ECE345 Midterm - TUT0101

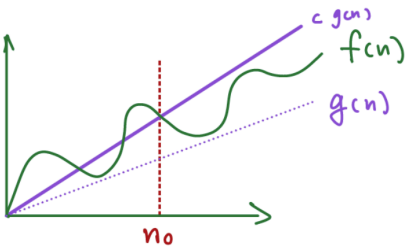
Asymptotics & Logarithms

General

1. How to evaluate an algorithm
   - Complexity: Speed &Memory
     - relationship of input size to the number of steps
     - Approximation of the speed of the algorithm
   - Identify the performance/complexity of the algorithm
     - Describe the asymptotic running time of an algorithm

Notations

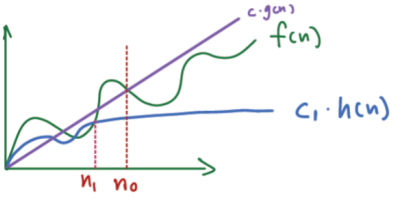
2. Big O
   - Definition:
     - Worst-case run time
     - Upper bound on the asymptotic behavior of a function
     - Func growth no faster than a certain rate



$O(g(n)) = \{f(n)$: there exist positive constants c and n_0 such that $0 \le f(n) \le cg(n)$ for all $n \ge n_0\}$.

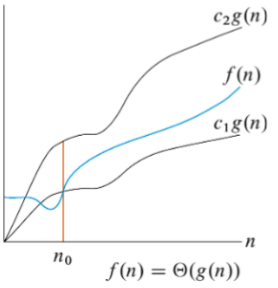
$g(n)$ is an ***asymptotic upper bound*** for $f(n)$.
If $f(n) \in O(g(n))$, we write $f(n) = O(g(n))$

   - rate/order of growth: highest-order coefficient and low-order terms don't matter
   - O(logn) < O(n)
3. Ω
   - Definition:
     - Best-case run time
     - Lower bound
     - Func growth is no slower than a certain rate



$\Omega(h(n)) = \{f(n)$: there exist positive constants c_1 and n_1 such that $0 \le c_1 h(n) \le f(n)$ for all $n \ge n_1\}$.

4. Θ
   - Definition:
     - Average-case run time
     - Tight bound
     - Func growth similar to this rate



$\Theta(g(n)) = \{f(n)$: there exist positive constants c_1, c_2, and n_0 such that $0 \le c_1 g(n) \le f(n) \le c_2 g(n)$ for all $n \ge n_0\}$.

5. o (Little o of g of n)
   - Upper bound but not asymptotically tight
   - To see whether one has a smaller big O than a little o
   - f(n) is asymptotically smaller than g(n) if f(n)=o(g(n))

$o(g(n)) = \{f(n)$: for all constants $c > 0$, there exists a constant $n_0 > 0$ such that $0 \le f(n) < cg(n)$ for all $n \ge n_0\}$.

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0.$$

6. □ (little omega of g of n)
   - Lower bound but not asymptotically tight
   - To see whether one has a bigger omega than a little omega
   - f(n) is asymptotically larger than g(n) if f(n)=□(g(n))

$\omega(g(n)) = \{f(n)$: for all constants $c > 0$, there exists a constant $n_0 > 0$ such that $0 \le cg(n) < f(n)$ for all $n \ge n_0\}$.

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty.$$

7. Asymptotic Properties
   - **Transitivity:** $f(n) = \Theta(g(n))$ and $g(n) = \Theta(h(n)) \Rightarrow f(n) = \Theta(h(n))$.
     Same for O, Ω, o, and ω.
   - **Reflexivity:** $f(n) = \Theta(f(n))$.
     Same for O and Ω.
   - **Symmetry:** $f(n) = \Theta(g(n))$ if and only if $g(n) = \Theta(f(n))$.
   - **Transpose symmetry:** $f(n) = O(g(n))$ if and only if $g(n) = \Omega(f(n))$.
     $f(n) = o(g(n))$ if and only if $g(n) = \omega(f(n))$.
8. Notations as comparator

$$
\begin{aligned}
f(n) = O(g(n)) \quad &\text{is like} \quad a \leq b\,, \\
f(n) = \Omega(g(n)) \quad &\text{is like} \quad a \geq b\,, \\
f(n) = \Theta(g(n)) \quad &\text{is like} \quad a = b\,, \\
f(n) = o(g(n)) \quad &\text{is like} \quad a < b\,, \\
f(n) = \omega(g(n)) \quad &\text{is like} \quad a > b\,.
\end{aligned}
$$

9. Useful equations

$n^a = \mathcal{O}(n^b) \Leftrightarrow a \leq b$

Proof: $a \leq b \Leftrightarrow \lim_{n\to\infty} \frac{n^a}{n^b} = 0 \text{ or } 1 \Leftrightarrow n^a = \mathcal{O}(n^b)$

$\log_a n = \mathcal{O}(\log_b n),\ \forall a, b > 1$

Proof: $\lim_{n\to\infty} \frac{\log_a n}{\log_b n} = \lim_{n\to\infty} \frac{\frac{\log_b n}{\log_b a}}{\log_b n} = \frac{1}{\log_b a} < \infty$

$c^n = \mathcal{O}(d^n) \Leftrightarrow c \leq d$

Proof: $c \leq d \Leftrightarrow \lim_{n\to\infty} \frac{c^n}{d^n} = \lim_{n\to\infty} \left(\frac{c}{d}\right)^n = 0 \text{ or } 1 < \infty \Leftrightarrow c^n = \mathcal{O}(d^n)$

- $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \Rightarrow f(n) = o(g(n))$[2] (The 2 and 3 here are refering to the footnote numbers.)
- $\lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty \Rightarrow f(n) = \mathcal{O}(g(n))$
- $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty \Rightarrow f(n) = \omega(g(n))$[3]
- $\lim_{n\to\infty} \frac{f(n)}{g(n)} > 0 \Rightarrow f(n) = \Omega(g(n))$
- $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c,\ c \in (0, \infty) \Rightarrow f(n) = \Theta(g(n))$

L'Hopital's rule: $\lim_{x\to a} \frac{f(x)}{g(x)} = \frac{0}{0} \text{ or } \frac{\infty}{\infty} \Rightarrow \lim_{x\to a} \frac{f(x)}{g(x)} = \lim_{x\to a} \frac{f'(x)}{g'(x)}$

- $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \Rightarrow f(n) = o(g(n))$
- $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c \in [0, \infty) \Rightarrow f(n) = \mathcal{O}(g(n))$
- $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c \in (0, \infty) \Rightarrow f(n) = \Theta(g(n))$
- $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c \in (0, \infty] \Rightarrow f(n) = \Omega(g(n))$
- $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty \Rightarrow f(n) = \omega(g(n))$

10. Logarithms

Notation (in this course): $(\log n)^2 = (\log n)(\log n)$ and $\log^{(2)} n = \log(\log n)$
$\log^* n = \min\{i \geq 0 : \log^{(i)} n \leq 1\}$

11. Compare functions

Short hand notation: $f(n) << g(n) \Leftrightarrow f(n) = \mathcal{O}(g(n))$
Assume f and h are eventually positive, i.e. $\lim_{n\to\infty} f(n) > 0$ and $\lim_{n\to\infty} h(n) > 0$

$1 << \log^*(n) << \log^{(i)} n << (\log n)^a << \sqrt{n} << n << n \log n << n^{1+b} << c^n << n!$, for all positive i, a, b, c

$f(n) << g(n) \Rightarrow h(n)f(n) << h(n)g(n)$
Proof: $\lim_{n\to\infty} \frac{h(n)f(n)}{h(n)g(n)} = \lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$

$f(n) << g(n) \Rightarrow f(n)^{h(n)} << g(n)^{h(n)}$
Proof: $\log\left(\lim_{n\to\infty} \frac{f(n)^{h(n)}}{g(n)^{h(n)}}\right) = \lim_{n\to\infty} h(n) \log \frac{f(n)}{g(n)} = \lim_{n\to\infty} h \log \lim_{n\to\infty} \frac{f}{g} = -\infty$, $\lim_{n\to\infty} \frac{f^h}{g^h} = 0$

$f(n) << g(n)$ and $\lim_{n\to\infty} h(n) > 1 \Rightarrow h(n)^{f(n)} << h(n)^{g(n)}$
Proof: $\log\left(\lim_{n\to\infty} \frac{h(n)^{f(n)}}{h(n)^{g(n)}}\right) = \lim_{n\to\infty} (f - g) \log h = -\infty$, $\lim_{n\to\infty} \frac{h^f}{h^g} = 0$

<u>Logarithms & Summations</u>

12. Logarithms equations

- Notation

- Properties

In the expression $\log_b a$:

- Hold b constant $\Rightarrow$ the expression is strictly increasing as a increases.
- Hold a constant $\Rightarrow$ the expression is strictly decreasing as b increases.

$$
\begin{aligned}
a &= b^{\log_b a}, \\
\log_c(ab) &= \log_c a + \log_c b, \\
\log_b a^n &= n \log_b a, \\
\log_b a &= \frac{\log_c a}{\log_c b}, \\
\log_b(1/a) &= -\log_b a, \\
\log_b a &= \frac{1}{\log_a b}, \\
a^{\log_b c} &= c^{\log_b a}.
\end{aligned}
$$

Iterated logarithm function

x	$\lg^* x$
$(-\infty, 1]$	0
$(1, 2]$	1
$(2, 4]$	2
$(4, 16]$	3
$(16, 65536]$	4
$(65536, 2^{65536}]$	5

Limit of logs: $\lim_{x\to a}(\log_b f(x)) = \log_b\left(\lim_{x\to a} f(x)\right)$ ($\log_b(\cdot)$ is continuous)

13. Different types of series

Solving continuous problems using numerical approximation.

$\sum_{i=1}^{n} i = 1+2+\cdots+n = n(n+1)/2 = \Theta(n^2)$, arithmatic series

$\sum_{i=0}^{n} x^i = 1+x+x^2+\cdots+x^n = (x^{n+1}-1)/(x-1)$, geometric series

$\sum_{i=0}^{\infty} x^i = 1/(1-x)$ when $|x|<1$, infinite series

ex1) Show $\sum_{k=0}^{\infty} kx^k = x/(1-x)^2$ if $|x|<1$

$x \cdot (d\sum x^i)/dx = d(1/(1-x))/dx \cdot x$

Telescoping

$\sum_{i=1}^{n} a_i - a_{i-1} = a_n - a_0$

$\sum_{i=0}^{n-1} a_i - a_{i+1} = a_0 - a_n$

ex1) Show $\sum_{k=1}^{n-1} 1/(k(k+1)) = \sum \frac{1}{k} - 1/(k+1) = 1 - 1/n$

Proof Methods (Induction, Contradiction, & Combinatorial Arguments)

Induction

1. Proof by induction
   - Steps
     - Proof by Induction [3-step] **Basis → Hypothesis → Inductive Step**

       IF Basis holds (for small numbers: 0, 1, 2, ...)

       AND

       IF Assume P(n) holds for an arbitrary number

       THEN we need to prove the Inductive Step follows from that P(n+1) holds

       THEN we have proven that ∀n, P(n) holds
   - Weak induction example

     Prove $11^n - 6$ is divisible by 5, $\forall n \geq 1$
     Let $P(n) = 5|(11^n - 6)$ (5 divides $11^n - 6$)
     Base Step: $n = 1$, $11^1 - 6 = 5$ $5|5$
     Induction Hypothesis: Assume $P(n)$ is true
     Induction Step: Check $P(n+1)$, $11^{n+1} - 6 = 11 \cdot 11^n - 6 = 11 \cdot (5m + 6 - 6)$
     $= 55m + 66 - 6 = 55m + 60 = 5(11m + 12)$ for some m
     So $5|11^{n+1} - 6$, $P(n+1)$ is true.
   - Strong induction
     1. Basis: show $P(n_0), P(n_1), \ldots$ are true
     2. Hypothesis: Assume $P(k)$ is true, $\forall k \leq n$
     3. Induction: Show $P(n_0) \wedge \cdots \wedge P(k) \wedge \cdots \wedge P(n) \Rightarrow P(n+1)$

     e.g. The Fundamental Theorem of Arithmetic: all integers $n \geq 2$ can be expressed as the product of one or more prime numbers
     Proof: **Base Step**: $n = 2$, 2 is a prime
     Induction Hypothesis: assume all $k \in [2, n]$ can be written as the product of one or more primes
     Induction Step:
     $n + 1$ is prime. Then it can be expressed as the product of itself.
     $n + 1$ is not prime. Then $n + 1 = k_1 k_2$ for some integers $k_1, k_2 < n + 1$. By IH, k_1, k_2 can be written as product of primes. Thus $n + 1$ can be written as product of primes
2. Power set

   Given a set S, powerset $2^{|S|}$, is every combination of the elements in S include the empty set.
   ex) $S = \{A, B, C\}$, show the powerset of $|S| = n$ has 2^n elements.
   $2^{|S|} = \{\phi, A, B, C, AB, BC, AC, ABC\}$.
   Basis $|S| = 1$, $S = \{A\}$, $2^{|S|} = \{\phi, A\} = 2$
   Hyp Assume $|S| = n$, then $|2^{|S|}| = 2^n$
   Step Need show for $|S| = n+1$ elements, $2^n + 2^n = 2^{n+1}$.

Contradiction

3. Proof by contradiction
   1. Assume toward a contradiction $\neg P$/ $\bar{P}$(not P)
   2. Make some argument
   3. arrive at a contradiction
   4. $\therefore P$ must be true

   e.g. Prove that there are infinitely many prime number
   Proof: Assume that there are a finite number of primes
   Let S be the complete set of primes
   let $P = \prod_{x \in S} x + 1$
   $P \notin S$ because $P > x$, $\forall x \in S$
   P is a prime because P is not divisible by any prime, $1 \equiv P \mod x$, $\forall x \in S$
   P is a prime but not in S, contradiction.

4. symbol

$\prod_{i=1}^{n} q_i = q_1 \times q_2 \times \cdots \times q_n$

- Set

  $\mathbb{N} = \{1, 2, 3, ...\}$: all natural numbers
  $\mathbb{Z} = \{..., -3, -2, -1, 0, 1, 2, 3, ...\}$: all integers
  $\mathbb{R}$ = all real numbers
  $\emptyset$: empty set

  $\mathcal{P}(X) = \{Y : Y \subset X\}$: the power set of X, i.e. the set of all subset of X

- Set operations

  Complement: Fix a universe U, $A \subset U$, $\bar{A} = C_U A = \{x \in U, x \notin A\}$

- General notations

  Logics:
  Negation: $\neg P$, $\sim P$, $\bar{P}$, (LaTeX: \lnot, \sim)
  And: $P \wedge Q$ (LaTeX: \land)
  Or: $P \vee Q$ (LaTeX: \lor)

  Quantifiers:
  $\exists$ there exists (LaTeX: \exists)
  $\forall$ for all, for any (LaTeX: \forall)

  Other symbols:
  s.t. such that
  $\Leftarrow$ implies (LaTeX: \Leftarrow)
  $\Leftrightarrow$ if and only if (equivalently) (LaTeX: \Leftrightarrow)

Combinatorial Arguments

5. General

Suppose A can happen in n ways and B can happen in m ways.
- Rule of Products: A and B can happen in nm ways.
- Rule of sums: A or B can happen in $n + m$ ways.

Factorials $n! = n(n-1)(n-2)\cdots 2 \cdot 1$
=# ways to arrange n distinct objects when order is important

Permutations $P(n, r) = n(n-1)\cdots(n-r+1) = \frac{n!}{(n-r)!}$
=# ways to arrange r out of n distince objects when order is important.
Suppose not all objects are distinct. Let q_i =# objects of the ith kind and t =# types of objects, then
ways to arrange n objects is $\frac{n!}{\prod_{i=1}^{t}(q_i!)}$

Combinations $C(n, r) = \binom{n}{r} = \frac{n!}{r!(n-r)!}$
=# ways to choose a set of r objects from n objects where order doesn't matter.

UNIVERSITY OF TORONTO

6. Method

Method:

1. Question: ask a question relating to the formula you would like to prove
2. LHS: argue why the LHS answers the question
3. RHS: argue why the RHS answers the question

You can choose either (easier) side to start.
Usually can be done by thinking about choosing k objects from a set of n (without replacement) $\rightarrow \binom{n}{k}$
Or by forming n-letter strings from an alphabet of size k (selection with replacement) $\rightarrow k^n$

$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$
Question: How many ways can you select k objects from a set of n objects?
LHS: by defintion
RHS: consider a specific object x in the set
x is in the k objects we choose $\Rightarrow \binom{n-1}{k-1}$ to choose the remaining
x is not in the k objects we choose $\Rightarrow \binom{n-1}{k}$ to choose the remaining
By Rule of sum.

Graph

7. Definition

Graph: $G = (V, E)$, $V = \{\text{vertices}\}$, $E = \{\text{edges}\}$.
Subgraph: $G' = (V', E')$ is a subgraph of a graph G if and only if $V' \subset V$, $E' \subset E$ and if $e = (v, u) \in E'$, then $v \in V'$, $u \in V'$
Adjacent vertices: $N(v) = \{u : (v, u) \in E\}$
Incident edges: $I(v) = \{(u, v) : (u, v) \in E\}$
Degree of a vertex: # of neighbors that a vertex has
Aside: Vertices are labelled, no pair of adjacent vertices is the same.

Directed graph: a graph where each edge has a direction from one vertex to another[a]

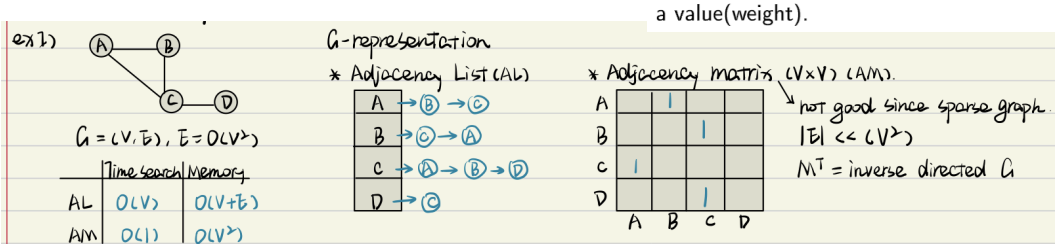
Edges (v_1, v_2) and (v_2, v_1) are different.

Undirected graph: a graph where each edge doesn't have a specific direction

Edges (v_1, v_2) and (v_2, v_1) are indifferent.

Undirected graphs can be thought of directed graphs, with bidirectional edges.

Weighted graph: a graph where each edge is associated with a value(weight).



8. Calculation

In a graph with n vertices, how many edges are there?

At least:

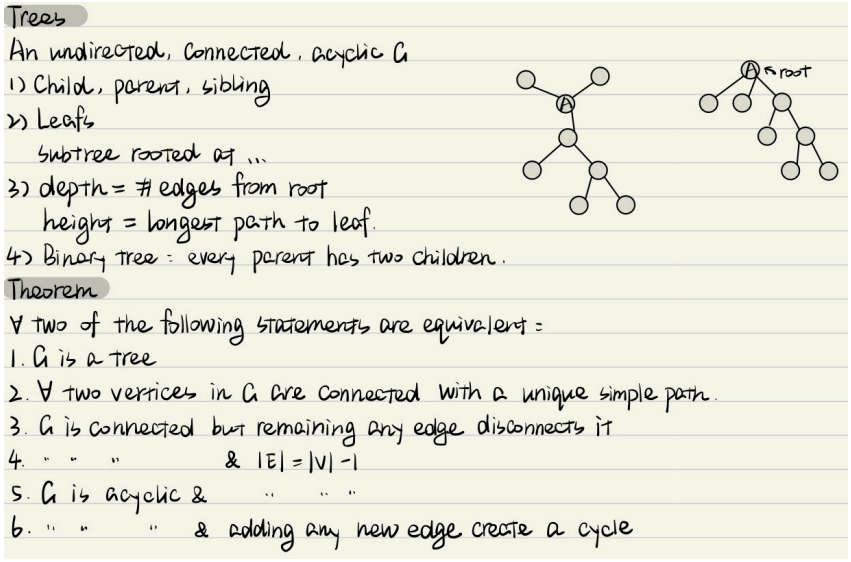
1. connected: $n - 1$
2. not connected: 0

At Most:

1. simple: $\binom{n}{2} = \frac{n(n-1)}{2}$
2. not simple: No upperbound

Number of edge and degrees of vertices (undirected graph): $\sum_{v \in V} \deg(v) = 2m$, where m is the number of edges

9. Tree



10. Probability

	order not important	order important
w/o replacement	$\binom{n}{r} = \frac{n!}{r!(n-r)!}$	$P(n,r) = \frac{n!}{(n-r)!}$
with replacement	$\binom{n+r-1}{r} = \frac{(n+r-1)!}{r!(n-1)!}$	n^r

Recurrences and the Master Theorem

Sorting

1. MergeSort

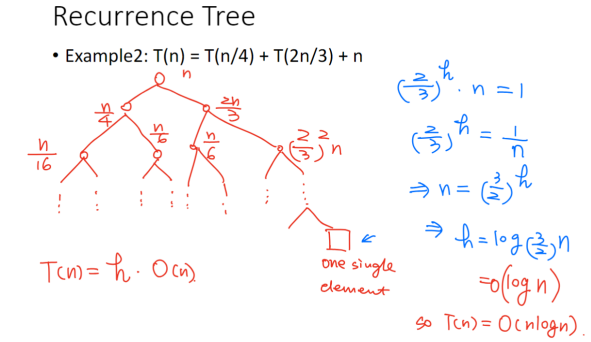
```
T(n)  MERGE-SORT(A, p, r)
Θ(1)  1  if p < r
Θ(1)  2      q = ⌊(p + r)/2⌋
T(n/2) 3     MERGE-SORT(A, p, q)
T(n/2) 4     MERGE-SORT(A, q + 1, r)
Θ(n)  5      MERGE(A, p, q, r)
```

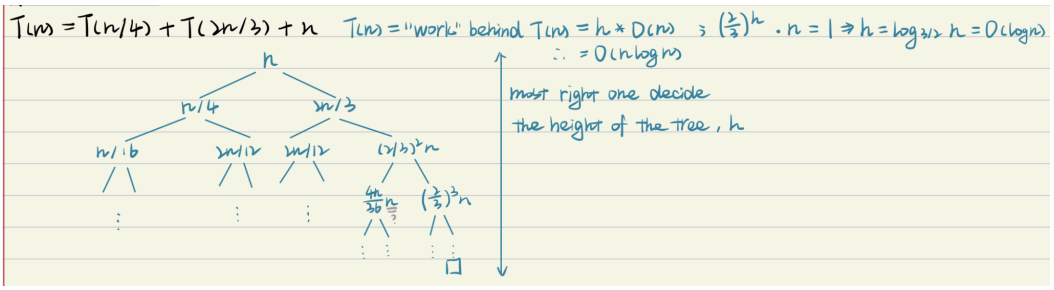
O(n) & Ω(n)

$A = \langle A[1], A[2], \ldots, A[n] \rangle$
p --- Left index
r --- Right index

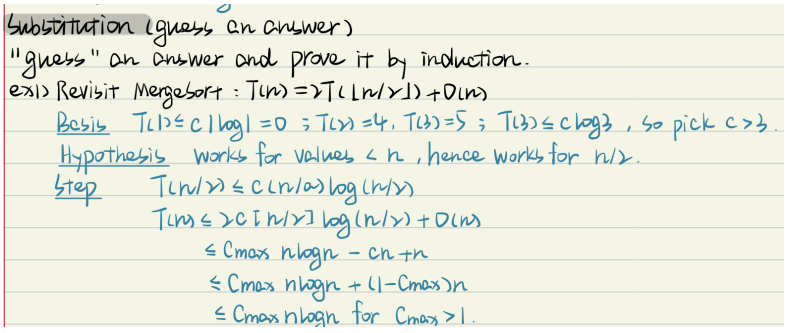
Calculating recursion

2. Recursion tree





3. Substitution



4. Master Theorem
   - Theorem

$T(n) = aT\left(\frac{n}{b}\right) + f(n)$, where $a \geq 1, b \geq 1$, $f(n)$ is asymptotically positive

- Case 1: $f(n) = \mathcal{O}(n^{\log_b a - \epsilon})$, $\epsilon > 0$. Then $T(n) = \Theta(n^{\log_b a})$ (cost of solving the sub-problems at each level increases by a certain factor, the last level dominates)
- Case 2: $f(n) = \Theta(n^{\log_b a})$. Then $T(n) = \Theta(n^{\log_b a} \log n)$ (cost to solve subproblem at each level is nearly equal)
- Case 3: $f(n) = \Omega(n^{\log_b a + \epsilon})$, $\epsilon > 0$ and $af\left(\frac{n}{b}\right) \leq cf(n)$ for some $c < 1$ and $n > n_0$ (regularity condition, always holds for polynomials). Then $T(n) = \Theta(f(n))$ (cost of solving the subproblems at each level decreases by a certain factor)

Method:

1. identify a and b and compute $\log_b a$
2. compare $n^{\log_b a}$ to $f(n)$ and decide which case applies
3. don't forget to check the regularity condition for case 3

