + 1$.
    - If $n + 1$ is prime, then done.
    - If $n + 1$ is composite $n + 1 = a \cdot b$ with $a, b < n + 1$.
      Then by IH, $a = p_1 p_2 \ldots p_i$, $b = q_1 q_2 \ldots q_j$, with p_k, q_k primes.
      $n + 1$ is then a product of primes, then $P(n+1)$ is true.

Contradiction

- To prove $P(n)$, assume by contradiction, $\neg P(n)$ is true.
- $\neg P(n) \Rightarrow$ some proposition known to be false, then $P(n)$ is true.
- E.g. $\sqrt{2}$ is irrational.
  - Assume $\sqrt{2}$ is rational, then $\sqrt{2} = \frac{a}{b}$ where a, b have no common factors.
    $a^2 = 2b^2 \Rightarrow a^2$ is even $\Rightarrow a$ is even, $a = 2c$.
    $\Rightarrow 4c^2 = 2b^2 \Rightarrow b^2 = 2c^2 \Rightarrow b$ is even.
    Contradiction.

Other proof techniques

- Direct proof
- Proof by counter example
- Contrapositive:
  - $A \Rightarrow B \Leftrightarrow \neg B \Rightarrow \neg A$.

Permutations and combinations

Rule of product: If event A can happen in m ways and event B can happen in n ways, then A and B can happen in mn ways
Rule of sum: If event A can happen in m ways and event B can happen in n ways, then A and B can happen in $m+n$ ways

Permutations

- $P(n,r) = \frac{n!}{(n-r)!}$: the way to arrange r objects out of n objects where order matters.
- e.g. # ways n people can be seated in a round table.
  - For linear, $P(n,n) = n!$.
  - For a ring, shifting doesn't affect the order, $(n-1)!$.
- If not all items are distinct, but we have q_1 of type 1, q_2 of type 2,… q_t of type t, then the permutation is $\frac{n!}{q_1!q_2!\dots q_t!}$.
- e.g. 5 dashes and 8 dots can be arranged in $\frac{13!}{5!8!}$ ways.
- e.g. show that $(k!)!$ Is divisible by $(k!)^{(k-1)!}$, $\forall k$.
  - Consider $(k!)$ objects, k of type 1, k of type 2,…, k of type $(k-1)!$.
  - # ways to arrange these objects: $\frac{(k!)!}{k!\cdot\ldots\cdot k!} = \frac{(k!)!}{(k!)^{(k-1)!}}$ is an integer.

Combinations

- Relative order does not matter
- $C(n,r) = \frac{P(n,r)}{r!} = \frac{n!}{(n-r)!r!} = C(n, n-r) = \binom{n}{r}$.
- How many diagonals in a decagon? $\binom{10}{2} - 10$.
- e.g. 11 scientists are working on a recent project. They want to lock documents in a vault such that vault opens if at least 6 scientists are present. What is the smallest number of locks required? What is the smallest number of keys each scientist should have?
  - Every group of 5 scientists, there should be 1 lock that cannot be opened
  - For every 2 or more groups of 5, this lock must be different, otherwise there would be a group of 6 scientist that cannot open the vault
  - $\Rightarrow \forall$ groups of 5, 1 lock cannot be opened $\Rightarrow \binom{11}{5} = 462$ locks at least.
  - Every time a new scientist join a group of 5, they have the key that the others don't.
  - #keys=how many scientists can be formed out of the rest 10 scientists.
  - #keys $= \binom{10}{5} = 252$ keys at least.

Combinatorial argument: (argument based on counting)

- Given some equation, prove using the following method
- Question: ask some counting question.
- LHS: argue why the LHS answers the question.
- RHS: argue why the RHS answers the question.
- E.g. $\binom{n}{k} = \binom{n}{n-k}$.
  - Question: how many ways can you select k objects from n total objects without replacement?
  - LHS: True by definition.
  - RHS: Instead of choosing k object, I choose $n-k$ objects to eliminate, leaving me with k objects.
- e.g. $\sum_{k=0}^{n} \binom{n}{k}^2 = \binom{2n}{n}$.
  - Question: I have n black balls and n red balls. How many ways can I select n balls out of the $2n$ total?
  - RHS: True by definition.
  - LHS: fix k to be the number of black balls chosen, then $n-k$ is the number of red balls.
    There are $\binom{n}{k}$ ways to choose black balls, and $\binom{n}{n-k} = \binom{n}{k}$ ways to choose $n-k$ red balls.

AND event, $\binom{n}{k}^2$ is the number of ways to choose k black and $n-k$ red.

$k \in [0, n]$ disjoint, OR event, adding them gives $\sum_{k=0}^{n} \binom{n}{k}^2$.

- e.g. $\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$.
  - Question: # ways to select k objects from n objects.
  - LHS: True by definition.
  - RHS: consider some particular object x in the set.
    - If x is in the k objects we selected, we choose $k-1$ objects from the rest $\binom{n-1}{k-1}$.
    - If x is not in the set of objects we selected, we choose k from the rest, $\binom{n-1}{k}$.
    - Disjoint, so addition.
- e.g. word length n from alphabet $\{0,1,2\}$.
  - $\binom{n}{0} 2^n + \cdots + \binom{n}{r} 2^{n-r} = \frac{3^n+1}{2}$.
  - RHS: proof by induction, if length k has odd number of zeros ($\frac{3^k-1}{2}$ ways), append a single 0, otherwise ($\frac{3^k+1}{2}$ ways), we can append 1 or 2 only.
- e.g. $\binom{n+m}{n}\binom{n+m}{m} = \sum \binom{n+m}{i}\binom{n+m-i}{n-i}\binom{m}{m-i}$.
  - Total $n+m$ balls, select n balls from them first, put back and select m balls.
  - RHS: first select i balls that will be in both the first and second set. Then select $n-i$ balls from $n+m-i$ balls to form the n ball group. Select $m-i$ balls from the rest m balls. Sum up over i.
- e.g. $n4^{n-1} = \sum_{k=0}^{n} \binom{n}{k} 3^k (n-k)$.
  - Question: string of length n, one blank position, alphabet of size 4. How many ways are there to create such string.
  - LHS: n ways to choose a single position for the blank. Then there are 4^{n-1} ways to assign 4 alphabets to the rest $n-1$ positions.
  - RHS: choose k positions from n to assign the rest 3 alphabets, then $n-k$ ways to choose a specific position for the blank. Fill in the rest with the final alphabet.

Probability

- Experiment
- Sample Space S
  - e.g. two fare coin $S = \{HH, HT, TH, TT\}$.
- Axioms
  - $\Pr(a \in S) \geq 0$,
  - $\Pr(S) = 1$,
  - $\Pr(A \cup B) = \Pr(A) + \Pr(B) - \Pr(A \cap B)$,
  - $\Pr(A \cap B) = \Pr(A)\Pr(B)$ if independent.
- e.g. Flip fair coins n times, there are 2^n outcomes uniformly distributed.
  - $\Pr(k\ heads) = \frac{C(n,k)}{2^n}$.
- Bayes theorem: $\Pr(A|B) = \frac{\Pr(A \cap B)}{\Pr(B)} = \frac{\Pr(B|A)\Pr(A)}{\Pr(A)\Pr(B|A) + \Pr(\bar{A})\Pr(B|\bar{A})}$.
  - e.g. 1 fair coin, 1 biased (always H), $\Pr(biased|2\ heads) = \frac{1 \cdot \frac{1}{2}}{\frac{1}{2} \cdot 1 + \frac{1}{2}\left(\frac{1}{2}\right)^2} = \frac{4}{5}$.

Discrete random variables

- For an r.v. X, $\Pr(X = x) = \sum_{\{s \in S, X(S) = x\}} \Pr(s)$.
- Expected value: $E(X) = \sum_{x \in X} x \Pr(X = x)$.
- e.g. flip two coins win \$3 for H, lose \$2 for T.
  - $E(X) = \frac{1}{4}6 + \frac{1}{4}(-4) + \frac{1}{2}1 = 1$.
- Properties:

- $E(X+Y) = E(X) + E(Y)$.
- $E(aX) = aE(X)$.

Graphs and trees

- $G = (V, E)$.
  - V: set of vertices.
  - E: set of edges.
  - Directed/undirected
  - Weighted/unweighted.
  - Representation
    - Adjacency list
    - Adjacency matrix
  - Path
  - Edge
  - Simple path
  - Cycle
  - Vertex degree
    - Undirected: $\deg(u)$= # all edges connected to u.
    - Directed: in-degree, out-degree.
  - Neighborhood: $N(u)$ all vertices directly connected to u.
  - For undirected $2|E| = \sum \deg(u)$.
- A tree is a connected, acyclic and undirected graph
  - Terminology: root, children, parent, internal nodes, leaves, subtree rooted at v.
  - Binary/k-ary tree: tree with nodes with at most 2 or k children.
  - Complete tree: all leaves have the same depth, all nodes have k children.
  - Depth at node u: length of path from root to u.
    - $depth(root) = 0$.
  - Height of node u: #edges in longest path from node u down to a leaf.

Recurrence

- Motivating example: Mergesort
  - Mergesort
    If $p < r$:
        $q = \left\lfloor \frac{p+r}{2} \right\rfloor$,
        Mergesort($A, q+1, r$)
        Mergesort(A, p, q)
        Merge(A, p, q, r).
  - Split to single element, then merge into a ordered manner
  - Runtime: $T(n) = T\left(\left\lceil \frac{n}{2} \right\rceil\right) + T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + \theta(n)$.
    - Recurrence is for # subproblems and size of subproblem.
    - $\theta(n)$ is for conquer part.
    - Base case omitted since we are only interested in asymptotic runtime.
      - $T(n) = 2T\left(\frac{n}{2}\right) + \theta(n)$.
- Master's theorem for $T(n) = aT(n/b) + f(n)$, $a \geq 1$, $b > 1$.
  - Case 1: if $f(n) = O\left(n^{\log_b a - \epsilon}\right)$ for $\epsilon > 0$, then $T(n) = \theta\left(n^{\log_b a}\right)$.
    - e.g. $T(n) = 9T\left(\frac{n}{3}\right) + n$, $a = 9$, $b = 3$, $n^{\log_b a} = n^2$, $f(n) = n = O\left(n^{2-\epsilon}\right)$, $T(n) = \theta\left(n^2\right)$.
  - Case 2: if $f(n) = \theta\left(n^{\log_b a} \log^k n\right)$, then $T(n) = \theta\left(n^{\log_b a} \log^{k+1} n\right)$.
    - e.g. $T(n) = 2T(n/2) + \theta(n)$, $a = b = 2$, $n^{\log_b a} = n$, $f(n) = \theta(n)$, $T(n) = n \log n$.
  - Case 3: if $f(n) = \Omega\left(n^{\log_b a + \epsilon}\right)$ for $\epsilon > 0$ and $af\left(\frac{n}{b}\right) \leq cf(n)$ for $0 < c < 1$, then $T(n) = \theta(f(n))$.
    - e.g. $T(n) = 3T(n/4) + n \log n$, $a = 3$, $b = 4$, $n^{\log_b a} \approx n^{0.8}$,

- ◻ $f(n) = n\log n = \Omega(n^{0.8+\epsilon})$ and $3\frac{n}{4}\log\left(\frac{n}{4}\right) \le \frac{3}{4}n\log n$.
- ◻ $T(n) = \theta(n\log n)$.

- Substitution
  - We guess a solution to $T(n)$ and use strong induction to prove guess was correct
  - $T(n) = 2T\left(\left\lfloor\frac{n}{2}\right\rfloor\right) + n$.
    - Guess $T(n) = O(n\log n)$, assume $T(k) \le k\log k$, $\forall k < n$.
    - Then $T(n/2) \le c\left\lfloor\frac{n}{2}\right\rfloor\log\left\lfloor\frac{n}{2}\right\rfloor$.
    - We have $T(n) = 2c\left\lfloor\frac{n}{2}\right\rfloor\log\left\lfloor\frac{n}{2}\right\rfloor + n \le 2c\frac{n}{2}\log\frac{n}{2} + n \le cn\log n + (1-c)n \le cn\log n$, for $c \ge 1$.
  - Erroneous guess
    - $T(n) = O(n)$ gives $T(k) \le ck$, $k < n$.
    - $T(n) = 2c\left\lfloor\frac{n}{2}\right\rfloor + n \le cn + n = (c+1)n$, not equivalent to $T(n) = O(n)$ since we are not explicitly proving the IH.
- Recursion tree
  - Helps find a good working guess for substitution
    - Longest path gives upper bound
    - Shortest path gives lower bound
  - e.g. $T(n) = T\left(\frac{n}{4}\right) + T\left(\frac{2n}{3}\right) + n$.
    - Imbalanced tree, longest path (height) is determined by the $\frac{2n}{3}$ path.
    - Consider the longest path:
      - ◻ Size at level i: $\left(\frac{2}{3}\right)^i n$.
      - ◻ At max level: $\left(\frac{2}{3}\right)^h n = 1$, gives $h = O(\log n)$.
    - For shortest path, $\left(\frac{1}{4}\right)^{h'} n = 1$, stil $h' = O(\log n)$.
    - Total work: h ×cost/level, $O(n\log n)$.
    - Need strong induction proof
    - Lower bound, still $\Omega(n\log n)$.
  - Generally, $\sum_{i=0}^{h} cost/level$.
  - $F(n) = F(\lfloor\log n\rfloor) + \log n = \Theta(\log n)$.
    - Base case if $\lfloor\log^{u} n\rfloor = 0$, where $u = \log^* n$.
    - $F(n) = \sum_{i=1}^{\log^* n}(\log^{(i)} n)$.

Let $G = (V, E)$ be an undirected graph, all of the following is equivalent.

- G is a free tree (connected, acyclic).
- Any two vertices in G are connected by a unique simple path.
- G is connected, but if you remove any edge, it becomes disconnected.
- G is connected and $|E| = |V| - 1$.
- G is acyclic and $|E| = |V| - 1$.
- G is acyclic and adding any edge to E creates a cycle.
- Proofs
  - (1)⇒(2): since G is connected, there must be at least one path.
    Assume by contradiction that a second path exits, $P_1: s \to t$, $P_2: t \to s$, P_1, P_2^{-1} forms a cycle, but G should be acyclic.
  - (2)⇒(3): since only one path exists between any 2 nodes, removing an edge must disconnect something.
  - (3)⇒(4): $|E| \ge |V| - 1$ by induction on $|V|$, same applies for $|E| \le |V| - 1$.
    Basis: $|V| = 1$, then $|E| = 0$, $0 \ge 1 - 1 = 0$.
    IH: if $|V| = n$, then $|E| \ge n - 1$.
    Induction: suppose G is any graph with $|V| = n + 1$.
    Remove some vertex to get $V' = V - \{v\}$, remove all edges connecting to v to get E'.
    $G' = (V', E')$ of size $|V'| = n$, so $|E'| \ge |V'| - 1$.

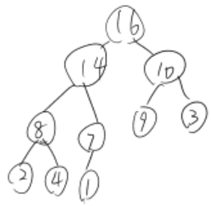
Now $|V| = |V'| + 1$, $|E| \geq |E'| + 1$, so $|E| \geq |V| - 1$.

- (4)⇒(5): assume by contradiction that G contains a cycle $v_1 \rightarrow v_2 \rightarrow \cdots \rightarrow v_k \rightarrow v_1$.
  Add vertices to G_k, one at a time, each vertex also adds at least 1 edge.
  $\left|V_{k+i}\right| = k + i$, $\left|E_{k+i}\right| \geq k + i$, then $|V| = n$ and $|E| \geq n$,
  contradiction, since $|E| = |V| - 1$.
- (5)⇒(6): G has k connected components. Each connected component is a free tree, so (1) to (5) is true.
  $\left|E_i\right| = \left|V_i\right| - 1, \forall i$, $|E| = \sum |E_i| = \sum_{i=1}^{k} \left|V_i\right| - 1 = |V| - k$, so $k = 1$, G is fully connected.
  G is a free tree means that adding any edges must create a cycle.
- (6)⇒(1): Consider any pair of nodes s and t.
  Adding edge (u, v) cause a cycle between s and t.
  Now remove (u, v) which leaves a path from s to t.
  G is connected, so G is a free tree.

Sorting

January 17, 2023 9:27 PM

Heap (binary)

- It is a tree
- Full except maybe at the bottom level, leaves must be starting from left
- Heap order property
  - Key(parent) $\geq$ key(children) is max heap.
  - Key(parent) $\leq$ key(children) is min heap.
- Heap as an array: Given index i,
  - Parent: $\left\lfloor \frac{i}{2} \right\rfloor$.
  - Left child: $2i$.
  - Right child: $2i + 1$.
- e.g. $A = [16,14,10,8,7,9,3,2,4,1]$.
  - 

Max-Heapify: enforce the heap order property if it is violated

- Compare $A[i]$ with $A[2i]$ and $A[2i + 1]$.
- Swap if $A[i]$ smaller, $A[i] \leftrightarrow \max(A[2i], A[2i + 1])$.
- Continue downwards swapping if necessary until either property not violated or you hit a leaf node.
- Runtime: $O(\log n)$ because of the balanced property.

Build-Max-Heap(A, n):

For $i \leftarrow \left\lfloor \frac{n}{2} \right\rfloor : 1$:

Do Max-Heapify(A, i, n)

e.g. $A = [4,1,3,2,16,9,10,14,8,7]$.

