1 Complexity and Master Theorem

Complexity. Symbols are $o, O, \Theta, \Omega, \omega$ for $<, \leq, \approx, \geq, >$, respectively. Recall that $f(n) = \text{symbol}(g(n))$ iff $\lim_{n\to\infty} f(n)/g(n)$ is 0, less than constant, constant, more than constant, ∞.

Formally, for big O, exists constant c, n_0 if $0 \leq \frac{f(n)}{g(n)} \leq c \forall n \geq n_0$.

Sometimes you need to explicitly write the constants.

Note: The definition of Θ that I made above is actually a bit inaccurate. $f(n) = \Theta(g(n))$ iff $f(n) = O(g(n))$ and $g(n) = O(f(n))$ (or, $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$, or something similar).

This implies that Θ means *for n sufficiently large*, there exists "a lower bound" and "an upper bound" such that

$$\text{lower bound} \leq \frac{f(n)}{g(n)} \leq \text{upper bound},$$

or, in other words, both

$$\liminf_{n\to\infty} \frac{f(n)}{g(n)} > 0 \text{ and } \limsup_{n\to\infty} \frac{f(n)}{g(n)} < +\infty$$

must hold.

Moreover, $f(n) = O(g(n))$ iff the limsup property holds and $f(n) = \Omega(g(n))$ iff the liminf property holds.

Master Theorem. Given $T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n)$. crit $:= \log_b(a)$, ϵ, k, c constants > 0, ≥ 0, < 1.

- If ϵ, s.t. $f = O(n^{\log_b(a)-\epsilon})$, then $T(n) = \Theta(n^{\log_b(a)})$.
- If k, s.t. $f = \Theta\left(n^{\log_b(a)} \log^k(n)\right)$, $T(n) = \Theta(n^{\log_b(a)} \log^{k+1}(n))$.
- If ϵ, s.t. $f = \Omega(n^{\log_b(a)+\epsilon})$, and f satisfy *regularity condition* $[af(n/b)/f(n) \leq c \ (< 1)$ for all n large$]$, $T(n) = \Theta(f(n))$.

Other than that, use **telescoping method**; i.e. expressing expression as $\frac{T(n)}{g(n)} = \frac{T(n/b)}{g(n/b)} + h(n)$, **guessing/substitution + induction** method, or **recursion tree**.

2 Correctness and DnC

Good Algorithm.

- Must be correct (include corner cases),
- efficient (economical in time, space, resources),
- well-documented with sufficient details,
- maintainable (not tricky, easy to modify and abstractions are easily understood, not computer-dependent, usable as modules by others).

Correctness of Iterative Algorithms. (Involves loop). *Key in reasoning is loop invariant*, which is supposed to be:

- true at the beginning of an iteration,
- and remains true after the iteration (i.e. at beginning of next iteration).

How to use said invariant to show correctness:

- Need to show three things ("Induction").
- **Initialization:** true before first iteration of loop,
- **Maintenance:** if true before an iter, remains true before next iter,
- **Termination:** when algo terminates, invariant has *useful property* for showing correctness.

Correctness of Recursive Algorithms. Use math induction on *size of problem*.

More specifically, use strong induction $\Rightarrow$ prove base cases and show algo works correctly assuming algo works correctly for **all** smaller instances/cases.

Divide-and-conquer design paradigm. Divide problem into subproblems, **conquer** subproblems by solving them recursively, **combine** subproblem solutions.

This often leads to efficient algorithms (note: except if it can be solved using "pure logic" e.g. Kadane's algo in Practice Set 1 #5.)

3 Sorting and Decision Trees

What is Decision Tree? Tree-like model, where

- every node is a comparison,
- every branch represents the outcome of node's comparison,
- every leaf represents "class label" (decision after all comparisons).
- Note: worst-case running time is *height of tree*.

Counting and Radix Sort. Counting sort algo:

Counting sort

Set C[i] be the number of elements equal i.

```
for i ← 1 to k
    do C[i] ← 0
for j ← 1 to n
    do C[A[j]] ← C[A[j]] + 1        ▷ C[i] = |{key = i}|
```

Set C[i] be the number of elements smaller than or equal i.

```
for i ← 2 to k
    do C[i] ← C[i] + C[i–1]         ▷ C[i] = |{key ≤ i}|
```

Move elements equal i to B[C[i-1]+1..C[i]].

```
for j ← n downto 1
    do B[C[A[j]]] ← A[j]
       C[A[j]] ← C[A[j]] – 1
```

Takes $\Theta(n+k)$ time to sort n numbers in range $[1, k]$.