UNIVERSITY OF TORONTO

- Example

$T(n) = 7T\left(\frac{n}{2}\right) + n^2$ (Works for $n^2 \log n$)
Solution: $a = 7, b = 2, \log_2 7 \approx 2.8, f(n) = n^2 = \mathcal{O}(n^{\log_2 7 - \epsilon})$ for $\epsilon \in (0, \log_2 7 - 2)$
Case 1, $T(n) = \Theta(n^{\log_2 7})$

$T(n) = 4T\left(\frac{n}{2}\right) + n^2\sqrt{n}$
Solution: $a = 4, b = 2, \log_2 4 = 2, f(n) = n^{2.5} = \Omega(n^{2+\epsilon})$ for $\epsilon \in (0, 0.5]$
Case 3, Check regularity: $af\left(\frac{n}{b}\right) = 4\left(\frac{n}{2}\right)^{2.5} \leq cn^{2.5}$. Choose $c \in \left[\frac{1}{\sqrt{2}}, 1\right)$
$T(n) = \Theta(f(n)) = \Theta(n^{2.5})$

$T(n) = T(\sqrt{n}) + 1$
Solution: let $n = 2^m$, $S(m) = T(2^m) = T(n)$
Then $T(2^m) = T(2^{\frac{m}{2}}) + 1 \Leftrightarrow S(m) = S\left(\frac{m}{2}\right) + 1$
$a = 1, b = 2, \log_2 1 = 0, f(m) = 1 = m^0 = \Theta(m^0)$
Case 2, $T(2^m) = S(m) = \Theta(\log m)$, $T(n) = \Theta(\log \log n)$

Heaps & Heapsort

Heap

1. Definition

Definition: A heap is an array $A = [a_1, a_2, ..., a_n]$ of elements such that:

- Heap shape property: the heap is an almost complete binary tree (all except the last row is full)
- Heap order property:
  - Max-heap: $\forall i,\ A[\text{parent}(i)] \geq A[i]$
  - Min-heap: $\forall i,\ A[\text{parent}(i)] \leq A[i]$

Indexing parents and children (Assume index of array starts at 1)

- $\text{parent}(i) = \lfloor \frac{i}{2} \rfloor$
- $\text{left}(i) = 2i$
- $\text{right}(i) = 2i + 1$

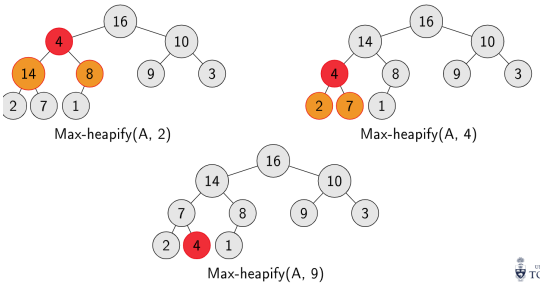
2. Properties

- $2^h \leq n \leq 2^{h+1} - 1 \Leftrightarrow h = \lfloor \log n \rfloor$
- For a max-heap, the maximum value will always occur at the root
- For a min-heap, the minimum value will always occur at the root
- Heapsort is $\Theta(n \log n)$ in time and $\Theta(1)$ in space (space complexity only considers extra memory needed excluding input)
- Heapsort is an in-place algorithm
- Heapsort is not stable

Note: A stable sorting algorithm is a sorting algo that preserves the relative order of the same value in the previous step

Heapsort

3. Max-Heapify (smaller num to the bottom)



Enforces the heap order property if it is violated

1: **function** Max-Heapify(A, i)
2: Compare $A[i]$ with $A[2i]$ and $A[2i+1]$
3: **if** $A[i]$ is smaller **then**
4: swap $A[i] \leftrightarrow \max(A[2i], A[2i+1])$
5: **end if**
6: Recurse downwards, until property is not violated or hit a leaf node
7: **end function**

Runtime: $\mathcal{O}(h) = \mathcal{O}(\log n)$

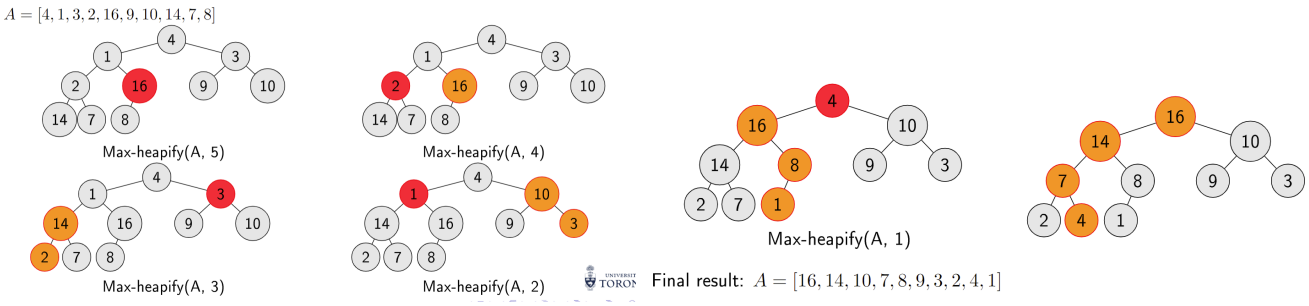
Show the worst-case running time of Max-Heapify is $\Omega(\log n)$
Proof: worst case is the minimum element is at the root.
Max-Heapify will terminate when it goes to the leaf
This takes $\Omega(h) = \Omega(\log n)$ steps

4. Build-Max-Heap (multiple call max-heapify to move max to the top)

1: **function** Build-Max-Heap(A, n)
2: **for each** $i = \lfloor \frac{n}{2} \rfloor : 1$ **do** ▷ the rest are leaf nodes
3: Max-Heapify(A, i)
4: **end for**
5: **end function**

Runtime:

- Simple: $\mathcal{O}(n \log n)$ (for loop $\times$ cost for Heapify)
- Actual: Time to run Max-Heapify is linear in the height of the node it is run on and most node have smaller height
  - At height h, there are at most $\lceil \frac{n}{2^{h+1}} \rceil$ nodes.
  - height of heap is $\lfloor \log n \rfloor$
  - Runtime: $\sum_{h=0}^{\lfloor \log n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil \mathcal{O}(h) \leq cn \sum_{h=0}^{\infty} \frac{h}{2^{h+1}} = \cdots = \mathcal{O}(n)$



5. Heapsort

```
1: function Extract-Max(A)
2:     Swap A[1] ↔ A[A.size()]
3:     A.size = A.size-1
4:     Max-Heapify(A, 1)
5: end function
```

```
1: function Heapsort(A, n)
2:     Build-Max-Heap(A, n)
3:     for each i = n : 2 do
4:         Extract-Max(A)
5:     end for
6: end function
```

Runtime: $\mathcal{O}(n) + \mathcal{O}(n \log n) = \mathcal{O}(n \log n)$

move the largest num out, others remain in heap shape.

6. Searching time

Searching for i^{th} biggest # in Data expected times
Maximum # 2^{nd} max, k^{th} max
Comparison $(n-1) + k\log n$ (total) since search from right branch left most up to start.
Minimum $(n/2) - 1$.

7. Priority Queue

Priority Queue (Lower Bounds. First-in first-out.
1) Insert
2) Delete / Extract max
3) Change priority
Lower Bound: ∀ Comparison-based sorting algorithm of unrestricted range #, take $\Omega(n \log n)$.
Heap is the best to do so.

FINDING THE MAXIMUM ELEMENT

Getting the maximum element is easy: it's the root.

MAX-HEAP-MAXIMUM(A)

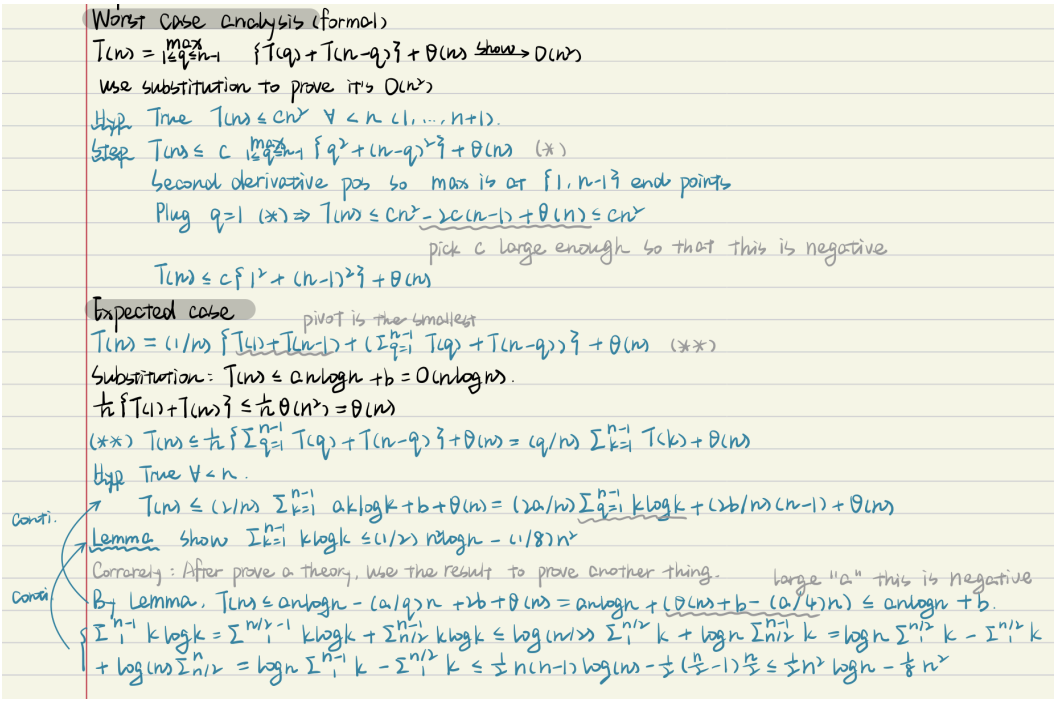
```
1  if A.heap-size < 1
2      error "heap underflow"
3  return A[1]
```

Time: $\Theta(1)$.

Quick Sort

General

1. Runtime analysis
   - Worst case run time: $\Theta(n^2)$ (the list is already sorted)
     - pivot always the largest/smallest. Everytime we get one empty array and one array of size $n-1$. $T(n) = T(n-1) + \mathcal{O}(n)$
   - Best case run time: $\Theta(n \log n)$
     - pivot always median, $T(n) = 2T(n/2) + \mathcal{O}(n)$
   - Average/Expected run time: $\Theta(n \log n)$
     - $T(n) = T(an) + T(bn) + \mathcal{O}(n)$, where $a + b = 1$ as we have to run through the full array.



2. Properties
   - Quicksort tends to have the smallest constant in front of its runtime
   - Quicksort is not stable (although can modify to be stable)
   - Quicksort is in-place (although recursion stores stuff on stack, based on exact definition of in-place)
   - The idea of a randomized algorithm. Why randomization helps and how to analyze a random algorithm

3. Codes

```
1: function Partition(A, p, r)
2:     x = A[r]
3:     i = p − 1
4:     for each j = p : r − 1 do
5:         if A[j] < x then
6:             i = i + 1
7:             Swap A[i] ↔ A[j]
8:         end if
9:     end for
10:    Swap A[i + 1] ↔ A[r]
11:    Return i + 1
12: end function
13: function Randomized-Partition(A, p, r)
14:    i = RAND(p, r)
15:    Swap A[i] ↔ A[r]
16:    Return Partition(A, p, r)
17: end function
```

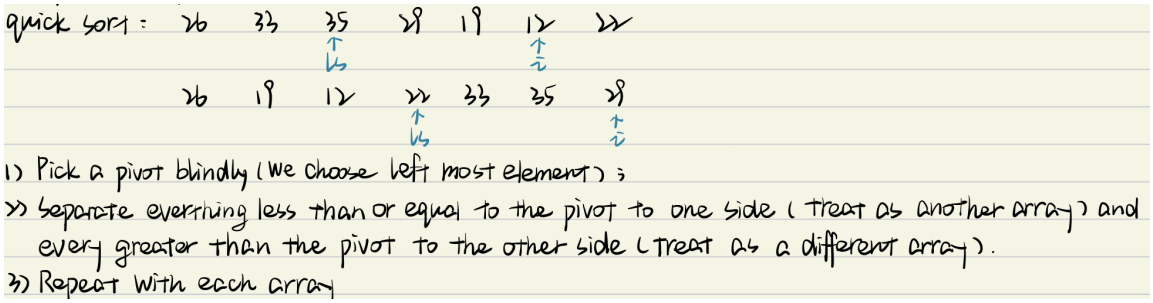
```
1: function Quicksort(A, l, r)          ▷ Checking A[l, ..., r]
2:     if l < r then                    ▷ Otherwise no subarray
3:         p =Partition(A, l, r)        ▷ Split the array by p
4:         Quicksort(A, l, p − 1)       ▷ Recurse on ≤ p
5:         Quicksort(A, p − 1, r)       ▷ Recurse on > p
6:     end if
7: end function
```

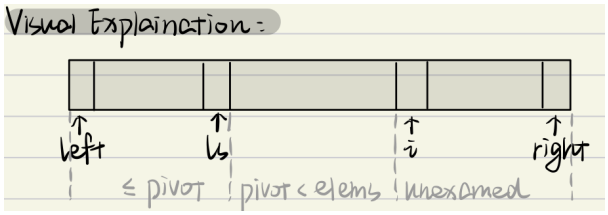
If we randomly shuffle input or choose pivot, we reduce the chance of getting the worst case scenario.
The worst case scenario is still $\mathcal{O}(n^2)$, but the chance is lower.

```
PARTITION(A, p, r)
1  x = A[r]                          // the pivot
2  i = p − 1                         // highest index into the low side
3  for j = p to r − 1                // process each element other than the pivot
4      if A[j] ≤ x                   // does this element belong on the low side?
5          i = i + 1                 // index of a new slot in the low side
6          exchange A[i] with A[j]   // put this element there
7  exchange A[i + 1] with A[r]       // pivot goes just to the right of the low side
8  return i + 1                      // new index of the pivot
```

- Rand-partition: to avoid the worst case take random num from array A, swap it with the left most num.

4. Steps to understand





5. example

$A = [\boxed{26}, 33, 35, 29, 19, 12, 22]$
Quicksort(A, 1, 7)
4=Partition(A, 1, 7) (number of elements smaller than or equal to the pivot value 26, i.e. where this pivot should be placed at)
$[22, 19, 12, \boxed{26}, 33, 35, 29]$
Quicksort(A, 1, 3) and Quicksort(A, 5, 7)
$[\boxed{22}, 19, 12]$ $[\boxed{33}, 35, 29]$
3=Partition(A, 1, 3) and 6=Partition(A, 5, 7)
$[12, 19, \boxed{22}]$ $[29, \boxed{33}, 35]$
Quicksort(A, 1, 2), Quicksort(A, 4, 3), Quicksort(A, 5, 5), Quicksort(A, 7, 7)
$[\boxed{12}, 19]$ $[29]$ $[35]$
1=Partition(A, 1, 2)
$[19]$
Quicksort(A, 1, 0), Quicksort(A, 2, 2)
$[12, 19, 22, 26, 29, 33, 35]$

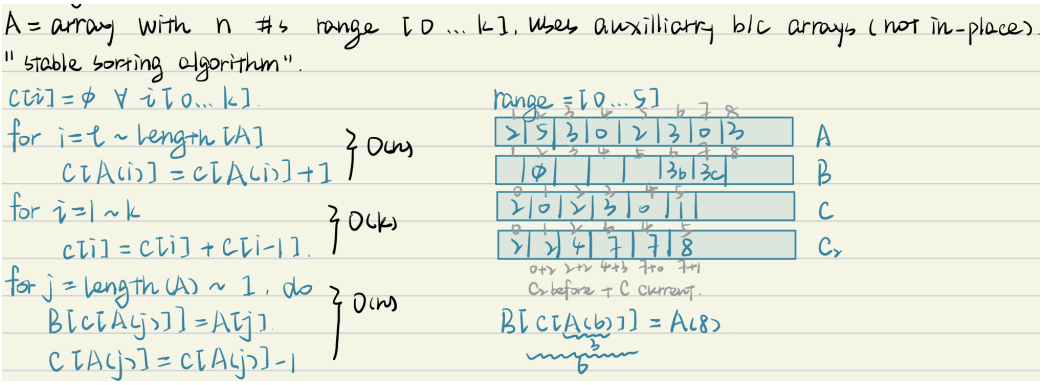
<u>Comparison-based sorting</u>

6. Lower-bound on comparison sorting
   - Any comparison-based sorting algorithm of n nums of unrestricted range cannot do better than $\Omega(nlogn)$
7. Counting Sort
   - Counting sort assumes elements are integers ranging from 0 to k
   - Runtime: $\Theta(n+k)$ (if $k = \mathcal{O}(n)$, then runtime is $\Theta(n)$)
   - Counting sort uses no comparisons (it uses values of elements to determine the position)
   - Counting sort is not in-place
   - Counting sort is stable

Note: A stable sorting algorithm is a sorting algo that preserves the relative order of the same value in the previous step

Input: $A[1, ..., n]$, $A[j] \in \{1, ..., k\}$
Output: $B[1, ..., n]$ sorted
Create an auxiliary array $C[1, ...k]$, recording the number of elements in A with value $\leq x$

```
1: function Counting-Sort(A[1, .., n], B[1, ..., n], n, k)
2:     for each i = 0 : k do
3:         C[i] = 0
4:     end for
5:     for each j = 1 : n do
6:         C[A[j]] = C[A[j]] + 1            ▷ # elements in A with value x
7:     end for
8:     for each i = 1 : k do
9:         C[i] = C[i] + C[i − 1]           ▷ # elements in A with value ≤ x
10:    end for
11:    for each j = n : 1 do                ▷ iterate A backwards
12:        B[C[A[j]]] = A[j]                ▷ C[A[j]] implies the sorted position of A[j]
13:        C[A[j]] = C[A[j]] − 1
14:    end for
15: end function
```

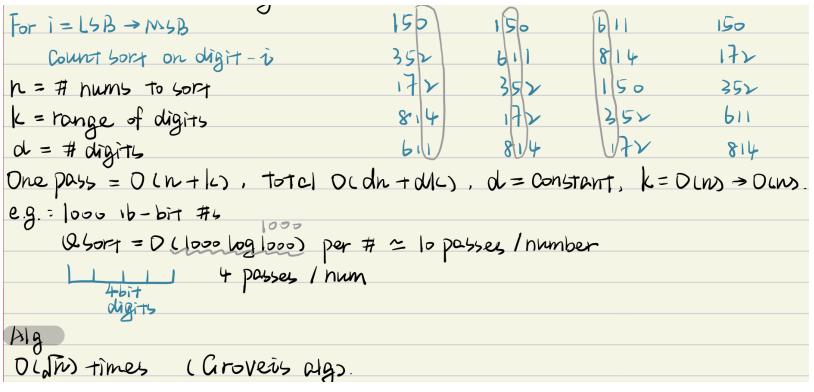


8. Radix Sort (stable sort)

- Radix sort assumes all elements have $\leq d$-digits
- Runtime $\Theta(d(n + k))$ for d-digit numbers and each digit $\in [0, k]$
- Runtime $\Theta\left(\frac{b}{r}(n + 2^r)\right)$ for b-bit numbers and $r = \min(b, \lceil \log n \rceil)$
- Overall, it sorts in $\Theta(n)$
- Radix sort uses no comparisons
- Radix sort is not in-place
- Radix sort is stable

Algorithm

```
1: function Radix-Sort(A, d)
2:     for each i = 1 : d do                ▷ Sort least sig digit first
3:         Stable sort A on digit i         ▷ Relative order in previous step is preserved
4:     end for
5: end function
```



9. Summary about sorting

Given an array of integers, the worst-case time for each algorithm:

Insertion sort: $\mathcal{O}(n^2)$
Merge sort: $\mathcal{O}(n \log n)$
Heap sort: $\mathcal{O}(n \log n)$
Quick sort: $\mathcal{O}(n^2)$
Counting sort: $\mathcal{O}(n + k)$

1. Note 1: k can be larger than n, and it will change the complexity
2. Note 2: Won't work with real numbers or any sets S that are not countable (*i.e.* no surjective $f : \mathbb{N} \to S$) Only work on finite sets.

Radix sort: $\mathcal{O}(d(n + k))$ (Uses counting sort, so restrictions of counting sort applies)
Bucket sort: $\mathcal{O}(n^2)$ (Worst case when all numbers in the same bucket)

(Tournament tree).

```
Randselect (A, p, r, i)
  if p = r then
      return A[p]
  q = rand-part
    k = q - p + 1
    if i ≤ k
        Randselect (A, p, q, i)
```

Sorting Algorithms	Time Complexity			Space Complexity
	Best Case	Average Case	Worst Case	Worst Case
Bubble Sort	Ω(N)	Θ(N^2)	O(N^2)	O(1)
Selection Sort	Ω(N^2)	Θ(N^2)	O(N^2)	O(1)
Insertion Sort	Ω(N)	Θ(N^2)	O(N^2)	O(1)
Quick Sort	Ω(N log N)	Θ(N log N)	O(N^2)	O(N)
Merge Sort	Ω(N log N)	Θ(N log N)	O(N log N)	O(N)
Heap Sort	Ω(N log N)	Θ(N log N)	O(N log N)	O(1)

Selection Sorting

1. Time complexity for search for min

   How many comparisons are necessary to determine the MIN/MAX of a set of n elements?

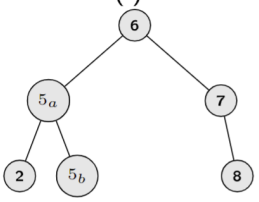
```
MINIMUM(A, n)
1  min = A[1]
2  for i = 2 to n
3      if min > A[i]
4          min = A[i]
5  return min
```

O($n-1$) comparisons

BST

2. Properties
   - 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$[x]$
3. Runtime analysis
   - In general, if a BST has n nodes, $h = \mathcal{O}(n)$. Only if the BST is balanced, $h = \mathcal{O}(\log n)$
   - Minimum/Maximum and searching for any arbitrary key $\mathcal{O}(h)$
   - Successor/Predecessor $\mathcal{O}(h)$
   - Insertion/Deletion $\mathcal{O}(h)$
   - Build-BST: $\mathcal{O}(n^2)$ worst case (chain)
4. Basic operations' code

(i)



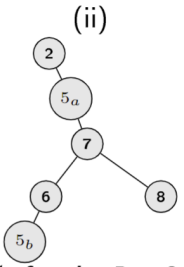

```
1: function In-Order(x)
2:     if x ≠NIL then
3:         In-Order(x.left)
4:         Process x
5:         In-Order(x.right)
6:     end if
7: end function
```

(i)2, 5_a, 5_b, 6, 7, 8
(ii)2, 5_a, 5_b, 6, 7, 8
Used for sorting

```
1: function Pre-Order(x)
2:     if x ≠NIL then
3:         Process x
4:         Pre-Order(x.left)
5:         Pre-Order(x.right)
6:     end if
7: end function
```

(i)6, 5_a, 2, 5_b, 7, 8
(ii)2, 5_a, 7, 6, 5_b, 8
Used for rotation

(ii)

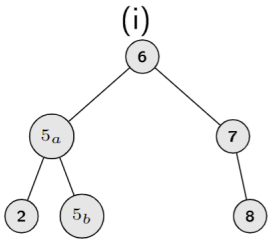



```
1: function Post-Order(x)
2:     if x ≠NIL then
3:         Post-Order(x.left)
4:         Post-Order(x.right)
5:         Process x
6:     end if
7: end function
```

(i)2, 5_b, 5_a, 8, 7, 6
(ii)5_b, 6, 8, 7, 5_a, 2
Used for deleting

$\mathcal{O}(n)$, since we process each node exactly once

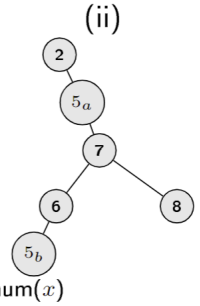
(i)




```
1: function Minimum(x)
2:     while x.left ≠ NIL do
3:         x = x.left
4:     end while
5:     Return x
6: end function
```

(i)$6 \to 5_a \to 2$
(ii)2

(ii)



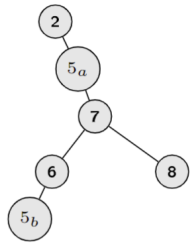

```
1: function Maximum(x)
2:     while x.right ≠ NIL do
3:         x = x.right
4:     end while
5:     Return x
6: end function
```

(i)$6 \to 7 \to 8$
(ii)$2 \to 5 \to 7 \to 8$

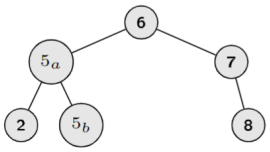
$\mathcal{O}(h)$, since in the worst case we have to traverse the full height

```
1: function Predecessor(x)
2:     if x.left ≠ NIL then
3:         Return Maximum(x.left)
4:     end if y =parent(x)
5:     while y ≠ NIL and x == y.right do
6:         x = y
7:         y =parent(y)
8:     end while
9:     Return y
10: end function
```

$\mathcal{O}(h)$, since in the worst case we have to traverse the full height



Predecessor(5_b) = 5_a



Predecessor(6) = 5_a

5. Search

```
1: function Search(x, k)
2:     if x==NIL or x.key == k then
3:         Return x
4:     end if
5:     if x.key > k then
6:         Return Search(x.left, k)
7:     end if
8:     if x.key < k then
9:         Return Search(x.right, k)
10:    end if
11: end function
```

$\mathcal{O}(h)$, since in the worst case we have to traverse the full height

6. Insert

```
TREE-INSERT(T, z)
1   x = T.root              // node being compared with z
2   y = NIL                 // y will be parent of z
3   while x ≠ NIL           // descend until reaching a leaf
4       y = x
5       if z.key < x.key
6           x = x.left
7       else x = x.right
8   z.p = y                 // found the location—insert z with parent y
9   if y == NIL
10      T.root = z          // tree T was empty
11  elseif z.key < y.key
12      y.left = z
13  else y.right = z
```


```
1: function Insert(T, z)
2:     y=NIL, x = T.root
3:     while x ≠NIL do
4:         y = x                ▷ Find where z should connect
5:         if z.key < x.key then
6:             x = x.left
7:         else
8:             x = x.right
9:         end if
10:    end while
11:    parent(z)=y
12:    if y==NIL then
13:        T.root = z
14:    else if z.key < y.key then
15:        y.left=z
16:    else
17:        y.right=z
18:    end if
19: end function
```

Build a BS

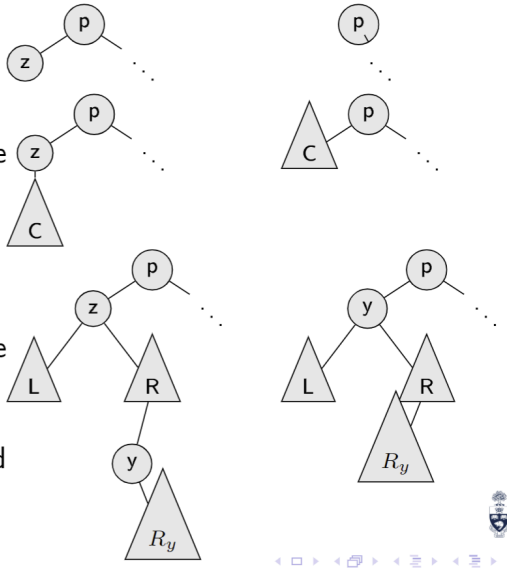
$\mathcal{O}(h)$, since in the worst case we have to traverse the full height

7. Delete

Case 1: z is a leaf, delete it.