- Start with 16, do nothing.
- Then at $i = 4$, $A[i] = 2$, $A[2i] = 14$, $A[2i + 1] = 8$, violated, swap with 14.
  - $A = [4,1,3,14,16,9,10,2,8,7]$.
- $i = 3$, $A[i] = 3$, $A[2i] = 9$, $A[2i + 1] = 10$ swap with 10.
  - $A = [4,1,10,14,16,9,3,2,8,7]$.
- $i = 2$, $A[i] = 1$, $A[2i] = 14$, $A[2i + 1] = 16$, swap with 16.
  - Then also need to swap with 7.
  - $A = [4, 16,10,14,7,9,3,2,8,1]$.
- $i = 1$, $A[i] = 4$, $A[2i] = 16$, $A[2i + 1] = 10$, swap with 16.
  - Then also need to swap with 14 and 8.
  - $A = [16,14,10,8,7,9,3,2,4,1]$.

Runtime for Build-Max-Heap:

- Simple: $O(n \log n)$ (for loop $\times$ cost at Heapify).
- Proper:
  - Time to run Max-Heapify is linear in the height of the node it is run on and most node have small height.
  - Lemma 1: at height h, there are at most $\left\lceil \frac{n}{2^{h+1}} \right\rceil$ nodes.
  - Lemma 2: height of heap is $\lfloor \log n \rfloor = O(\log n)$.

- Runtime $= \sum_{h=0}^{\lfloor \log n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(n \sum_{h=0}^{\lfloor \log n \rfloor} \frac{h}{2^h}\right)$.
  - Apply $\sum kx^k = \frac{x}{(1-x)^2}$ for $x = \frac{1}{2}$, we get $O(n)$.

Heapsort(A, n)
    Build-Max-Heap(A, n)
    For $i \leftarrow n: 2$:
        Swap $A[1] \leftrightarrow A[i]$.
        Max-Heapify($A, 1, i-1$).

E.g. $A = [7,4,3,1,2]$.

- $[7,4,3,1,2] \to [2,4,3,1,7] \to [4,2,3,1,7] \to [1,2,3,4,7] \to$.

Runtime for Heapsort: $O(n) + O(n \log n) = O(n \log n)$.

Priority Queue implementation using heaps

- Treat each element in the heap array as a pointer to an object in the priority queue.
- Each element has a key value $A[i].key$.
- Insert(S, x, k): inserts the element x with key k into the set S.
  - $O(\log n)$.
- Maximum(S): returns the element of S with the largest key.
  - $\Theta(1)$.
- Extract-Max(S): removes and returns the element of S with the largest key.
  - $O(\log n)$.
- Increase-Key(S, x, k): increases the value of element x's key to the new value k which is assumed to be at least as large as x's current key value.
  - $O(\log n)$.

```
MAX-HEAP-MAXIMUM(A)
if A.heap-size < 1
    error "heap underflow"
return A[1]

MAX-HEAP-EXTRACT-MAX(A)
max = MAX-HEAP-MAXIMUM(A)
A[1] = A[A.heap-size]
A.heap-size = A.heap-size - 1
MAX-HEAPIFY(A, 1)
return max
```

```
MAX-HEAP-INCREASE-KEY(A, x, k)
if k < x.key
    error "new key is smaller than current key"
x.key = k
find the index i in array A where object x occurs
while i > 1 and A[PARENT(i)].key < A[i].key
    exchange A[i] with A[PARENT(i)], updating the information that maps
        priority queue objects to array indices
    i = PARENT(i)

MAX-HEAP-INSERT(A, x, n)
if A.heap-size == n
    error "heap overflow"
A.heap-size = A.heap-size + 1
k = x.key
x.key = -∞
A[A.heap-size] = x
map x to index heap-size in the array
MAX-HEAP-INCREASE-KEY(A, x, k)
```

Quicksort

- Sort in place
- Constant in $\theta(n \log n)$ runtime are small
- But $\theta(n \log n)$ only in expected case.
- $O(n^2)$ in worst case.

Partition(A, p, r)

$x \leftarrow A[r]$ (pivot is the right most element in the array).
$i \leftarrow p - 1$.
For $j \leftarrow p$ to $r - 1$.
  If $A[j] \leq x$:
    $i \leftarrow i + 1$.
    Swap $A[i] \leftrightarrow A[j]$.
Swap $A[i+1] \leftrightarrow A[r]$.
Return $i + 1$.

Runtime: $\theta(n)$.

e.g. $A = [8,1,6,4,0,3,9,5]$.

- Initially, $p = 1$, $r = 8$, $i = 0$.
- $j = 1$, $A[1] > A[r]$, skip.
- $j = 2$, $A[2] = 1 \leq 5 = A[r]$, $i = 1$, swap $A[1], A[2]$, get $[1,8,6,4,0,3,9,5]$.
- $j = 3$, $A[3] = 6 > A[r]$ skip.
- $j = 4$, $A[4] = 4 \leq 5 = A[r]$, $i = 2$ swap $A[2], A[4]$, get $[1,4,6,8,0,3,9,5]$.
- Finally, get a partial ordering $[1,4,0,3,5,8,9,6]$.
  - Left elements smaller than the pivot.
  - Right elements larger than the pivot

Quicksort(A, p, r)

If $p < r$:
  $q \leftarrow$Partition(A, p, r).
  Quicksort$(A, p, q - 1)$
  Quicksort$(A, q + 1, r)$

Initial call: Quicksort$(A, 1, n)$.

Performance of quicksort:

- Worst case: when input is already sorted, pivot is always the largest/smallest element. Every time, we get

an empty array and an array of size $p-1$.
  - $T(n) = T(n-1) + \theta(n) = \theta(n^2)$.
- Best case: pivot always median $T(n) = 2T\left(\frac{n}{2}\right) + \theta(n) = \theta(n \log n)$.
- Balanced case: $T(n) = T(an) + T(bn) + \theta(n)$, where $a + b = 1$, $T(n) = \theta(n \log n)$.

Randomized quicksort
- We can randomly shuffle input or choose pivot to reduce the chance of getting the worst case scenario
- The worst case scenario is still $O(n^2)$, but the chance is lower.

Randomized-Partition

i ←RAND(p, r);

$A[r] \leftrightarrow A[i]$;

Return Partition(A, p, r).

Worst case analysis (applies to both versions)
- $T(n) = \max_{q\in[0,n-1]}\{T(q) + T(n-q+1)\} + \theta(n)$.
- We guess $T(n) = O(n^2)$, and prove by induction.
- Assume $T(k) \leq ck^2$ for some c and all $k < n$.
- Then $T(n) \leq \max_{q\in[0,n-1]}\left\{cq^2 + c(n-q-1)^2\right\} + \theta(n)$.
- $cq^2 + c(n-q-1)^2$ obtains max at $q = 0$ and $q = n-1$.
  - $\max_{q\in[0,n-1]}\left\{cq^2 + c(n-q-1)^2\right\} \leq c(n-1)^2$.
- $T(n) \leq c(n-1)^2 - c(2n-1) + \theta(n) \leq cn^2$. Choose c such that $c(2n-1)$ dominates $\theta(n)$.
- $T(n) = O(n^2)$.
- Can also show that $T(n) = \Omega(n^2)$, $T(n) = \theta(n^2)$.

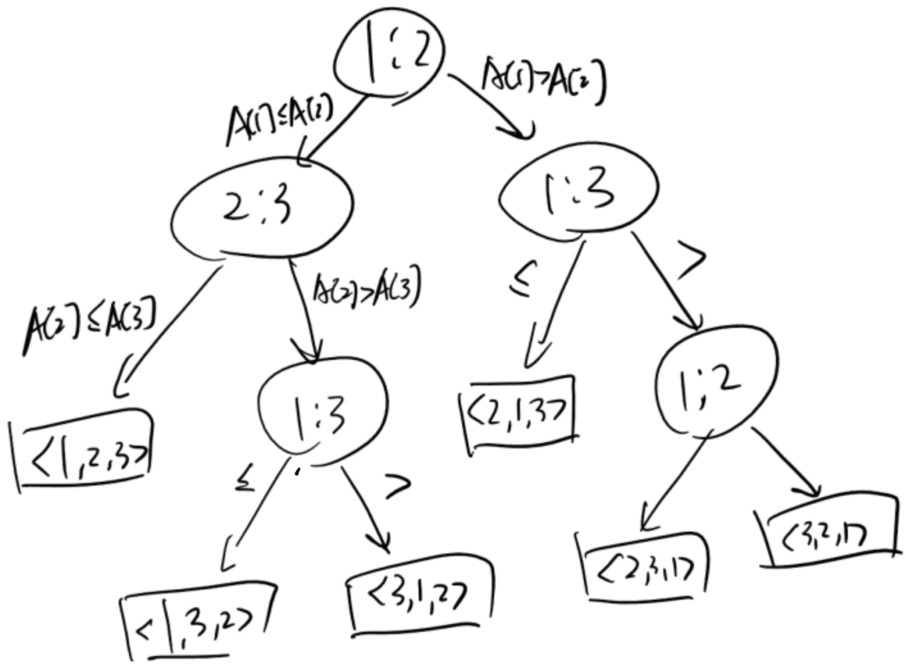
Expected case analysis
- $T(n) = \frac{1}{n}\sum_{i=0}^{n-1}\big(T(i) - T(n-i-1)\big) + n - 1$.

  $= (n-1) + \frac{2}{n}\sum_{i=1}^{n-1} T(i)$.
- Guess $T(i) \leq ci \log i$ for $i < n$.
- Use $\sum_{i=1}^{n-1} f(i) \leq \int_1^n f(x)dx$ and $\int cx \log x \, dx = \left(\frac{c}{2}\right) x^2 \log x - \frac{cx^2}{4}$.
- $T(n) \leq (n-1) + \frac{2}{n}\sum_{i=1}^{n-1} ci \log i \leq n - 1 + \frac{2}{n}\int_1^n cx \log x \, dx$.

  $= (n-1) + \frac{2}{n}\left(\frac{c}{2}n^2 \log n - \frac{cn^2}{4} + \frac{c}{4}\right) \leq cn \log n$ for $n = 2$.
- So $T(n) = O(n \log n)$.

Lower bounds for sorting
- Consider comparison-based sorting only
  - Only operation to determine order info about a sequence of elements is pairwise comparison
- Trivial: $\Omega(n)$ to examine all elements.
- Claim: $\Omega(n \log n)$ is lower bound for comparison based sorting in the worst case.

Decision tree
- Abstraction of comparison-based sorting
- Every tree is for one sorting algorithm on inputs of a given size
- No control flow, no data movements are modeled
- We count only comparisons as cost
- e.g. $A[1,2,3]$.



Observation: decision tree must have at least one leaf for every permutation of input sequence

- Number of leaves: $l \geq n!$.
- Height: h, we need to show $h = \Omega(n \log n)$.
- Lemma: any binary tree of height h has $\leq 2^h$ leaves (proof by induction on h).
- $n! \leq l \leq 2^h$, $2^h \geq n$, $h \geq \log n!$, $h \geq \log(n^n/e^n)$ (by Stirling).
- $h \geq n \log n - n \log e = \Omega(n \log n)$.
- Since h represents worst case execution trace, any comparison-based sorting takes $\Omega(n \log n)$ in worst case.

Sorting in linear time

- Only algos that use operations other than pairwise comparisons
- Counting, radix, bucket sort.

Stable sort: sorting that preserves the relative order of the same value in the previous step

Counting sort

- Input: $A[1 \dots n]$, $A[j] \in \{0,1,\dots,k\}$ (n, k are parameters).
- Output: $B[1 \dots n]$ sorted (not in place).
- Auxiliary array: $C[0 \dots k]$.
- Algo:
  CountingSort(A, B, n, k)
  - For $i \leftarrow 0{:}k$, $c[i] \leftarrow 0$.
  - For $j \leftarrow 1{:}n$, $C\left[A[j]\right] \leftarrow C\left[A[j]\right] + 1$.
  - For $i \leftarrow 1{:}k$, $C[i] \leftarrow C[i] + C[i-1]$ (accumulation).
  - For $j \leftarrow n{:}1$,
    - $B[C[A[j]]] \leftarrow A[j]$.
    - $C\left[A[j]\right] \leftarrow C\left[A[j]\right] - 1$.
- Example: $A[2_1, 5_1, 3_1, 0_1, 2_1, 3_2, 0_2, 3_3]$.
  - First for loop: $C = [0,0,0,0,0,0]$.
  - Second for loop: $C = [2,0,2,3,0,1]$.
  - Third for loop: $C = [2,2,4,7,7,8]$.
  - Sorted: $[0_1, 0_2, 2_1, 2_2, 3_1, 3_2, 3_3, 5_1]$.
- Total time: $\theta(n + k)$.
  - Linear if and only if $k = \theta(n)$.
- Auxiliary array can be used to do Range Query in $O(1)$.

- e.g. to find number of elements in $[a, b]$, do $c[b] - c[a-1]$, in (a, b) do $c[b-1] - c[a]$.

Radix sort

- Key idea: sort LSD (least significant digit first)
- RadixSort(A, d)

  For $i \leftarrow 1: d$,

  Stable sort to sort A on digit i. (relative order in previous step is preserved. e.g. Counting sort)
- Example:

initial	Right	middle	left
326	690	704	326
453	751	608	435
608	453	326	453
835	704	835	608
751	835	435	690
435	435	751	704
704	326	453	751
690	608	690	835

- Time: d passes, each pass $\theta(n+k)$.
  - $\theta(d(n+k))$ if $k = \theta(n)$, then we get $\theta(dn)$.
- Suppose we have n words, b bits/word, and use r-bit digits.
  - $d = \left\lceil \frac{b}{r} \right\rceil$, $k = 2^r - 1$.
  - Plug into the time, get $\theta\left(\frac{b}{r}(n + 2^r)\right)$.
  - When $r = \log n$, $\theta\left(\frac{b}{\log n}(n+n)\right) = \theta\left(\frac{bn}{\log n}\right)$. (balanced)
  - When $r = 2\log n$, $\theta\left(\frac{b}{2\log n}(n+n^2)\right) = \theta\left(\frac{bn^2}{\log n}\right)$. (worst)
  - When $r < \log n$, no improvement.

BucketSort(A, n)

For $i \leftarrow 1: n$,

Insert $A[i]$ into $B[\lfloor nA[i] \rfloor]$ (B is a list of buckets).

e.g. with $n = 100$, 0.5 and 0.505 goes to $B[50]$, 0.51 goest to $B[51]$.

For $i \leftarrow 0: n-1$,

Sort $B[i]$ with insertion sort.

Concat $B[0], \dots, B[n-1]$.

Return concatenated B.

Correctness

- Consider $A[i], A[j]$, WLOG, assume $A[i] \le A[j]$.
- Then $\lfloor nA[i] \rfloor \le \lfloor nA[j] \rfloor$.
- Two cases
  - $A[i]$ in the same bucket as $A[j]$, then insertion sort imposes the correct order within the bucket.
  - $A[i]$ in a bucket with smaller index than $A[j]$'s bucket, after concatenation, order is preserved.

Runtime in expected case

- Define r.v. n_i =# elements placed in $B[i]$.
- $T(n) = \theta(n) + \sum_{i=0}^{n-1} O(n_i^2)$.
- $E[T(n)] = E\left[\theta(n) + \sum_{i=0}^{n-1} O(n_i^2)\right] = \theta(n) + \sum_{i=0}^{n-1} E\left(O(n_i^2)\right) = \theta(n) + \sum_{i=0}^{n-1} O\left(E(n_i^2)\right)$.
- Claim: $E(n_i^2) = 2 - \frac{1}{n}$, $\forall i = 0, \dots, n-1$.
  - Proof: define indicator r.v.s $X_{ij} = I\{A[j] \in B[i]\} = \begin{cases} 1, if\ A[j]\ is\ in\ Bucket\ i \\ 0, else \end{cases}$.

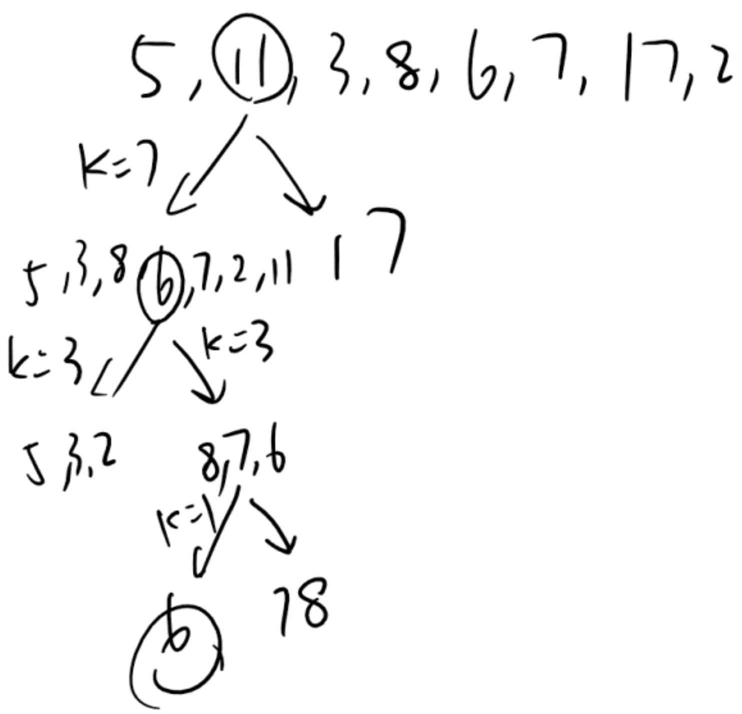
- $\Pr\left[A[j] \in B[i]\right] = \frac{1}{n}$, since the values are uniformly distributed.
- $n_i = \sum_{j=1}^{n} X_{ij}$.
- $E[n_i^2] = E\left[\left(\sum_{i=1}^{n} X_{ij}\right)^2\right] = E\left[\sum_{i=1}^{n} X_{ij}^2 + 2\sum_{j=1}^{n-1}\sum_{k=j+1}^{n} X_{ij}X_{ik}\right]$,
- $= \sum_{j=1}^{n} E[X_{ij}^2] + 2\sum_{j=1}^{n-1}\sum_{k=j+1}^{n} E\left[X_{ij}X_{ik}\right]$.
- $E\left[X_{ij}^2\right] = 0^2 \Pr(A[j] \notin B[i]) + 1^2 \Pr(A[j] \in B[i]) = \frac{1}{n}$.
- Since X_{ij}, X_{ik} are independent, $E\left[X_{ij}X_{ik}\right] = E\left[X_{ij}\right]E\left[X_{ik}\right] = \frac{1}{n^2}$.
- Then $E[n_i^2] = \sum_{j=1}^{n}\frac{1}{n} + 2\sum_{j=1}^{n-1}\sum_{k=j+1}^{n}\frac{1}{n^2} = 1 + 2\frac{1}{n^2}\binom{n}{2} = 2 - \frac{1}{n}$.