Radix sort: suppose there are T "digits". Do sort by the t^{th} least significant digit ($t \leftarrow 1$ to T) using a stable sorting algo.

Suppose the n b—bit "words" (radix sort) is broken into r—bit pieces, each pass of counting sort takes $\Theta(n+2^r)$ time.

Since there are b/r passes, we have:

$$T(n,b) = \Theta\left(\frac{b}{r}(n+2^r)\right).$$

Choose r to minimize $T(n,b)$: as $r > \log(n)$, time grows exponentially in n; so pick r close to $\log(n)$ (slightly smaller) by differentiation.

Correctness of radix sort. Induction on digit position.

Note: radix sort is fast when number of passes is small. Downside is little locality of reference, and a well-tuned quicksort fares better on modern processors (which features steep memory hierarchies).

Conclusion. Each sorting algorithm can handle different issues, as follows.

- Memory issue: InPlace sort.
- Need to preserve original order of *equal elements*: Stable sort.
- Average case fast: QuickSort.
- Worst case fast (comparison model): worst case $O(n\log(n))$ time sort.
- Worst case fast (non-comparison model): radix sort.

4 Randomized Algoroithms, illustrated by QuickSort

QuickSort Divide and Conquer Algorithm.

- *Divide:* Partition array into two subarrays around a ***pivot*** x such that the elements in lower subarray $\leq x \leq$ elements in upper subarray.
- *Conquer:* Recursively sort two subarrays.
- *Combine:* Trivial.

Suppose the "pivot" produces the subarrays with size j and $(n-j-1)$, then

$$T(n) = T(j) + T(n-j-1) + \Theta(n).$$

Average-case analysis of QuickSort. Uses $A(n) :=$ average running time of QuickSort, input size n. (Input chosen uniformly at random from the set of $n!$ permutations.)

$A(n) \approx 1.39 n \log_2(n)$.

Serious problem with QuickSort: time taken depends on initial/input permutation (is real data random?). Can we make it distribution insensitive?

If we pick pivot uniformly at random, time taken depends on random choices of pivot elements.

What makes Randomized Algorithms so Popular? Answer, based on lecture slides:

What makes Randomized Algorithms so Popular? [A study by *Microsoft* in **2008**]

Title: Cycles, Cells and Platters: An Empirical Analysis of **Hardware Failures** on a Million Consumer PCs

Authors: **Edmund B. Nightingale, John R. Douceur, Vince Orgovan**

Available at : research.microsoft.com/pubs/144888/eurosys84-nightingale.pdf

Event	Probability
My desktop will **crash** during this lecture	$> 10^{-7}$
RandQsort takes time at least **double** the average	$< 10^{-15}$

$n \geq 10^6$

\+ Simplicity

Note: in the lecture, we sum over i and j from the perspective of e_i (i^{th} smallest element) and e_j, we have $\Pr[e_i \text{ and } e_j \text{ get compared}] = \frac{2}{j-i+1}$.

Types of Randomized Algorithms.

- **Las Vegas Algo.** Output always correct, runtime is a random variable.
- **Monte Carlo Algo.** Output may be incorrect with (very) small probability, runtime is deterministic.

5 Weirder Stuff (Hashing, Worst-case Lin-time Order Stats)

Universal Hashing. Given hash table T with m slots, stores n elements, the **load factor** $\alpha := n/m$ is the average number of elements in a chain.

The hash function h is a randomized procedure satisfying **universal hashing** assumption: for any pair of **distinct** keys u, v,

$$\Pr[h(u) = h(v)] \le \frac{1}{m}$$

where *probability is over the* **random** *choice of the hash function* h.

For a key u contained in T, expected length of the chain $h(u)$ is at most $1+\alpha$.

Reason: length of chain at $h(u)$ [key means the number being inserted] is

$$1 + \sum_{v \neq u, v \in T} C_{uv} \, (:= \text{int}(h(u) == h(v))),$$

so length of chain is "at least one" since u itself belong to u's chain.

Worst-case linear-time order statistics. "This is a ketok-magic algo that was not here in my semester" —Edbert.

Developing the recurrence

T(n) **SELECT(*i*, *n*)**

Θ(n) { 1. Divide the n elements into groups of 5. Find the median of each 5-element group by rote.

T(n/5) { 2. Recursively **SELECT** the median x of the $\lfloor n/5 \rfloor$ group medians to be the pivot.