Case 2: z has one child, delete z and replace with child.

Case 3: z has 2 children.
Let y be z's successor, y never has a left subtree by defintion.
If y has no children, replace z with y
If y has a right subtree, replace z with y and replace y with right child



TRANSPLANT

TRANSPLANT(T, u, v) replaces the subtree rooted at u by the subtree rooted at v:

- Makes u's parent become v's parent (unless u is the root, in which case it makes v the root).
- u's parent gets v as either its left or right child, depending on whether u was a left or right child.
- Doesn't update $v.left$ or $v.right$, leaving that up to TRANSPLANT's caller.

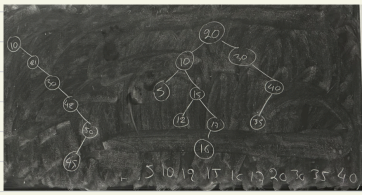
```
TRANSPLANT(T, u, v)
1  if u.p == NIL
2      T.root = v
3  elseif u == u.p.left
4      u.p.left = v
5  else u.p.right = v
6  if v ≠ NIL
7      v.p = u.p
```

```
TREE-DELETE(T, z)
1  if z.left == NIL
2      TRANSPLANT(T, z, z.right)          // replace z by its right child
3  elseif z.right == NIL
4      TRANSPLANT(T, z, z.left)           // replace z by its left child
5  else y = TREE-MINIMUM(z.right)         // y is z's successor
6      if y ≠ z.right                     // is y farther down the tree?
7          TRANSPLANT(T, y, y.right)      // replace y by its right child
8          y.right = z.right              // z's right child becomes
9          y.right.p = y                  //      y's right child
10     TRANSPLANT(T, z, y)                // replace z by its successor y
11     y.left = z.left                    // and give z's left child to y,
12     y.left.p = y                       //      which had no left child
```

Note that the last three lines execute when z has two children, regardless of whether y y is z's right child.

8. example

Binary Search Trees
A binary tree with property key(left) ≤ key(pivot) ≤ key(right)
Search key O(h)
Start at root and go left or right
Insert key O(h)
Search for it and insert it as a leaf
Build BST(A[1...n]) O(nh)
For i=1...n, insert(A[i])
Sort BST
In order: left → print → right



Show if a BST node has 2 children, then the successor has no left child and the predecessor has no right child

Proof: Assume that node x has 2 children, and its successor s (minimum element of the BST rooted at x.right) has left child l, s.key $> l$.key $> x$.key, then l is the successor of x. Contradiction

Similarly, if its predecessor p (maximum element of the BST rooted at x.left) has right child r, we have x.key $> r$.key $> p$.key, r is the predecessor of x. Contradiction.

BST-sort works by constructing a BST out of the array and then calling In-Order traversal. What is the best and worst case?

Solution: In-Order always take $\Theta(n)$

Worst case: array is already sorted, BST is a linked list

Each BST at position i insert takes $\mathcal{O}(i)$, summing up gives $\sum_{i=1}^{n} \mathcal{O}(i) = \mathcal{O}(\sum_{i=1}^{n} i) = \mathcal{O}(n^2)$

Best case: BST is always perfectly balanced, height is always $\mathcal{O}(\log i)$

$\sum_{i=1}^{n} \mathcal{O}(\log i) = \mathcal{O}(\sum_{i=1}^{n} \log i) = \mathcal{O}(\sum_{i=1}^{n} \log n) = \mathcal{O}(n \log n)$

UNIVERSITY OF TORONTO

9. NIL
    - Absence of a node

RBT

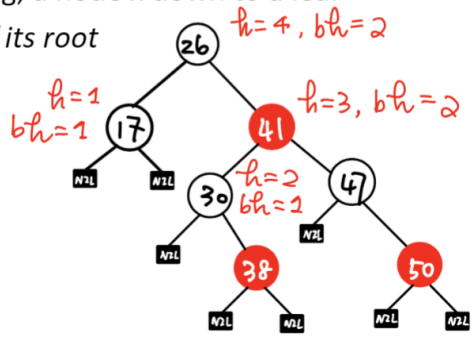
10. Properties

    Red Black Tree is a Binary Search Tree with the following additional properties:
    - Every node is either red or black
    - The root is black
    - All leaves (NIL) are black
    - If a node is red, both its children are black
    - For all nodes, all paths to all leaves have the same black-height (number of black nodes on path to a leaf, not including itself, but including NIL)

11. Balanced and run time
    - Balanced

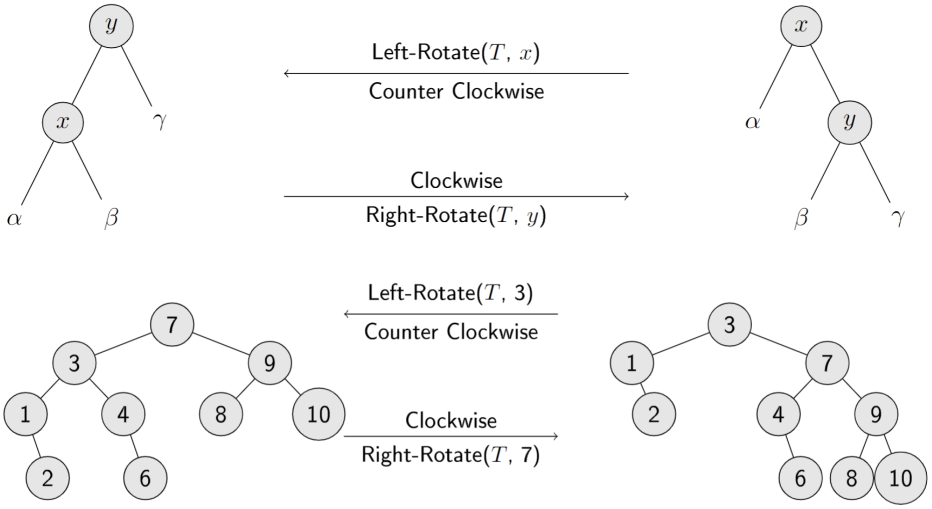
      By constraining the node colors on any simple path from the root to a leaf, red-black trees ensure that no such path is more than twice as long as any other, so that the tree is approximately balanced
        - All read-only operations (e.g. traversals) are all the same as BST
        - Rotation
    - Run time
        - RB Treees are balanced $h = \mathcal{O}(\log n)$

12. Definition
    - Height of the node: #edges in longest path to leaf
    - **Black-Height of node x**, denoted *bh(x)* -- number of black nodes on any simple path from, but not including, a node x down to a leaf
    - ***Black-Height of a RBT**: black-height of its root*



13. Operations and rotations

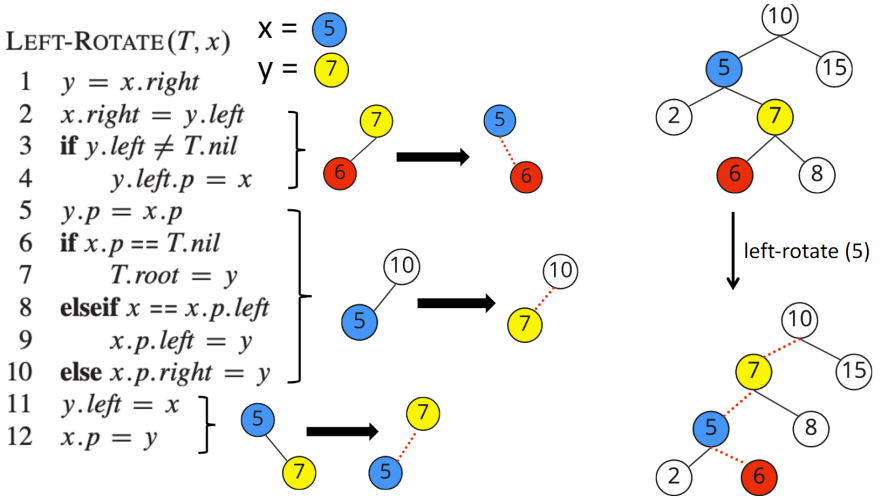
- Operations
  - MINIMUM, MAXIMUM, SUCCESSOR, PREDECESSOR, SEARCH are same as in BST
  - Insert, delete require ROTATION to fix the properties when they are violated
    - To restore these properties, we must change the colors of some of the nodes in the tree and also change the pointer structure.
- Rotations



LEFT-ROTATE(T, x)

```
1   y = x.right                 // set y
2   x.right = y.left            // turn y's left subtree into x's right subtree
3   if y.left ≠ T.nil
4       y.left.p = x
5   y.p = x.p                   // link x's parent to y
6   if x.p == T.nil
7       T.root = y
8   elseif x == x.p.left
9       x.p.left = y
10  else x.p.right = y
11  y.left = x                  // put x on y's left
12  x.p = y
```

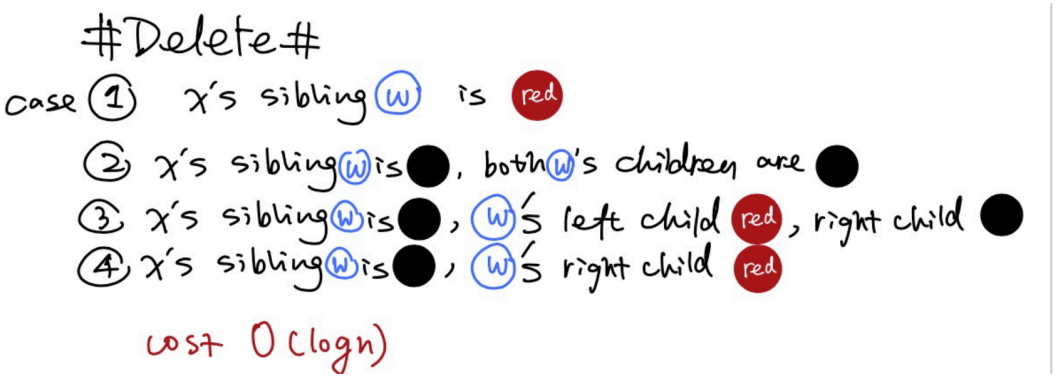
assumes that x.right != T.nil and that the root's parent is T.nil.



14. Insert

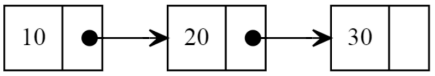
```
RB-INSERT-FIXUP(T, z)
 1  while z.p.color == RED
 2      if z.p == z.p.p.left
 3          y = z.p.p.right
 4          if y.color == RED
 5              z.p.color = BLACK                    // case 1
 6              y.color = BLACK                      // case 1
 7              z.p.p.color = RED                    // case 1
 8              z = z.p.p                            // case 1
 9          else if z == z.p.right
10                  z = z.p                          // case 2
11                  LEFT-ROTATE(T, z)                // case 2
12              z.p.color = BLACK                    // case 3
13              z.p.p.color = RED                    // case 3
14              RIGHT-ROTATE(T, z.p.p)               // case 3
15      else (same as then clause
                with "right" and "left" exchanged)
16  T.root.color = BLACK
```

15. Delete



LinkedList

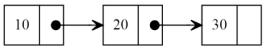
1. Properties
   - Objects arranged in a linear order.
   - Order is determined by a pointer in each object.
   - A flexible representation for dynamic set.
     Supporting all the dynamic-set operations (Search, insert, delete, minimum, maximum, successor, predecessor)



Linked List

2. Different types

   Linked lists come in several types:



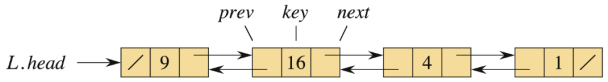
   - ***Singly linked***: each element has a next attribute but not a *prev* attribute.
   - ***Doubly linked: each element x has the following attributes"***
     - $x.key$
     - $x.next$: the successor of x, NIL if x has no successor so that it's the ***tail***
     - $x.prev$: the predecessor of x, NIL if x has no predecessor so that it's the ***head***

   $L.head$ points to the first element of the list, NIL if the list is empty.

   Linked lists come in several types:

   - ***Sorted***: the linear order of the list follows the linear order of keys stored in elements of the list.
   - ***Unsorted***: the elements can appear in any order.
   - ***Circular***: the *prev* pointer of the head of the list points to the tail, and the next pointer of the tail of the list points to the head.

3. Searching



LIST-SEARCH finds the first element with key k in list L by a linear search.

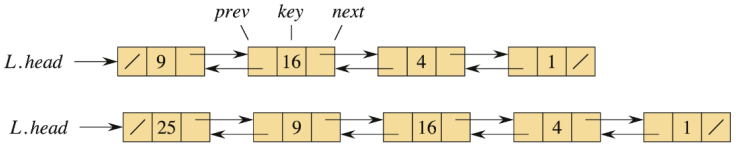
LIST-SEARCH(L, k)

```
1  x = L.head
2  while x ≠ NIL and x.key ≠ k
3      x = x.next
4  return x
```

4. Insert

   There are two scenarios for inserting into a doubly linked list:

   1. inserting a new first element: LIST-PREPEND, $O(1)$
   2. inserting anywhere else: LIST-INSERT , $O(1)$



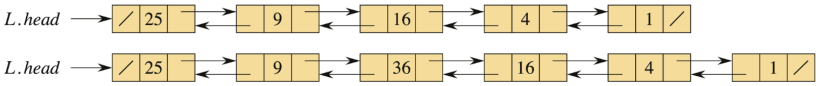
LIST-PREPEND(L, x)

```
1  x.next = L.head
2  x.prev = NIL
3  if L.head ≠ NIL
4      L.head.prev = x
5  L.head = x
```

To insert elsewhere, LIST-INSERT "splices" a new element x into the list, immediately following y. Since the list object L is not referenced, it's not supplied as a parameter.

```
LIST-INSERT(x, y)
1  x.next = y.next
2  x.prev = y
3  if y.next ≠ NIL
4      y.next.prev = x
5  y.next = x
```

e.g., inserting 36 after 9.

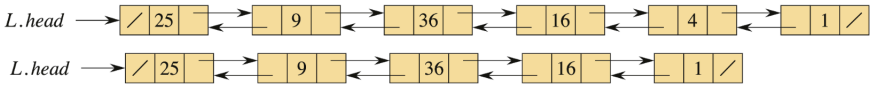


5. Delete

Given a point to x, LIST-DELETE removes x from L in $O(1)$ time.

```
LIST-DELETE(L, x)
1  if x.prev ≠ NIL
2      x.prev.next = x.next
3  else L.head = x.next
4  if x.next ≠ NIL
5      x.next.prev = x.prev
```

e.g., deleting 4.



To delete an element just given a key, first call LIST-SEARCH, then call LIST-DELETE. This makes the worst-case running time $\Theta(n)$.

6. Runtime Comparison

Linked List, Array, BinarySearchTree (operation by key)

		Insert	Delete	Search		Space
				By index	By key	
Array	Sorted / Un-Sorted	Sorted: O(N)	O(N)	O(1)	O(N)	O(N)
BST	Average	O(lgN)	O(lgN)	O(lgN)		O(N)
	Worst	O(N)	O(N)	O(N)		
Linked List		O(N) / O(1)	Search first O(N) / O(1)	O(N)		O(N)

Hash Tables

7. Definitions

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

Good hashing scheme:

- Simple uniform hashing: any given element is equally likely to be hashed into any of the m slots.
- Good mechanism for collision resolution (since $m < |U|$, there could be collision)

8. Hash function types

- Division method

Division method: divide the element with the size of the hash table and use the remainder as the index of the element in the hash table.

$h(k) = k \bmod m$.

Example: $m = 20$ and $k = 91 \Rightarrow h(k) = 11$.

Advantage: Fast, since requires just one division operation.

Good choice for m: A prime not too close to an exact power of 2.

- Multiplication method

  1. Choose constant A in the range $0 < A < 1$.
  2. Multiply key k by A.
  3. Extract the fractional part of kA.
  4. Multiply the fractional part by m.
  5. Take the floor of the result.

  Put another way, $h(k) = \lfloor m\,(kA \bmod 1)\rfloor$, where $kA \bmod 1 = kA - \lfloor kA\rfloor =$ fractional part of kA.

  Disadvantage: Slower than division method.

  Advantage: Value of m is not critical. Can choose it independently of A.

Collision Resolution

9. Open Addressing

Use hash function of the form $h(k, i)$, try to insert k at $h(k, 0)$, $h(k, 1)$, ..., until we find an empty slot

Linear probing: $h(k, i) = (h'(k) + i) \mod m$

Quadratic probing: $h(k, i) = (h'(k) + c_1 i + c_2 i^2) \mod m$ (slightly better performance, but still m probing sequences)

Double hashing: $h(k, i) = (h_1(k) + i h_2(k)) \mod m$

- No clusters
- $h_2(k)$ must be relatively prime to m ($\gcd(h_2(k), m) = 1$) for all k
- Can choose m prime, $h_2(k) < m$, $\forall k$
- $\Theta(m^2)$ different probe sequences

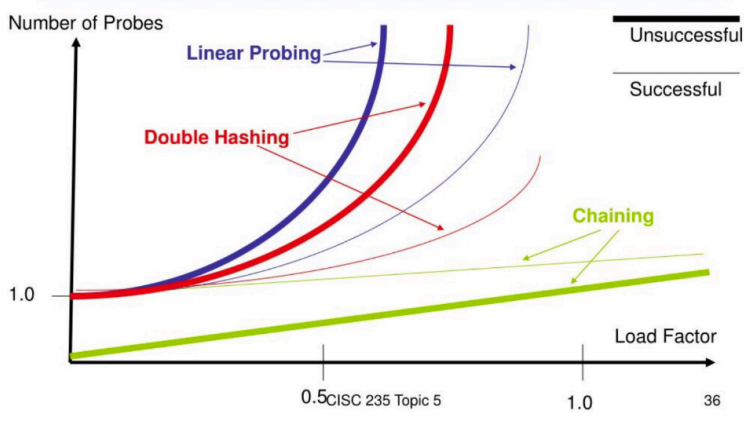
UNIVERSITY OF TORONTO

Insert 22, k=22
h1(22) = 22 mod 11 = 0, collision with 88,
∴ use secondary hash function,
∴ h2(22) = 22×3 mod 4 = 66 mod 4 = 2, so probing step 2 start at idx 0,
Check idx 2, occupided by number 23,
∴ Check idx 4, empty ∴ idx 4 has value 22

Runtime

10. runtime

Performance:
Chaining = 1+α/2
Linear probing: ½(1+1/(1-α)).
Double hashing: (1/α)ln(1/(1-α)).



Dynamic Programming

1. definitions

   Simple defintion: Efficient recursion for solving well-behaved optimization problems

   Optimization problems: trying to find an optimal solution given some constraints (often discrete problems)

   Used for problems with the following properties

   - Optimal substructure: optimal solutions incorporate optimal solutions to related subproblems
   - Overlapping subproblems: solving the same subproblems over and over again (Memorization exploits this redundancy)

   In general, DP good for tasks with small but repetitive search spaces

   Fibonacci series: $F_n = F_{n-1} + F_{n-2}$ with $F_1 = F_2 = 1$. Each value F_i needs to be solved as subproblem for F_{i+1} and F_{i+2}. Similarly, Pascal's triangle.

   String/Array algorithms: longest common subsequence, longest increasing subsequence, longest common substring, edit distance, interleaving strings, balanced array, etc.

   Graph algorithms: Bellman-Ford, Dijkstra (to be discussed later in the course).

   Backpropagation: compute the gradients for one layer at a time starting from the last layer. When calculating gradients for layer i, use the result from layer $i+1$.

   Viterbi algorithm: with a hidden markov model defined by $(S,\ O,\ P(s_{t=0} = s_i),\ P(s_{t+1} = s_j | s_t = s_i),\ P(O_t | s_t = s_i))$, find the most probable state path given the output path. Used for part-of-speech tagging, speech recognition etc. in NLP.

   More examples on Leetcode, but in the course, backtracking is not discussed and a correct working code is not enough.

2. Methodology

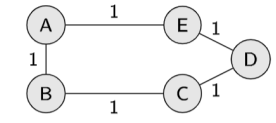
   1. Visualize Examples (start small, with base cases, and gradually increase the problem size)
   2. Determine how the problem exhibits optimal substructure, i.e. characterize the structure of an optimal solution
   3. Find a relationship among subproblems, i.e. defining the rule of an optimal solution recursively in terms of the optimal solution to subproblems
   4. Compute the value of an optimal solution, typically bottom-up solving subproblems in order and use memorization
   5. Construct the optimal solution from the computed information (if you want more than just the value of the optimal solution, you can still use memory)

3. Optimal Substructure proof

   - Definition

     An optimal solution contains within it optimal solutions to subproblems.

     E.g. shortest path problem: find shortest path between nodes in a graph without reapting an edge.

     ShortestPath(A,D)=$A \to E \to D$
     ShortestPath(A,E)=$A \to E$
     ShortestPath(E,D)=$E \to D$

     Note: Optimal substructure does not mean you can combine any optimal solutions to subproblems to arrive at another optimal solution
     E.g. ShortestPath(A,C)=$A \to B \to C$, and ShortestPath(C, E)=$C \to D \to E$, ShortestPath(A,E)$\neq$ ShortestPath(A,C) + ShortestPath(C, E)

- Steps

  Proving Optimal substructure (not required)

  1. The solution to the problem requires some choice. Suppose an optimal solution contains choices k, this breaks the problem into 2 subproblems.
  2. Argue that you would now want to solve these subproblems optimally because if you didn't, the original solution wouldn't be optimal (Proof by contradiction)

  Recurrence relation:

$$A[i, j, ...] = \begin{cases} \text{base case} \\ \min / \max\{A[\text{subproblem 1}] + A[\text{subproblem 2}] + \cdots + \text{cost of breaking problem}\} \end{cases}$$

  Runtime: $\approx \mathcal{O}(n^{\#\text{subproblems per choice}})\mathcal{O}(n^{\#\text{choices}})$ (not necessarily true, use code to analyze)
  For bottom-up approach, runtime is obvious.

4. Problem templates
   - Answer parts
     - Optimal substructure: Simply describe the method to divide the subproblems/subgroups, e.g. cut from where, to divide the total number of stuffs into how many groups, etc.
     - Recursive relationship among subproblems: fix one group (suppose we know one group's optimal solution), then calculate the other side's optimal solution.
     - Code: Usually two loops, the outer loops iterate from 1 to n as the total number of stuff, inner loop considers all possible combinations.
   - Rod cutting problem

     Given: a rod of length n and a table of prices p_i for each rod length $i = 1, ..., n$
     Goal: Determine maximum revenue you can obtain from the rod by cutting it into various size pieces and selling them

     Visualization:
     Suppose we have a rod of length 4 with prices:

length	1	2	3	4
price ($)	1	5	8	9

     Possible cuts:
     $[1, 1, 1, 1]$: \$4; $[2, 2]$: \$10 (optimal);
     $[1, 3]$: \$9; $[1, 1, 2]$: \$7; $[4]$: \$9

     Naive solution: Try all possible cuts.
     This will be $\mathcal{O}(2^n)$, since you either cut or not (2 possibilities for $n - 1$ spots)

     Optimal Substructure:
     Suppose in the optimal solution, we make a cut at position k and we have 2 rods
     The optimal solution must now contain the optimal way to cut the 2 sub-rods. If not, we could replace them with the optimal sub-solution to obtain a better optimal solution.
     $\therefore$ this problem exhibits optimal substructure

     Recursive relationship among subproblems:
     Let r_n be the maximum revenue you can get from a rod of length n
     $r_n = \max(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, ..., r_{n-1} + r_1) = \max_{k \in [1,n]} (r_k + r_{n-k})$
     We can simplify things. Instead of considering 2 subproblems, we fix the length of the rod on one side. $r_n = \max_{k \in [1,n]} (p_k + r_{n-k})$

     Naive implementation:

```
function Rod-Cut(p, n)
    if n == 0 then
        Return 0
    end if
    q = -∞
    for each i = 1 : n do
        q = max(q, p[i] + Rod-Cut(p, n − i))
    end for
end function
```

     However, we can make it more efficient by adding a memory. This gives a top-down approach.