- Hence $E[T(n)] = \theta(n) + \sum_{i=0}^{n-1} O\left(E\left(2 - \frac{1}{n}\right)\right) = \theta(n) + O(n) = O(n)$.

Order statistics
- Given $A[1, \ldots, n]$, interested in finding ith order statistics.
  - Element in A, s.t. $i - 1$ elements are smaller than it.
- 1st order statistic: min.
- Nth order statistic: max.
- Lower/upper median, etc.
- Simultaneous min and max requires at most 3 $\left\lfloor\frac{n}{2}\right\rfloor$ comparisons.

Selection in expected linear time
Randomized-Select(A, p, r, i)
    If $p = r$: return $A[p]$
    q =Randomized-Partition(A, p, r).
    $k = q - p + 1$.
    If $i = k$: return $A[q]$ (pivot is the ith order statistic).
    If $i < k$: return Randomized-Select($A, p, q - 1, i$) (We have more elements than needed).
    Else: return Randomized-Select($A, q, r, i - k$) (We have fewer elements than needed).

e.g. $A = [5,11,3,8,6,7,17,2]$, $i = 4$.



6 is the 4th order statistics in this case

Worst case: $\theta(n^2)$.

Expected runtime: $T(n) \le \sum_{k=1}^{n} X_k\big(T(\max\{k-1, n-k\})\big) + O(n)$.

- $E[T(n)] \le \sum_{k=1}^{n} E\big[X_k\big]E[T(\max\{k-1, n-k\})] + O(n)$,
- $= \sum_{k=1}^{n} \frac{1}{n} E[T(\max\{k-1, n-k\})] + O(n)$,
- Note: $\max\{k-1, n-k\} = \begin{cases} k-1, k > \left\lceil \frac{n}{2} \right\rceil \\ n-k, k \le \left\lceil \frac{n}{2} \right\rceil \end{cases}$.
- If n is even, terms from $T\left(\left\lceil \frac{n}{2} \right\rceil\right)$ to $T(n-1)$ appear twice.
- If n is odd, terms also appear twice except $T\left(\left\lfloor \frac{n}{2} \right\rfloor\right)$ which appears once.
- Then $E[T(n)] \le \frac{2}{n} \sum_{k=\left\lfloor \frac{n}{2} \right\rfloor}^{n-1} E[T(k)] + O(n)$.
- Replace $O(n)$ with αn, guess $T(k) \le ck$ for $k < n$.
- $E[T(n)] \le \frac{2}{n} \sum_{k=\left\lfloor \frac{n}{2} \right\rfloor}^{n-1} ck + \alpha n = \frac{2c}{n}\left(\sum_{k=1}^{n-1} k - \sum_{k=1}^{\left\lfloor \frac{n}{2} \right\rfloor - 1} k\right) + \alpha n$.
- $\le \frac{2c}{n}\left(\frac{n(n-1)}{2} - \frac{\left(\frac{n}{2}-2\right)\left(\frac{n}{2}-1\right)}{2}\right) + \alpha n$.
- $= cn - \left(\frac{cn}{4} - \frac{c}{2} - \alpha n\right)$.
- Thus $E[T(n)] \le cn$ for $\frac{cn}{4} - \frac{c}{2} - \alpha n \ge 0$ or $n \ge \frac{2c}{c-4\alpha}$.

E.g. sort an array of integers in worst case $O\big(n \log n\big)$ time

- Insertion sort ($\theta\big(n^2\big)$)
- Merge sort ($\theta\big(n \log n\big)$)
- Heap sort ($\theta\big(n \log n\big)$)
- Randomized quicksort ($O\big(n^2\big)$)
- Counting sort ($\theta(n+k)$)
  - k can be larger than n, assume all integers in $[0, k]$.
- Radix sort ($\theta\big(d(n+k)\big)$)
- Bucket sort ($O\big(n^2\big)$)
  - Worst case when all numbers in the same bucket

e.g. sort an array of integers ranging from -100 to 100 in $O(n)$ time worst case.

- Shift all integers by +100
- Sort the array by counting sort
- Shift output by -100

e.g. sort the above array using bucket sort, in $O(n)$ expected time.

- $\forall x \in A, y = \frac{x+100}{201}$
- Sort using bucket sort.
- Then $\forall y \in A', x = 201y - 100$.

e.g. sort n integers ranging from 0 to $n^3 - 1$ in $O(n)$ time.

- Counting sort won't work, since $k = n^3 - 1, \theta(n+k) = \theta(n + n^3 - 1)$.
- Any number $x \in [0, n^3 - 1]$ can be written as $= a_2 n^2 + a_1 n + a_0$ for $a_0, a_1, a_2 \in [0, n)$.
- Run radix sort base n.
- $\theta\big(d(n+k)\big) = \theta\big(3(n+n)\big)$.
  - k is given by the base (n), d is given by number of digits (# a_i).

e.g. weighted medians

- Let $x_1, \dots, x_n$ be n distinct (unsorted) elements, each with positive edge weight $w_1, \dots, w_n$ s.t. $\sum_{i=1}^{n} w_i = 1$, the weighted (lower) median is the element x_k s.t. $\sum_{x_i < x_k} w_i < \frac{1}{2}$, $\sum_{x_i > x_k} w_i \le \frac{1}{2}$.
- Show that the weighted median is the same as the median if $w_i = \frac{1}{n}$, $\forall i \in [1, n]$.
- Find the weighted median in $O\big(n \log n\big)$ time using sorting.

Sort using heapsort/mergesort.
$s_0 = 0$.
For $i = 1: n$,
$s_i = s_{i-1} + w_i$. ($s_i = \sum_{j=1}^{i} w_j$)
If $s_{i-1} < \frac{1}{2}$ and $(1 - s_i) \leq \frac{1}{2}$.
Return x_i.

- Find the weighted median in $O(n)$ expected time using selection.
  Modify the Randomized Selection algorithm
  Let X be the randomly chosen partition.
  Let $l = r = \frac{1}{2}$.
  Partition the input array by x and compute $a = \sum_{x_i < x} w_i$, $b = \sum_{x_i > x} w_i$.
  If $a \geq l$, then recurse on the left side with $r = r - b$.
  Else if $b > r$, then recurse on the right side with $l = l - a$.
  Else return x.

e.g. merge k sorted list where each list is size n/k.
- Method 1: concatenate and run merge sort $O(n) + O(n \log n) = O(n \log n)$.
- Method 2:
  - initialize a pointer in these k lists, starting at the first elements.
  - Each iteration, finds the min of the k elements, then increment the corresponding pointer.
  - There is a total of n iterations.
  - Time: $O(nk)$.
- Method 3:
  - initialize a pointer in these k lists, starting at the first elements.
  - Build a heap containing all pointer values $O(k)$.
  - Extract min pointer, $O(\log k)$.
  - Insert the next pointer, $O(\log k)$.
  - Do this n times, get $O(n \log k)$.
- Method 4:
  - Merge the arrays 1 by 1, $\sum_{i=1}^{k-1} O\left(\frac{(i+1)n}{k}\right) = O(nk)$.
  - Pairwise merge, $\sum_{i=1}^{\log k} O(n) = O(n \log k)$.

Selection in worst-case linear time
- Idea: guarantee good split (using median)
- Select algo:
  - Divide the n elements into groups of 5. Get $\left\lceil \frac{n}{5} \right\rceil$ groups ($\left\lfloor \frac{n}{5} \right\rfloor$ with 5 elements, possibly 1 with n mod 5 elements) $O(n)$ time.
  - Find median of each group $O(n)$.
    - Insertion sort on each group $O(1)$.
    - Take median from each group $O(1)$.
  - Find lower median x of the $\left\lceil \frac{n}{5} \right\rceil$ medians from step 2 using recursive call to Select, $T\left(\left\lceil \frac{n}{5} \right\rceil\right)$.
  - Partition by using x as pivot. Assume x is kth element $\begin{cases} k-1\ left \\ n-k-1\ right \end{cases}$, $O(n)$.
  - If $i = k$, return x.
  - If $i < k$, recurse on lower side.
  - If $i > k$, recurse on greater side, searching for $i - k$.
- After insertion sort, we will be able to find medians sorted in increasing order.
  - $a_{11}, a_{12}, a_{13}, a_{14}, a_{15}, a_{21}, .., a_{25}, a_{31}, \dots, a_{35}, a_{41}, \dots, a_{45}, a_{51}, .., a_{55}, a_{61}, a_{62}, a_{63}$.
  - Medians are $a_{13} < a_{23} < a_{33} < a_{43} < a_{53} < a_{62}$.
  - Lower median of them is a_{33}.
- For the final 3 if statements
  - Take the lower median of medians, then $a_{11:3}, a_{21:3}, a_{31}, a_{32} < a_{33}$ and $a_{34}, a_{35}, \ldots > a_{33}$.
  - So at least half of medians $\geq x$ (pivot).

    - Groups with medians $\geq x$ contribute exactly 3 elements $> x$, except x's group and the leftover grpup wich contribute less.
    - Ignore these 2 groups, we have $\left\lceil \frac{1}{2} \left\lceil \frac{n}{5} \right\rceil \right\rceil - 2$ contributing with 3 elements $> x$.
        - At least $3\left(\left\lceil \frac{1}{2} \left\lceil \frac{n}{5} \right\rceil \right\rceil - 2\right) \geq \frac{3n}{10} - 6$ elements.
    - Symmetrically, at least $\frac{3n}{10} - 6$ elements $< x$.
    - In step 5, worst case, we recurse on partition size $\leq \frac{7n}{10} + 6$.
- $T(n) \leq \begin{cases} O(1), n < 140 \\ T\left(\left\lceil \frac{n}{5} \right\rceil\right) + T\left(\frac{7n}{10} + 6\right) + O(n), n \geq 140 \end{cases}$.
    - Guess $T(k) \leq ck$ for $k < n$.
    - $T(n) \leq c\left\lceil \frac{n}{5} \right\rceil + c\left(\frac{7n}{10} + 6\right) + \alpha n \leq cn + \left(-\frac{cn}{10} + 7c + \alpha n\right)$.
    - $\leq cn$ if $-\frac{cn}{10} + 7c + \alpha n \leq 0$ or $c \geq 10\alpha\left(\frac{n}{n-70}\right)$.
    - For $n \geq 140$, $\frac{n}{n-70} \leq 2$, so choosing $c \geq 20\alpha$ gives $c \geq 10\alpha\left(\frac{n}{n-70}\right)$.
    - Could work for $n \geq 71$ with $c \geq 710\alpha$.

Trees

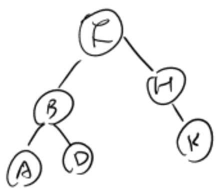
2023年2月2日 18:12

Binary search trees (BST)

- Tree: T.
- Root: $root(T)$.
- Each node has key, left, right, parent.

BST property:

- If y is in the left subtree of x, then $key(y) \leq key(x)$.
- If y is in the right subtree of x, then $key(y) \geq key(x)$.



Traversals

- In-order: A,B,D,F,H,K.
- Pre-order: F,B,A,D,H,K.
- Post-order: A,D,B,K,H,F.

Min: leftmost node, $O(h)$.
Max: rightmost node, $O(h)$.
Successor: next element in in-order walk (min of right subtree)
Predecessor: previous element in in-order walk (max of left subtree, in case of empty left subtree, find y whose successor is x)

Basic operations

- Tree-min: $O(h)$.
- Tree-max: $O(h)$.
- Predecessor: $O(h)$.
- Successor: $O(h)$.
- Insert: $O(h)$.
  - Search and place new node as a leaf
- Delete: $O(h)$.
  - Case 1: z is a leaf, make the parent point to null.
  - Case 2: z has one child, make parent point to z's child.
  - Case 3: z has 2 children, swap the value of z with its predecessor or successor, then delete the successor/predecessor by case 1 or 2.
- Build a BST
  - Worst case: $O(n^2)$ (insertion into a chain).
  - Expected case: $O(n \log n)$ (based on lower bound of sorting).

Red black trees (RBTs)

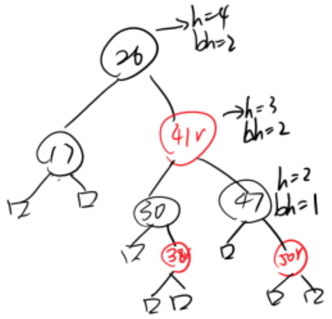
- Motivation: want $h = O(\log n)$ guaranteed in worst case.

RBT properties

- BST property assumed
- Every node is either red or black (0/1 bit).
- The root is black
- Every leaf is black

- If node is red, then both children black
- For each node, all path from that node to descendant leaves contain the same number of black nodes

Heights

- h: heights.
- bh: black height, number of black nodes from this node to leaf, excluding start node.
- 

Claim 1: any node of height h has black height $\geq h/2$.

- Proof: by property 4, at most $h/2$ nodes on the path can be red, so $\geq \frac{h}{2}$ black nodes.

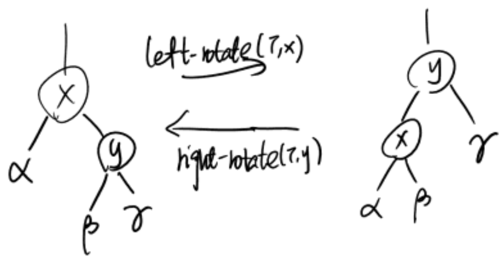
Claim 2: the subtree rooted at node x contains $\geq 2^{bh(x)} - 1$ internal nodes.

- Proof by induction on height of x.
- Basis: if height of x is zero, then it is leaf ⇒ bh(x)=0,$2^{bh(x)} - 1 = 0$.
- I.H.: true for height $< h$ where h is height of x.
- I.S.: height of x is h, say black height is $bh(x) = b$.
- Any child of x has height $\leq h - 1$ and black height $b - 1$ if child is black or b if child is red.
- By IH, each child has $\geq 2^{b-1} - 1$ internal nodes.
- So subtree at x contains $\geq 2\left(2^{b-1} - 1\right) + 1 = 2^b - 1$ internal nodes.

Lemma: RBT with n internal nodes has height $\leq 2\log(n+1)$.

- Claim 1+2 gives $n \geq 2^b - 1 \geq 2^{\frac{h}{2}} - 1 \Rightarrow n + 1 \geq 2^{\frac{h}{2}} \Rightarrow h \leq 2\log(n+1)$.
- i.e. height of RBT is $O(\log n)$.

Operations:

- Search, max, min, predecessor/successor are same as in BST
- Insert, delete need special case
- Rotation
  - Runtime $O(1)$.
  - 

RB-Insert(T,z)

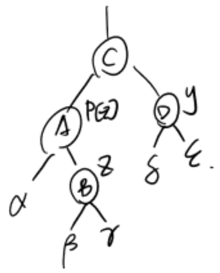
- Search for z.
- Insert as leaf
- Color it red
- Use RB-Insert-fixup(T,z) to fix violated properties.
  - $O(\log n)$.

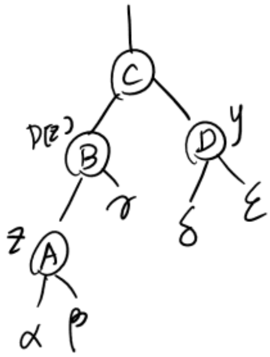
Properties that might be violated by 3

- Property 2: if z is root, violation, but easy to fix by recoloring.

- Property 4: If p(z) is red, violation.