Θ(n) 3. **Partition** around the pivot x. Let $k = \text{rank}(x)$.

4. **If** $i = k$ **then** return x

T(3n/4) { 5. **Else if** $i < k$ **then** recursively **SELECT** the ith smallest element in the lower part

6. **Else** recursively **SELECT** the $(i-k)$th smallest element in the upper part

Solving the recurrence. Substitution: $T(n) \le cn$.

$$\begin{aligned} T(n) &\le T\left(\frac{1}{5}n\right) + T\left(\frac{3}{4}n\right) + \Theta(n) \\ &\le \frac{19}{20}cn + \Theta(n) \\ &= cn - \left(\frac{1}{20}cn - \Theta(n)\right) \\ &\le cn. \end{aligned}$$

Choose c large enough to handle both $\Theta(n)$ and the initial conditions.

6 Tutorials and Examples

Term test for Sem 2, AY19/20.

- $e^x \ge x+1$ for all $x \in \mathbb{R}^+$, in particularly, $x \ge \log(x+1) > \log(x)$ **for all x positive**. This is particularly useful in proving $n\log(n)$ in $O(n^2)$, for example.
- (Q2) Trick for $\sum_{j=1}^{n} j \cdot 2^j$ and $\sum_{j=1}^{n} j/(2^j)$:

$$\begin{aligned} \sum_{j=1}^{n} j \cdot 2^j &= \sum_{j=1}^{n} \sum_{k=1}^{j} 2^j \\ &= \sum_{k=1}^{n} \sum_{j=k}^{n} 2^j \\ &= \sum_{k=1}^{n} 2^{n+1} - 2^k \\ &= n \cdot (2^{n+1}) - (2^{n+1} - 2) \\ &= \Theta(n \cdot 2^n). \end{aligned}$$

- Similarly,

$$\begin{aligned} \sum_{j=1}^{n} \frac{j}{2^j} &= \sum_{j=1}^{n} \sum_{k=1}^{j} \frac{1}{2^j} \\ &= \sum_{k=1}^{n} \sum_{j=k}^{n} \frac{1}{2^j} \\ &= \sum_{k=1}^{n} \left(\frac{1}{2^{k-1}} - \frac{1}{2^n}\right) \\ &= 2 - \frac{n}{2^n} = \Theta(1). \end{aligned}$$

- Note that some parts of the computation may be wrong; but I have checked that asymptotic bound is indeed correct.
- log tricks: $\frac{1}{\log_a(b)} = \log_b(a)$, "base transfer" $\frac{\log_c(b)}{\log_c(a)} = \log_a(b)$ (proof can be done using "chain rule" as follows: $\log_a(b) * \log_c(a) = \log_c(b)$, and then homogenize), $a^{\log_a(b)} = b$.
- $\log(x!) = x\log(x)$ for any x. More specifically, **Stirling's** says that $n! \sim \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$.
- This can be applied in weird shit like $\log((\log(\log(n)))!)$ which is equal to $\log(\log(n)) * \log(\log(\log(n)))$.
- $2^{\log(\log(\log(n)))}$ has different complexity with $10^{\log(\log(\log(n)))}$. Use the third and second log trick, and we can deduce that the first one is $\log(\log(n))$ and second is $\text{first}^{\log_2(10)}$.
- Similarly, 2^{2n} and 2^n has different complexity. Don't take constants in powers (especially "after taking log") for granted.
- Letting $n = 2^k$ does wonders to weird log stuff, and taking log of both sides does wonders to weird power stuff.
- For algorithm-constructing questions, make sure you **read the question carefully** and understand wholly what the algorithm does (*at all its phases/stages.*) This paper's Q3 is honestly quite confusing and tedious.

Term test for Sem 2, AY20/21. This test is very technical/math-heavy, so I won't put too much here.

- For Q3, remember to explicitly **state base cases** for runtime analysis,
- and then also remember to provide **a valid construction** for the "decision tree classes"; i.e. configurations must satisfy the property that "the peak must be in the specified location".

Term test for Sem 2, AY18/19.

- Insertion sort: we can always compare the functions iteratively, and "insert" them to an existing total-order-of-functions. This reduces the need to "compare all functions with one another".
- **Be careful with asymptotic constants.** For example, to prove $(1+\cos(n\pi/2)) = \Theta(1)$, need to set $c \ge 2$, as left function will get arbitrarily close to 2. Cannot just say "works for any c".
- Weird phrasing of Q4(b). Read question multiple times and understand what its "best meaning is"; in this case, it's *expected number of rounds where score is positive, given that* ***they play an infinite round of games***.