     DP with bottom up approach ($\mathcal{O}(n^2)$):

```
function Rod-Cut(p, n)
    r[0, ..., n] = 0
    for each j = 1 : n do
        q = -∞
        for each i = 1 : j do
            q = max(q, p[i] + r[j − i])
        end for
        r[i] = q
    end for
end function
```

p_i=price of rod with length i
r_i=optimal revenue of rod with length i

$j = 1$: $p_1 = 1$, $r_1 = \max(1) = 1$
$j = 2$: $p_2 = 5$, $p_1 + r_1 = 1 + 1 = 2$, $r_2 = \max(5, 2) = 5$
$j = 3$: $p_3 = 8$, $p_1 + r_2 = 1 + 5 = 6$, $p_2 + r_1 = 5 + 1 = 6$, $r_3 = \max(8, 6, 6) = 8$
$j = 4$: $p_4 = 9$, $p_1 + r_3 = 1 + 8 = 9$, $p_2 + r_2 = 5 + 5 = 10$, $p_3 + r_1 = 8 + 1 = 9$,
$r_4 = \max(9, 9, 10, 9) = 10$

- Typesetting problem

Typeset words of length $w_1, ..., w_n$ into lines of length L, assuming $\forall i,\ w_i \leq L$.

Length of a line consisting of words i to j is $l(i, j) = \sum_{k=i}^{j} w_k + (j - i)$

We want to minimize the amount of spaces on the RHS across all lines
The "raggedness" is defined as $r(i, j) = (L - l(i, j))^2$

Optimal Substructure:
Suppose the optimal solution has words w_i to w_j on line k
We now have 2 similar subproblems: optimally typesetting words $w_1, ..., w_{i-1}$ and $w_{j+1}, ..., w_n$

Recursive relationship among subproblems:
Let $R[i]$= the minimum raggedness to typeset words $w_1, ..., w_i$

$$R[i] = \begin{cases} 0, \text{ if } i = 0 \\ \min_{j \in [0,i], l(j+1,i) \leq L}(R[j] + r(j + 1, i)), \text{ if } i > 0 \end{cases}$$

where $r(j + 1, i) = (L - l(j + 1, i))^2$.

Implementation:

```
function Typeset(L, l)
    R[0, ..., n] = 0
    for each i = 1 : n do
        R[i] = ∞
        for each j = 0 : i − 1 do
            if l(j + 1, i) ≤ L then
                r = (L − l(j + 1, i))^2
                R[i] = min(R[i], R[j] + r)
            end if
        end for
    end for
    Return R[n]
end function
```

Runtime: $\mathcal{O}(n^2)$
Space: $\mathcal{O}(n)$

Greedy Algorithms

Greedy

1. Definition & Algorithms

   Simplified Definition: an algorithm where locally optimal decisions lead to a globally optimal solution
   Idea: when making a choice, take the one that looks the best right now. Local optimality leads to global optimality.
   Note: Greedy is not always optimal, but good as approximation algorithms

   Properties:

   - Greedy choice property: the optimal solution agrees with the first greedy choice
   - Smaller subproblems: after making the greedy choice, the resulting subproblem reduces in size
   - Optimal substructure: an optimal solution contains optimal solutions to subproblems

   - 2 principles/properties
     - Optimal sub-structure
     - Greedy Choice: a global optimal is reached by doing local greedy choices

2. Types of greedy

   There are only two types of greedy problems you need to consider in this course:

   1. Car building: rearrange the full set of tasks to minimize the total penalty
   2. Scheduling: find a feasible subset of tasks to maximize the profit

3. Proof of correctness

   Notation:
   Optimal solution: $J = j_1, j_2, ..., j_n$
   Greedy solution: $G = g_1, g_2, ..., g_n$
   j_i and g_i are indices into some object set. If the task set $T = \{t_1, t_2, t_3\}$ and $J = j_1, j_2, j_3$, the optimal ordering is $t_{j_1}, t_{j_2}, t_{j_3}$
   We treat J and G as ordered sets and use set operations $J \backslash \{j_1\} = j_2, j_3, ..., j_n$, $\{j_0\} \cup J = j_0, j_1, ..., j_n$

   Proof of correctness:

   1. let $G = g_1, g_2, ..., g_n$
   2. First greedy choice: $\exists J = j_1, ..., j_n$ s.t. $g_1 = j_1$
   3. Smaller subproblems: After making choice $j_1 = g_1$, we are left with a smaller subproblem
   4. Optimal substructure: we must solve the subproblem optimally for the original solution to be optimal. $J \backslash \{j_1\} = j_2, j_3, ..., j_n$ must be an optimal solution to the subproblem
   5. Recurse the argument to conclude $j_2 = g_2$, $j_3 = g_3$

   $\therefore \exists J$ optimal solution s.t. $J = G$

4. Comparison

	Greedy Approach	Dynamic Programming
Main Concept	Choosing the best option that gives the best profit for the current step	Optimizing the recursive backtracking solution
Optimality	Only if we can prove that local optimality leads to global optimality	Gives an optimal solution
Time Complexity	Polynomial	Polynomial, but usually worse than the greedy approach
Memory Complexity	More efficient because we never look back to other options	Requires a DP table to store the answer of calculated states
Examples	Dijkstra and Prim's algorithm	0/1 Knapsack and Longest Increasing Subsequence

5. Binomial distribution

$$P(x) = \binom{n}{x} p^x q^{n-x} = \frac{n!}{(n-x)!\,x!} p^x q^{n-x}$$

where
n = the number of trials (or the number being sampled)
x = the number of successes desired
p = probability of getting a success in one trial
$q = 1 - p$ = the probability of getting a failure in one trial

6. Problem templates
   - Answer parts
     - Devise a greedy algorithm
     - Prove the greedy choice property
     - Prove the problem is reduced to a smaller subproblem after making the first greedy choice
     - Prove optimal substructure: (If we remove the first greedy choice from optimal solution, the the remaining solution must be optimal for a subproblem)
   - Car building
     - Sort in descending order, build from high penalty to lower [O(nlogn)]
     - Let J (1~n) be the optimal solution, G (1~n) be the greedy solution.
       If j_1 = g_1 then done.
       Suppose j_1 != g_1.
       Since must build all cars at the end, so g must be somewhere in J.
       By definition of greedy choice, penalty_j1<=penalty_g1=penalty_jm for m > 1.
       Let J' be the set of solutions that j_1 and j_m swapped.
       According to the equation of penalty write P(J) and P(J').
       Find the relation between them by using P(J) to subtract to rewrite P(J'), proving that P(J') <= P(J).
       Then, J' is an optimal solution that agrees with the first greedy choice.
     - Simply describe what will be the left occasion if we built a car on day one.
     - J is optimal with j_1 = g_1, J'=J/j_1 is not optimal and acts as a subproblem.
       N from n2 as a new optimal solution that better than J'.
       H as j1+N (new optimal solution plus the first greedy choice) solution set.
       Prove that P(H) = p_j1+P(N) is better than same plus P(J') hence P(J).
       Proved by contradiction.
   - scheduling

     Given $T = \{t_1, t_2, ..., t_n\}$ a set of tasks, $D = \{d_1, ..., d_n\}$ a set of corresponding deadlines in units of days, $Q = \{p_1, ..., p_n\}$ a set of corresponding profits if we complete the task by the deadline.
     Each task takes exactly 1 day to complete. A feasible set is a set of tasks s.t. it is possible to complete all tasks before their deadline
     Goal: Find a feasible set of tasks s.t. their profits $\sum_{j \in J} p_j$ are maximized.

     e.g.

i	1	2	3	4
t_i	a	b	c	d
d_i	2	1	2	1
p_i	50	10	15	30

     Invalid solution: $\{b, d\}$, both needs to completed on day 1
     Solution 1: $\{d\}$, profit 30
     Solution 2: $\{d, a\}$, profit 30 + 50

(conti next page)

Devise a greedy algorithm assuming you have a function Feasible(J) that runs in constant time and check if J is a feasible schedule
Sort tasks in order of decreasing profit. $J = \{\}$, iterate through the tasks, if Feasible($J \cup \{j\}$), then $J = J \cup \{j\}$.

Prove the greedy choice property:
Let $J = j_1, j_2, ..., j_m$ be an optimal solution.
Let $G = g_1, g_2, ..., g_m$ be the greedy solution.
Note that the sequence of selection doesn't matter here. We need to finish all tasks in J and all tasks in G before the corresponding deadlines. If $g_1 \in J$, then done
Assume $g_1 \notin J$, then $p_{g_1} \geq p_j$, $\forall j \in J$ by construction of g_1
Construct $J' = J \backslash \{j\} \cup \{g_1\}$ for any $j \in J$ s.t. J' is feasible.
$\therefore P(J') = P(J) - P(\{j\}) + P(\{g_1\}) = P(J) - p_j + p_{g_1} \geq P(J)$
$\therefore J'$ is optimal, and J is not the optimal solution. Contradiction.

Prove the problem is reduced to a smaller subproblem after making the first greedy choice:
Start with $T = \{t_1, ..., t_n\}$, make first greedy choice g_1 with profit p_{g_1} where $p_{g_1} \geq p_k$, $\forall k \in \{1, ..., n\}$
Now, we have $T \backslash \{t_{g_1}\}$ possible tasks remaining, but with less days to complete them
So we have a subproblem T'.
Also, there are possibly some tasks that is now impossible to complete because we don't have enough days and we can remove them. $T' \subset T \backslash \{t_{g_1}\}$
$|T'| \leq |T| - 1 < |T|$, T' is a strictly smaller problem

Prove optimal substructure:
Let $J = j_1, j_2, ..., j_m$ s.t. $g_1 \in J$ be an optimal solution
Let $J' = J \backslash \{g_1\}$ be a solution to the subproblem
Assume that J is optimal and J' is not
Let S be an optimal solution to the subproblem $P(S) > P(J')$ and construct $J'' = \{g_1\} \cup S$. J'' is feasible because we can just do g_1 first and S is feasible for the subproblem.
$P(J'') = P(S) + P(\{g_1\}) > P(J') + P(\{g_1\}) = P(J)$. Contradiction, since J'' is now more optimal.

ECE345 Final - TUT0101

Amortized Analysis

General

1. Definition
   - Show that although some individual operations may be expensive, on average the cost per operation is small.
   - Guarantee the average performance of each operation in the worst case.
   - Give a much tighter bound on the true cost of using the data structure than a standard worst-case-per-operation bound.

Aggregate Analysis

2. Definition
   - Determine an upper bound T(n) on the total cost of a sequence of n operations.
   - Worst case: the average cost (amortized cost) per operation is T(n)/n
3. Example
   - Stack
     - Scenario

PUSH(S, x): $O(1)$ each $\Rightarrow$ $O(n)$ for any sequence of n operations.
POP(S): $O(1)$ each $\Rightarrow$ $O(n)$ for any sequence of n operations.

```
MULTIPOP(S, k)
  while not STACK-EMPTY(S) and k > 0
      POP(S)
      k = k − 1
```

     - What is the largest possible cost of a sequence of N-stack operations, starting from empty stack?
       - Simple-worst-case-bound:
         O(n)*n = O(n^2)
       - Aggregate analysis:
         Any sequence of push/pop/multi pop on an initial empty stack cost at most O(n)
         T(n) = Tpop(n) + Tpush(n) <= 2Tpush(n) <= 2n = O(n)
         Therefore, the amortized cost per operation = T(n)/n = O(1)
   - Binary Counter
     - Scenario

- k-bit binary counter $A[0:k-1]$ of bits, where $A[0]$ is the least significant bit and $A[k-1]$ is the most significant bit.
- Counts upward from 0.

```
INCREMENT(A)
1  i = 0
2  while i < A.length and A[i] == 1
3      A[i] = 0
4      i = i + 1
5  if i < A.length
6      A[i] = 1
```

k − bit in total.
Θ(k) worst case
if all the bits = 1
naive solution:
Θ(n·k) for n-oper.

     - Amortized Analysis
       - Initialization: all n bits are set to 0, so n bit flips cost
       - Increment Operation:

Accounting Method

4. Definition
   - Assign different charges to different operations
   - Amortized cost = amount we charge an operation
   - When amortized cost > actual cost, store the difference on specific objects in the data structure as credit.
   - Use credit later to pay for operations whose actual cost > amortized cost
   - Differs from aggregate analysis:
     - Different operations can have different costs in accounting method

5. example
   - Stack

operation	actual cost	amortized cost
PUSH	1	2$
POP	1	0
MULTIPOP	$\min(k, s)$	0

2$ → { 1 — push; 1 — deposit to pay pop

Q: can we allocate 100 $ on push(x)?

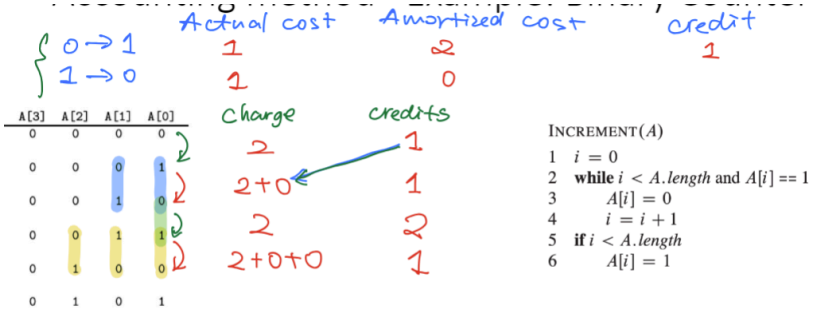
A: It's not a tight-bound

	Charge($)	Credit in total
Push(A)	2	1
Push(X)	2	2
Push(W)	2	3
Pop(W)	0	2
Multipop (S,2)	0	0

$T(n) = 2n = O(n)$

amortized cost per opt = $O(1)$

     - Needs to ensure the credit in total is non-negative
     - Design a cost-credit model to satisfy the requirements
   - Binary counter



INCREMENT(A)

```
1  i = 0
2  while i < A.length and A[i] == 1
3      A[i] = 0
4      i = i + 1
5  if i < A.length
6      A[i] = 1
```

for N- INCR. $T(n) = 2n = O(n)$.

Amortized cost per operation: $O(1)$

6. Dynamic Tables
   - Hash table
     - Scenario
       - Don't know in advance how many objects will be stored in it.
       - When it fills, must reallocate with a larger size, copying all objects into the new, larger table.

N

#item	1	2	3	4	5	6	7	8	9
Table size	1	2	4	4	8	8	8	8	16
INSERT cost	1	2	3	1	5	1	1	1	9
		(1+1)	(1+2)		(1+4)		2n		(1+8)

     - Answer
       - When the table is full, double its size and copy all previous elements into the new table.
       - Hence, every time create a new table and insert a number will be cost 1, and every time create a new table to copy all previous elements will cost curr-1, hence:

$T(N) = \underbrace{(1+1+1+\cdots+1)}_{N\text{- items}} + (1+2+4+8+\cdots) = 3n$

       -

Time for reshaping: $n, \frac{n}{2}, \frac{n}{4}, \ldots$

$n \sum_{i=0}^{\infty} \frac{1}{2^i} = 2n$

$T(N) = 3N$

The amortized cost per operation $= \frac{3}{} = O(1)$

- Run Time

  Charge 1 per elementary insertion. Count only elementary insertions, since all other costs together are constant per call.

  c_i = actual cost of ith operation

  - If not full, $c_i = 1$.
  - If full, have $i - 1$ items in the table at the start of the ith operation. Have to copy all $i - 1$ existing items, then insert ith item $\Rightarrow c_i = i$.

  n operations $\Rightarrow c_i = O(n) \Rightarrow O(n^2)$ time for n operations.

  Of course, not every operation triggers an expansion:

  $$c_i = \begin{cases} i & \text{if } i-1 \text{ is exact power of 2,} \\ 1 & \text{otherwise.} \end{cases}$$

- Aggregate analysis

  $$\begin{aligned} \text{Total cost} &= \sum_{i=1}^{n} c_i \\ &\le n + \sum_{j=0}^{\lfloor \lg n \rfloor} 2^j \\ &= n + \frac{2^{\lfloor \lg n \rfloor + 1} - 1}{2 - 1} \\ &< n + 2n \\ &= 3n \end{aligned}$$

  Therefore, **aggregate analysis** says amortized cost per operation = 3.

7. Steps
   - First try \$1
     - Charge 1 for each insert
     - Nothing to pay for expanding the array
   - Second try \$2
     - Charge 1 for each insert
     - Charge 1 for insert-to-new-table (stored credit)
     - Nothing to pay for second expand
   - Third try \$3
     - Charge 1 for each insert
     - Charge 1 for insert-to-new-table
     - Charge 1 for recharge-dollar: used to recharge the old elements that have spent their copy-dollars ; useful when double the table
   - Explanation scenario

     Charge \$3 per insertion of x.

     - \$1 pays for x's insertion.
     - \$1 pays for x to be moved in the future.
     - \$1 pays for some other item to be moved.

     Suppose the table has just expanded, $size = num = m$ just before the expansion, and $size = 2m$ and $num = m$ just after the expansion. The next expansion occurs when $size = num = 2m$.

     - Each insertion will put \$1 on one of the m items that were in the table just after expansion and will put \$1 on the item inserted.
     - Have \$2m of credit by next expansion, when there are 2m items to move. Just enough to pay for the expansion, with no credit left over!

     for a sequence of m insertion operations, the amortized cost per operations is 3, which is in O(1).

8. Credit invariant
   - At any step each element has a const
   - Used to show the credit is always positive
   - Example by a Binary Counter
     - Credit Invariant: At any step, each bit of the counter that is equal to 1 will have $1 credit
     - Initial condition: counter is zero with no credit
     - Proof by induction

       Induction step: assume true up to some value of x and now consider next increment

       Case 1: x = b .. b b 0 1 … 1 → b .. b b 1 0 … 0

       (i least significant bits are 1, i+1st bit is 0)

       i+1 = actual cost: use i credits to pay for i flips 1→0

       use 1 out of 2 to pay for 0→1,

       use 1 out of 2 for credit on the new "1"

       Case 2: x = 1 1 … 1 → 0 0 … 0

       (all bits are 1)

       actual cost is k

       use k credits to pay for k flips 1→0

       extra $2 isn't needed.

       Thus invariant is always true

       Credit invariant: a claim about the value of the cummulative stored credit $\sum_{i=1}^{n}(\hat{c}_i - c_i)$. Usually of the form "element with property P contains x stored credit (x can be dependent on property P)"

       If we show that the credit invariant always maintains positive stored credit i.e. $\sum_{i=1}^{n}(\hat{c}_i - c_i) \geq 0$, $\forall n$, then each operation i is $\mathcal{O}(\hat{c}_i)$ amortized.

   - Example by HashTable
     - Credit Invariant: Each element in the second half of array has $2 credit
     - Scenario:
       - $3
     - Proof by induction

       Proof:

       • Base case: no elements in the table, so it is true

       Inductive step.

       Case 1: array not full

       $1 to append, $2 stored on new item

       Case 2: Array full; make new array

       Copy all items using stored credit

       Add new item ($1) plus $2 credit

9. General answer steps for both methods

When using the aggregate method, you can follow these steps:

1. State your costs as accurately as possible
2. Calculate a bound $f(m)$ on those costs
3. Divide the bound $f(m)$ by m to get a bound on the amortized sequence complexity

When using the accounting method, you can follow these steps:

1. State your costs as accurately as possible
2. State what you are going to charge for each operation
3. State how and where your credits are going to be stored
4. State your credit invariant
5. Prove that your credit invariant is valid initially and across all possible operations
6. Show that you can always pay for each operation, based on the credit invariant
7. Calculate the amortized sequence complexity

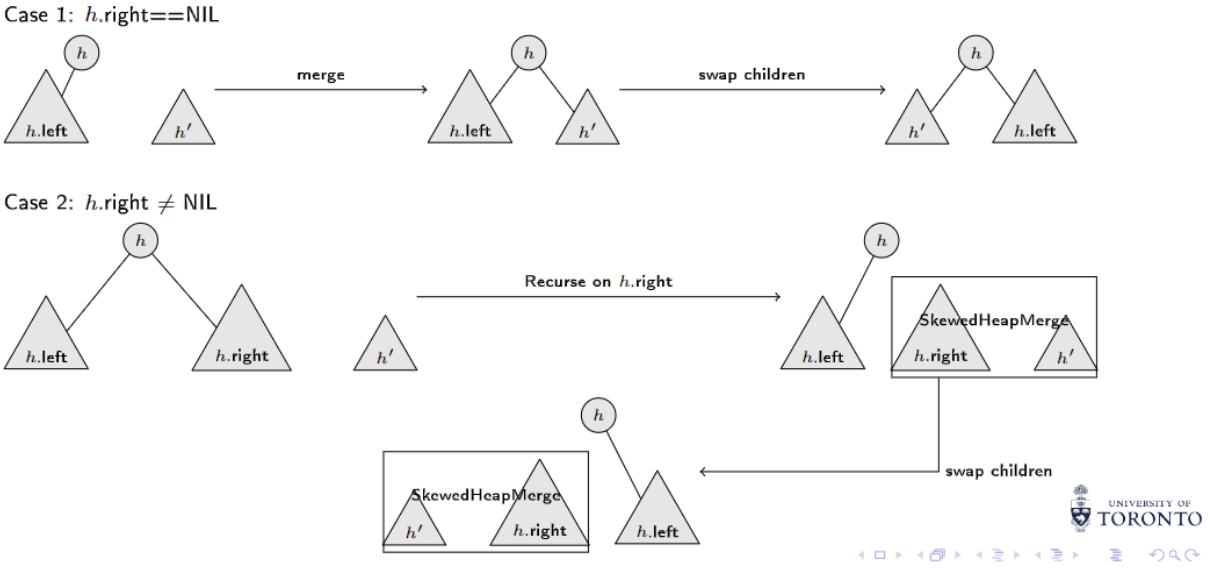
Skewed-Heap-Merge

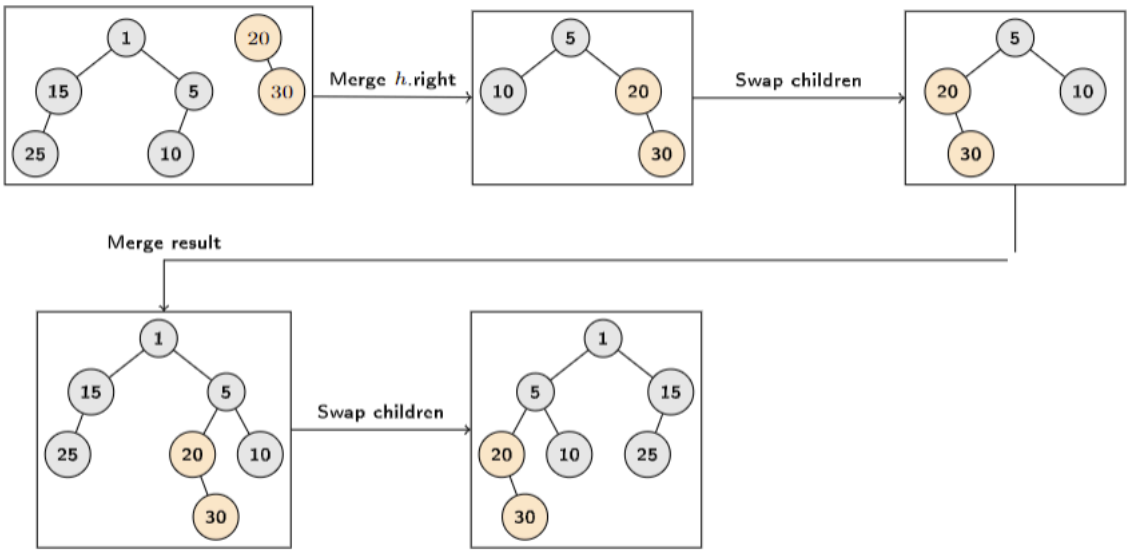
10. Definition and Code

A skewed heap is a heap that attempts to maintain balance by unconditionally swapping all nodes in the merge path when merging 2 heaps

```
function SkewedHeapMerge(h, h')
    if h =NIL or h'=NIL then
        Return the other
    if h'.root≤ h.root then
        Swap h ↔ h'        ▷ force h to have the smaller root
    h.right=SkewedHeapMerge(h.right, h')
    Swap h.right ↔ h.right
    Return h
```

11. Abstract Example





12. Advices

Part (a): prove SkewedHeapMerge(h, h') is $\mathcal{O}(\log n)$ amortized

- Need to charge \$$c \log n$ per operation
- Let $wT(x)$ =# nodes in subtree rooted at x
- heavy node: $wT(x) \geq \frac{1}{2} wT(\text{parent}(x))$
- light node: $wT(x) < \frac{1}{2} wT(\text{parent}(x))$
- Cost of merge is minimized if the rightmost path is all light nodes
- Misplaced node: a heavy node on the rightmost path
- Find the # light nodes on any path from the root to a leaf
- Cost of merging along rightmost path is bounded by the # light nodes (also bound for the credit invariant)
- Pay for the swapping of children using the credit invariant and additional funds
- Re-establish the credit invariant, how many misplaced nodes remain after swap?