Fixup:

- Assume p[z] is left child (right child is symmetric)
- Let y be p[z]'s sibling.
- Case 1: y is red (z is left/right child of p[z]), not now p[p[z]] is black.
    - Color p[z] and y black, p[p[z]] red, call RB-Insert-Fixup(T,p[p[z]]).
    - 

- Case 2: y is black z is right child.
    - Left rotate(T, p[z]). Now the original p[z] becomes z. We get case 3
- Case 3: y is black, z is left child
    - Make p[z] black, p[p[z]] red.
    - Right rotate on p[p[z]].
    - No further calls
    - 

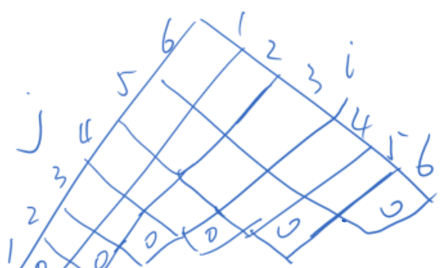
DP & Greedy

February 9, 2023 7:35 PM

Dynamic programming
- Optimal substructure
- Overlapping subproblems: memorization exploits this redundancy

Steps:
- Optimal substructure
- # subproblems
- Recursion
- Memorization: store a table and implement recursion using the table

e.g. Fibonacci numbers
- $F_0 = 0, F_1 = 1, F_n = F_{n-1} + F_{n-2}$.
- Easy to compute recursively, but lots of redundancies
- To get F_6 by recursion, requires solving F_3 3 times
- Memorization would store intermediate results and reuse

Problem 1: Matrix-chain multiplication (matrix parenthesization)
- e.g. $A_1 \in \mathbb{R}^{10\times 100}, A_2 \in \mathbb{R}^{100\times 5}, A_3 \in \mathbb{R}^{5\times 50}$, calculate $A_1A_2A_3$.
  - Option 1: $(A_1A_2)A_3$, #multiplication=$10\cdot 5\cdot 100 + 10\cdot 50\cdot 5 = 7500$ (final matrix size × multiplications needed for each cell).
  - Option 2: $A_1(A_2A_3)$, #multiplication=$100\cdot 50\cdot 5 + 10\cdot 50\cdot 100 = 75000$.
- Goal: fully parenthesize n matrices while minimizing total number of multiplications
- Input: $A_1, A_2, \dots, A_n$.
- Brute force: enumerate all possible parenthesizations
  - $P(n) = \sum_{k=1}^{n-1} P(k)P(n-k) = \Omega\left(\frac{4^n}{n^{3/2}}\right)$.
- Key idea: an optimal parenthesization for $A_1, \dots, A_n$ involves optimal parenthesization for L: $A_1, \dots, A_k$, and R: $A_{k+1}, \dots, A_n$ for some k.
- Proof of optimality: suppose L is not optimal, then exists some other $1 \le k' < k$ such that L is more optimal, and total number of multiplication is smaller.
- # subproblems= $O(n^2)$, since we require optimal on any subsequence $A_1, \dots, A_j$.
- Recurrence
  - Let A_i be a matrix with dimension $p_{i-1} \times p_i$.
  - $m[i,j]$ be the optimal value (minimized cost) for sub problem $A_i, \dots, A_j$.
    - $m[1,n]$ is the entire problem we want to solve.
  - $m[i,j] = \begin{cases} 0, i = j \\ \min_{k\in[i,j)}\{m[i,k] + m[k+1,j] + p_{i-1}p_jp_k\}, i < j \end{cases}$
- Memorization
  - A naïve recursive implementation and is inefficient (you do not expect redundancy).
  - Use a table to store intermediate results
  - e.g. A_1: 30×35, A_2: 35×15, A_3: 15×5, A_4: 5×10, A_5: 10×20, A_6: 20×25.
    - 

    - $m[1,6]$ (top) is what we want to get.
    - To get $m[2,5]$, we need $m[2,2], m[2,3], m[2,4], m[3,5], m[4,5], m[5,5]$.
  - The dependence dictates the order in which the table must be filled

- Runtime: $O(\#sub\ problems) \times O(time\ per\ sub\ problem) = O(n^2)O(n) = O(n^3)$.

Problem 2: longest common subsequence (LCS)

- Given sequences $X_m = x_1 \ldots x_m$, $Y_n = y_1 \ldots y_n$, find a subsequence common to both such that the subsequence length is maximal, not necessarily consecutive.
- e.g. X=springtime, Y=pioneer, result=pine.
- Brute force runtime: $O(n2^m)$.
- Theorem: suppose $Z_k = z_1 \ldots z_k$ is LCS of X_m and Y_n.
  - If $x_m = y_n$, then $z_k = x_m = y_n$ and Z_{k-1} is LCS of $x_1 \ldots x_{m-1}$ and $y_1 \ldots y_{n-1}$.
    - If not, can find a Z'_{k-1} such that $|Z'_{k-1} \cup \{z_k\}| > |Z_{k-1} \cup \{z_k\}|$.
  - If $x_m \neq y_n$, then $(z_k \neq x_m) \Rightarrow Z_k$ is LCS of X_{m-1} and Y_n.
  - If $x_m \neq y_n$, then $(z_k \neq y_n) \Rightarrow Z_k$ is LCS of X_m and Y_{n-1}.
- Recurrence:
  - Let $c[i,j]$ be the optimal length of LCS of X_i and Y_j, $c[m,n]$ is the optimal value for the problem.
  - $c[i,j] = \begin{cases} 0, i = 0\ or\ j = 0 \\ c[i-1,j-1]+1, x_i = y_j \\ \max\{c[i-1,j], c[i,j-1]\}, x_i \neq y_j \end{cases}$.
- Pseudo Code
  LCS(X,Y,m,n)
  For $i = 1{:}m$: $c[i,0] = 0$.
  For $j = 1{:}n$: $c[0,j] = 0$.
  For $i = 1{:}m$
  For $j = 1{:}n$
  If $x_i == y_i$, then $c[i,j] = c[i-1,j-1]+1$, tag with arrow pointing to $(i-1,j-1)$.
  Else if $c[i-1,j] > c[i,j-1]$, then $c[i,j] = c[i-1,j]$, tag with ↑.
  Else $c[i,j] = c[i,j-1]$, tag with ←.
- Runtime: $O(mn)$.

Greedy Algorithm

- Idea: when making a choice, take the one that looks the best right now
  - Locally optimal leads to globally optimal (need to prove)
- Greedy is not always optimal, but good as approximation algorithms
- Steps
  - Find optimal substructure
  - Prove Greedy Choice Property

Problem 1: activity selection

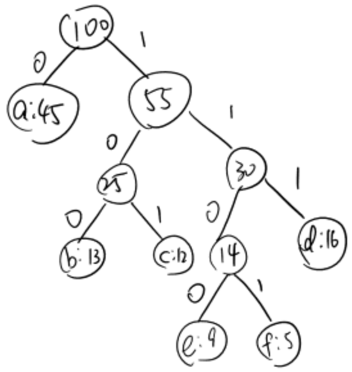
- Inputs: set of activities: $S = \{a_1, .., a_n\}$.
  - Each a_i needs resource during period $[s_i, f_i]$ where s_i is the start time, f_i is the finish time.
- Goal: select the largest possible set of mutually compatible activities.
- e.g. $t = [0,16]$, $a_1 = [1,3]$, $a_2 = [2,5]$, $a_3 = [4,7]$, $a_4 = [1,8]$, $a_5 = [5,9]$, $a_6 = [8,10]$, $a_7 = [9,11]$, $a_8 = [11,14]$, $a_9 = [13,16]$.
  - $S = \{a_1, \ldots, a_9\}$.
  - $S^{opt} = \{a_1, a_3, a_6, a_8\}$ (not unique).
- Greedy: at each step, from compatible activities, choose the one with smallest finish time.
- Optimal structure:
  - Let $S_{ij} = \{a_k \in S : f_i \leq s_k < f_k \leq s_j\}$ = activities that start after a_i finishes and finish before a_j starts.
  - $A_{ij} = opt\ sol\ to\ s_{ij}$.
  - $\{sol\ to\ s_{ij}\} = \{sol\ to\ s_{ik}\} \cup \{a_k\} \cup \{sol\ to\ s_{kj}\}$.
  - $A_{ij} = A_{ik} \cup \{a_k\} \cup A_{kj}$.
- Greedy Choice property:
  - Let $S_{ij} \neq \emptyset$ and a_m be activity in S_{ij} with earliest finish time, $f_m = \min\{f_k : a_k \in S_{ij}\}$.
  - a_m is used in some max-size(optimal) subset of compatible activities of S_{ij}.
    - Let A_{ij} be max size set of compatible activities in S_{ij}.
    - Order activities in A_{ij} in increasing order of finish time.

            - Let a_k be the first one in A_{ij}.
            - If $a_k = a_m$, done.
            - If $a_k \neq a_m$, then construct $A'_{ij} = A_{ij} - \{a_k\} \cup \{a_m\}$.
                - $\left|A'_{ij}\right| = \left|A_{ij}\right| - 1 + 1 = \left|A_{ij}\right|$.
            - Activities in A'_{ij} are still compatible, since a_k if the first in A_{ij} to finish, but $f_m \leq f_k$ ($a_k \neq a_m$ and a_m is min finish time in S_{ij}).
            - a_m doesn't overlap with $A_{ij} - \{a_k\}$.
            - A'_{ij} is optimal for S_{ij}, i.e. greedy is optimal.
        - $S_{im} = \emptyset$.
            - Suppose $\exists a_k \in S_{im}$.
            - $f_i \leq s_k < f_k \leq s_m < f_m$, then $f_k < f_m$, contradiction.
- Runtime: $O(n \log n)$.

Huffman coding (data compression)

-

	A	B	C	D	E	F
$F(c)$	45	13	12	16	9	5
$d(c_1)$ (fixed length coding)	000	001	010	011	100	101
$d(c_2)$ (variable)	0	101	100	111	1100	1101

- Must be prefix codes
- 

- $B(T) = \sum_c f(c)d(c)$ (number of bits needed to encode given input).
- Goal: to find T that minimizes $B(T)$.
- Greedy algorithm

HuffmanCoding

- Unite/merge the 2 lowest frequency characters, represent them as nodes in the tree
- Create new char in vocabulary representing the two chars merged
- Repeat until vocabulary is single char

Greedy Choice property:

- Consider 2 smallest frequency chars (x and y), show there exists optimal code tree in which x and y are max-depth siblings
- Proof:
    - Let T be any optimal prefix code tree with b and c the two siblings at max depth, assume $f(b) \leq f(c)$.
    - If $\{x, y\} = \{b, c\}$, done.
    - If $\{x, y\} \neq \{b, c\}$, then $f(x) \leq f(b)$ and $f(y) \leq f(c)$.
    - We know that b and c are deepest, $d_T(b) \geq d_T(x)$ and $d_T(c) \geq d_T(y)$.
    - First swap b with x to get T',
        - $B(T) = \sum_{c \neq b,x} f(c)d_T(c) + f(b)d_T(b) + f(x)d_T(x)$.
        - $B(T') = \sum_{c \neq b,x} f(c)d_T(c) + f(b)d_T(x) + f(x)d_T(b)$.
        - $B(T) - B(T') = \left(f(b) - f(x)\right)\left(d_T(b) - d_T(x)\right) \geq 0$.
        - So $B(T') \leq B(T)$.
    - Swap c with y to get T'', similary, we can show $B(T'') \leq B(T')$.
    - So $B(T'') \leq B(T)$.

Optimal structure + Greedy

- Let T_n be any tree that satisfies greedy choice property.
- Let T_{n-1} be the tree that results from replacing the two lowest frequency char and their parent with a single leaf z with frequency $f(z) = f(x) + f(y)$. We show that $B(T_n) = B(T_{n-1}) + f(z)$.
- Proof: Let d denote the depth of x, y in T_n, z is in depth $d - 1$ in T_{n-1}.
  - $B(T_n) = B(T_{n-1}) - (cost\ of\ z\ in\ T_{n-1}) + (cost\ of\ x\ and\ y\ in\ T_n)$,
  - $= B(T_{n-1}) - f(z)(d-1) + (f(x) + f(y))\,d$,
  - $= B(T_{n-1}) - f(z)(d-1) + f(z)d = B(T_{n-1}) + f(z)$.

Hashing

2023年2月16日 19:16

Let U be the universe, $K \subset U$ a set of keys, T a table of size m with indices $\{0,1,\dots,m-1\}$.
A hash function $h: U \to \{0,\dots,m-1\}$ hashes key k into index $h(k)$.

Desired from hashing scheme

- Simple uniform hashing
- Good mechanism for collision resolution
  - Chaining: if $h(x) = h(y)$, x, y are in the same list, (delete is easy).
  - Open addressing: if collision, use a probing sequence to find an empty slot (delete is not trivial).
    - Linear probing: $h(k,i) = (h'(k) + i) \ mod\ m$ when hashing key k for ith time.
    - Quadratic: $h(k,i) = (h'(k) + c_1 i + c_2 i^2) \ mod\ m$.
    - Double hashing: $h(k,i) = (h_1(k) + h_2(k)) \ mod\ m$.

Hashing design

- Multiplication: $\lfloor n(kA \ mod\ 1) \rfloor$, $A \in [0,1]$ constant.
- Division: $h(k) = k \ mod\ n$.

Analysis of chaining

- $n =$ #elements.
- $m =$#slots.
- Load factor: $\alpha = \frac{n}{m}$.
- If we assume simple uniform hashing (a key if equally likely to hash into any slot)
  - Worst case: single list of n element.
  - Expected case: $j = 0,1,\dots,m-1$, denote length of $T(j)$ by n_j,
    - then $n_0 + \cdots + n_{m-1} = n$.
    - $E[n_j] = \frac{n}{m}$, also assume $O(1)$ to compute h.
- Expected cost of search
  - Case 1: unsuccessful search $\theta(1+\alpha)$, compute the hash and search to end of list, taking $\theta(d)$.
  - Case 2: successful search.
    - # elements examined during successful for key x is one more than the number of elements before x in x's list=#elements that hash to same slot as x after x is hashed into slot.
    - For $i = 1,2,\dots,n$, let x_i be the ith element inserted into the table and k_i is key(x_i).
    - $\forall i, j$, define $X_{ij} = 1\left\{h(k_i) = h(k_j)\right\}$.
    - Simple uniform hashing $\Rightarrow \Pr\left(h(k_i) = h(k_j)\right) = \frac{1}{m} \Rightarrow E\left[X_{ij}\right] = \frac{1}{m}$.
    - $E\left[\frac{1}{n}\sum_{i=1}^{n}\left(1 + \sum_{j=i+1}^{n} X_{ij}\right)\right] = \frac{1}{n}\sum_{i=1}^{n}\left(1 + \sum_{j=i+1}^{n}\frac{1}{m}\right) = 1 + \frac{n-1}{2m}$.
      - $\sum_{j=i+1}^{n} X_{ij}$ is # elements after i that collides with i.
    - $= 1 + \frac{\alpha}{2} - \frac{\alpha}{2n} = \theta(1+\alpha)$.

For any h, if $|U| \geq (n-1)m + 1$ then there is set S of n elements that all hash to same slot.

- Proof: contrapositive, if every slot had at most $n-1$ element of U hashing to it, then $|U| \leq m(n-1)$.

Universal hashing

- A randomized algorithm H for constructing hash function $h: U \to \{0,\dots,m-1\}$ is universal if $\forall x \neq y \in U$, it holds that $\Pr[h(x) = h(y)] \leq \frac{1}{m}$.
- Theorem: if H is universal, then $\forall S \subset U$ with $|S| = n$, $\forall x \in U$, the expected number of

collision between x and other elements in $S \leq \frac{n}{m}$.

- Corollary: if H is universalm, any sequence of λ operations (insert, search, delete) has expected total cost $O(\lambda)$.

Construction of universal hash family (matrix based)

- Assume keys are u bits long, table size m is power of 2, index is b bits ($m = 2^b$).
- Algo: choose n to be a random $b \times u$ 0/1 matrix and have $h(x) = h \cdot x$, where addition is mod 2.
- Claim: $x \neq y$, $\Pr[h(x) = h(y)] = \frac{1}{m} = \frac{1}{2^b}$.
  - In worst case, only 1 bit is different, select the column in the matrix.
  - 2^b combinations, each of them creates different output.