September 2008 Test.

- My standard tips for MCQ test: **Remember that MCQs is created to facilitate learning**. Pick answer that is **most related** to others, and **try to avoid** "None of the Above" answers unless everything else seems either trap-py, plain wrong, or unrelated.
- **Searching** for a particular thing means not assuming the thing is in there (based on Q12 of this paper's answer key).
- Preorder traversal is **not** the reverse order of postorder traversal. Preorder is parent, left, right and postorder is left, right, parent.

Sample Midterm Review. 2 more materials "unique" to this problem set: adversarial argument and hashing.

- **Adversarial** means creating 2 inputs that produces **same output** given an (arbitrary) "iteration" of an algorithm. Note that the second input can be "dependent" on how the algorithm behaves, as "the algorithm is not supposed to know everything".
- Namely, we can fix input I, and we can adjust an *adversary* I' based on how the algorithm acts (deterministically) on I. So, in other words, we prove that the "degree of freedom" of the algorithm is "not enough".

- In Q2's case, we craft a nice I and set "$2n-1$ degrees of freedom", and we set I' to be based on a "degree of freedom" that the algorithm has not "touched". In this case, it is a pair of consecutive numbers.
- Hashing lingo: choose h from a family $\mathcal{H}$ of hash functions mapping $\mathcal{U}$ (the keys/input values) to $[M = \Theta(n)]$ (the storages/memory/buckets).
- *Insert* to hash table.
- Collisions (of expected $\Theta(1)$ occurences) to be handled via chaining (no need fancy stuff such as quadratic probing).
- Correctness is assured, as for a given sampled hash function, *everything from there is deterministic.*
- ***Note:*** if U is so large that $\log(U)$ does not fit in a machine word, then (all) operations cannot be done in constant time. In practice, such situations are rare, as memory is typically big enough to store *pointers to input elements.*

7 Amortized Analysis

Why need this?

- There is a sequence of n operations $\sigma_1, \sigma_2, \ldots, \sigma_n$.
- Let $t(i)$ be time complexity of σ_i.
- Let $T(n)$ time complexity of all n operations. $T(n) = nf(n)$ may be grossly wrong.

Examples: binary counter.

Amortized analysis. Strategy for analyzing a sequence of operations to show average cost per operation is small, even if a single operation within a sequence might be expensive.

Note: no probability involved! It guarantees *average-case* performance of each operation *in the worst case.*

There are 3 types of amortized analysis: aggregate, accounting, potential. Aggregate is just summing, and it lacks precision of the other two methods.

Accounting (Banker's) Method

- Charge ith operation a fictitious ***amortized cost*** $c(i)$, assuming \$1 pays for 1 unit of work (time) e.g., bit-flips in the previous example.
- This fee is consumed to perform the operation.
- Any amount not immediately consumed is stored in the ***bank*** for use by subsequent operations.
- The idea is *to* impose an extra charge on inexpensive operations and use it to pay for expensive operations later on.
- The bank balance ***must not go negative***!
- We must ensure that $\sum_{i=1}^{n} t(i) \le \sum_{i=1}^{n} c(i)$ for all n.
- Thus, **the total amortized costs provides an upper bound on the total true costs.**

Example: charge \$2 each time we set a bit $0 \to 1$, and \$ 1 is used to flip the bit while \$1 is stored in the bank (as the number of "1s" left in the number).

Potential Method

ϕ: Potential function associated with the algorithm/data-structure

$\phi(i)$: Potential at the end of ith operation

Important conditions to be fulfilled by ϕ
$\phi(0) = 0$
$\phi(i) \ge 0$ for all i

$\Delta\phi_i$ = Potential difference

Amortized cost of ith operation $\stackrel{\text{def}}{=}$ Actual cost of ith operation $+ (\phi(i) - \phi(i-1))$

Amortized cost of n operations $\stackrel{\text{def}}{=} \sum_i$ Amortized cost of ith operation
$=$ Actual cost of n operations $+ (\phi(n) - \phi(0))$
$=$ Actual cost of n operations $+ \phi(n)$
$\ge$ Actual cost of n operations

If we want to show *actual cost* of n operations is $O(g(n))$ then it suffices to show that *amortized cost* of n operations is $O(g(n))$.