Part (b) Show Insert(h, x) and DeleteMin(h) are $\mathcal{O}(\log n)$ amortized
Simply define/implement both functions using SkewedHeapMerge with some $\mathcal{O}(1)$ operations

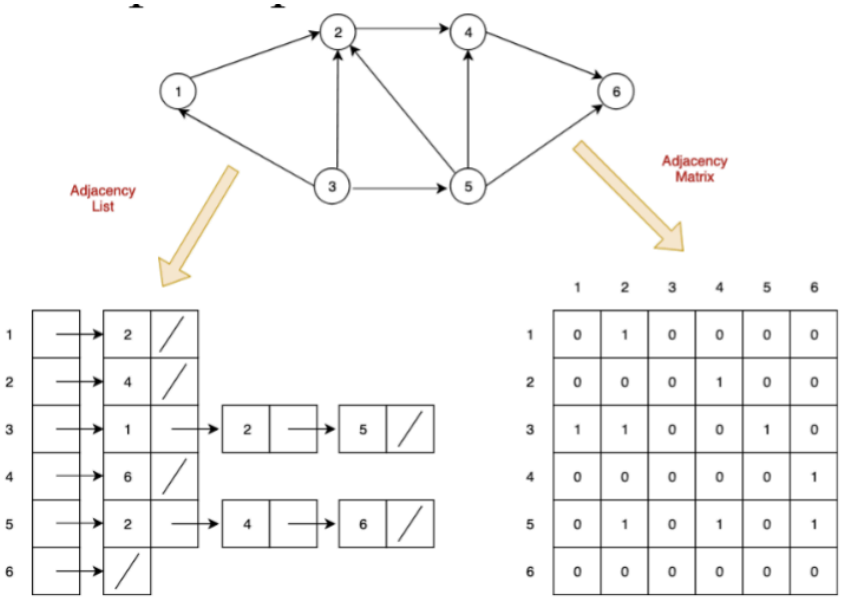
Graph Algorithm

Graph

1. Definition

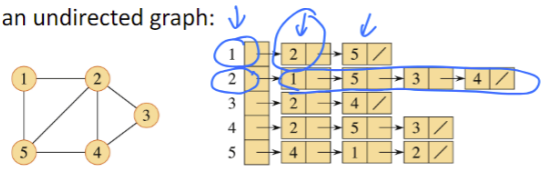
$$G = (V, E) \text{ defined over set of vertices } |V| \text{ and set of edges } |E|$$

2. Properties
   - Weighted vs un-weighted
     - Weight: a numerical value attached to each individual edge
   - Directionality
     - Directed: one-way relationship
     - Un-directed: bi-directional
   - Path: sequence of edges, between adjacent vertices
     - Simple path: no vertex is repeated (A -> B -> C)
     - Path: have repeated vertex (A -> B -> C -> B -> C)
   - Cycle: simple path with same start/end vertex
     - e.g.: A -> B -> C -> A
   - Connectivity
     - Connected graph (G): there exists a path between every 2 vertices.
     - Otherwise, G is disconnected
3. Representation
   - Adjacency list
   - Adjacency matrix
   - Example



4. Example of a graph

For an undirected graph:



If edges have *weights*, can put the weights in the lists.

Weight: $w : E \to \mathbb{R}$

We'll use weights later on for spanning trees and shortest paths.

Space: $\Theta(V + E)$.

Time: to list all vertices adjacent to u: $\Theta(\text{degree}(u))$.

Time: to determine whether $(u, v) \in E$: $O(\text{degree}(u))$.

5. Adjacency matrix

$|V| \times |V|$ matrix $A = (a_{ij})$

$$a_{ij} = \begin{cases} 1 & \text{if } (i,j) \in E, \\ 0 & \text{otherwise.} \end{cases}$$

	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

	1	2	3	4
1	0	1	0	0
2	0	0	0	1
3	1	1	0	0
4	0	0	1	1

Space: $\Theta(V^2)$.

Time: to list all vertices adjacent to u: $\Theta(V)$.

Time: to determine whether $(u, v) \in E$: $\Theta(1)$.

Can store weights instead of bits for weighted graph.

<u>BFS (Breadth-first search)</u>

6. BFS model
   - Input: Graph G = (V, E), either directed or undirected, and source vertex s belongs to V.
   - Output:
     - v.d: shortest distance from s to v for all v belongs to vertices V
     - v.π: v's predecessor on the shortest path from s
     - (u,v): the last edge on the shortest path from s to v
     - Predecessor subgraph: contains edges (u,v) such that v.π = u
       - It forms a tree called the BF-tree
7. Complexity Analysis
   - Runtime: O(V+E) overall
     - Queue Operation: O(V)
     - Scanning adjacent list: O(E)
       - Directed: O(E)
       - Un-directed: O(2E)
   - Space complexity: O(V)
8. Key Take-aways (run time)

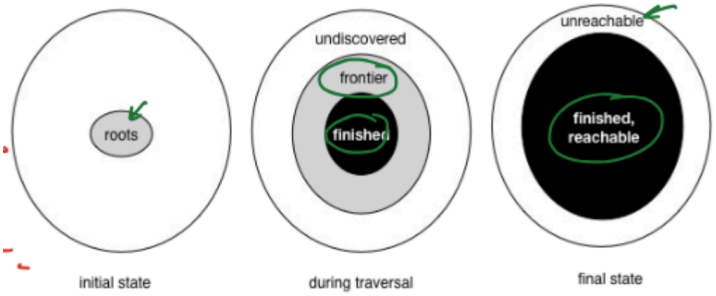
Key Take-Aways

- Discover all nodes of depth k before discovering any node of depth $k+1$
- we only discover nodes **reachable** from s (i.e. $\exists\ s \xrightarrow{p} t$)
- Runtime: $\mathcal{O}(V+E)$
- Node types: WHITE (undiscovered), GREY (processing), BLACK (finished)
- maintain $v.d/d[v]$ and $v.\pi/\pi[v]$, $\forall v \in V$
  - $\forall v \in V$, $d[v] = \min\{\#\text{edges in any path from } s \text{ to } v\}$
  - if $\nexists\ s \xrightarrow{p} v$, $d[v] = \infty$
  - $\pi[v]$ is the predecessor of v on the shortest path $s \to v$

BFS can be thought of as a SSSP algorithm for a graph with all edge weights equal to 1.
$d[v] = \delta(s, v)$.

<u>DFS (Depth-first search)</u>

9. Model
   - Input: G = (V, E), directed or undirected, no src vertex
   - Output:
     - DF-forest
     - d[v]: discovery time
     - f[v]: finishing time



10. Code
    - Global search starts a local search on each vertex to explore entire graph.

```
DFS(G)
1  for each vertex u ∈ G.V
2      u.color = WHITE
3      u.π = NIL
4  time = 0
5  for each vertex u ∈ G.V
6      if u.color == WHITE
7          DFS-VISIT(G, u)
```

```
DFS-VISIT(G, u)
1   time = time + 1            // white vertex u has just been discovered
2   u.d = time
3   u.color = GRAY
4   for each vertex v in G.Adj[u]   // explore each edge (u, v)
5       if v.color == WHITE
6           v.π = u
7           DFS-VISIT(G, v)
8   time = time + 1
9   u.f = time
10  u.color = BLACK            // blacken u; it is finished
```

    - At current position, mark as white since haven't visit
    - Goes to visit function, mark current as grey since currently visited, then look at the neighbours
    - If the neighbour is white (haven't visited), assume current position goes to that neighbour, calculate neighbours by visit function again, until all graph has been reached.
    - Time is to track the structure of the graph, more useful for detect back edge in cycle detection. d stands for detection time, and f stands for finish time.

11. Complexity Analysis
    - Run time: O(V+E)
        - For each vertex: DFS visits each vertex exactly once, because the coloring is done once per vertex.
        - For each edge: for each vertex u, the algorithm explores every edge (u, v) emerging from u.
    - Space complexity: O(V)

12. Properties of DFS: Parenthesis theorem
    - rules

For all u, v, exactly one of the following holds:

1. $u.d < u.f < v.d < v.f$ or $v.d < v.f < u.d < u.f$ (i.e., the intervals $[u.d, u.f]$ and $[v.d, v.f]$ are disjoint) and neither of u and v is a descendant of the other.
2. $u.d < v.d < v.f < u.f$ and v is a descendant of u. (v is discovered after and finished before u.)
3. $v.d < u.d < u.f < v.f$ and u is a descendant of v. (u is discovered after and finished before v.)

So $u.d < v.d < u.f < v.f$ (v is both discovered and finished after u) *cannot* happen.

    - When DFS starts exploring from a vertex, it is the opening a parenthesis
    - When DFS finishes with a vertex (has looked at all its edges and nowhere to go), it as closing a parenthesis.
    - Finish and discovery time will not overlap with each other unless they are fully nested
    - If two vertices, U and V, the interval for U is from its discovery to its finish time, same for V, then:
        - The intervals are completely separate (U and V are on different branches of the DFS tree)
        - The internal for U completely contains the interval for V (V is a descendant of U in the DFS tree)
        - The interval for V completely contains the interval for U (U is a descendant of V in the DFS tree)

13. Key Take-Aways (Run time)
    - Discover all "deeper" nodes first.
    - Runtime: $\mathcal{O}(V+E)$.
    - Maintains $v.d/d[v]$, $v.f/f[v]$, $v.\pi/\pi[v]$.
        - $\pi[u]$=parent/predecessor of u in the DFS.
        - $d[u]$=time of discovery of u in the DFS.
        - $f[u]$=time when u is finished processing in the DFS
        - **Note** that $d[u] < f[u]$ by construction in the DFS algorithm.
    - We will discover all nodes in V (produces a DFS forest).
        - WHITE (undiscovered) vs GREY (processing) vs BLACK (finished) nodes.
        - Tree vs Back vs Forward vs Cross Edges
            - **Tree Edge**: in the DFS format, i.e. we explored edge (u, v).
            - **Back Edge**: (u, v) such that u is a descendent of v (includes self-loops).
            - **Forward Edge**: (u, v) such that u is an ancestor of v, but is not a tree edge.
            - **Cross Edge**: all other edges.
        - Back edge exists DFS forest $\Leftrightarrow$ cycle exists in graph.
        - If a graph is undirected, then all edges are either tree or back edges.

    When exploring edge (u, v) in DFS:
    - color[v]=WHITE: (u, v) is **Tree Edge**
    - color[v]=GREY: (u, v) is **Back Edge**
    - color[v]=BLACK:
        - $d[u] < d[v]$, (u, v) is **Forward Edge**
        - $d[u] > d[v]$, (u, v) is **Cross Edge**

14. Example

    Determine if an undirected graph $G = (V, E)$ contains a cycle in $O(V)$ time.
    Solution:

    ```
    1: Run DFS
    2: if encounter a back edge or |V| distinct edges then
    3:     Retrun True
    4: else
    5:     Return False
    ```

    Correctness and running time:
    Claim: back edge $\Leftrightarrow$ cycle $\rightarrow \mathcal{O}(V+E)$
    Theorem B.2: $|E| > |V| - 1 \Rightarrow$ cycle, so G must contain a cycle
    If $|E| < |V|$, then we will return True iff we find a back edge in $\mathcal{O}(V+E) = \mathcal{O}(2V) = \mathcal{O}(V)$ time.
    If $|E| > |V|$, either we find back edge first or we discover $|V|$ distinct edges first, either way we did $\mathcal{O}(V)$ item times.

Classification of edges

15. Different edges
    - Tree edge: found by exploring (u, v) in DFS
    - Back edge: (u, v), u is a descendant of v
    - Forward edge: (u, v), where b is a descendant of u, but not a tree edge
    - Cross edge: any other edge, can go between vertices in same DF tree or in different DF trees
16. Application
    - Cycle detection: perform DFS with edge classification on the entire graph and ask whether there exists a back edge anywhere in the graph.
    - Connected components problem: edge represents jobs need to be done in the same day. Find out the max num of days that are needed to carry out all jobs, only need to count the components.

Topological sort

17. Directed acyclic graph (dag)
    - Definition: A directed graph with no cycles.
    - Good for modeling processes and structures that have a partial order: a > b and b > c then a > c

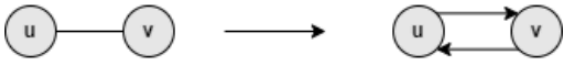
- Usually used to indicate precedences among events

18. Topological sort
    - Topological sort of a dag: a linear ordering of vertices such that if (u,v) belongs to E, then u appears somewhere before v (not like sorting numbers).

```
TOPOLOGICAL-SORT(G)
1  call DFS(G) to compute finish times v.f for each vertex v
2  as each vertex is finished, insert it onto the front of a linked list
3  return the linked list of vertices
```

19. Key Take-Aways (Run time, code)
    - Produces total ordering from partial ordering.
    - $G = (V, E)$ must be a DAG to produce a valid topological sort. (This is also the only algorithm you learn and can use for DAG in this course.)
    - Runtume: $\mathcal{O}(V + E)$.

    Can convert undirected graph to a directed graph, $\forall (u, v) \in E$ do the following transformation. Number of edges doubles, which shouldn't change asymptotic behavior.




```
1: function TopologicalSort(G)
2:     DFS(G)
3:     Output vertices in decreasing order of finish time
```

20. Example

A directed graph $G = (V, E)$ is said to be semiconnected if $\forall u, v \in V$ either $u \to v$ or $v \to u$. Give an $\mathcal{O}(V + E)$ algorithm to determine whether a given directed graph is semiconnected.

Algorithm: Find G^{SCC} the strongly connected components of G using CLRS 20.5. Then run Toposort on G^{SCC}. (G^{SCC} is a DAG), G is semiconnected if there exists edge between all adjacent vertices in the topological order of G^{SCC}.

Runtime: $\mathcal{O}(V + E)$
Correctness:
Claim:
Let $v_1, v_2, \ldots, v_n \in V^{SCC}$ be the topological ordering
$(v_i, v_{i+1}) \in E^{SCC}\ \forall i \Leftrightarrow G^{SCC}$ semiconnected.
Proof:
$\Rightarrow \exists$ path $v_i \to v_{i+1}\ \forall i$
so take $i < j$, $\forall v_i \to v_j$ via path $v_i \to v_{i+1} \to \cdots \to v_{j-1} \to v_j$
$\Leftarrow$ either exists path $v_i \to v_{i+1}$ or $v_{i+1} \to v_i$
$v_{i+1} \to v_i$ cannot exist because that breaks topological ordering.
consider $v_i \to v_k \to v_{i+1}$, which cannot exist because then $k > i$ and $k < i + 1$, but k integer.
So $v_i \to v_{i+1}$ is the edge (v_i, v_{i+1})

Minimum Spanning Tree

General

1. Definition of ST

   Spanning Tree: $T \subseteq E$ such that $\forall v \in V$, $\exists (u, v) \in T$ or $\exists (v, u) \in T$. (contains all vertices)

2. Definition of MST
   - A spanning tree whose weight is minimum over all spanning trees.

     Minimum Spanning Tree: Spanning tree $T \subseteq E$ such that $w(T) = \sum_{(u,v)\in T} w(u, v)$ is
   - minimized.

3. Model
   - Input
     - Undirected graph G = (V, E)
     - Weight w(u, v) on each edge (u, v) belongs to E
   - Goal
     - Find an acyclic subset T included by E such that the total weight is minimized:

       $$w(T) = \sum_{(u,v)\in T} w(u, v)$$

4. Properties of an MST
   - It has |V| - 1 edges
   - It has no cycles
   - It might not be unique

   - An MST always has $|V|$ vertices and $|V| - 1$ edges ($|E_{\text{MST}}| = |V_{\text{MST}}| - 1$)
   - Consider $(u, v) \in T$ and $(x, y) \notin T$, then $T \cup \{(x, y)\}$ always contains a cycle and $T \setminus \{(u, v)\} \cup \{(x, y)\}$ is a spanning tree.

5. Example

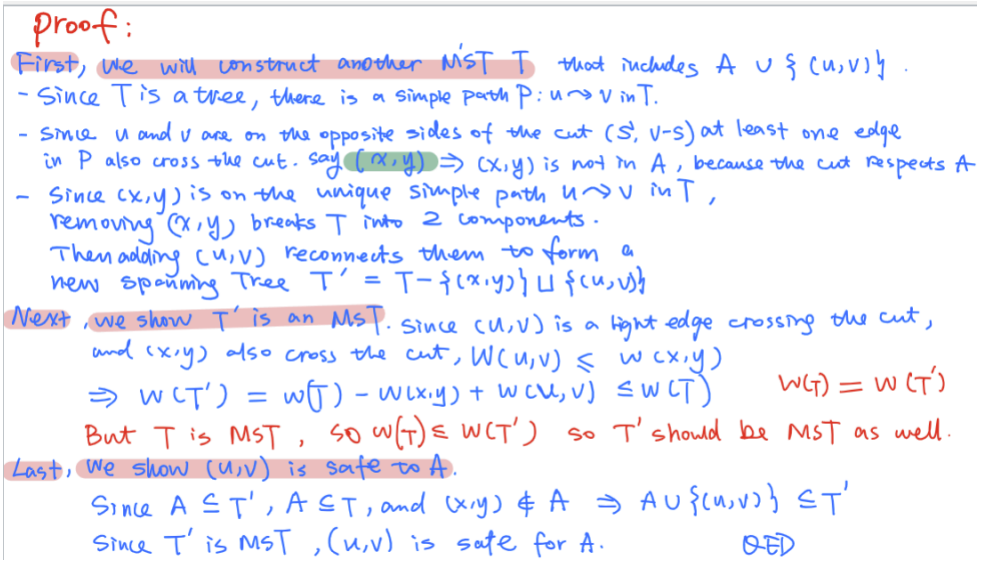
   Let $G = (V, E)$ with distinct edge weights, show that the edge connecting two separate components with the least weight must be included in every MST.
   Proof: Let e be the minimum-weight edge
   Assume $\exists T$ an MST such that $e \notin T$
   $T \cup \{e\}$ must form a cycle, take any edge $f \neq e$ in that cycle
   $T' = T \cup \{e\} \setminus \{f\}$ is a spanning tree
   since $w(e) < w(v), \forall v \neq e$, then $w(T') = w(T) + w(e) - w(f) < w(T)$, Contradiction.

Generic Algorithm (greedy approach)

6. Code

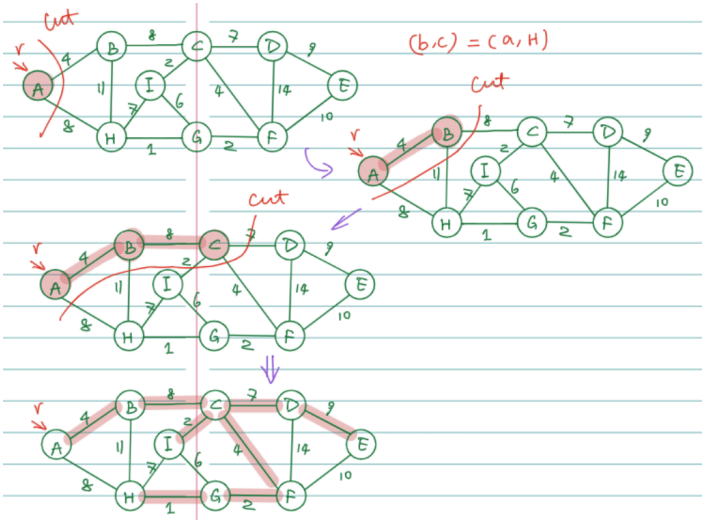
   GENERIC-MST(G, w)
   1 $A = \emptyset$
   2 **while** A does not form a spanning tree
   3 find an edge (u, v) that is safe for A
   4 $A = A \cup \{(u, v)\}$
   5 **return** A

7. Terms and Theorem
   - A cut (S, V - S)
     - It is a partition of vertices into disjoint sets S and V - S
     - Edge (u, v) belongs to E crosses cut(S, V - S) if one endpoint is in S and the other is in V - S
     - A cut respects A iff no edge in A crosses the cut
     - Light edge crossing a cut: iff the edge's weight is min over all edges crossing the cut. For a given cut, there can be >1 light edge crossing it
   - Theorem: Let A be a subset of some MST, (S, V-S) be a cut that respects A, and (u, v) be a light edge crossing (S, V - S), then (u, v) is safe for A.



Prim's Algorithm (greedy approach)

8. Kruskal and Prim theorem
    - Kruskal: The set A is a forest whose vertices are all those of the given graph. The safe edge added to A is always a least-weight edge in the graph that connects two distinct components.
    - Prim: set A forms a single tree. The safe edge added to A is always a least-weight edge connecting the tree to a vertex not in the tree.
9. Prim's Algorithm
    - Builds one tree, so A is always a tree.
    - Starts from an arbitrary "root" r
    - At each step, find a light edge connecting A to an isolated vertex. Such an edge must be safe or A. Add this edge to A.
    - Run time: O(ElogV)
10. Finding a light edge
    - Use a priority queue Q:
        - Each object is a vertex not in A
        - v.key is the minimum weight of any edge connecting v to a vertex in A. v.key = inf if no such edge.
        - v.π is v's parent in A
        - Maintain A implicity as: $A = \{(v, v.\pi) : v \in V - \{r\} - Q\}$
        - At completion, Q is empty and the minimum spanning tree is: $A = \{(v, v.\pi) : v \in V - \{r\}\}$
    - Key operations
        - Insert (u, key): Insert u with key value in Q
            - Run time: O(log|Q|)
        - Extract_MIN(): Extract the item with min_key
            - Run time: O(log|Q|)
        - Decrease-key(u, new-key): Decrease n's key with new key
            - Run time: O(log|Q|)
            - Alternatively: Delete + insert
            - Note: cannot just update the key of u without affecting the heap, need to readjust the weight.



11. Code

```
MST-PRIM(G, w, r)
1  for each u ∈ G.V
2      u.key = ∞
3      u.π = NIL
4  r.key = 0
5  Q = G.V
6  while Q ≠ ∅
7      u = EXTRACT-MIN(Q)
8      for each v ∈ G.Adj[u]
9          if v ∈ Q and w(u, v) < v.key
10             v.π = u
11             v.key = w(u, v)
```

O(V)

|V|

O(log Q)

degree(u)

Decrese-key (V, key(v))

O(log Q)

- Total runtime:

$O(V) \cdot T(\text{Extract-MIN}) + \left(\sum_{v \in V} \text{degree}(v)\right) \cdot T_{\text{Decrease-key}}$

$\sum_{v \in V} \text{degree}(v) = O(E)$

Implementation of Q	T extract-MIN	T decrease-key	TOTAL
Binary-Heap	V·O(log V)	+ E·O(log V)	O(E log V)
Array	O(V)	O(1)	
Fibonacci-Heap	O(log V)	O(1)	

Amortized time.

12. Priority queue operations, time and application

recap

readjust hight

readjust height

Operations (Implementation)

insert (item, priority) a new element into the heap. O(log n)

get-Highest-Priority ( ) O(1)

delete-Highest-Priority ( ) O(log n)

Application

① CPU scheduling

② Dijkstra's shortest path.

Single Source Shortest Paths

General

How to find the shortest route between two points on a map.

Input:

- Directed graph $G = (V, E)$
- Weight function $w : E \to \mathbb{R}$

Shortest-path weight u to v:

$$\delta(u, v) = \begin{cases} \min\{w(p) : u \overset{p}{\rightsquigarrow} v\} & \text{if there exists a path } u \rightsquigarrow v \text{ ,} \\ \infty & \text{otherwise .} \end{cases}$$

Shortest path u to v is any path p such that $w(p) = \delta(u, v)$.

1. Shortest Path Properties: Optimal substructure
   - Any subpath of the shortest path is the shortest path.
   - Negative-weight edges:
     - If we have a neg-weight cycle, we can keep going around it, and get w(s, v0 = -inf for all v on the cycle.
     - The negative-weight cycle is not reachable from the source is also good.
     - Some algorithms work only if there are no negative-weight edges in the graph.
2. Procedure for finding SSSP
   - For all single source shortest-paths algorithms:
     - Start by calling Initialize-single-source
     - Relax edges
   - The algorithms differ in the order and how many times they relax each edge.
   - Code

INITIALIZE-SINGLE-SOURCE(G, s)

```
1  for each vertex v ∈ G.V
2      v.d = ∞          shortest-path estimate
3      v.π = NIL
4  s.d = 0
```

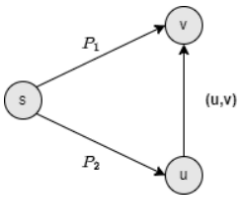
For each vertex $v \in V$:

- $v.d = \delta(s, v)$.
  - Initially, $v.d = \infty$.
  - Reduces as algorithms progress. But always maintain $v.d \geq \delta(s, v)$.
  - Call $v.d$ a ***shortest-path estimate***.
- $v.\pi$ = predecessor of v on a shortest path from s.
  - If no predecessor, $v.\pi$ = NIL.
  - π induces a tree—***shortest-path tree***.