Amortized Analysis

March 3, 2023 8:01 PM

Unlike best/worst/expected case for single operations. Here we care about average cost/operation in sequence of operations

- Aggregate: simple to understand/calculate for simple data structure.
- Accounting: identify cheap/expensive operations. Use cheap operations to justify expensive cost
  - Charge $k for each operation (amount is amortized cost for each operation)
  - Goal is to maintain a credit invariant
  - If amortized cost > actual cost, remain difference in deposit
  - If amortized cost < actual cost, use credit stored to compensate (pay) for difference
  - Should never end up with negative credit (if not enough, bump up the deposit)
- Potential (not used)

Stack

	Actual cost	Amortized cost
Push(x)	O(1)	2
Pop()	O(1)	0
Multipop(k)	O(k)	0
- Sequence of push/pop/multipop operations (n operations)
- Naïve:
  - $O(nk)$ total, so $O(k)$ average. Wrong since to have multiple, we must have pushed k times.
- Aggregate
  - You never pop more than you push.
  - $O(n)$ total, so $O(1)$ average.
- Accounting
  - Charge $2 for each push. $1 for actual cost of push, $1 stays as credit.
  - Charge $0 for each pop. $1 credit in pushed elements pays for cost of pop
  - Charge $0 for multipop. $k credit in k pushed elements pay for the cost
- if multipop(k) is $O(k^3)$, need to consider $O(n^2)$.
- Queue is the same

Counter

- k-bit counter $A[0, \dots, k-1]$, $A[0]$ is the least significant bit.
- Increment(A, k)

  $i = 0,$

  While $i < k$ and $A[i] == 1$:

  $A[i] = 0,$

  $i = i + 1.$

  If $i < k$: $A[i] = 1.$
- Naïve: $O(k)$ per operation.
Cost	A
0	0:000
1	1:001
3	2:010
4	3:011
7	4:100
8	5:101

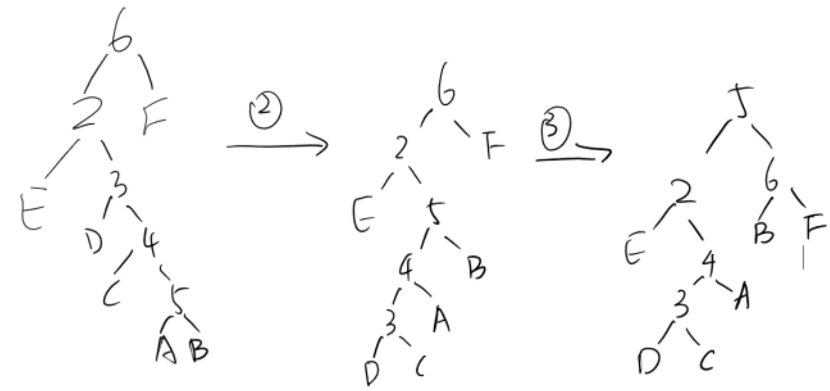
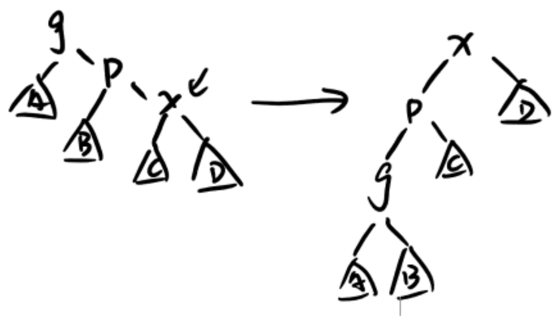
10	6:110
11	7:111

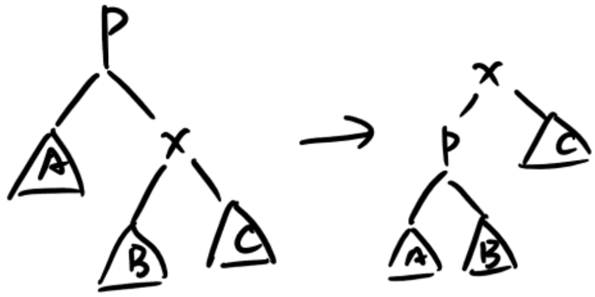
- LSB flips everytime
- ith bit flips $\frac{n}{2^i}$ times.
- Aggregate: $\#flips = \sum_{i=0}^{k-1} \left\lfloor \frac{n}{2^i} \right\rfloor \leq n \sum_{i=0}^{k-1} \frac{1}{2^i} = n \cdot \frac{1}{1-\frac{1}{2}} = 2n.$
    - $O(n)$ total, $O(1)$ amortized.
- Accounting method
    - Charge \$2 for every 1 we set ($0 \to 1$).
    - Every increment costs \$2 because there's only one single $0 \to 1$ flip
    - Every $1 \to 0$ flip is paid for by the \$1 credit left after the $0 \to 1$ flip
    - For n operations, $O(1)$ per operation.
- Binary counter with reset
Operation	Actual cost	Amortized cost
increment	$O(n)$	\$3
reset	$O(n)$	\$0
    - The number of bits used by the counter will be less than the number of increment operations.
    - If not, charge \$4 for increment and \$1 for reset
    - \$1 pays for flipping 0 to 1, \$1 saved for flipping 1 to 0.
    - \$1 to update max, \$1 to pay for flipping to a 0 during reset.
- Ternary counter (increment by 3)
    - Charge \$3 per increment.
    - Invariant: A trit with value 0 has \$0 credit, value 1 has \$2 credits, value 2 has \$1 credit.
    - At most one 0-1 flip, \$1 from the charge pays for the flip. Remaining \$2 stored as credit.
    - Increment changes states in the order 0-1-2-0. Credit used to do 1-2 and 2-0.

Dynamic hash table

- Insert
    -
      ```
      TABLE-INSERT(T, x)
      if T.size == 0
          allocate T.table with 1 slot
          T.size = 1
      if T.num == T.size
          allocate new-table with 2 · T.size slots
          insert all items in T.table into new-table
          free T.table
          T.table = new-table
          T.size = 2 · T.size
      insert x into T.table
      T.num = T.num + 1
      ```
    - Aggregate:
        - Cost of ith insert $c_i = \begin{cases} i, i-1 = 2^k \\ 1, else \end{cases}$.
        - $\sum_{i=1}^{n} c_i \leq n + \sum_{j=0}^{\lfloor \log n \rfloor} 2^j < n + 2n = 3n.$
        - Amortized $O(1)$ on average.
    - Accounting:
        - Charge \$3 on insert.
        - \$1 used for insert.
        - \$1 store as credit.
        - \$1 stored for $\frac{m}{2}$ items already in the table.
        - Each \$1 pay for it to be reinserted during the expansion.
- Delete
    - Shrink the table size when $T.num \leq T.size/2$.
- Amortized cost of each operation is bounded above by a constant. The actual time for any sequence of n operations on a dynamic table is $O(n)$.

Splay tree

- Weighted dictionary problems: given keys $\langle x_1, \dots, x_n \rangle$ and frequencies $\langle w_1, \dots, w_n \rangle$, the goal is to minimize cost of accessing high frequency elements.
    - If w_i known a priori, then we can build a static optimal tree using dynamic programming in $O(n^3)$.
    - If w_i not known, splay tree, $O(\log n)$ average cost for insert/delete/search.
- Properties
    - No explicit balancing conditions.
    - BST property holds.
    - Pre-emptively rotate element that is accessed until it becomes the root.
- SPLAY(x)
  While x is not the root:
  If $p(x)$ is the root: rotate $p(x)$,
  Else if $p(x)$, x both left or right children: rotate $p(p(x))$, then rotate $p(x)$.
  Else: rotate $p(x)$, then rotate at new $p(x)$.
- 
- Cost of splay
    - Let $w(x)$ be the number of nodes in subtree rooted at x plus x itself.
    - Define $rank(x) = \lceil \log(w(x)) \rceil$.
    - Credit invariant: every node has $rank(x)$ credit on it.
    - We need to show that every SPLAY operation can be paid with $O(\log n)$ additional credit to account for rotations and maintain the invariant.
    - Claim: every operation in while loop costs $3(newrank(x) - oldrank(x))$ except for $p(x)$ =root case, which needs $+1$ credit.
        - Proof:
        - Case 2 and 3
            - 
            - Compare $or(x) + or(p) + or(g)$ with $nr(x) + nr(p) + nr(g)$.
            - $nr(p) \le nr(x), nr(g) \le nr(p), nr(x) = or(g), or(p) \ge or(x)$.
            - $nr(x) + nr(p) + nr(g) - or(x) - or(p) - or(g) \le 2(nr(x) - or(x))$.
            - Amount charged covers this cost.
            - If $nr(x) = or(x)$, more than half of tree nodes were under x. Otherwise its rank would have incresed
                - Less than half of the nodes are in A and B.
                - $rank(g)$ is reduced by at least 1.
                - Leftover credit on g pays for costs of rotations.

        - Case 1:
            - $or(x) \le nr(x)$, $or(p) \ge nr(p)$.
            - $nr(x) + nr(p) - or(p) - or(x) \le nr(x) - or(x)$.
            - If $nr(x) = or(x)$, we don't know if p's rank is affected/reduced.
            - Pay \$1 for the rotation.
            - 

    - Let $rank_0, \dots, rank_k$ be the sequence of ranks for x until x becomes root. We need $1 + 3(rank_k - rank_{k-1}) + \dots + 3(rank_1 - rank_0) = 1 + 3(rank_k - rank_0)$.
    - But $rank_k \le \lceil \log n \rceil$, so credit required $\le 1 + 3\log n$, which is $O(\log n)$ amortized.

Average cost: mean over all possible inputs
Expected cost: assume uniform, then same as average
Amortized cost: average over a particular sequence of inputs.

Worst cast upper bound: $\hat{c} \ge c(x_i), \forall i$.
Amortized upper bound: $\frac{1}{n}\sum \hat{c} \ge \frac{1}{n}\sum_{i=1}^{n} c(x_i), \forall n$.

Aggregate analysis
- Given an operation $f(x)$ and a sequence $(x_1, \dots, x_n)$, let $c(x_i)$ be the cost of $f(x_i)$.
- Compute $T(n) = \sum_{i=1}^{n} c(x_i)$.
- Amortized cost: $\frac{T(n)}{n}$.

Accounting method
- Declare that $\$\hat{c}$ will be charged per operation
- Describing a procedure for how we use $\hat{c}$.
- Assert a credit invariant (some claim about the stored credit in the data structure).
- Argue that the credit invariant is true.
- Use the credit invariant to argue why the credit is never negative.

E.g. (array doubling) suppose $f(x)$ has cost $c(x) = \begin{cases} x, x = 2^m \\ 1, else \end{cases}$.
- Aggregate method:
    - $\sum_{x=1}^{n} c(x_i) = \sum_{x \ne 2^m} 1 + \sum_{x=2^m} x = n + \sum_{m=0}^{\log n} 2^m = n + (2n-1) < 3n = O(n)$.
    - Amortized cost: $\frac{O(n)}{n} = O(1)$.
- Accounting method
    - Charge \$3 for each operation
    - If $x \ne 2^m$, use \$1 to pay for operation and store \$2
    - If $x = 2^m$, store \$2, and use the stored \$x to pay for the operation.
    - Credit invariant: when $x = 2^m$, all elements in the range $(2^{m-1}, 2^m]$ have \$2 stored.
        - True by construction
    - $\$2(2^{m-1}) = \$2^m$, since $x = 2^m$, we have exactly enough, so never go negative.
    - Amortized cost is $O(3) = O(1)$.

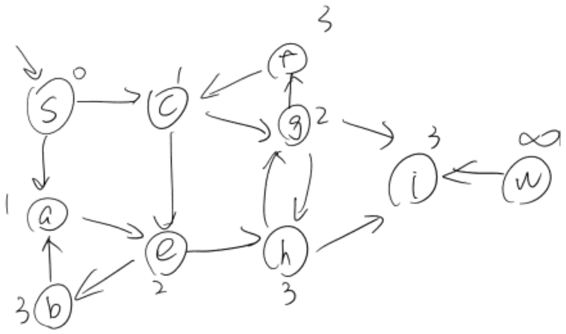
Graph Algorithms

March 3, 2023 8:01 PM

Graph $G = (V, E)$, size $|V|, |E|$.
Representation
- Adjacency list:
  - Space: $O(V + E)$.
  - Check edge (u, v) $O(\deg(u))$.
- Adjacency matrix:
  - Space: $O(V^2)$.
  - Check edge (u, v) $O(1)$.

Breadth-First-Search (BFS)
- Input: $G = (V, E)$ directed/undirected, source vertex $s \in V$.
- Output:
  - $d[v]$: distance from s to v, $\forall v \in V$.
  - $\pi[v]$: v's predecessor.
- Idea: start at s, and in each iteration i, visit nodes that are i edges away from s.
- BFS(V,E,s)

```
For each u ∈ V − {s}:
    d[v] = ∞.
d[s] = 0.
Q = ∅ (FIFO).
Enqueue(Q,s).
While Q ≠ ∅:
    u =Dequeue(Q)
    For each v ∈ adj(u):
        If d[v] = ∞:
            d[v] = d[u] + 1;
            π[v] = u;
            Enqueue(Q,v);
```

- BFS may not reach all vertices
- Runtime: $O(V + E)$.
- 

Depth-First-Search (DFS)
- Input: $G = (V, E)$ directed/undirected.
- Output:
  - $d[v]$: discovery time.
  - $f[v]$: finishing time.
- Idea: as soon as we discover a vertex, we explore from it. Every vertex has one of three colors as DFS progresses
  - White: undiscovered
  - Gray: discovered but not done exploring from
  - Black: finished
- DFS(G)

```
For each u ∈ V:
    Color[u]=white
Time=0;
For each u ∈ V:
    If color[u]==white:
        DFS-VISIT(G,u)
```

- DFS-Visit(G,u)

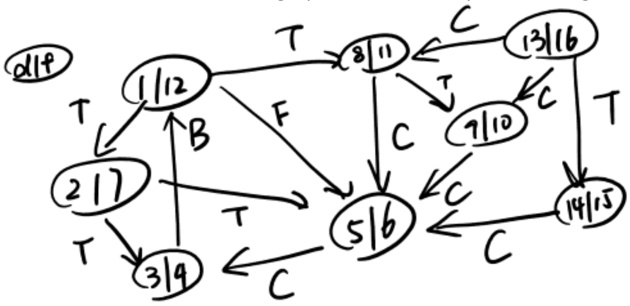
```
Time=time+1
d[u] =time
Color[u]=gray
For each v ∈ adj(u):
    If color[v]==white:
```

DFS-Visit(G,v)

Color[u]=black

Time=time+1

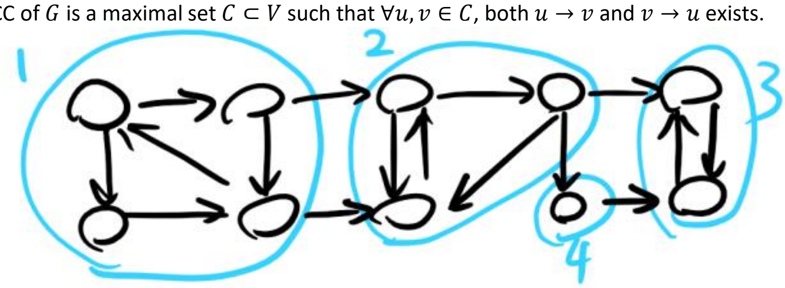
$f[u]$ =time

- Runtime: $\theta(V+E)$.
- Edge classification
  - Tree edge: edges in the depth first forest found when exploring (u,v).
  - Back edge: (u,v) where u is descendant of v.
  - Note: v is a descendant of u if and only if at time $d[u]$, $\exists u \to v$ consisting of only white vertices.
    - u is discovered first while none of the vertices on $u \to v$ is discovered.
  - Forward edge: (u,v) where v is descendant of u, but not tree edge.
  - Cross edge: any other edge.
- Parenthesis theorem: $\forall u, v$, the following cannot happen: $d[u] < d[v] < f[u] < f[v]$.
  - v must finish before u.
- Theorem: in DFS of undirected graph, there are only T and B edges.
- 

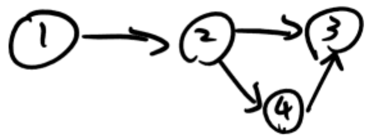
Topological sort

- Works on directed acyclic graphs (DAGs). DAGs model partial order
  - $a > b$ and $b > c \Rightarrow a > c$.
  - But may have a and b such that neither $a > b$ nor $b > c$.
- Topo sort produces a total order that respects partial order
- Lemma: a directed graph G is acyclic if and only if DFS yields no back edges.
  - Proof ($\Rightarrow$): if $\exists(u,v)$ that is a back edge, then $\exists$ path $v \to u$ and $v \to u - v$ is a cycle.
  - ($\Leftarrow$) suppose G contains a cycle. Let v be the first vertex discovered in that cycle, and let (u,v) be preceding edge in the cycle.
    At time $d[v]$, vertices of the cycle form a white path $v \to u$.
    By white path theorem, u is descendant of v, (u,v) is a back edge.
- Topo-sort(G):
  DFS(G) gives $f[v]$ $\forall v$.
  Output vertices in order of decreasing finish time
- Runtime: $\theta(V+E)$.
- Correctness proof: show if $(u,v) \in E$, then $f_u > f_v$.
  - When we explore (u,v), what are colors of u, v.
  - u is gray.
  - v cannot be gray, otherwise v would be ancester of u, (u,v) is a back edge, and we get a cycle (contradiction).
  - v can be white, v is the decendant of u in DFS tree, $d_u < d_v < f_v < f_u$.
  - v can be black (finished), $f_v < d_u < f_u$.

Strongly Connected Components (SCCs)

- Given directed $G = (V, E)$.
- SCC of G is a maximal set $C \subset V$ such that $\forall u, v \in C$, both $u \to v$ and $v \to u$ exists.
- 

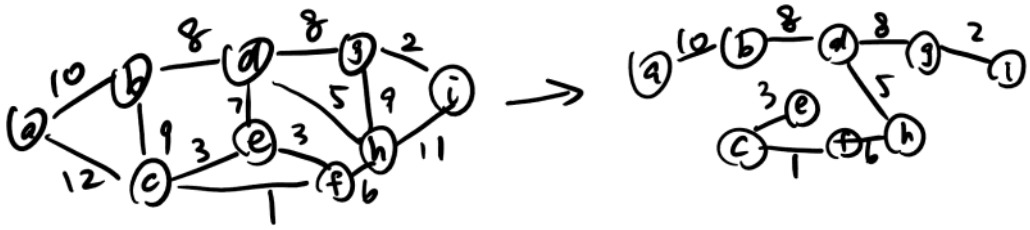
- Definition
  - G^T=transpose of G, $G^T = (V, E^T)$ such that $E^T = \{(u,v) : (v,u) \in E\}$.
    - G^T and G have the same SCCs.
    - Runtime: $\theta(V+E)$.
  - $G^{SCC} = (V^{SCC}, E^{SCC})$ component graph.
    - V^{SCC} has one vertex per SCC.
    - E^{SCC} has edge if $\exists$edges between components.

- 

- G^{SCC} is DAG.
  - Proof: let C, C' be distinct SCCs and $u, v \in C$, $u', v' \in C$ and suppose $\exists u \to u' \in G$. Then we show there is n $v' \to v$.
  - Suppose $\exists v' \to v$, then there is $u \to u' \to v' \to v \to u$, so u, v' are reachable from each other.
  - C, C' not maximal, contradiction.
- SCC(G):

  DFS(G) and compute $f_u, \forall u$.
  Compute G^T.
  DFS(G^T), but in main loop, visit nodes in decreasing order of f_u.
  Output vertices of each DFS(G^T) tree as separate SCCs.
- Runtime: $\theta(V + E)$.