8 DP and Greedy

Honestly just read the lecture notes for this one. Very logic heavy, especially the LCS DP recursive algo.

The theory of DP algorithm paradigm is "just":

- Expressing solution recursively (i.e. define base cases clearly!)
- Overall there are only small (e.g. polynomial) number of subproblems, but there is a huge overlap among the subproblems.

Then, we compute recursive solution iteratively in a bottom-up fashion.

KNAPSACK Problem. Formulation is input: $(w_1, v_1), (w_2, v_2), \ldots, (w_n, v_n)$ and W, and output: subset $S \in \{1, 2, \ldots, n\}$ that maximizes

$$\sum_{i \in S} v_i, \text{such that } \sum_{i \in S} w_i \le W.$$

Divide into two cases, item n (last one) is taken and item n (last one) is not taken.

Note: no probability involved! It guarantees *average-case* performance of each operation *in the worst case.*

Greedy Algorithms. A very general technique, to recast the problem so that *only one subproblem* needs to be solved at each step. Beats DnC **when it works**.

FRACTIONAL-KNAPSACK Problem. Same input, but output are weights $x_1, \ldots, x_n$ that maximize $\sum_i v_i \cdot \frac{x_i}{w_i}$ subject to:

$$\sum_i x_i \le W, \text{and } 0 \le x_j \le w_j \forall j \in [n].$$

It has **optimal substructure** property (cut-and-paste argument): if we remove w kgs of one item j from the optimal knapsack, then the remaining load must be the optimal knapsack weighing at most $W - w$ kgs that one can take from the $n-1$ original items and $w_i - w$ kgs of item j.

It also has the **greedy-choice** property: let j^* be item (i.e. item j^* has index j) with the maximum value/kg, v_j/w_j. Then, exists an optimal knapsack containing $\min(w_{j^*}, W)$ kgs of item j^*.

Huffman Encoding. Mostly just read lecture notes, but there are two main ideas:

First is

- $A = a_1, \ldots, a_n$ n alphabets in non-decreasing order of frequencies,
- $A' = a_3, \ldots, a', \ldots, a_n$ be $n-1$ alphabets in non-decreasing order of frequencies with $f(a') = f(a_1) + f(a_2)$

The relation between OPTIMAL AVERAGE-BIT-LENGTH of A and A' is that A is A' plus $f(a_1)$ plus $f(a_2)$. If this relation is true, we have an algo for optimal prefix codes.

The second idea is the algorithm itself (implementation):

The algorithm based on $OPT_{ABL}(A) = OPT_{ABL}(A') + f(a_1) + f(a_2)$

```
OPT(A)
{   If |A|=2, return 0 1 (a1 a2) ;
    else
    {  Let a1 and a2 be the two alphabets with least frequencies.
       Remove a1 and a2 from A;
       Create a new alphabet a';
       f(a') ← f(a1) + f(a2);
       Insert a' into A;
       T ← OPT(A);
       Replace node a' in T by 0 1 (a1 a2) ;
       return T ;
    }
}
```

Build a heap for the alphabets with frequencies as key takes $O(n \log n)$ time

Perform **Remove** and **Insert** in $O(\log n)$ time

Overall time = $O(n \log n)$

Additional Details. ABL per symbol using encoding γ (and alphabet set A) is

$$\text{ABL}(\gamma) = \sum_{x \in A} f(x)|\gamma(x)|.$$

Prefix coding. Coding $\gamma(A)$ prefix coding if there does not exist $x, y \in A$ s,t, $\gamma(x)$ prefix $\gamma(y)$.

Algorithmic problem: γ is prefix and ABL(γ) minimum (given A alphabet sets and frequencies).

Labeled binary tree. Leaf notes $\leftrightarrow$ alphabets. Code of alphabet is label of path from root.

Most important observation(s).

- Intuitively, more frequent alphabets should be closer to the root (and conversely, less freq alphabets should further from the root).
- Observation about local structure in the tree: if $a_1 \le a_i$ and a_i is on deepest level, swapping them cannot increase ABL (equality happens iff a_1 is also at the deepest level).
- Do the same for a_1's sibling; switch it with a_2.
- Important observation: *exists* (not always, but 1 of optimal encoding(s)) in which a_1, a_2 appear as siblings. We just need to focus on that binary tree (of optimal prefix coding) which a_1, a_2 are siblings.

9 Reductions and NP-completeness

The main idea of reductions.