3. Relaxing an edge (u, v)
   - Run time: O(1)

```
1: function Relax(u,v,w)
2:     d[v] = min(d[v], d[u] + w(u, v))
```

```
1: function Relax(u,v,w)
2:     if d[v] > d[u] + w(u, v) then
3:         d[v] = d[u] + w(u, v)
4:         π[v] = u
```



if $w(p_1) > w(p_2) + w(u, v)$, then we want to take $p_2 + (u, v)$ because it is shorter.

4. Shortest-paths properties
   - Based on calling initialize-single-source once and then calling relax zero or more times.

Triangle inequality: For all $(u, v) \in E$, we have $\delta(s, v) \leq \delta(s, u) + w(u, v)$.

Upper-bound property: Always have $v.d \geq \delta(s, v)$ for all v. Once $v.d$ gets down to $\delta(s, v)$, it never changes.

No-path property: If $\delta(s, v) = \infty$, then $v.d = \infty$ always.

Convergence property: If $s \rightsquigarrow u \rightarrow v$ is a shortest path, $u.d = \delta(s, u)$, and edge (u, v) is relaxed, then $v.d = \delta(s, v)$ afterward.

Path-relaxation property: Let $p = \langle v_0, v_1, \ldots, v_k \rangle$ be a shortest path from $s = v_0$ to v_k. If the edges of p are relaxed, *in the order*, $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$, even intermixed with other relaxations, then $v_k.d = \delta(s, v_k)$.

- Triangle Inequality: $\forall (u, v) \in E,\ \delta(s, v) \leq \delta(s, u) + w(u, v)$
- Upper-bound property: $d[v] \geq \delta(s, v)\ \forall v \in V$ and once $d[v] = \delta(s, v)$ it never changes
- No-path property: if $\nexists s \xrightarrow{p} v$ then $d[v] = \delta(s, v) = \infty$
- Convergence property: if $s \rightarrow u \rightarrow v$ is the shortest path in G, $\exists u, v \in V$. if $d[u] = \delta(s, u)$ prior to relaxing edge (u, v) then $d[v] = \delta(s, v)$ at all times afterwards.
- Path-relaxation property: if $p = \langle v_0, v_1, \ldots, v_k \rangle$ is shortest path from $s = v_0 \xrightarrow{p} v_k$ and paths are relaxed in order $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$ then $d[v_k] = \delta(s, v_k)$
- Predecessor-subgraph property: Once $d[v] = \delta(s, v) \forall v \in V$, the predecessor subgraph is a shortest-paths tree rooted at s.

Dijkstra's algorithm

5. Characteristics
   - No negative-weight edges
   - Use priority queue
   - Keys are shortest-path weights (v.d)
6. Run time analysis
   - Q list: $\mathcal{O}(V^2 + E) = \mathcal{O}(V^2)$
   - Q priority queue (heap): $\mathcal{O}((V + E) \log V) = \mathcal{O}(E \log V)$ (if sufficiently sparse, $E = \mathcal{O}(V^2 / \log V)$)
   - Q Fibonacci heap: $\mathcal{O}(V \log V + E)$
   - Time: O((E+V)logV) with binary heap, O(VlogV+E) with Fibonacci heap
     - O(V) visit each node once
     - O(E) scan adjacent lists
       - G is directed: O(E)
       - G is undirected: O(2E)
   - Space: O(V)
7. Code

```
DIJKSTRA(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  S = ∅
3  Q = ∅
4  for each vertex u ∈ G.V
5      INSERT(Q, u)
6  while Q ≠ ∅
7      u = EXTRACT-MIN(Q)
8      S = S ∪ {u}
9      for each vertex v in G.Adj[u]
10         RELAX(u, v, w)
11         if the call of RELAX decreased v.d
12             DECREASE-KEY(Q, v, v.d)
```

Conti next page…

```
function Dijkstra(G,w,s)
    Init-Single-Source(G,s)
    S = ∅                                  ▷ set of vertices whose first shortest path weights have been determined
    Q = ∅
    for each v ∈ V do
        INSERT(Q, v)                       ▷ priority queue sorted by d[v]
    while Q ≠ ∅ do
        u =EXTRACT-MIN(Q)
        S = S ∪ {u}
        for each v ∈ adj[u] do
            Relax(u, v, w)                 ▷ priority queue sorted by d[v]
            Decrease-key(Q, v)
```

- S: vertices whose final shortest-path-weight is determined
- Q: Priority queue = V - S
- Key: v.d

Bellman-ford's algorithm

8. characteristics
    - Allows negative-weight edges
    - Returns true if no negative-weight cycles are reachable from s, false otherwise.
9. Code and run time

```
BELLMAN-FORD(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  for i = 1 to |G.V| − 1
3      for each edge (u, v) ∈ G.E
4          RELAX(u, v, w)
5  for each edge (u, v) ∈ G.E
6      if v.d > u.d + w(u, v)
7          return FALSE
8  return TRUE
```

Θ(V) — |V|−1 — relax each edge once } Θ(E)

detect NW-C } O(E) ← No Solution

runtime: O(VE)

```
function Bellman-Ford(G,w,s)
    Init-Single-Source(G,s)
    for each i = 1, ..., |V| − 1 do
        for each (u, v) ∈ E do
            Relax(u,v,w)                   ▷ Relaxing all edges |V| − 1 times
    for each (u, v) ∈ E do
        if d[v] > d[u] + w(u, v) then
            Return False                   ▷ Negative cycle
    Return True                            ▷ valid shortest path
```

- Steps
    - Set every entry in D as inf, set D[s] = 0
    - Relax each edge V-1 times
    - Update every node each time by the shortest cost
    - Previous shortest path might be updated by go to a new vertex and get back

10. Path-relaxition property

Lemma 24.2

Let $G = (V, E)$ be a weighted, directed graph with source s and weight function $w : E \to \mathbb{R}$, and assume that G contains no negative-weight cycles that are reachable from s. Then, after the $|V| - 1$ iterations of the **for** loop of lines 2–4 of BELLMAN-FORD, we have $v.d = \delta(s, v)$ for all vertices v that are reachable from s.

Proof We prove the lemma by appealing to the path-relaxation property. Consider any vertex v that is reachable from s, and let $p = \langle v_0, v_1, \ldots, v_k \rangle$, where $v_0 = s$ and $v_k = v$, be any shortest path from s to v. Because shortest paths are simple, p has at most $|V| - 1$ edges, and so $k \le |V| - 1$. Each of the $|V| - 1$ iterations of the **for** loop of lines 2–4 relaxes all $|E|$ edges. Among the edges relaxed in the ith iteration, for $i = 1, 2, \ldots, k$, is (v_{i-1}, v_i). By the path-relaxation property, therefore, $v.d = v_k.d = \delta(s, v_k) = \delta(s, v)$. ■

Path-relaxation property: Let $p = \langle v_0, v_1, \ldots, v_k \rangle$ be a shortest path from $s = v_0$ to v_k. If the edges of p are relaxed, *in the order*, $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$, even intermixed with other relaxations, then $v_k.d = \delta(s, v_k)$.

SSSP in DAGs (single-source shortest paths in a directed acyclic graph)

11. DAG
    - Characteristic: no negative-weight cycles
    - code

```
DAG-SHORTEST-PATHS(G, w, s)
1  topologically sort the vertices of G
2  INITIALIZE-SINGLE-SOURCE(G, s)
3  for each vertex u ∈ G.V, taken in topologically sorted order
4      for each vertex v in G.Adj[u]
5          RELAX(u, v, w)
```

    - Overall O(V+E) complexity

Difference Constraints and Shortest Paths

12. Linear programming
    - Linear approximation tool
    - Need to satisfy a set of constraints set
13. Rules
    - Feasibility problem
      - Wish to find any feasible solution, or determine that no feasible solution exists.
    - Systems of difference constraints
      - Each row of the linear programming matrix A contains one 1 and one -1, all others are 0.
      - constraints given by Ax b are a set of m difference constraints involving n unknowns, in which each constraint is a simple linear inequality of the form $x_j - x_i \le b_k$ where $1 \le i, j \le n, i \ne j$, and $1 \le k \le m$.
    - Example

Finding a 5-vector x = (x_i) that satisfies

$$\begin{pmatrix} 1 & -1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & -1 \\ 0 & 1 & 0 & 0 & -1 \\ -1 & 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & 1 & 0 \\ 0 & 0 & -1 & 1 & 0 \\ 0 & 0 & -1 & 0 & 1 \\ 0 & 0 & 0 & -1 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{pmatrix} \le \begin{pmatrix} 0 \\ -1 \\ 1 \\ 5 \\ 4 \\ -1 \\ -3 \\ -3 \end{pmatrix}$$

$$\begin{aligned} x_1 - x_2 &\le 0, \\ x_1 - x_5 &\le -1, \\ x_2 - x_5 &\le 1, \\ x_3 - x_1 &\le 5, \\ x_4 - x_1 &\le 4, \\ x_4 - x_3 &\le -1, \\ x_5 - x_3 &\le -3, \\ x_5 - x_4 &\le -3. \end{aligned}$$

$x = (-5, -3, 0, -1, -4)$ $x' = (0, 2, 5, 4, 1)$

Lemma 24.8

Let $x = (x_1, x_2, \ldots, x_n)$ be a solution to a system $Ax \le b$ of difference constraints, and let d be any constant. Then $x + d = (x_1 + d, x_2 + d, \ldots, x_n + d)$ is a solution to $Ax \le b$ as well.

Proof For each x_i and x_j, we have $(x_j + d) - (x_i + d) = x_j - x_i$. Thus, if x satisfies $Ax \le b$, so does $x + d$. ■

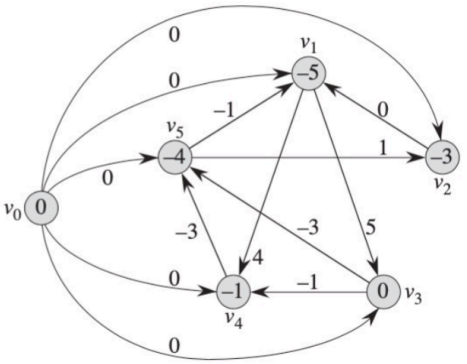
-

14. Constraint graph

$G = (V, E)$, weighted, directed.

- $V = \{v_0, v_1, v_2, \ldots, v_n\}$: one vertex per variable + v_0
- $E = \{(v_i, v_j) : x_j - x_i \le b_k \text{ is a constraint}\} \cup \{(v_0, v_1), (v_0, v_2), \ldots, (v_0, v_n)\}$
- $w(v_0, v_j) = 0$ for all $j = 1, 2, \ldots, n$
- $w(v_i, v_j) = b_k$ if $x_j - x_i \le b_k$

$$
\begin{aligned}
x_1 - x_2 &\le 0, \\
x_1 - x_5 &\le -1, \\
x_2 - x_5 &\le 1, \\
x_3 - x_1 &\le 5, \\
x_4 - x_1 &\le 4, \\
x_4 - x_3 &\le -1, \\
x_5 - x_3 &\le -3, \\
x_5 - x_4 &\le -3.
\end{aligned}
$$



- Theorem: given a system of difference constraints, let G = (V, E) be the corresponding constraint graph.
  - If G has no negative weight cycles, then:
    $x = (\delta(v_0, v_1), \delta(v_0, v_2), \ldots, \delta(v_0, v_n))$
  - If G has a negative weight cycle, then there is no feasible solution.

15. Negative cycle
    - The sum of a cycle is negative
16. Steps to find a feasible solution

1. Form constraint graph.
   - $n + 1$ vertices.
   - $m + n$ edges.
   - $\Theta(m + n)$ time.
2. Run BELLMAN-FORD from v_0.
   - $O((n + 1)(m + n)) = O(n^2 + nm)$ time.
3. BELLMAN-FORD returns FALSE ⇒ no feasible solution.
   BELLMAN-FORD returns TRUE ⇒ set $x_i = \delta(v_0, v_i)$ for all $i = 1, 2, \ldots, n$.

17. Find feasible solution by using BF method
    - Draw a constraint graph by using the constraints
    - Draw an invisible graph about adding vertex V0, the weight from V0 to all other vertices are all zero
    - Find the shortest path for each vertex from V0:
      - First, determine negative cycle
        - If the summation of the weights of a cycle is negative, then it is a negative cycle
        - If there exist any negative cycle, then there is no solution, write NEGATIVE CYCLE (according to the requirement)
      - Find the smallest number (negative number as smaller than positive number).
      - According to that edge's direction, set the starting point as zero, and the ending point as zero+weight of that edge.
      - Each time, move on to next connected vertex, and the score stored within each vertex is by supperposition of previous vertex's score and the weight of that edge.
      - Calculate every possible path. If one path is smaller cost than the other, update the vertex score.
      - Write the final score of each vertex as $\delta(v_i)$

- Rewrite the solution as $x_i = value$, value is the final score calculated from previous step.
    - If negative cycle, write NO SOLUTION.

18. Procedure for difference constraints

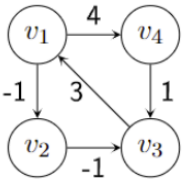
To solve a difference constraint problem with n variables, $x_1,...,x_n$,

1. Build constraint graph (weighted, directed) $G = (V, E)$
   - $V = \{v_0, v_1, ..., v_n\}$: one vertex per variable. Define v_0 as the pseudo-start.
   - $E = \{(v_i, v_j) : x_j - x_i \leq b_k \text{ is a constraint.}\} \cup \{(v_0, v_1), (v_0, v_2), ..., (v_0, v_n)\}$: one edge per constraint. Direction is from the subtrahend to the minuend[4]. Also connect pseudo-start to all vertices in the constraints.
2. Assign weights
   - $w(v_0, v_i) = 0$, for all i
   - $w(v_i, v_j) = b_k$, for all constraints $x_j - x_i \leq b_k$
3. Theorem:
   - If G has no negative weight cycle, then $x_1 = \delta(v_0, v_1)$, ..., $x_n = \delta(v_0, v_n)$ is a feasible solution
   - If G has a negative weight cycle, then there is no feasible solution
4. Build graph and run Bellman-Ford to find the shortest path from v_0 to all of $v_1, ..., v_n$, which gives the solution, or detect negative cycle (no solution).

19. Example

Find a feasible solution or determine that no solution exists for the following system of difference constraints:

$x_1 - x_3 \leq 3$
$x_2 - x_1 \leq -1$
$x_3 - x_2 \leq -1$
$x_3 - x_4 \leq 1$
$x_4 - x_1 \leq 4$



Running Bellman-Ford gives:
$\delta(v_0, v_1) = 0$, $\delta(v_0, v_2) = -1$, $\delta(v_0, v_3) = -2$, $\delta(v_0, v_4) = 0$

	v_0	v_1	v_2	v_3	v_4
iter0	0	0	0	0	0
iter1	0	0	-1	-1	0
iter2,3,4	0	0	-1	-2	0

A feasible solution is:

$x_1 = 0$, $x_2 = -1$, $x_3 = -2$, $x_4 = 0$.

Maximum Flow

Maximum-flow problem define

1. Flow networks

$G = (V, E)$ directed.

Each edge (u, v) has a ***capacity*** $c(u, v) \geq 0$.

If $(u, v) \notin E$, then $c(u, v) = 0$.

if an edge $(v_1, v_2) \in E$, then $(v_2, v_1) \notin E$

Source vertex s, ***sink*** vertex t, assume $s \rightsquigarrow v \rightsquigarrow t$ for all $v \in V$, so that each vertex lies on a path from source to sink. $|E| >= |V| - 1$

A ***flow*** in G is a real-valued functic $f : V \times V \rightarrow \mathbb{R}$

that satisfies the following two properties:

Capacity constraint: For all $u, v \in V$, we require $0 \leq f(u, v) \leq c(u, v)$.

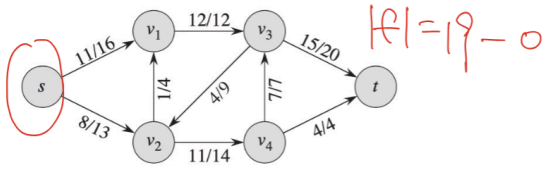
Flow conservation: For all $u \in V - \{s, t\}$, we require

$$\sum_{v \in V} f(v, u) = \sum_{v \in V} f(u, v) .$$

When $(u, v) \notin E$, there can be no flow from u to v, and $f(u, v) = 0$.

- The flow from vertex u to vertex v -- nonnegative quantity $f(u, v)$
- The value $|f| = \sum_{v \in V} f(s, v) - \sum_{v \in V} f(v, s)$

$=$ flow out of source $-$ flow into source .



- Input: a flow network with source and sink
- Goal: find a flow of maximum value

2. Antiparallel edges
   - If an edge (v1, v2) belongs to E, then (v2, v1) not belongs to E.
   - Solution: adding a new vertex and replacing the original edge with two new edges connect with the new vertex.
3. Networks with multiple src and sinks
   - Reduce the problem by superposition through adding a super source and a super sink

Flow mathematics

Maximum Flow mathematics

$f(X, Y) = \sum_{x \in X} \sum_{y \in Y} f(x, y)$

Properties:

① $f(X, X) = 0$

② $f(X, Y) = -f(Y, X)$

③ $f(X \cup Y, w) = f(X, w) + f(Y, w)$ if $X \cap Y = \emptyset$

④ $f(w, X \cup Y) = f(w, X) + f(Y, w)$ if $X \cap Y = \emptyset$

Ford Fulkerson method

4. Residual capacity

Given a flow f in network $G = (V, E)$.

Consider a pair of vertices $u, v \in V$.

How much additional flow can be pushed directly from u to v?

That's the ***residual capacity***,

$$c_f(u,v) = \begin{cases} c(u,v) - f(u,v) & \text{if } (u,v) \in E\,, \\ f(v,u) & \text{if } (v,u) \in E\,, \\ 0 & \text{otherwise (i.e., } (u,v), (v,u) \notin E)\,. \end{cases}$$

5. Residual network
   - Similar to a flow network but it may contain antiparallel edges ((u,v) and (v,u)).

   The ***residual network*** is $G_f = (V, E_f)$, where
   Residual edge $E_f = \{(u,v) \in V \times V : c_f(u,v) > 0\}$.

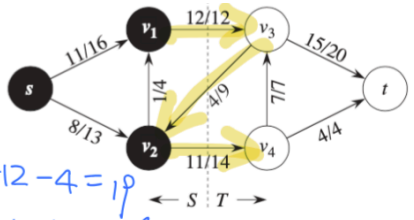
6. Augmenting Path (AP)
   - Admits more flow along each edge
   - Like a sequence of pipes through which can squirt more flow from s to t
7. Residual capacity
   - How much more flow can be pushed from s to to along augmenting path p

   $c_f(p) = \min\{c_f(u,v) : (u,v) \text{ is on } p\}$
8. Cuts of flow networks
   - A cut (S, T) of flow network G = (V, E) is a partition of V into S and T = V - S such that s belongs to S and t belongs to T.
   - If f is a flow, then the net flow f(S, T) across the cut (S, T) is defined to be:
   - Minimum cut: a cut whose capacity is minimum over all cuts of G.
     - After compute the final flow by applying the FF method, sum all capacity that outward from t as the minimum grade, then from the original graph find the cut that satisfy this capacity grade.

   $$f(S,T) = \sum_{u \in S}\sum_{v \in T} f(u,v) - \sum_{u \in S}\sum_{v \in T} f(v,u)\,.$$

   The ***capacity*** of the cut (S,T) is

   $$c(S,T) = \sum_{u \in S}\sum_{v \in T} c(u,v)\,.$$

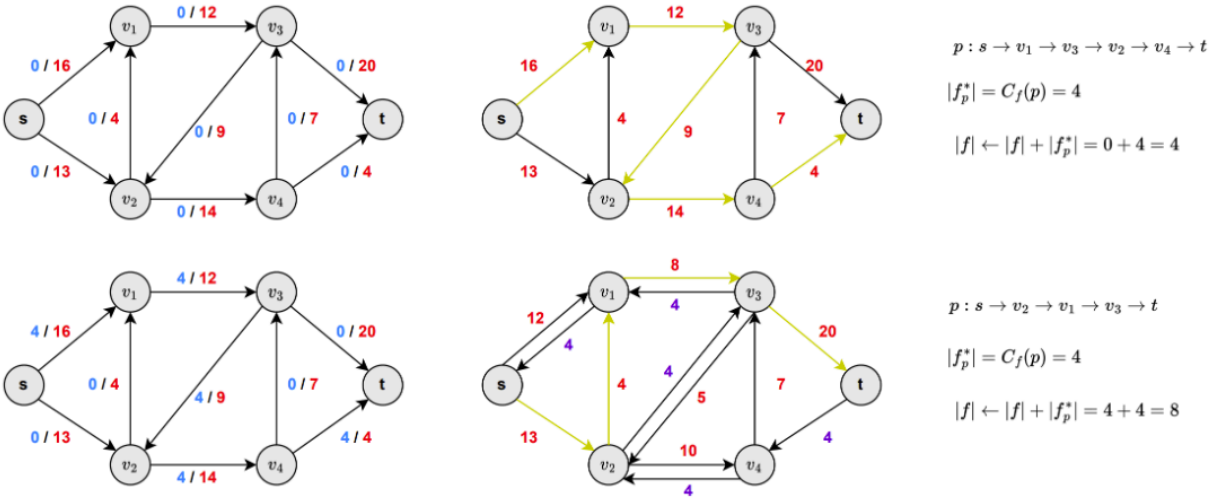
   f(S,T) = 11+12-4 = 19
   c(S,T) = 12+14 = 26

9. Theorem (Max-flow min-cut theorem)
   - The following are equivalent:
     - F is a maximum flow
     - Gf has no augmenting path
     - |f| = c(S, T) for some cut (S, T)
   - Corollary:
     - The value of any flow <= capacity of any cut
     - So maximum flow <= capacity of minimum cut
   - When we talk about the cut, we usually calculate the capacity of the cut, which is only count from u to v.
10. Code
    - Keep augmenting flow along an augmenting path until there is no augmenting path. Represent the flow attribute using (u, v).f

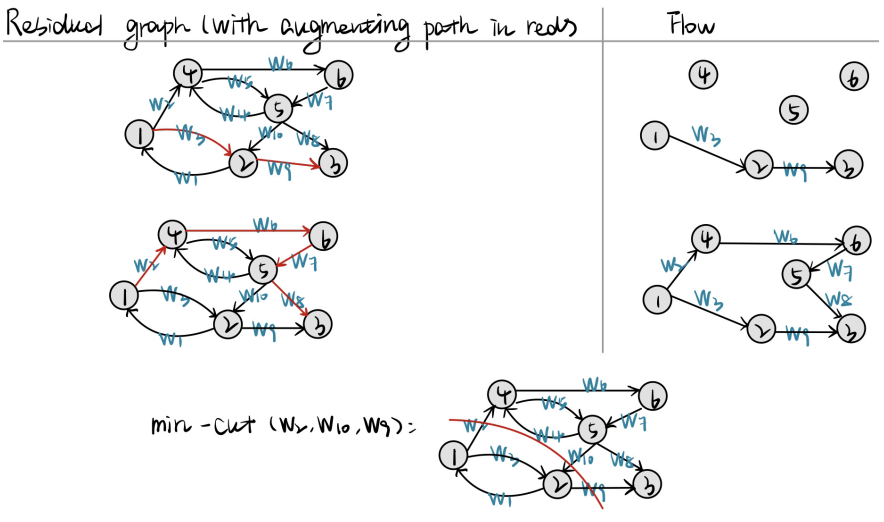
```
FORD-FULKERSON(G, s, t)
1  for each edge (u, v) ∈ G.E
2      (u, v).f = 0
3  while there exists a path p from s to t in the residual network G_f
4      c_f(p) = min {c_f(u, v) : (u, v) is in p}
5      for each edge (u, v) in p
6          if (u, v) ∈ G.E
7              (u, v).f = (u, v).f + c_f(p)
8          else (v, u).f = (v, u).f − c_f(p)
9  return f
```

- It is a problem of optimizing the distribution of some quantity (data and goods) through a network with capacity constraints.
- This algorithm will return true if possible route is found, false otherwise.

11. Max-flow capacity working steps
    - Initialize all scores as zero.
    - Along the arrow direction, find one possible path from src to des, and mark the score as the weight of each edge. Mark capacity as the minimum score.
    - Traverse backward from des to src, and subtract the minimum score with each weight of the edges as the edge's new score.
    - Find another possible path, if zero score then use the edge's weight, if already score a value then use the scored value subtract from the weight to get current score. Mark capacity as the minimum score.
    - Repeat step three, and if it has original score, add the original score with the newly computed score as the current score.
    - The total capacity is the sum of all capacity calculated.



12. Procedure for solving: using FF method to find max flow
    - Graph
        - Residual Graph (with augmenting path bolded)
        - Flow



    - Minimum cut on original graph

Edmonds Karp Algorithm

13. How to run it
    - Do ford-fulkerson, but compute augmenting paths by BFS of Gf.
    - Augmenting paths are the shortest paths from src to des (s to t) in Gf, with all edge weights = 1.

14. Run time
    - $O(VE^2)$
    - Proof:

To prove, need to look at distances to vertices in G_f.
Let $\delta_f(u, v)$ = shortest path distance u to v in G_f, with unit edge weights.

Lemma
For all $v \in V - \{s, t\}$, $\delta_f(s, v)$ increases monotonically with each flow augmentation.