Minimum spanning trees (MSTs)

- Input: undirected $G = (V, E)$, weight $w(u, v)$ for each edge $(u, v) \in E$.
- Goal: find a tree $T \subset E$ such that T connects all vertices and $w(T) = \sum_{(u,v)\in T} w(u, v)$ is minimized.
- 
- MST facts
  - $|V| - 1$ edges.
  - No cycles
  - Not necessarily unique
- Generic-MST(G,w):

  $A = \emptyset$;
  While A is not a spanning tree:
      Find safe edge (u, v).
      $A = A \cup \{(u, v)\}$.
  Return A.
- Proof:
  - A: set of edges (initially empty).
  - Expanding A by maintaining loop invariant (A is a subset of some MST).
  - Edges that maintain invariant:
    - If $A \subset$MST, (u, v) is safe if and only if $A \cup \{(u, v)\} \subset$MST.
- Definitions:
  - Cut(S,V-S) is a partition of V into disjoint sets $S, V - S$.
  - Edges $(u, v) \in E$ crosses cut(S,V-S) if one of (u, v) is in S and the other in $V - S$.
  - Cut respects A if and only if no edge in A crosses the cut.
  - An edge is light edge crossing cut if and only if its weight is minimum across all edges crossing the cut.
- Theorem: let $A \subset$MST, cut$(S, V - S)$ respecting A and (u, v) light edge crossing $(S, V - S)$, then (u, v) is safe for A.
  - Proof: let T be MST that includes A.
  - If T contains (u, v), done.
  - Assume T does not contain (u, v), we will construct T' that includes $A \cup \{(u, v)\}$.
  - T is MST, then exists unique path p from u to v.
  - Path p must cross (S,V-S) at least once. Let (x, y) be the edge of p that cross the cut.
  - We choose (u, v) to be light, so $w(u, v) \le w(x, y)$.
  - Since cut(S,V-S) respects A, then $(x, y) \notin A$.
  - To form T' from T, remove (x, y) to break T into 2 components, then add (u, v) to combine.
  - $T' = T - \{(x, y)\} \cup \{(u, v)\}$, $w(T') = w(T) - w(x, y) + w(u, v)$, $w(T') \le w(T)$, T' is MST.
  - Need to show that (u, v) is safe for A.
  - $A \subset T$ and $(x, y) \notin A$, so $A \subset T'$.
  - $A \cup \{(u, v)\} \subset T'$, since T' is MST, $A \cup \{(u, v)\} \subset$MST.
- If weights of edges are all unique, then there is only one MST. Reverse doesn't hold.

Kruskal's

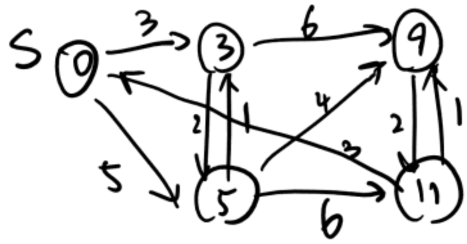
- Each vertex is its own component initially.
- Merge 2 components by choosing light edge, scanning edges in monotonically non-decreasing order.
- Uses disjoint set data structure to ensure edges cross different components.
- Runtime: $O(E \log E)$.

Prim's

- Expands a tree (A is always a tree).
- Each step, find light edge crossing $(V_A, V - V_A)$, where V_A is the set of vertices A is incident on.
- Use a priority queue Q.
  - Each element corresponds to a vertex in $V - V_A$.
  - Key[v] is min weight of any edge (u, v) such that $u \in V_A$.
- Prim(V,E,w,r).

```
# r is an arbitrary root.
Q = ∅.
Foreach u ∈ V:
    Key[u]=∞;
    π[u] = NIL;
    Insert(Q,u);
Decrease-key(Q,r,0) # set key[r]=0
While Q ≠ ∅:
    u=Extract-min(Q);
    For v ∈ adj(u):
        If v ∈ Q and w(u,v) < key[v]:
            π[v] = u;
            Decrease-key(Q,v,w(u,v));
```

- Runtime
  - Assume Q is a binary heap.
  - Initialization: $O(V \log V)$.
  - Decrease-key: $O(\log V)$.
  - While loop.
    - Extract-min V times: $O(V \log V)$.
    - Decrease-key E times: $O(E \log V)$.
  - Total: $O(E \log V)$.
    - $O(V \log V + E)$ if Fibonacci heaps.

Shortest path
- Input: directed $G = (V, E)$, weight function $w: E \to \mathbb{R}$.
- Def:
  - Weight of path $p = \langle v_0, v_1, \dots, v_k \rangle$ is $\sum_{i=1}^{k} w(v_{i-1}, v_i)$.
  - Shortest path weight from u to v is $\delta(u, v) = \begin{cases} \min\{w(p)\}, if\ \exists p: u \to v \\ \infty, otherwise \end{cases}$.
- 

- Optimal solution (shortest path tree) is not unique
- Variants
  - Single source.
  - Single destination.
  - Single pair
  - All pairs shortest path $u \to v, \forall u, v$.
- Negative weight edges
  - OK as long as no neg-weight cycle reachable from source
  - Some algorithms only work with positive weight edges.
- Cycles: Algorithms will not output shortest path with cycles
- Output:
  - for each $v \in V$, $d[v] = \delta(s, v)$.
    - Initially, $d[v] = \infty$, reduces as algorithm progresses.
  - $\pi[v]$ =predecessor of v in shortest path tree.
- Init-single-source(V,s)

```
For each v ∈ V:
    d[v] = ∞;
    π[v] = NIL;
d[s] = 0.
```

- Relax(u,v,w):

```
If d[v] > d[u] + w(u,v):
    d[v] = d[u] + w(u,v);
    π[v] = u.
```

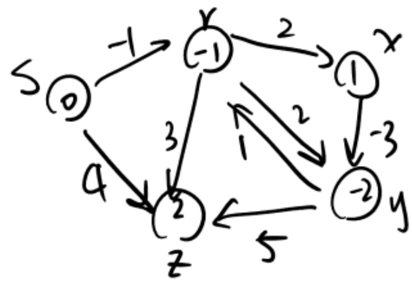
- Properties
  - Optimal substructure: any subpath of a shortest path is a shortest path
    - If p_{uv} is shortest path, then p_{ux}, p_{xy}, p_{yv} are shortest path for x, y on $u \to v$.
    - Proof similar to Greedy, DP cut based approach.

- Triangle inequality: $\forall (u,v) \in E, \delta(s,v) \le \delta(s,u) + w(u,v)$.
    - Proof: $\delta(s,v)$ is the shortest path, must be shorter than $s \to u \to v$, $\forall u$ by definition.
- Upper bound property: always have $d[v] \ge \delta(s,v)$, $\forall v$. Once $d[v] = \delta(s,v)$, it never changes.
    - Proof: initially true. Assume $\exists v$ s.t. $d[v] < \delta(s,v)$ and WLOG, assume v is the first vertex for which this happens.
    - Let u be the vertex that causes $d[v]$ to change.
    - Then $d[v] = d[u] + w(u,v)$, $d[v] < \delta(s,v) \le \delta(s,u) + w(u,v)$.
    - Since u is not a violator, $d[u] \ge \delta(s,u)$. Then $d[v] < d[u] + w(u,v)$, contradiction.
    - Once $d[v] = \delta(s,v)$, the assertion in Relax will be false.
- No-path property: if $\delta(s,v) = \infty$, then $d[v] = \infty$ (because of upper bound property).
- Convergence property: If $s \to u \to v$ is a shortest path, $d[u] = \delta(s,u)$ and call Relax(u,v,w), then $d[v] = \delta(s,v)$ afterwards.
    - After relaxation, $d[v] \le d[u] + w(u,v) = \delta(s,u) + w(u,v) = \delta(s,v)$ by optimal substructure.
    - Since $d[v] \ge \delta(s,v)$ by upper bound property, then $d[v] = \delta(s,v)$.
- Path relaxation property: Let $p = \langle v_0, v_1, \dots, v_k \rangle$ be a shortest path from v_0 to v_k. If we relax in the order (v_0, v_1), (v_1, v_2),….,(v_{k-1}, v_k), even mixed with other relaxation. Then $d[v_k] = \delta(v_0, v_k)$.
    - Apply convergence property from $i = 1$ iteratively.

Bellman-Ford

- Allows neg-weight cycles
- Returns True if no neg-weight cycle reachable from s, False otherwise. Can also compute the shortest path from s to any other vertex in the graph.
- Bellman-Ford(V,E,w,s)

```
Init-single-source(v,s)
For i = 1:|V| - 1:
    For each edge (u,v) ∈ E:
        Relax(u,v,w)
For each edge (u,v) ∈ E:
    If d[v] > d[u] + w(u,v):
        Return False.
Return True
```

- Runtime: $O(VE)$.
- Proof of correctness
    - For $d = \delta$, path relaxation property.
    - For True/False
        - No neg-weight cycle: $d[v] = \delta(s,v) \le \delta(s,u) + w(u,v) = d[u] + w(u,v)$.
            - Returns True
        - If there is a neg-weight cycle $C = \langle v_0, v_1, \dots, v_k \rangle$ with $v_0 = v_k$, reachable from s, $\sum_{i=1}^{k} w(v_{i-1}, v_i) < 0$.
            - Assume it returns True, then $d[v_i] \le d[v_{i-1}] + w(v_{i-1}, v_i)$, $\forall i = 1, \dots, k$.
            - Sum around C, $\sum_{i=1}^{k} d[v_i] \le \sum_{i=1}^{k} d[v_{i-1}] + \sum_{i=1}^{k} w(v_{i-1}, v_i)$.
            - Since $\sum_{i=1}^{k} w(v_{i-1}, v_i) < 0$, $\sum_{i=1}^{k} d[v_i] < \sum_{i=1}^{k} d[v_{i-1}]$, but $\sum_{i=1}^{k} d[v_i] = \sum_{i=1}^{k} d[v_{i-1}]$ for a cycle, contradiction.
- Example
    - 

Edge order	(r,x)	(x,y)	(y,r)	(y,z)	(r,y)	(s,z)	(s,r)	(r,z)
Iter 1	0	0	0	0	0	1	1	1
Iter 2	1	1	0	0	0	0	0	0
Iter 3	0	0	0	0	0	0	0	0

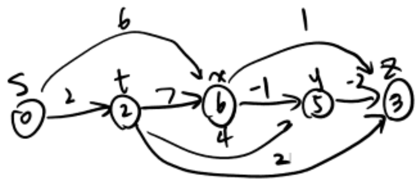
- 0 means no update, 1 means update

Single Source Shortest Paths in Direct Acyclic Graphs (SSSPs in DAGs)

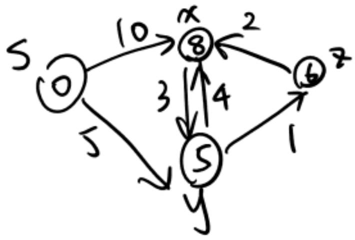
- DAG-Shortest-Paths(V,E,w,s)

```
Topological sort (θ(V + E))
Init-Single-Source(V,s) (θ(V))
Foreach u in topological order: (θ(E))
    Foreach v ∈ adj(u):
        Relax(u,v,w).
```

- Runtime: $\theta(V + E)$.

- 

Dijkstra's algorithm

- No negative-weight edges
- Idea:
  - Maintain a priority queue Q, with keys=$d[*]$ estimates.
  - S=vertices where final shortest path distance is determined.
  - $Q = V - S$.
- Dijstra(V,E,w,s)
  Init-Single-Source(V,s)
  $S = \emptyset$;
  $Q = V$;
  While $Q \neq \emptyset$:
      u=Extract-min(Q);
      $S = S \cup \{u\}$;
      Foreach $v \in adj(u)$:
          Relax(u,v,w) (Requires Decrease-Key)
- 

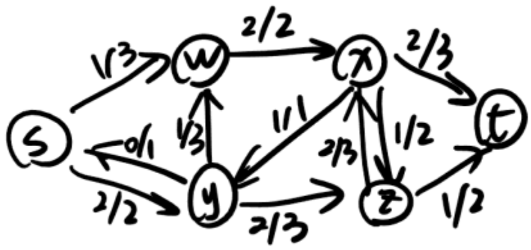
- Proof of correctness
  - Need to show that $d[u] = \delta(s,u)$ when u is added to S.
  - Assume $\exists u$ such that $d[u] \neq \delta(s,u)$. WLOG, let u be the first vertex for which this happens when u is added to S.
    - $u \neq s$, $d[s] = 0 = \delta(s,s)$, $s \in S$, $s \neq \emptyset$.
    - u is reachable from s, otherwise $d[u] = \delta(s,u) = \infty$. (there exists a shortest path from s to u)
  - Just before u is added to S, path $p: s \to u$ connects a vertex in S to a vertex in $V - S$.
  - Let y be the first vertex along p that is in $V - S$, x be the predecessor
  - Let $p_1: s \to x$, $p_2: y \to u$, $p = p_1 + (x,y) + p_2$.
  - Claim: $d[y] = \delta(s,y)$ when u is added to S.
    - $x \in S$ and u is the first vertex such that $d[u] \neq \delta(s,u)$, then $d[x] = \delta(s,x)$.
  - Relax (x,y) at that time, then $d[y] = \delta(s,y)$ by convergence property.
  - y is on shortest path $s \to u$, and all edge weights are positive.
    - Then $\delta(s,y) \leq \delta(s,u)$.
  - So $d[y] = \delta(s,y) \leq \delta(s,u) \leq d[u]$.
  - Observe y and u were in Q when we choose u, thus $d[u] \leq d[y]$, thus $d[u] = d[y]$.
  - $d[y] = \delta(s,y) = \delta(s,u) = d[u]$, contradiction.
- Runtime: $O((V+E)\log V)$.

Difference constraints

- Build constraint graph (weighted, directed)
- $V = \{v_0, v_1, \dots, v_n\}$: one vertex per variable, v_0 is pseudo-start.
- $E = \{(v_i, v_j): x_j - x_i \leq b_k \text{ a constraint}\} \cup \{(v_0,v_1),(v_0,v_2),\dots,(v_0,v_n)\}$.
- $w(v_0, v_i) = 0$.
- $w(v_i, v_j) = b_k$ if $x_j - x_i \leq b_k$.
- Theorem:
  - If G has no negative weight cycle, then $x = (\delta(v_0,v_1), \delta(v_0,v_2), \dots, \delta(v_0,v_n))$ is a feasible soltuion.
  - If G has a neg-weight cycle, then no solution.
- $x_j \leq x_i + b_k$ is equivalent to $d[v_j] \leq d[v_i] + w(v_i, v_j)$.
- Build graph and run Bellman-Ford.
  - Runtime: $O(VE)$.

Maximum flow

- $G = (V,E)$ directed, each edge (u,v) has a capacity $c(u,v) \geq 0$.
- Source vertex s, sink vertex t, and assume $\exists p: s \to u \to t$, $\forall u \in V$.

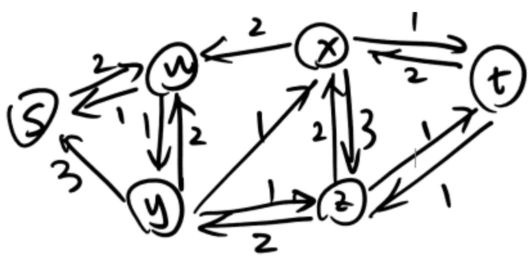
- 

    - In the graph: $f(s,w)=1$, $f(w,s)=-1$, even there is no edge (w,s).
    - For x, $\sum_{v\in V} f(x,v) = f(x,s)+f(x,y)+f(x,w)+f(x,z)+f(x,t) = 0+1+(-2)+(1-2)+2=0$.
        - Input =4 from w,z, output=4 to y,z,t.
    - $|f|=3$ (output from $s=3$, input to $t=3$).
- Net flow: $f: V\times V \to \mathbb{R}$ such that
    - Capacity constraint: $\forall u,v\in V$, $f(u,v)\le c(u,v)$.
    - Skew symmetry: $\forall u,v\in V$, $f(u,v)=-f(v,u)$.
    - Flow conservation: $\forall u\in V-\{s,t\}$, $\sum_{v\in V} f(u,v)=0$.
- Value of flow f: $|f|=\sum_{v\in V} f(s,v)$= total flow out from s.
    - Value comes from s goes to t.
- Cancellation:
    - 5 units $u\to v$ with 0 units $v\to u$ is equivalent to 8 units $u\to v$, 3 units $v\to u$.

Maximum flow problem:
- Given G,s,t,c, find $|f|$ that is maximum.
- Implicit summation: if X,Y are sets of vertices $f(X,Y)=\sum_{x\in X}\sum_{y\in Y} f(x,y)$.
- Flow conservation: $f(u,V)=0$, $\forall u\in V-\{s,t\}$.
- Lemma: for any flow in $G=(V,E)$.
    - $\forall X\subset V$, $f(X,X)=0$.
    - $\forall X,Y\subset V$, $f(X,Y)=-f(Y,X)$.
        - Proof: $f(X,Y)=\sum_x\sum_y f(x,y)=\sum_x\sum_y -f(y,x)=-\sum_y\sum_x f(y,x)=-f(Y,X)$.
    - $\forall X,Y,Z\subset V$ such that $X\cap Y=\emptyset$, $f(X\cup Y,Z)=f(X,Z)+f(Y,Z)$, $f(Z,X\cup Y)=f(Z,X)+f(Z,Y)$.

  Lemma: $|f|=f(s,V)=f(V,t)$.
    - Proof:
    - (i) show $f(V,V-s-t)=0$.
        - $f(u,V)=0$, $\forall u\in V-\{s,t\}$, so $f(V-s-t,V)=0$ (sum up on $V-s-t$), then $f(V,V-s-t)=0$ by skew symmetry.
    - $|f|=f(s,V)=f(V,V)-f(V-s,V)=-f(V-s,V)=f(V,V-s)=f(V,V-s-t)+f(V,t)=f(V,t)$.
        - Since $f(V,V)=f(V,V-s-t)=0$.