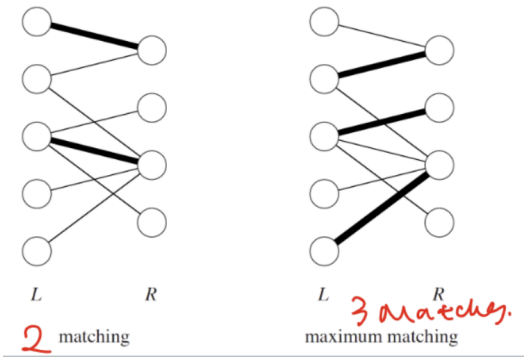
Theorem
Edmonds-Karp performs $O(VE)$ augmentations.

Proof Suppose p is an augmenting path and $c_f(u, v) = c_f(p)$. Then call (u, v) a ***critical*** edge in G_f, and it disappears from the residual network after augmenting along p.
≥ 1 edge on any augmenting path is critical.
Will show that each of the $|E|$ edges can become critical $\leq |V|/2$ times.

Use BFS to find each augmenting path in $O(E)$ time $\Rightarrow O(VE^2)$ time.
Can get better bounds.

Maximum Bipartite Matching

15. Bipartite
    - If G = (V, E) (undirected) has a partition of the vertices V=L并R such that all edges in E go between L and R.
16. Matching
    - Definition: A subset of edges M included by E such that for all v belongs to V, <= 1 edge of M is incident on v. (Vertex v is matched if an edge of M is incident on it, otherwise unmatched)
    - Maximum matching: a matching of maximum cardinality.
        - M is a maximum matching if |M| >= |M'| for all matchings M'



Tutorials

17. Useful results

The maximum flow $|f^*|$ of any flow network is upper bounded by the sum of capacities of any cut.
Proof: The flow going across that cut cannot be any larger, so it bottlenecks the flow of the graph.

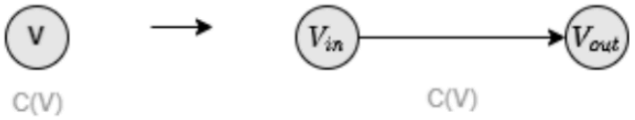
In the max flow network, the min-cut has capacity $|f^*|$.
Proof: By Max-Flow Min-Cut Theorem.

Let $G = (V, E)$ be a flow network with all edge capacities of 1. ($c(u, v) = 1$, $\forall u, v \in V$), then $|f^*|$ = the number of edge-disjoint paths in G
The min cut has capacity $|f^*|$, since all edge weighs 1, each path through the cut accounts for 1 capacity.
Therefore, there are net $|f^*|$ independent paths across the cut.

To find the number of independent paths in G
We just find the max flow in G.

18. More transformations
    - Combine many node into one node $\rightarrow$ make edge capacities ∞.
    - Isolate nodes $\rightarrow$ make edge capacities 0.
    - Count number of paths $\rightarrow$ make edge capacities 1.
    - Restrict flow $\rightarrow$ make edge capacity C.
19. Vertex capacities

    Suppose each vertex has capacity $c(v)$. Find the maximum flow from s to t.
    Transformation:



    Formally: $G' = (V', E')$ s.t.
    - $V' = \{v_{\text{in}} : v \in V\} \cup \{v_{\text{out}} : v \in V\}$
    - $E' = \{(v_{\text{in}}, v_{\text{out}}) : v \in V\} \cup \{(u_{\text{out}}, v_{\text{in}}) : (u, v) \in E\}$
    - $c(u_{\text{out}}, v_{\text{in}}) = c(u, v), \forall (u, v) \in E$
    - $c(v_{\text{in}}, v_{\text{out}}) = c(v), \forall v \in V$

NPC (non-deterministic poly complete)

Introduction

1. Alphabet
   - $\sum_1 = \{a,\ldots, z\}$: alphabet of every string that make from the letters
   - $\sum_2 = \{0,\ldots, 9\}$: alphabet of every string that make from the numbers
   - If a mix of letters and numbers, then not a string from any alphabet
   - Empty strings: denoted by ϵ, a string over any alphabet
2. Basic notations

   $\sum^*$: The set of all the strings over an alphabet Σ

   $\sum^+$: $\Sigma^* - \{\epsilon\}$

   - **Binary Alphabet**: An alphabet of cardinality 2
   - **Binary strings**: strings over a binary alphabet

   $\{0,1\}$ *is a binary alphabet, and* $\{1\}$ *is a unary alphabet.*

   11 *is a binary string over the alphabet* $\{0,1\}$
   a unary string over the alphabet $\{1\}$
3. Languages

   if Σ is an alphabet
   and L is a (possibly infinite) subset of Σ^*
   then L is said to be a *language* over Σ
   Each element of L is said to be a *string* of the language.

   $\sum^*$: The set of all the strings over an alphabet Σ

   - Example:

   $\{0,11,001\}$, $\{\epsilon,10\}$, *and* $\{0,1\}^*$ *are subsets of* $\{0,1\}^*$

   ➔ They are Language over *alphabet* $\{0,1\}$

   $\{aba, czr, d, f\}$ is a language over $\{a,b,\ldots,z\}$
4. Operations of Languages
   - Union
   - Intersection
   - Difference
   - Kleene star $L^* = \{w \in \Sigma^* : w = w_1 \cdots w_k$ for some $k \geq 0$ and some $w_1,\ldots,w_k \in L\}$.

     w is a string over Σ

     "concatenating zero or more strings from L"

   For example, if $L = \{01, 1, 100\}$, then $110001110011 \in L^*$, since $110001110011 = 1 \circ 100 \circ 01 \circ 1 \circ 100 \circ 1 \circ 1$, and each of these strings is in L.

   - For a property P of strings to be admissible as a specification of a language, there must be an algorithm for deciding whether a given string belongs to the language.

     $L = \{w \in \Sigma^* : w \text{ has property } P\}$
   - Language recognition device: An algorithm that is specifically designed for some language L, to answer questions of the form "Is string w a member of L?"
5. Finite Automaton/Finite State Machine
   - Automaton (plural: automata) is a machine designed to respond to encoded instructions; a robot
   - a language recognition device, as a specifically designed algorithm for some language L
   - Given language L, is x ∈ L ?
   - We say that a finite automaton accepts language L, if it accepts (i.e. answers yes for) all strings in L and rejects (i.e. answers no for) all strings outside L.
   - In the context of a compiler, a finite automaton corresponds to a lexical analyzer, which recognizes and groups the tokens of a language (such as 'begin' and '+').

6. Some knowns

   Some knowns and open questions: $P \subset$ NP $\cap$ co-NP; If $X \in$ NPC and $X \in$ co-NP, then NP = co-NP; If P=NP, then NP=co-NP.

Motivation

7. Polynomial-time algorithms
   - on inputs of size n, their worst-case running time is $O(n^k)$ for some constant k.
8. NP-completeness
   - Applied to decision problems
   - We can cast a given optimization problem as a related decision problem by imposing a bound on the value to be optimized.
     - the hardest problems in NP.
     - Definition: A language $L \subseteq \{0,1\}^*$ is ***NP-complete*** if

       1. $L \in$ NP, and
       2. $L' \leq_P L$ for every $L' \in$ NP. "NP-hard"
9. NP-hard
   - If a language L satisfies property 2 in the previous point, but not necessarily property 1.
10. Class P
    - Problems that are solvable in polynomial time.
    - Complexity class P: the set of concrete decision problems that are polynomial-time solvable

      $$\text{P} = \{L \subseteq \{0,1\}^* : \text{there exists an algorithm } A \text{ that decides } L \text{ in polynomial time}\}.$$
11. Class NP
    - Problems that are verifiable in polynomial time.
    - If given a certificate of a solution, then the certificate can be verified to be correct in time polynomial in the size of the input to the problem.
    - Complexity class NP: A language L belongs to NP iff there exist a two-input polynomial-time algorithm A and a constant c such that:

      $$L = \{x \in \{0,1\}^* : \text{there exists a certificate } y \text{ with } |y| = O(|x|^c) \text{ such that } A(x,y) = 1\}.$$

      - We say that algorithm A verifies language L in polynomial time.

      HAM-CYCLE $\in$ NP

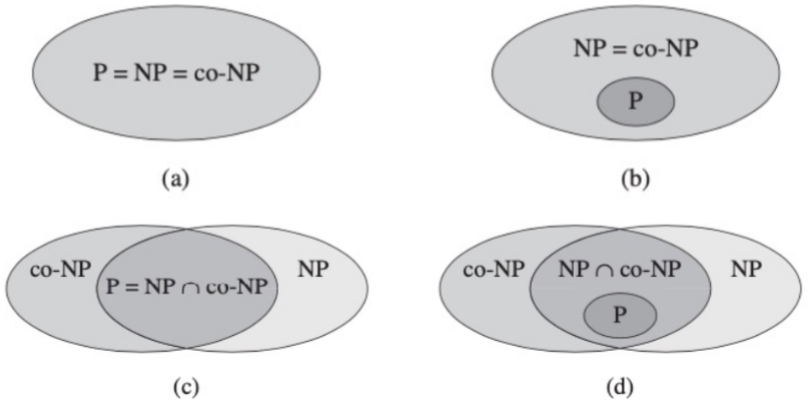
      if $L \in$ P, then $L \in$ NP $\Rightarrow$ P $\subseteq$ NP
12. Class Co-NP
    - Complexity class co-NP: the set of languages L such that $\overline{L} \in$ NP
13. Class NPC
    - If it is in NP and is as hard as any problem in NP
14. Polynomial-time verification
    - Algorithms that verify membership in languages.
    - For a given instance i=<G,u,v,k> of the decision problem path, also given a path p from u to v. We can easily check whether p is a path in G and whether the length of p is at most k.
    - View p as a certificate that the instance indeed belongs to the path.
15. Hamiltonian cycles (ham-cycle)
    - Use for a no polynomial-time decision algorithm, given a certificate, verification is easy.
    - A Hamiltonian cycle of an undirected graph is a simple cycle that contains each vertex.
    - Hamiltonian-cycle problem: Does a graph G have a Hamiltonian cycle.

      $$\text{HAM-CYCLE} = \{\langle G \rangle : G \text{ is a hamiltonian graph}\}$$

16. Verification algorithms
    - A two-argument algorithm A, where onr argument is an ordinary input string x and the other is a binary string y called a certificate.
    - A two-argument algorithm A verifies an input string x if there exists a certificate y such that A(x, y) = 1
    - The language verified by a verification algorithm A is:

      $L = \{x \in \{0,1\}^* : \text{ there exists } y \in \{0,1\}^* \text{ such that } A(x,y) = 1\}$

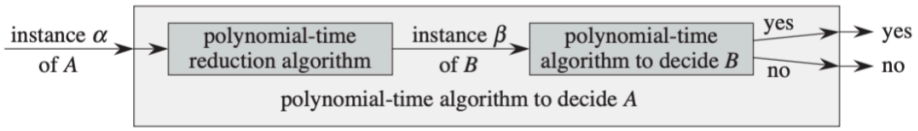
      HAM-CYCLE = $\{\langle G \rangle : G \text{ is a hamiltonian graph}\}$
17. Determine different class



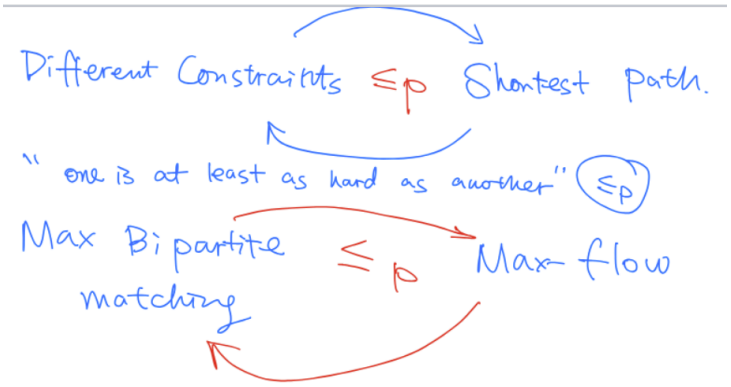
18. Reducibility

a language L_1 is ***polynomial-time reducible*** to a language L_2, written $L_1 \leq_P L_2$, if there exists a polynomial-time computable function $f : \{0,1\}^* \to \{0,1\}^*$ such that for all $x \in \{0,1\}^*$,

$x \in L_1$ if and only if $f(x) \in L_2$. (34.1)

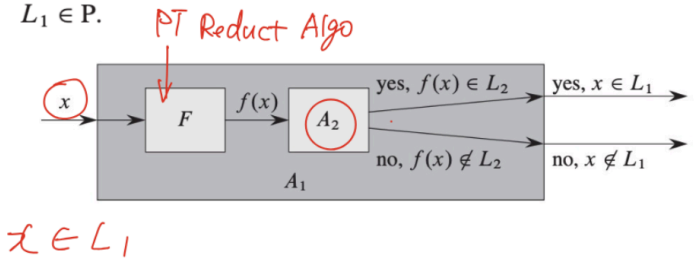


- Polynomial-time reductions: showing that one problem is at least as hard as another, ro within a polynomial-time factor.



Lemma 34.3

If $L_1, L_2 \subseteq \{0,1\}^*$ are languages such that $L_1 \leq_P L_2$, then $L_2 \in$ P implies $L_1 \in$ P.



x ∈ L1

Clique

19. Clique
    - **Definition**: in an undirected graph $G = (V, E)$, a *clique* is a subset of the vertices $V' \subseteq V$, such that each pair of vertices in V' is connected by an edge $\in$ E
    - **Clique problem**: optimization problem for finding max size clique
    - **Application**: social network, chemistry networks, computational biology
    - **Decision problem:** Does G has a clique of size K?

    $\text{CLIQUE} = \{\langle G, k\rangle : G \text{ is a graph containing a clique of size } k\}$

    - **Naïve Algorithm:**

    brute-force. $\Omega\left(k^2\binom{|V|}{k}\right)$ $\frac{k(k-1)}{2}$

20. Theorem: the Clique problem is NP-complete
    - Clique belongs to NP

    Proof To show that CLIQUE $\in$ NP, for a given graph $G = (V, E)$, we use the set $V' \subseteq V$ of vertices in the clique as a certificate for G. We can check whether V' is a clique in polynomial time by checking whether, for each pair $u, v \in V'$, the edge (u, v) belongs to E.

    V on inputs $(G = (V, E), k), c = \{v_1, \ldots, v_m\}$:
    If $k < 0$ or $k > |V|$ or $m \neq k$ or $c \not\subseteq V$, **reject**
    For $i = 1$ to m, $j = i + 1$ to m :
    If $(v_i, v_j) \notin E$, **reject**
    Accept

    - Clique belongs to NP-hard

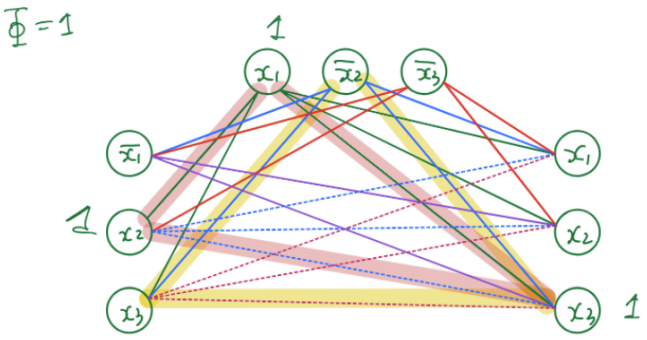
    3-CNF-SAT $\leq_P$ CLIQUE

    - $\phi \in$ 3SAT $\Rightarrow$ $f(\phi) \in$ CLIQUE
    - $\phi \notin$ 3SAT $\Rightarrow$ $f(\phi) \notin$ CLIQUE

21. 3-CNF-SAT
    - A satisfying assignment simultaneously satisfies each clause, so that each clause has at least one literal that is true.
    - For a formula with m clauses to be satisfiable, there must be some set of m literals, one from each clause, that can simultaneously be true.
        - If no such set exists, the formula is unsatisfiable.
22. Reduction algorithm: 3-CNF-SAT <= $_p$Clique
    - Given a circuit equation **Φ**
    - List all three clauses in three directions, each consisting of all elements in that clause.
    - An edge is placed between two vertices v_i' *and* v_j' if:
        - they are from different clauses
        - Their corresponding literals are consistent (one is not the 'head' one of the other)
            - e.g. if one is x_1, then $\neg x_1$ cannot connect with it
    - If this graph has a clique of size k, k is num o clauses, then the circuit is satisfiable.

$\phi = (x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee x_3)$



- B is at least as hard as A
- B is harder than A
- A is poly time reducible to B
- $\exists$ poly time computable function f s.t. $\alpha \in A \Rightarrow f(\alpha) \in B$
- we can decide $B \Rightarrow$ we can decide A
- $A \leq_p B \Rightarrow$ running time of $A \leq$ running time of $B + \mathcal{O}(n^k)$

23. Prove NP-hard by reduction

Informally: a problem is in NPC if it is at least as hard as any problem in NP.

Formally: a decision problem L s.t.

1. $L \in NP$
2. $L' \leq_P L$, $\forall L' \in NP$ (L is NP-hard)

Using reductions:
Suppose $A \leq_p B$, then $A \leq B + O(n^k)$, $\exists k \in N$
if $A \in P$, then $B \in$? we don't know
if $A \in$NPC, then $L' \leq_p A$ $\forall L' \in$NPC. Also $L' \leq_p A \leq_p B$ $\forall L' \in$NPC, which implies $B \in$NP-Hard
if $B \in P$, then $A \in P$
if $B \in$NPC, then $A \in$? we don't know

Given decision problem A whose complexity class is unknown

1. Show that $A \in NP$ (i.e. that A can be verified in polynomial time)
   - provide a certificate: the evidence that the solution is an instance of A
   - provide a verification procedure that checks if the certificate is valid
2. Show that $A \in$NP-Hard (i.e. A is at least as hard as all other problems in NP)
   - determine problem $L' \in$NPC you will reduce from (often given)
   - show $L' \leq_p A$, i.e. find polynomial reduction f s.t. $l \in L' \Leftrightarrow \alpha \in A$

Note: we reduce known to unknown, i.e. start with an instance of a known NPC problem, and tranform it to unknown problem. You should not start with the unknown problem and reduce it to known problem.

24. Conversion algorithm: 3-CNF-SAT to clique

- Code

```
f on input φ (a 3CNF formula with m clauses) :
  G := an empty graph
  For each literal ℓ_i ∈ φ :
    Add a vertex v_i to G
  For each pair of literals ℓ_i, ℓ_j ∈ φ from different clauses :
    If ℓ_i ≠ ¬ℓ_j :
      Add an edge (v_i, v_j) to G
  Return (G, m)
```

- Proof: show that this transformation of **Φ** into G is a reduction.

➔ Suppose that ϕ has a satisfying assignment.

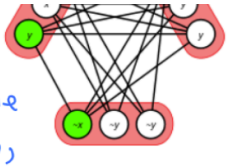
3-CNF-SAT $(\square \vee \neg\square \vee \neg\square) \wedge (\square \vee \square \vee \square) \wedge (\neg\square \vee \neg\square \vee \neg\square) \cdots$

suppose the formula is satisfiable, then a satisfying assignment means each clause has at least 1 literal=1

thus, picking 1 true literal from each clause, with non-contradictory assignment, yields a set V' of k vertices

$\Rightarrow$ V' is a clique

➔ Suppose that G has a clique V' of size k



suppose G has a clique V of size k

No edge from the same triple $\Rightarrow$ V' has one vertex per triple

$\Rightarrow$ one vertex per clause, none-contradiction, yields an consistent assignment satisfying all clauses

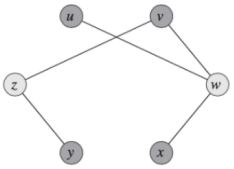
show there isn't a satisfying assignment for the formula, then the maximum clique size is $<m$.

Alternative

25. Vertex cover
    - Definition:

      A ***vertex cover*** of an undirected graph $G = (V, E)$ is a subset $V' \subseteq V$ such that if $(u, v) \in E$, then $u \in V'$ or $v \in V'$ (or both)

      - Each vertex covers its incident edges, and a vertex cover for G is a set of vertices that covers all the edges in E.
    - Size: the number of vertices in it.

      has a vertex cover {w, z} of size 2.

      - It is the set of vertex that combine together can cover all edges of the graph (the edge that comes out of the vertex)
26. Vertex-cover problem
    - The ***vertex-cover problem*** is to find a vertex cover of minimum size in a given graph.
    - **Decision problem**: determine whether a graph has a vertex cover of a given size k.
27. Theorem: vertex-cover problem is NP-complete

    1 VERTEX-COVER $\in$ NP

    Certificate: subset $V' \subseteq V$

    Verification algorithm:

    - $|V'|=k$
    - for each edge $(u, v) \in E$, that $u \in V'$ or $v \in V'$.

    2 CLIQUE $\leq_P$ VERTEX-COVER

    Intuition: This reduction relies on the notion of the "complement" of a graph.

    Given an undirected graph $G = (V, E)$, we define the ***complement*** of G as $\overline{G} = (V, \overline{E})$, where $\overline{E} = \{(u, v) : u, v \in V, u \neq v, \text{ and } (u, v) \notin E\}$.

    $\overline{G}$ is the graph containing exactly those edges that are not in G

28. Traveling Sales Man problem

In the ***traveling-salesman problem***, which is closely related to the hamiltonian-cycle problem, a salesman must visit n cities. Modeling the problem as a complete graph with n vertices, we can say that the salesman wishes to make a ***tour***, or hamiltonian cycle, visiting each city exactly once and finishing at the city he starts from. The salesman incurs a nonnegative integer cost $c(i, j)$ to travel from city i to city j, and the salesman wishes to make the tour whose total cost is minimum, where the total cost is the sum of the individual costs along the edges of the tour.

$$\begin{aligned}\text{TSP} = \{\langle G, c, k\rangle : \ & G = (V, E) \text{ is a complete graph,} \\ & c \text{ is a function from } V \times V \to \mathbb{Z}, \\ & k \in \mathbb{Z}, \text{ and} \\ & G \text{ has a traveling-salesman tour with cost at most } k\}.\end{aligned}$$

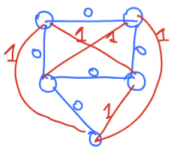
Theorem: TSP is NP-complete.

1. **TSP belongs to NP**

   Certificate: sequence of n vertices

   Verification algorithm: this sequence contains each vertex exactly once, sums up the edge costs, and checks whether the sum is at most k. (ploy-time)

2. **TSP is NP-hard** HAM-CYCLE $\leq_P$ TSP



$G = (V, E)$ be the instance of HAM-CYCLE

$G' = (V, E')$ - - - - - - $\text{TSP}(G', c, 0)$

$E' = \{(i, j) : i, j \in V \text{ and } j \neq i\}$

Theorem: TSP is NP-complete.

Complete answer

1. **TSP belongs to NP**

   Certificate: set of edges. / sequence of vertices

   Verification algorithm: ① check the cost, $\leq k$?

2. **TSP is NP-hard** ② check if we visit all the vertices exactly once. $\Rightarrow$ poly-time.

HAM-CYCLE $\leq_p$ TSP

① let a graph $G = (V, E)$, we try to find a HAM-CYCLE on G,

② complete G to a complete graph G'

③ assign cost to each edge.

TSP – near-optimal solution

Idea: Construct a minimum spanning tree, do a preorder walk of the tree, and construct the TSP tour in the order in which each vertex is first visited during the preorder walk.

APPROX-TSP-TOUR(G, c)

1 select a vertex $r \in G.V$ to be a "root" vertex
2 compute a minimum spanning tree T for G from root r using MST-PRIM(G, c, r)
3 let H be a list of vertices, ordered according to when they are first visited in a preorder tree walk of T
4 **return** the hamiltonian cycle H

Time: Easy to make Prim's algorithm run in $O(V^2)$ time. The rest is $\Theta(V)$, so total running time is $O(V^2)$.

Theorem

APPROX-TSP-TOUR is a polynomial-time 2-approximation algorithm for the traveling-salesperson problem with the triangle inequality.

29. NPC solving method
    - Remember to write QED overall
    - Show that one belongs to NP:
        - Use the conversion algorithm
        - Certificate: usually uses the question proposed in the content.
        - Verification Process: Verify that the check process of the certificate can be run in polynomial time.
    - Show that one belongs to NP-hard:
        - Use the reduction algorithm
            - e.g. 3-CNF-SAT ⇔ 4-CNF-NAE-SAT
        - Transformation: create a 4-CNF-NAE-SAT from 3_CNF-SAT by adding a z component to every clause, name it as a circuit prime.
        - Equivalence proof: giving the instances from both sides, to prove one is equal to another.
            - Forward way proof
            - Backward way proof
30. NPC Ham-Cycle solving method
    - Hamiltonian Cycle: Given graph G=(V,E), find a simple cycle through all vertices in V.
    - Hamiltonian Path: Given graph G=(V,E), find a simple path through all vertices in V

Ham-Cycle ∈ NP
- Certificate: the cycle $v_1 \to \cdots \to v_n \to v_1$ (poly size)
- Verification procedure: check all vertices appear exactly once except the first and last which equal each other $\mathcal{O}(V)$
  check that all edges in path exist in E, $\mathcal{O}(E)$ (poly time)

Example:

HAM-CYCLE ∈ NP-Hard via Ham-Path $\leq_p$ Ham-Cycle
Transformation: Given: $G = (V, E)$ be an instance of Ham-Path
Contruct: $G' = (V', E')$ with $V' = V \cup \{x\}$ and $E' = E \cup \{(x, v), (v, x) : v \in V\}$ in $\mathcal{O}(V + E)$ (poly time)
Claim: $\langle G \rangle \in$ Ham-Path ⇔ $\langle G' \rangle \in$ Ham-Cycle
Equivalence Proof: (⇒) Suppose $v_1 \to \cdots \to v_n$ is Ham-Path in G then $v_1 \to \cdots \to v_n \to x \to v_1$ is a Ham-Cycle in G'
(⇐) Suppose $v_1 \to \cdots \to v_n \to x \to v_1$ is the Ham-Cycle in G'
then $v_1 \to \cdots \to v_n$ is a Ham-Path in G.