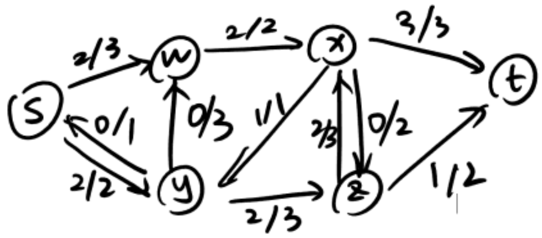
Cut:
- A cut (S,T) of G is a partition of V into S, $T=V-S$ such that $s\in S$, $t\in T$.
- For flow f, net flow across $(S,T)=f(S,T)$, capacity of $(S,T)=c(S,T)$.
- e.g. in the same graph above, let $S=\{s,w,y\}$, $T=\{x,z,t\}$.
    - $f(S,T)=f(w,x)+f(y,z)+f(y,x)=2+2-1=3$.
    - $c(S,T)=c(w,x)+c(y,z)=5$ (directional, only consider the path from S to T).
- Lemma: for any cut (S,T), $f(S,T)=|f|$.
- Corollary: the value of any flow $\le$ capacity of any cut ($|f|\le c(S,T), \forall S,T,f$).
    - Max flow $\le$ capacity of min cut

Residual network
- Given flow f in $G=(V,E)$, residual capcity: $c_f(u,v)=c(u,v)-f(u,v)\ge 0$.
- Residual network $G_f=(V,E_f)$ where $E_f=\{(u,v)\in V\times V : c_f(u,v)>0\}$.
- E.g.
    - 

    - Note: $f(w,s)=-1$, $c(w,s)=0$, $c_f(w,s)=1>0$.
    - $c_f(y,s)=c(y,s)-f(y,s)=1-(-2)=3$.
    - $c_f(z,x)=c(z,x)-f(z,x)=3-(2-1)=2$.
    - $c_f(x,z)=c(x,z)-f(x,z)=2-(1-2)=3$.
- Flow sum of f_1,f_2: f_1+f_2.
- If f' is flow in G_f, then $f+f'$ is flow in G with value $|f+f'|=|f|+|f'|$.

Augmenting path:

- A path $p: s \to t$ in G_f.
- Can push $c_f(p)$ flow from s to t along this path, with $c_f(p) = \min\{c_f(u,v) : (u,v) \in p\}$.
- e.g. $p = \langle s, w, y, z, x, t \rangle$, $c_f(p) = 1$.
  - Updated original:



- Lemma: given flow net G, and p augmenting path in G_f, define f_p as flow in G_f with value $c_f(p)$, then $f' = f + f_p$ is flow in G with value $|f'| = |f| + c_f(p) > |f|$.

Theorem (maxflow-mincut): the following 3 are equivalent:

- f is max flow.
- f admits no augmenting path.
- $|f| = c(S,T)$ for some cut (S,T).
- (The maximum value of an s-t flow is equal to the minimum capacity over all s-t cuts.)

Ford-Fulkerson(V,E,s,t)
  Foreach $(u,v) \in E$:
    $f[u,v] = f[v,u] = 0;$
  While $\exists$ augmenting path $p \in G_f$:
    Augment f by $c_f(p)$;
Runtime: assume integer capacity, and max flow f^*, $O(E|f^*|)$.

- Not polynomial, since $|f^*|$ is not an input size.

Edmonds-Karp
  Do Ford-Fulkerson, but compute augmenting path by BFS in G_f (shortest path $s \to t$ with least number of edges).
Runtime: $O(VE^2)$.

- Proof: Let $\delta_f(u,v)$ be the shortest path distance $u \to v$ in G_f.
- Lemma: $\forall v \in V - \{s,t\}$, $\delta_f(s,v)$ increases monotonically with every augmentaion.
  - Proof: assume $\exists v \in V - \{s,t\}$ such that exists flow augmentation making $\delta_f(s,v)$ decrease.
  - Let f be flow before and f' flow after. Let v be a vertex with minimum $\delta_{f'}(s,v)$ whose distance was decreased ($\delta_{f'}(s,v) < \delta_f(s,v)$).
  - Let $s \to u \to v$ be shortst path in $G_{f'}$, $(u,v) \in E_{f'}$ and $\delta_{f'}(s,v) = \delta_{f'}(s,u) + 1$.
  - So $\delta_{f'}(s,u) < \delta_{f'}(s,v)$.
  - This implies $\delta_{f'}(s,u) \ge \delta_f(s,u)$ (u cannot be one of vertices whose distance is decreased, otherwise u will be chosen).
  - Claim: $(u,v) \notin E_f$.
    - If $(u,v) \in E_f$, then $\delta_f(s,v) \le \delta_f(s,u) + 1 < \delta_{f'}(s,u) + 1 = \delta_{f'}(s,v)$ contradiction, since $\delta_{f'}(s,v) < \delta_f(s,v)$.
  - Thus $(u,v) \in E_{f'}$ and $(u,v) \notin E_f$.
  - Augmentation increases flow $v \to u$.
  - Shortest path $s \to u$ in G_f has (v,u) as last edge.
  - $\delta_F(s,v) = \delta_f(s,u) - 1 \le \delta_{f'}(s,u) - 1 = \delta_{f'}(s,v) - 2$.
  - Contradiction to $\delta_{f'}(s,v) < \delta_f(s,v)$.
- Theorem: Edmonds-Karp does $O(VE)$ augmentation.
  - Proof: p is augmenting path, $c_f(u,v) = c_f(p)$. Call edge (u,v) critical in G_f.
  - At least 1 critical edge per augmenting path.
  - We show that each of $|E|$ edes become critical at most $\frac{|V|}{2} - 1$ times.
  - Assume $u, v \in V$ s.t. $(u,v) \in E$ or $(v,u) \in E$ or both.
  - Since augmenting path are shortest path, (u,v) become critical means that $\delta_f(s,v) = \delta_f(s,u) + 1$.
  - Augmenting $\Rightarrow$ (u,v) disappears, can reappear if flow $u \to v$ decreases.
  - $\Rightarrow$ (v,u) is on augmenting path in $G_{f'}$, $\delta_{f'}(s,u) = \delta_{f'}(s,v) + 1$.
  - Using the lemma, $\delta_{f'}(s,v) \ge \delta_f(s,v) \Rightarrow \delta_{f'}(s,u) = \delta_{f'}(s,v) + 1 \ge \delta_f(s,v) + 1 = \delta_f(s,u) + 2$.
  - Every time an edge become critical, δ increases at least by 2.
  - Longest number of edges $= |V| - 2$.
  - In the worst case, become critical $\frac{|V|-2}{2} = \frac{|V|}{2} - 1$ times.
  - Have $O(E)$ pairs of nodes $\Rightarrow O(VE)$ critical edges $\Rightarrow O(VE)$ augmentations.
  - $\Rightarrow O(VE^2)$ total time (augmenting $\times$ BFS).

e.g. find the min weight cycle in $G = (V,E)$ in $O(VE^2)$ time (assume no neg wight cycle).
  Foreach $(u,v) \in E$:
    Let $G' = (V, E - \{(u,v)\})$;
    Bellman-Ford(G', v) gives $d[u] = v \to u$ shortest path;
  Take min of each cycle;

e.g. Find the min weight cycle in $O\big(VE\log V\big)$ time.

For $v \in V$: ($O\big(VE \log V\big)$)

Dijkstra(G,v);

Store results in matrix D;

// Now $D[u, v] = \delta(u, v)$.

Compute $\min_{u,v\in V} \delta(u, v) + \delta(v, u)$ ($O\big(V^2\big)$).

Bellman-Ford will be $O\big(V^2 E\big)$.

e.g. Maximum-bottleneck path

- Let $G = (V, E)$ be a directed weighted path with positive edge weights. Imagine each edge weight represents width of the edge. The bottleneck of a path is the minimum edge width on a path. We want the maximum bottleneck path from $s \in V$, computed in $O\big((V + E) \log V\big)$ time.
- Modify Dijkstra:
  - In Relax:
    - $d[v] = \max\{d[v], \min\{d[u], w(u, v)\}\}$.
    - Record the parent accordingly, (if $d[v] < \min\{d[u], w(u, v)\}$: $\pi[v] = u$).
  - In Init-Single-Source:
    - $d[v] = -\infty$, $\forall v \neq s$.
    - $d[s] = \infty$.

NP-Completeness

March 3, 2023 8:03 PM

Theory of computation

Alphabet (Σ): finite set of symbols, nonempty, ordered
String: possibly infinite sequence of symbols from alphabet
e.g. Σ_1={a,..,z}, Σ_2={0,...,9}.

- abc is a string on Σ_1.
- 123 is a string on Σ_2.
- a1b is not a string of Σ_1 or Σ_2.

Empty string: ϵ.

Conventions:

- Concatenate: 01 with 011 gives 01011.
- Self-concatenation: $a = 01$, then $a^0 = \epsilon$, $a^1 = 01$, $a^2 = 0101$.
- Reverse: a^R is the reverse of a.
- Σ^*: Set of all strings in Σ.
- Σ^+: Set of all strings in Σ with ϵ.

Language (L):

- L is a possibly infinite subset of Σ^*.
- L is language over Σ^*, then each element in L is string of the language.
- e.g.
  - {0,11,0011}, {ϵ,10} are languages over {0,1} (all subsets of $\{0,1\}^*$).
- With languages L_1 and L_2:
  - Union: $L_1 \cup L_2$.
  - Intersection: $L_1 \cap L_2$.
  - Subtraction: $L_1 - L_2$ (in L_1 but not in L_2).
- L^i: concatenate i copies of the language.
  - $L^0 = \{\epsilon\}$.
  - e.g. $L_1 = \{\epsilon, 0{,}1\}$, $L_1^2 = \{\epsilon, 0{,}1{,}00{,}01{,}10{,}11\}$.
- Kleene closure: $L^0 \cup L^1 \cup L^2$.

Regular languages

- A regular expression (RE) over Σ is defined with the following rules:
  - ϵ is RE.
  - $\forall a \in \Sigma$, a is RE.
  - If R,S are RE, then R+S (R or S) is a RE.
  - If R,S are RE, then RS (concatenation) is a RE.
  - If R is RE, then R^* is RE (R^* is infinite copies of R).
  - If R is RE, then (R) is RE (parenthesize).
- e.g.
  - $L(0) = \{0\}$.
  - $L\big((0+1)(0+1)\big) = \{00{,}01{,}10{,}11\}$.
  - $L(0^*) = \{\epsilon, 0{,}00{,}000\}$.
  - $L\big((0+1)^*1\big)$ ={any string ending with 1}.
  - $L\big((1^*01^*01^*)^*\big)$ ={any string with even number of 0s}.
  - $L\left(c^*\big(a+(bc^*)\big)^*\right)$ ={any string over {a,b,c} that do not contain substring ac}.

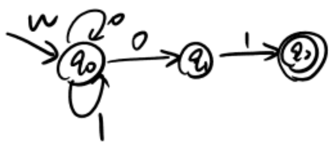
Deterministic finite automata (DFA)

- Language recognition devices: given string x as input, does $x \in L$ or $x \notin L$?
- Given finite number of states $q_0, q_1, .., q_n$, with some terminal state, if a string ends in a terminal state, we accept it, otherwise, reject.
- Theorem: a language is regular if and only if it is recognized (accepted) by some DFA.
- e.g.
  - 

  - q_1 is terminal state.
  - $x = 1011$ is accepted.
  - $x = 0110$ is rejected.
  - Accepts all $L\big((0+1)^*1\big)$.

Non-deterministic finite automata (NFA)

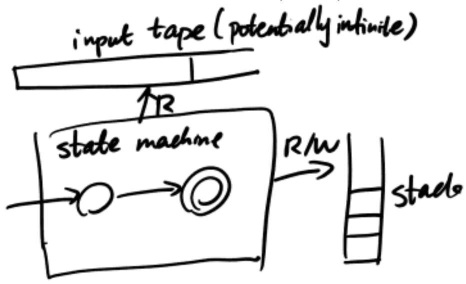
- A single input can cause the state transition towards more than 1 state.
- When we reach a non-deterministic state, we go to all possible next state to check.
- e.g.

  - 

  - Accepts all w on $\{0,1\}^*$ that ends with 01.
- Theorem: for each NFA, there exists equivalent DFA.

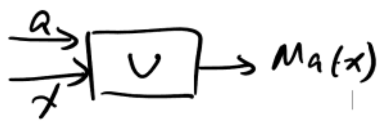
All the following are not regular languages, and cannot be recognized by DFAs
- $\{0^p$: p is a prime$\}$.
- $\{0^n 1^n : \forall n \in \mathbb{N}\} = \{\epsilon, 01, 0011, 000111, \ldots\}$.
- $\{ww^R : w \in \{0,1\}^*\}$.
- Issue: no memory

Context-free languages(CFLs)
- They rise from production rule.
- s: string in language.
- e.g.
  - $s \to \epsilon, s \to 0s1$ gives $\{0^n 1^n : \forall n \in \mathbb{N}\}$.
  - $s \to 0s0|1s1|\epsilon$ gives $\{ww^R : w \in \{0,1\}^*\}$.

Nondeterministic Pushdown automata (NPDA)
- 

- Push when 0, pop when 1, stack empty then accept: $\{0^n 1^n\}$.

Turing machine
- Finite state machine
- Infinite length tape
- Can read/write tape
- Can leave an answer on the tape
- Special state: halting state
  - Finished computation
  - Read tape: 0 for yes, 1 for no.
- Can enter infinite loop
- A Turing machine T accepts language L if T accepts $x \in L$ and rejects or enters infinite loop for $x \notin L$.
- A Turing machine T decides a language L if:
  - Yes: $x \in L$.
  - No: $x \notin L$.
  - There should be no infinite loop

Universal Turing Machine
- A Universal Turing Machine U takes in an input $\langle M, w \rangle$, it simulates a Turing Machine M on an input w.
- Let M_a be the Turing machine with specification a, it simulates M_a on input x.
- 

- Algorithm=Turing Machine=hardware=computer.
- Theorem: There always exists universal Turing machine such that $\forall x, a \in \{0,1\}^*, U(x,a) = M_a(x)$ such that if $M_a(x)$ halts within T steps, then $U(x,a)$ halts within $cT \log T$ steps where constant c depends on the alphabet size, number of tapes etc of M_a.

Uncomputability
- Theorem: There is uncomputable functions $UC: \{0,1\}^* \to \{0,1\}^*$ not computed by any Turing machine.
  - Define UC as follows, $\forall a \in \{0,1\}^*$:
    - If $M_a(a) = 1$ (accept), then $UC(a) = 0$.
    - If $M_a(a) = 0$ (reject), then $UC(a) = 1$.
  - Proof: Assume UC is computable, i.e. there exists Turing machine M such that $M(x) = UC(x), \forall x \in \{0,1\}^*$.
    Then $M(M) = UC(M)$ contradiction, because by definition, $M(M) = 1$ iff $UC(M) = 0$.

Halting problem:
- Define HALT(a, x)=1 if $M_a(x)$ halts. HALT is uncomputable.
  - Proof: Assume there exists Turing machine TM_{halt}, then use TM_{halt} to compute UC function.
  - To build machine on UC (M_{UC}):

- On input a, M_{UC} runs HALT(a, a).
- If HALT(a, a)=0 (M_a does not halt on a), then $M_{UC} = 1$.
- If HALT(a, a)=1, then run universal Turing machine U on $M_a(a)$, get result b.
  - If $b = 1$, output 0.
  - If $b = 0$, output 1.
- However, this is not computable, contradiction.

Decision v.s. optimization

- Decision: Is there a path $x \rightarrow y$ which is at most k-edges?
  - HAM-CYCLE: Is there a simple cycle traversing all vertices of G?
- Optimization: What's the shortest path between vertices x and y?
- Decision problems ≤ optimization problems.
  - If we solve an optimization problem, we have the solution to the corresponding decision problem.

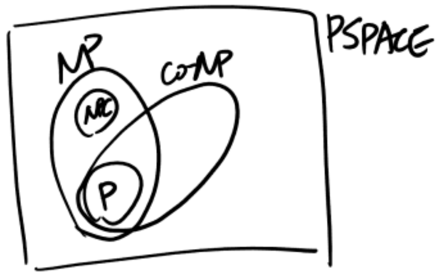
Complexity class P

- P ={$L \in \{0,1\}^*$: ∃ poly time algorithms that decides L in poly time}.
- Def: Algorithm A verifies a problem L if and only if given instance $x \in L$, ∃ certificate (or witness, candidate solution) y such that $A(x, y) = 1$.
  - The language verified is $L = \{x \in \{0,1\}^* : \exists y \in \{0,1\}^*\ s.t.\ A(x, y) = 1\}$.
  - e.g. in HAM-CYCLE, x is a graph, y is a proposed solution of HAM-CYCLE.

Complexity class NP

- Informally: all problems verified in poly-time.
- Formally: $L \in NP$ if there exists poly-time algorithm A and constant c such that L ={$x \in \{0,1\}^*$: ∃ certificate y where $|y| = O(|x|^c)$ such that $A(x, y) = 1$ and A runs in poly-time}.
  - The size of certificate (solution) must be polynomial to the size of the input.

Hierarchy

- $P \subset NP$: problems that can be solved in polynomial time can be verified in polynomial time.
- Co-NP: $L \in NP \Rightarrow \bar{L}$ ∈co-NP.
  - e.g. NP=all graphs that have HAM-CYCLE, co-NP=problems that are:
    - Not a graph
    - A graph without HAM-CYCLE
- Theorem: P is closed under complement that is P=co-P.
  - $L \in P \Rightarrow \bar{L} \in P$ (simply reverse the problem and solution).
- PSPACE: problems that can be solved by Turing machine using poly space
- 

Open problems

- NP=co-NP?
- P=NP∩co-NP? (primality checking is NP∩co-NP)
- P=NP?

Poly-reducibility

- Informally: if an instance of problem Q can be transformed in poly-time to an instance of problem Q' such that a solution to Q' provides a solution to Q.
  - i.e. Q is not harder than Q', $Q \leq Q'$.
- Formal: language (problem) L_1 is poly-reducible to L_2 denoted as $L_1 \leq_p L_2$ if and only if ∃ poly-time algorithm $f()$ such that $x \in L_1$ if and only if $f(x) \in L_2$.
- Theorem: if $L_1 \leq_p L_2$ and $L_2 \in P$, then $L_1 \in P$.
  - Given x, reduce x to $f(x)$ in poly-time, check $f(x) \in L_2$ is poly-time, map back to x is poly time.

NP Complete (NPC)

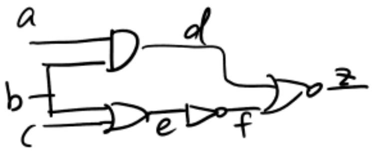
- A problem is NPC if
  - $L \in NP$ (verified in poly time)
  - $\forall L' \in NP$, $L' \leq_p L$ (if only this property is satisfied, then L is NP-hard)
- Theorem:
  - If $L \in NPC$ and $L \in P$, then $P = NP$.
  - NP=co-NP if and only if $\exists L \in NPC$ such that $\bar{L} \in NP$.
    - ⇒: easy since NP and co-NP now overlaps.
    - ⇐: pick $L' \in NP$, show that $\overline{L'} \in NP$.
      - Since $L \in NPC$, $L' \leq_p L$, equivalently, $\overline{L'} \leq_p \bar{L}$.
      - Since $\bar{L} \in NP$, then $\overline{L'} \in NP$.

Methodology: Given L, to prove $L \in NPC$.

- Prove $L \in NP$(verified in polytime).

- Provide a certificate: the evidence that the solution is an instance of L.
  - e.g. for SAT, assignment, for Ham-Cycle: a ham-cycle.
- Select known $L' \in NPC$ and:
  - Find algorithm f that given instance x, $x \in L'$ if and only if $f(x) \in L$.
    - Show the transformation
    - Then prove the if and only if equivalence
  - Show f takes poly time, i.e. $L' \leq_p L$.
- If $L' \leq_p L$ for some $L' \in NPC$, then $\forall L'' \in NP, L'' \leq_p L' \leq_p L$.

Circuit SAT is NPC

- Is there an assignment to primary inputs a, b, c, making $z = 1$?
- Circuit SAT→SAT→3-CNF-SAT→ $\begin{cases} clique \rightarrow vertex\ cover \\ HAM - CYCLE \rightarrow TSP \end{cases}$.
- 

Reduce circuit SAT to formula SAT

- Formula SAT: ϕ is a formula of n-boolean variables and connections ∧, ∨, ¬, (), ⇒, ⇔.
  - e.g. $\phi = (x_1 \Leftrightarrow \overline{x_2}) \wedge \left(\overline{x_4} \Rightarrow (x_1 \vee \overline{x_2})\right)$.
- Decision version: Is there a 0/1 assignment to variables such that $\phi = 1$? $|\phi| = n$.
- Formula SAT∈NP:
  - Number of connections is poly in n, given a solution, it takes polytime to evaluate and verify.
- Circuit-SAT$\leq_p$Formula SAT.
  - Given a single output circuit C, create a formula ϕ such that C has satisfying assignment is equivalent to $\exists x_1, x_2, \dots, x_n$, s.t. $\phi = 1$.
  - If $\phi = 1$, the corresponding a, b, c must give $z = 1$ in the circuit.
  - If C has satisfying assignment, by construction $\phi = 1$.
  - Reduction is polynomial time, since number of gates is polynomial in n.

3-CNF-SAT

- CNF: a conjunction of disjunction of clauses with any number of boolean variables
  - $\phi = (a \vee b) \wedge (a \vee b \vee c) \wedge (b \vee d)$.
- 3-CNF: a conjunction of disjunction of clauses with exactly 3 boolean variables
  - $\phi = (x_1 \vee \overline{x_2} \vee \overline{x_3}) \wedge (\overline{x_3} \vee x_5 \vee x_7) \wedge (x_1 \vee x_3 \vee \overline{x_7}) \wedge \cdots$.
- Literal: variable or complement of a variable
- Clauses: each ($\square$) is a clause.
- Disjunction: connected by ∨.
- Conjunction: clauses connected by ∧.
- Decision version: Given ϕ with n =# variables, $O(n)$ =# clauses, does it have a satisfying $x_1, \dots, x_n$ assignment?
- Side note:
  - 2.4-SAT∈ P: if each clause have 2.4 literals on average, then it is P.
  - 2.41∈NPC.
- 3-CNF-SAT is NP: given assignment $x_1, \dots, x_n$, it takes poly time to plug in $O(n)$ clauses to check.
- Circuit-SAT $\leq_p$ 3-CNF-SAT.
  - Given a circuit, it has a satisfying input assignment ⇔ some 3-CNF-SAT ϕ is satisfiable.
  - 

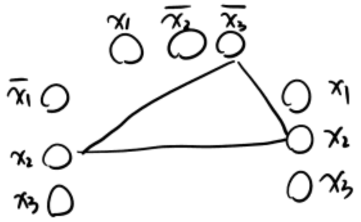
  - Consider a gate $d = a \wedge b$, it has a characteristic function:

a	b	d	And
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

    - The maxterm is: $\phi^{and}_{max} = \left(\bar{a} \wedge b^{-} \wedge d\right) \vee (\bar{a} \wedge b \wedge d) \vee \left(a \wedge \bar{b} \wedge d\right) \vee \left(a \wedge b \ \wedge \bar{d}\right)$.
    - $\phi_{and} = \left(a \vee b \vee \bar{d}\right) \wedge \left(a \vee \bar{b} \vee \bar{d}\right) \wedge \left(\bar{a} \vee b \vee \bar{d}\right) \wedge \left(\bar{a} \vee \bar{b} \vee d\right)$. (complement everything)
  - The overall circuit can be represented by $\phi = \phi_{and} \wedge \phi_{nor} \wedge \phi_{nand} \wedge (w \vee \bar{p} \vee q) \wedge (w \vee p \vee \bar{q}) \wedge (w \vee \bar{p} \vee \bar{q}) \wedge (w \vee p \vee q)$.

- Note: the final 4 terms is equivalent to $w = 1$.
  - If there exists satisfying assignment to the circuit, then ϕ is satisfiable.
  - If ϕ is satisfiable, we use the same input, and w must be 1.
- f (transformation) takes poly-time, since we just translate $O(n)$ clauses to $O(n)$ gates.

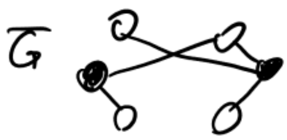
Clique

- A clique is a graph that every vertex is connected with all other vertices.
  - K4:
  - 
- Both the clique and approximating clique are NPC.
- Decision version: Does G have a clique of size k?
- Clique is NP: given the k vertices, check if they are pair-wise connected takes poly time $O(V + E)$.
- 3-SAT $\leq_p$ clique
  - Consider $\phi = (\overline{x_1} \vee x_2 \vee x_3) \wedge (x_1 \vee \overline{x_2} \vee \overline{x_3}) \wedge (x_1 \vee x_2 \vee x_3)$.
    - 

  - ϕ has a satisfying assignment ⇔ some G has a clique of size = # clauses.
  - Reduction procedure:
    - For each clause, introduce 3 vertices.
    - Connect vertices from different clauses if and only if they are not complement of themselves.
  - Given a satisfying assignment to ϕ, the connection in G is a clique.
  - Given a clique in G, the vertex assignment satisfies ϕ.
- Given ϕ, we create a graph is polynomial time.

Vertex cover

- Given a graph G, a vertex cover $V' \subset V$ is one that $\forall (u, v) \in E$, u or v or both in V'.
- Decision version: Does there exist a vertex cover of size k?
- Vertex cover is NP: Iterate through the vertices V', check if all edges are adjacent to V', $O(E^2)$.
- Clique$\leq_p$Vertex-Cover
  - G has a clique of size k ⇔ $\bar{G}$ has a vertex cover of size $|V| - k$.
  - $\bar{G}$ is the complement graph, with the same vertices, if $e \in E, e \notin \bar{E}$.
  - 



  - Assume they are not vertex cover, there is an additional edge in $\bar{G}$ not covered, then there is no clique of size k in G.
  - Assume there is no clique, then there will be an additional edge in $\bar{G}$, the vertex cover has a larger size.
- Transformation from G to $\bar{G}$ is polynomial time.

Travelling Salesman Problem

- Informal: a salesman needs to go to every city only once to sell his merchandise and wants to minimize the mileage
- Formally: Given a complete, undirected, weighted graph, find a Ham-Cycle of minimum weight.
- Decision version: Does G have a TSP with weight k?
- TSP is NP: Iterate through the given solution, check if it is weight k and Ham-Cycle. Poly-time
- Ham-Cycle$\leq_p$TSP
  - Assign unit weight to all edges in the original Ham-cycle graph G.
  - Make G a complete graph by assigning infinite weight to the additional edges.
  - The transformation is poly-time, since we add $O(V^2)$ edges.
  - Is there a TSP with $k = |V|$?

Suppose $A \leq_p B$:

- If $B \in P$, then $A \in P$.
- If $B \in NPC$, then $A \in NP$ (can use B's verification procedure).
- If $A \in P$, cannot conclude on B.
- If $A \in NPC$(NP-hard), then B ∈NP-Hard.

Half-Vertex-Cover

- A=Half-Vertex-Cover={$\langle G \rangle$: G has even number of vertices and a vertex cover of size $\frac{|V|}{2}$}.
- B=k-vertex cover.
- $A \in NP$:

- Certificate: $S \subset V$.
- Verification: check that $|S| = \frac{|V|}{2}$ and check that $\forall (u, v) \in E$, either $u \in S$ or $v \in S$, takes $O(E) + O(1)$.

- $B \leq_p A$.
  - Given G and k, construct G that has vertex cover of size $\frac{|V|}{2}$.
  - case 1: $k = \frac{|V|}{2}$, nothing to do.
  - Case 2: $0 \leq k < \frac{|V|}{2}$.
    - Transformation: let $m = |V| - 2k$, given $G = (V, E)$ and k, construct $G' = (V', E')$ by adding $v_1, \dots, v_m$ new vertices to G that are disconnected and contain self-loops. $O(V + E + m) = O(V + E + k)$.
    - Claim: $\langle G, k\rangle$ ∈k-VC ⇔ $\langle G'\rangle$ ∈Half-VC.
      - ⇒Let $S \subset V$ be the k-vertex cover of G, $|S| = k$, consider $S' = S \cup \{v_1, v_2, \dots, v_m\}$, which is a vertex cover of G'.
        Notice $|S'| = |S| + m = k + |V| - 2k = |V| - k = \frac{|V'|}{2}$.
      - ⇐let $S' \subset V'$ be a vertex cover of G', notice $v_1, \dots, v_m \in S'$ otherwise we miss the self loop. Consider $S = S' - \{v_1, \dots, v_m\}$.
        $|S| = |S'| - m = \frac{|V'|}{2} - |V| + 2k = \frac{|V|+|V|-2k}{2} - |V| + 2k = k$ is a k-vertex cover of G.
  - Case 3: $\frac{|V|}{2} < k \leq |V|$.
    - Transformation: Let $p = 2k - |V|$, given $G = (V, E)$ and k, construct $G' = (V', E')$ by adding $v_1, \dots, v_p$ new vertices to G that are disconnected, $|V'| = |V| + 2k - |V| = 2k$. $O\big(V + E + p\big) = O(V + E + k)$.
    - Claim: $\langle G, k\rangle$ ∈k-VC ⇔ $\langle G'\rangle$ ∈Half-VC.
      - ⇒Let $S \subset V$ be the k-vertex cover of G, $|S| = k$, consider $S' = S$, $|S'| = |S| = k = \frac{|V|}{2}$.
      - ⇐let $S' \subset V'$ be the half-vertex cover of G'. Let $S = S' - \{v_1, \dots, v_p\}$.
        $|S'| - p \leq |S| \leq |S'| = \frac{|V'|}{2} = k$, so $|S| \leq k$.
        If $|S| < k$, add any vertex until $|S| = k$

Approximation Algorithms

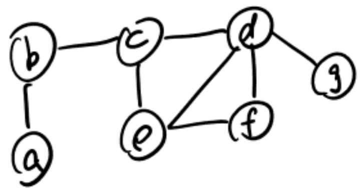
March 3, 2023 8:03 PM

Approximation algorithm with approximation ratio $\rho(n)$ (or a $\rho(n)$-approximation)

- $\rho(n) \geq 1$, often constant, can be abbreviated to ρ-approx.
- Minimization: $\frac{C}{C^*} \leq \rho(n)$, C is approximation, C^* is optimal.
- Maximization: $\frac{C^*}{C} \leq \rho(n)$.
- If algorithm is poly-time, then we have poly-time $\rho(n)$-approximation.

Vertex cover

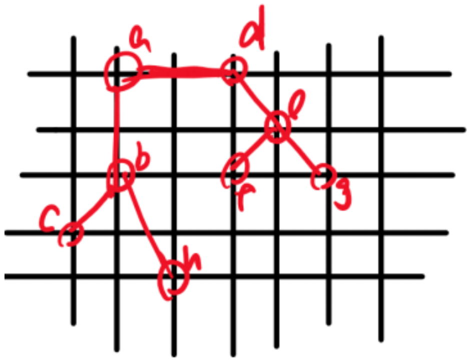
- Optimization: find vertex cover of minimum size
- 2-approximation algorithm in poly time
- Approx-Vertex-Cover(G)

  $C = \emptyset$;
  $E' = G.E$ (copy edges);
  While $E' \neq \emptyset$:
  Choose $(u, v) \in E'$ arbitrarily;
  $C = C \cup \{u, v\}$;
  Remove from E', every edge incident on u or v;
  Return C.
- e.g.
  - 

  - $C = \emptyset$.
  - $C = \{b, c\}$.
  - $C = \{b, c, e, f\}$.
  - $C = \{b, c, e, f, d, g\}$.
  - Optimal: $\{b, e, d\}$.
- Proof: the algorithm is 2-approximation of optimal vertex cover
  - Observations:
    - C is a vertex cover.
    - Need to create a bound for C^*.
  - Let A denote set of edges the algorithm picks.
  - An optimal vertex cover C^* is a vertex cover, must cover at least one endpoint of each edge in E, and each edge in A.
  - No 2 edges in A share common endpoints ⇒ no 2 edges in A are covered by the same vertex in C^*.
  - $|C^*| \geq |A|$.
  - Also, $|C| = 2|A|$, thus $|C| \leq 2|C^*|$.

Travelling salesman in 2D plane

- Complete undirected $G = (V, E)$ and integer cost $C(u, v)$ for each $(u, v) \in E$.
- Denote $c(A) = \sum_{(u,v)\in A} c(u, v)$.
- TSP in 2D ⇒ edge costs satify triangle inequality because edge costs are the ordinary Euclidean distance between nodes.
  - $c(u, w) \leq c(u, v) + c(v, w)$.

- 

- Approx-TSP-Tour(G,c)

  Select vertex $v \in V$ to to be some root vertex

  Compute MST T of G from root r using MST-Prim(G,c,r)

  Let H be a list of vertices ordered according to first visit in preorder walk of T.

  Return Hamiltonian cycle H.
- e.g.
  - $T = \{(a,b),(b,c),(a,d),(d,e),(e,f),(e,g),(b,h)\}$.
  - Preorder walk of T: $a, b, c, b, h, b, a, d, e, f, e, g, e, d, a$.
  - Only count first visit: a, b, c, h, d, e, f, g.
  - H: $a \to b \to c \to h \to d \to e \to f \to g \to a$ ($\to$ direct shortest path straight line).
- Proof: let H^* be the optimal tour if remove any single edge from that tour H^*, get a spanning tree.
  - $c(T) \leq c(H^*)$.
  - A full walk of T traverses every edge in preorder walk of T exactly twice.
  - Let W be the full walk, $c(W) = 2c(T) \Rightarrow c(W) \leq 2c(H^*)$.
  - From W to walk that only uses first visit of each vertex, we are deleting v from W between u and w.
  - By triangle inequality, $c(H) \leq c(W)$, $c(H) \leq 2c(H^*)$.
- Theorem: if $P \neq NP$, then for any constant $\rho > 1$, there does not exist poly-time approximation algorithm with approximation ratio ρ for the general TSP problem (triangle inequality does not hold).
  - Proof (by contradiction): Ham-Cycle$\leq_p$TSP-opt.
  - Reduction from G to G', c, where G' is the completion of G, $c = \begin{cases} 1, (u,v) \in E \\ \rho|V| + 1, else \end{cases}$ is the cost function, where ρ is the approxmiation rate, $|V|$ =# vertices.
  - TSP tour have total cost $|V|$ using Ham-Cycle edges.
  - For sub optimal, total cost will be at least $(\rho + 1)|V|$.
  - This will tell if there exists a Ham-Cycle in G in polynomial time.