31. Combinatorial Equivalence Checking (CEC)
    - CEC: given boolean formulas $A(x_1, ..., x_n)$ and $B(x_1, ..., x_n)$ does there exist an assignment s.t. $A(x_1, ..., x_n) \neq B(x_1, ..., x_n)$
    - Formula-SAT: given boolean formula $F(y_1, ..., y_n)$ does there exist an assignment s.t. $F(y_1, ..., y_n) = 1$

CEC ∈ NP
- Certificate: The assignment of $x_1, ..., x_n$ (poly size)
- Verification procedure: Evaluate $A(x_1, ..., x_n)$ and $B(x_1, ..., x_n)$. This takes polynomial time w.r.t. number of clauses. Check if $A \neq B$ $\mathcal{O}(1)$

CEC ∈ NP-Hard via Formula-SAT $\leq_p$ CEC
Transformation: Given: a boolean formula F
Construct: A = F and B = 0 in poly time
Claim: $\langle F \rangle \in$ Formula-SAT ⇔ $\langle A, B \rangle \in$ CEC
Equivalence Proof: (⇒) suppose $\exists y_1, ..., y_n$ s.t. $F(y_1, ..., y_n) = 1 \Rightarrow A = F(y_1, ..., y_n) \neq 0 = B$
⇐ suppose $\exists y_1, ..., y_n$ s.t. $A \neq B \Rightarrow F(y_1, ..., y_n) \neq 0 \Rightarrow F(y_1, ..., y_n) = 1$

32. Half-3-CNF-SAT

Half-3-CNF-SAT: Given 3-CNF formula Φ with n variables and m clauses. Does there exist an assignment to Φ s.t. exactly m/2 clauses evaluate to 1 and other m/2 evaluate to 0?
Note: $(x \vee \bar{x} \vee y)$ is allowed.

Half-3-CNF-SAT $\in$ NP

- Certificate: the assignment (poly size)
- Verification procedure: evaluate all clauses and check exactly half evaluate to 1 and half to 0, $\mathcal{O}(m)$ (poly time)

3-CNF-SAT $\leq_p$ Half-3-CNF-SAT
Transformation: Given: 3-CNF formula Φ
Construct: add 3 variables p, q, r and let $\Phi' = \Phi \wedge (p \vee \bar{p} \vee q)^m \wedge (p \vee q \vee r)^{2m}$ (first bracket is always true, second bracket either all true or all false) in $\mathcal{O}(m)$
Claim: $\langle \Phi \rangle \in$ 3-CNF-SAT $\Leftrightarrow \langle \Phi' \rangle \in$ Half-3-CNF-SAT
Equivalence Proof: ($\Rightarrow$) Suppose $\exists$ assignment s.t. $\Phi = 1 \Rightarrow$. Let $p = q = r = 0$ and exactly $2m$ out of $4m$ clauses in Φ' are 1
($\Leftarrow$) Suppose exactly $2m$ clauses in Φ' are 1. We know m of them come from $(p \vee \bar{p} \vee q)$ clauses, so the other m clauses must come from Φ

33. Partition

- Partition: Let S be set of positive integers, then exists $S_1, S_2 \subseteq S$ s.t. $S_1 \cup S_2 = S$ and $S_1 \cap S_2 = \emptyset$ with $\sum S_1 = \sum S_2$
- Subset-Sum: Let S be a set of position integers and t be an integer, $\exists T \subseteq S$ s.t. $\sum T = t$

Partition $\in$ NP

- Certificate: the subsets S_1 and S_2
- Verification procedure: check that S_1 and S_2 are a partition of S, $\mathcal{O}(|S|)$. Check that $\sum S_1 = \sum S_2$, $\mathcal{O}(|S|)$.

Subset-Sum $\leq_p$ Partition
Transformation: Given: set S and target t
Construct: Let $s = \sum S$, let $S' = S \cup \{s - 2t\}$ in $\mathcal{O}(|S|)$
Claim: $\langle S, t \rangle \in$ Subset-sum $\Leftrightarrow \langle S' \rangle \in$ Partition
Equivalence Proof: ($\Rightarrow$) suppose $\exists T_1 \subseteq S$ s.t. $\sum T_1 = t$, then $T_2 = S \setminus T_1$ has $\sum T_2 = s - t$. So $\sum T_1 \cup \{s - 2t\} = s - t$. Let $S_1 = T_1 \cup \{s - 2t\}$ and $S_2 = T_2$
($\Leftarrow$) suppose $\exists S_1, S_2 \subseteq S'$ s.t. $\sum S_1 = \sum S_2$, note $\sum S_1 + \sum S_2 = \sum S + (s - 2t) = 2s - 2t$, thus $\sum S_1 = \sum S_2 = s - t$
suppose $s - 2t \in S_1$, then $S_1 \backslash \{s - 2t\} = t \Rightarrow \exists T = S_1 \backslash \{s - 2t\} \subseteq S$ with $\sum T = t$

34. Graph 3-Coloring

- 3-col: given $G = (V, E)$ can you assign 1 of 3 colours to each vertex s.t. no vertices that share an edge are the same color.
- k-clique-cover: given $G = (V, E)$ and integer k, can we partition V into $V_1, V_2, ..., V_k$ s.t. each V_i

3-col $\in$ NP

- Certificate: the color assignment of each vertex
- Verification procedure: check color assignment contains $\leq$ 3 colors, $\mathcal{O}(V)$. Check for each edge (u, v) that color[u] $\neq$ color[v], $\mathcal{O}(E)$.

3-clique-cover $\leq_p$ 3-col
Transformation: Given: $G = (V, E)$
Construct: $G' = (V, E')$ with $E' = \{(u, v) \in V \times V : (u, v) \notin E\}$ (Compliment graph) in $\mathcal{O}(V + E)$
Claim: $\langle G \rangle \in$ 3-clique-cover $\Leftrightarrow \langle G' \rangle \in$3-col
Equivalence Proof: ($\Rightarrow$) suppose G has clique cover, color each vertex in V_i a different color, then color[u] = color[v] $\Rightarrow u, v \in V_i \Rightarrow (u, v) \notin E'$
($\Leftarrow$) suppose G' is 3-colorable, color[u] = color[v]$\Rightarrow (u, v) \notin E' \Rightarrow (u, v) \in E$
so the set of all vertices with the same colour forms a clique in G.

Approximation Algorithm

General

1. Approximation Ratio

Trade-off: *Time complexity vs* $\rho(n)$

Denote:

- Input size: n
- the solution value of the approximation algorithm as C
- the optimal solution value is C*

For maximization problem: $0 < C \leq C^*$

For minimization problem: $0 < C^* \leq C$

- We say that an algorithm for a problem has an ***approximation ratio*** of $\rho(n)$

minimization maximization

$$\max\left\{\frac{C}{C^*}, \frac{C^*}{C}\right\} \leq \rho(n)$$

If an algorithm achieves an approximation ratio of , we call it a ***ρ(n)-approximation algorithm***.

Tutorial

2. Definition

Approximation algorithm: An algorithm that returns **near-optimal** solution.

Approximation ratio $\rho(n)$: if for any input of size n the cost C of the solution produced by the algorithm is within a factor of $\rho(n)$ of the cost C^* of an optimal solution. $\rho(n) \geq 1$

- Minimization problem: $\frac{C}{C^*} \leq \rho(n)$
- Maximization problem: $\frac{C^*}{C} \leq \rho(n)$

$\rho(n)-$approximation: if the algorithm achieves an approximation ratio of $\rho(n)$.
Note: usually, $\rho(n)$ is a constant.

Most of the problems (also relevant to real-life, e.g. 0-1 integer programming[1] and games) are NP-complete[2][3]. We want to be able to efficiently find near-optimal solutions to those problems in poly-time. Here is where the approximation algorithm comes into place.

Polynomial-time $\rho(n)$-approximation: a $\rho(n)$-approximation that runs in polynomial time.

Examples[4]:

- Local Search (Greedy): start with a random feasible solution, then incrementally modify the solution to achieve local optimality using heuristics.
- Relaxation: drop the hard terms. e.g. in integer programming, $\min c^T x$ s.t. $x_i \in \{0,1\}, \mathbb{1}^T x \geq 1$ can be relaxed to $\min c^T x$ s.t. $\mathbb{1}^T x \geq 1$

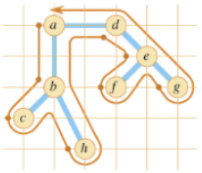
3. Vertex Cover 2-approximation

Algorithm: Choose any edge $e = (u, v)$ from $G = (V, E)$, add u, v to the covered set C, remove all incident edges on u, v. Iterate until no edges remain in G.

Prove that the algorithm is 2-approximation.
Proof: Let A denote the set of edges the algorithm picks, C^* be the optimal vertex cover. C^* must cover at least one endpoint of each edge in E, and thus at least one endpoint of each edge in A.
Since no 2 edges in A share common endpoint, then no 2 edges in A are covered by the same vertex in C^*.
For every $v \in C^*$, there is at most one edge in A connecting v, $|C^*| \geq |A|$ (1).
Each edge in A should connect to 2 vertices in C, so $|C| = 2|A|$ (2).
By (1) and (2), $|C| = 2|A| \leq 2|C^*|$

4. 2D TSP with triangle inequality

Assume $G = (V, E)$, with the cost for each $e = (u, v) \in E$ satisfying $c(u, w) \leq c(u, v) + c(v, w)$.
Algorithm: Compute MST T of G from a random start node v.
Let H be a list of vertices ordered according to pre-order traversal of T, return Ham-cycle of H.



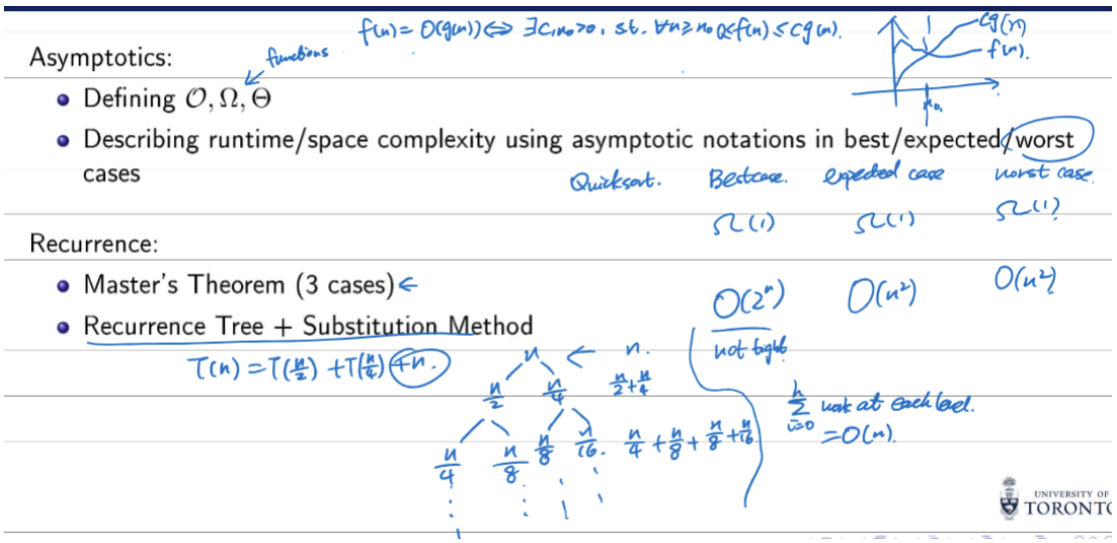
Prove that the algorithm is 2-approximation.
Proof: Let H^* be the optimal tour (a Ham-cycle), deleting any of the edge gives a spanning tree with non-negative edges.
A MST T gives a lower bound $c(T) \leq c(H^*)$ (1).
A full walk W of T traverses every edge of T exactly twice, $c(W) = 2c(T)$ (2).
By (1) and (2), $c(W) = 2c(T) \leq 2c(H^*)$
Full walk visits some vertices more than once, so not a tour, but by triangle inequality, deleting a vertex v in the sequence u, v, w, which gives a direct connection $u \to w$, decreases the cost, $c(H) \leq c(W)$. Thus $c(H) \leq 2c(H^*)$

UNIVERSITY OF TORONTO

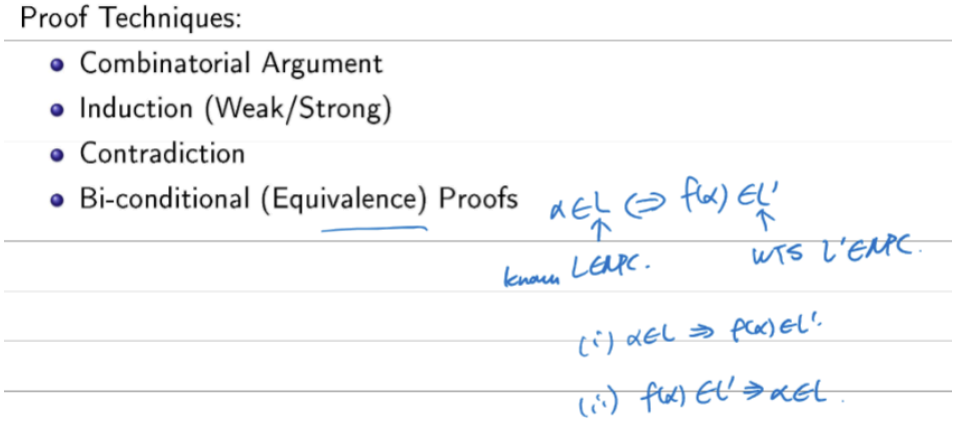
Final Review Slides

Topics

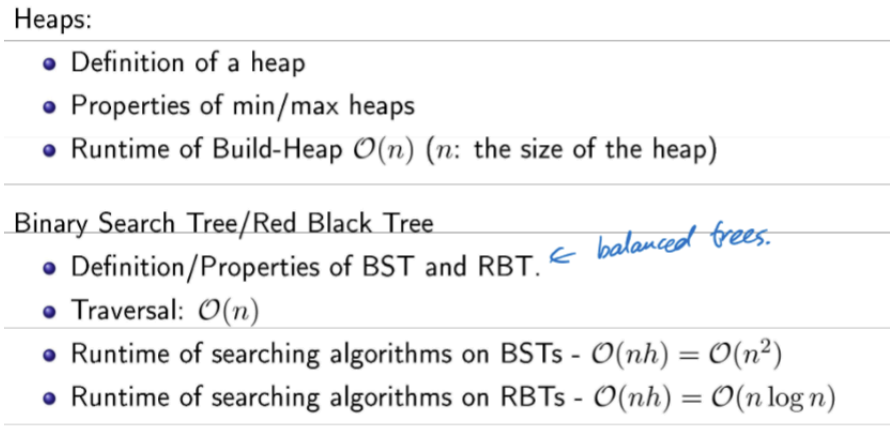
1. Asymptotics and Recurrence



2. Proof Techniques



3. Heaps and BST and RBT



4. Sorting

Comparison based sorting:

- Mergesort - $\mathcal{O}(n \log n)$
- Heapsort - $\mathcal{O}(n \log n)$
- Quicksort - $\mathcal{O}(n^2)$
  - Idea of Partition
  - Idea of Randomized Algorithms
  - Strong induction proof of correctness for recurrent algorithms

Non-comparison based (finite key sets only):

- Counting sort - $\mathcal{O}(n + k)$
- Radix sort - $\mathcal{O}(d(n + k))$

5. Algorithm Design

Algorithm Design

- Algorithm: start with some algorithm that correctly solves the problem, rather than trying to figure out the most optimal algorithm at first. For example,
  1. Start with linear search (brute force)
  2. Optimize with sorting + binary search (or other divide and conquer algorithms)
  3. Apply data structures (*e.g.* Heap) if they help
- Proof of Correctness
- Runtime Complexity
- Space Complexity

A sub-optimal but correct algorithm gives you most of the marks. But an incorrect algorithm with correct complexity gives no mark. More detail in the midterm review slides. HW2 has good examples.

6. DP and Greedy

Dynamic Programming:

- Optimal substructure
- Overlapping subproblems
- Recurrence + Memorization (Memoization)

Greedy:

- Idea: locally optimal solution leads to a global optimal solution. Good as approximation.
- Reduction to smaller subproblem
- First Greedy choice property
- Optimal substructure

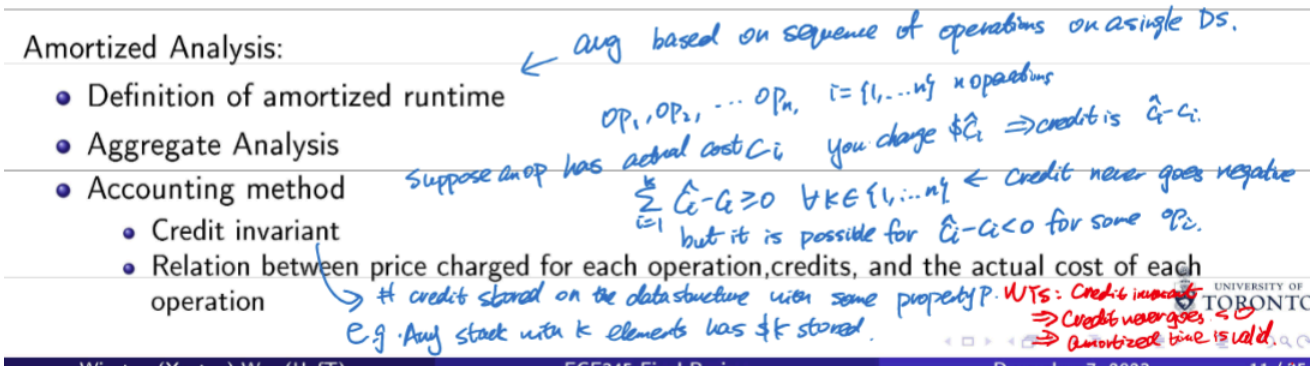
7. Hashing and Amortized Analysis

Hashing:

- Definition of a hash function, load factors etc.
- Collision resolution
- Chaining (could have bad run time)
- Linear probing (subject to clustering)

Amortized Analysis:

- Definition of amortized runtime
- Aggregate Analysis
- Accounting method
  - Credit invariant
  - Relation between price charged for each operation,credits, and the actual cost of each operation



8. Graphs and MST

Graphs

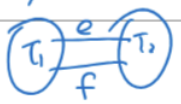
- Terminology for graphs: vertices, edges, paths, cycles, etc.
- Basic proofs relevant to graphs.

Elementary Graph Algorithms:

- Breath-First Search - $\mathcal{O}(V + E)$ ← SSSP with all edges weights = 1
- Depth-First Search - $\mathcal{O}(V + E)$ ← directed – forward edge / cross edge. undirected – only tree/back edge. → cycle in the graph
- Topological Sort - $\mathcal{O}(V + E)$ ← only useful for DAG. if not DAG, need to find SCC (20.5)

Minimum Spanning Trees:

- Prim's Algorithm - $\mathcal{O}(E \log V)$
- General proof for MST by contradiction: Adding an edge to an MST creates a cycle



Removing e disconnects T into T_1 and T_2.
Can you find another edge f to reconnect
And what will happen to the weight?
$w(T \setminus \{e\} \cup \{f\}) < w(T)$
UNIVERSITY OF TORONTO

9. SSSP and Max Flow

Single-Source Shortest Paths:

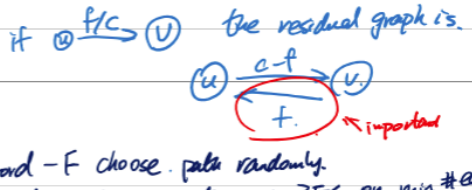
- Edge relaxation - $\mathcal{O}(1)$ ←
- Bellman-Ford - $\mathcal{O}(VE)$ (no negative cycles) ← can detect negative weight cycles
- Dijkstra - $\mathcal{O}((V + E) \log V)$ (no negative edges)
- Difference Constraints (Bellman-Ford)
  - Pseudo Start
  - Negative cycle = no solution
  - Valid SP Distance = feasible solution

if $d[v] > d[u] + w(u,v)$ then
$d[v] = d[u] + w(u,v)$ ← change this to achieve different applications.
$\pi[v] = u$.

SPSP.
APSP
all solvable by SSSP.

Max Flow

- Definition of flow and residual networks
- Max-Flow-Min-Cut Theorem
- Edmonds-Karp (Ford-Fulkerson + BFS)



Ford – F choose path randomly.
EK chooses path using BFS. on min # edges from start.
UNIVERSITY OF TORONTO

10. Complexity Theory

Topics

Complexity Theory:

- Definition of P, NP, and NPC
- Prove that $L \in$NPC i.e. $L \in$ NP ∧ NP-Hard.
  - Prove $L \in$NP
    - a proposed solution.
    - **Certificate:** the evidence that the solution is an instance of A (poly size) 1
    - **Verification Procedure:** checks if the certificate is valid (poly time) 1
  - Prove $L' \leq_p L$ for some known $L' \in$NPC 15'
    - Meaning of $L' \leq_p L$ Given an instance of L', how you can construct an instance of L in polynomial time
    - **Transformation:** Polynomial time reduction from L' to L
    - **Equivalence:** Prove that $l \in L' \Leftrightarrow \alpha \in L$. i.e. solving an instance in $L' \Leftrightarrow$ solving an instance in L

L is harder than L'

P: you can reduce/transform instance of L' to instance of L in polynomial time.

Always transform from easier problem (known) to unknown

$A \leq_p B$, $B \leq_p C$
if $B \in NP$, A can be P, NP.
C can be NP, NP-hard
if $B \in NPC$, $C \in$ NP-Hard.
A doesn't know.

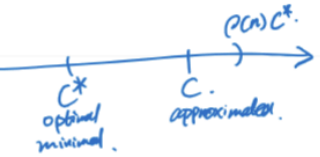
UNIVERSITY OF TORONTO

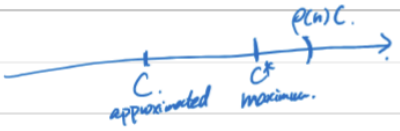
11. Approximation Algorithms

Approximation Algorithms:

- Definition of $\rho(n)$-Approximation
- $\rho(n) \geq 1$ in CLRS and this course
- Minimization $\frac{\text{approx}}{\text{opt}} \leq \rho(n)$
- Maximization $\frac{\text{opt}}{\text{approx}} \leq \rho(n)$

for different input size n, we can always bound C with $\rho(n) \cdot C^*$.





Q why is it $\rho(n)$? A. As n grows, the problem becomes harder, approximation becomes worse. $\rho(n) \uparrow$.

Usually $\rho(n)$ is constant

minimization $\rho(n)=2$. $C \leq 2C^*$. $\frac{C}{C^*} \leq 2$.
approx optimal

UNIVERSITY OF TORONTO

Data Structures

12. Heaps

Definition: A heap is an array $A = [a_1, a_2, ..., a_n]$ of elements such that:

- Heap shape property: the heap is an almost complete binary tree (all except the last row is full)
- Heap order property:
  - Max-heap: $\forall i$, $A[\text{parent}(i)] \geq A[i]$
  - Min-heap: $\forall i$, $A[\text{parent}(i)] \leq A[i]$

Indexing parents and children (Assume index of array starts at 1)

- $\text{parent}(i) = \lfloor \frac{i}{2} \rfloor$
- $\text{left}(i) = 2i$
- $\text{right}(i) = 2i + 1$

13. BST

BST properties:

- 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$[x]$

Key Take-Aways

- In general, if a BST has n nodes, $h = \mathcal{O}(n)$. Only if the BST is balanced, $h = \mathcal{O}(\log n)$
- Minimum/Maximum and searching for any arbitrary key $\mathcal{O}(h)$
- Successor/Predecessor $\mathcal{O}(h)$
- Insertion/Deletion $\mathcal{O}(h)$
- Build-BST: $\mathcal{O}(n^2)$ worst case (chain)

14. RBT

Red Black Tree is a Binary Search Tree with the following additional properties:

- Every node is either red or black
- The root is black
- All leaves (NIL) are black
- If a node is red, both its children are black
- For all nodes, all paths to all leaves have the same black-height (number of black nodes on path to a leaf, not including itself, but including NIL)

Key Take-Aways

- RB Trees are balanced $h = \mathcal{O}(\log n)$
- All read-only operations (e.g. traversals) are all the same as BST
- Rotation $\mathcal{O}(1)$

UNIVERSITY OF TORONTO

15. Hash Tables

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

Good hashing scheme:

- Simple uniform hashing: any given element is equally likely to be hashed into any of the m slots.
- Good mechanism for collision resolution (since $m < |U|$, there could be collision)

Collision Resolution:

- Chaining
- Linear probing