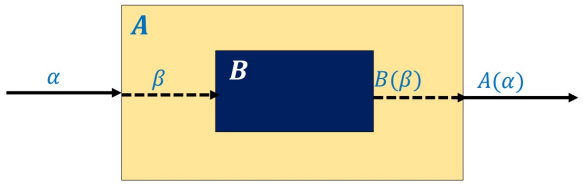
- Using another (computational) problems, as a paradigm in algo **design**.
- How do we solve problems? Using a solution of some old problem, applied at a "new"problem.
- Another way to view it is to give a way to compare the **hardness** of two problems.

What are reductions? Consider two problems A and B. A can be solved as follows: *Input:* An instance (another word for "input") α of A is to be

- Convert α to an instance β of B,
- Solve β (obtain a solution for β),
- Based on β's solution, obtain the solution of α.

This way, we say A **reduces to** B.

What is a reduction?



Example is *longest palindromic subsequence* in a string reduces to finding *longest common subsequence* of a pair of strings. In particular, I think of LCS as a more generalized problem of LPS (hence, LPS is more specific).

Another example: MAT-MUL and MAT-SQR reduces to each other. Another example, T-SUM reduces to 0-SUM (and obviously, 0-SUM can be reduced to T-SUM).

$p(n)$-time reduction. Consider two problems A, B. If for any instance α of A of (input) size n:

- An instance β for B can be constructed in $p(n)$ time,
- and a solution to input α of A can be constructed/recovered from a solution of input β of B in $p(n)$ time,

then we say there is a $p(n)$-time reduction from A to B.

Example: LPS of x is an LCS of x and reverse(x), with an $O(n)$ time reduction.

Running Time.

Running Time Composition

Claim: If
- there is a $p(n)$-time reduction from problem ***A*** **to problem** ***B***, and
- there exists a $T(n)$**-time algorithm to solve problem** ***B*** on instances of **size** n,

then there is a $\left(T(p(n)) + O(p(n))\right)$ time algorithm to solve problem A on instances of **size** n.

Implications of Poly-Time reduction. We define a notation for A poly-time reduces to B by $A \leq_P B$.

- If B has poly-time algo, then so does A!
- Or, if B is "easily solvable", then so is A!
- Its contrapositive: if A is "hard", then so is B!
- Intuition: A is a special case of B.

A note on encoding.

- Polynomial in the *length of the encoding of the problem instance.*
- For many problems, use "standard" encoding; binary encoding of integers; for mathematical objects (graphs, matrices, etc): list of parameters enclosed in braces, separated by commas.
- An algorithm that runs in time *polynomial in the numeric value* of the input but is *exponential in the length of the input* is called a **pseudo-polynomial** time algorithm.
- Example, KNAPSACK has pseudo-poly time algo, FRACTIONAL-KNAPSACK has a poly time algo.

Intractability and Decision Problems. Need a framework to talk about all problems in the same language.

A decision problem is a function that maps an instance space I to the solution set {YES, NO}.

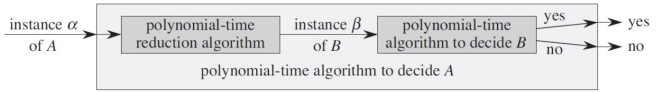
Note that **decision reduces to optimization**, i.e. decision problem is no harder than the optimization problem.

This is because given the value of the optimal solution, simply check whether it is $\leq k$. This implies that if cannot solve decision problem quickly, cannot solve optimization problem quickly!

Reductions between Decision Problems (Karp Reduction). Given two decision problems A, B, define $A \leq_P B$ by a transformation from instances α of A to instances β of B s.t.

- α is a YES-instance for A **iff** β is a YES-instance for B, and
- transformation takes poly-time in the size of α.

Reductions



Suffices to show:
- Reduction runs in polynomial time
- If α is a YES-instance of A, β is a YES-instance of B
- If β is a YES-instance of B, α is a YES-instance of A

Example: $VC \leq_P IS$, and the converse is true, too.

NP. Is a class of problem, dates back to the 1960's. It is the set of all **decision** problems which have **efficient certifier** ("non-deterministic polynomial time").

NP-complete. A problem X in NP class is NP-complete if for every $A \in$ NP, $A \leq_P X$.

Does any such problem exist?

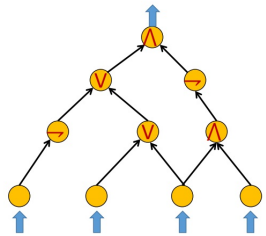
Does any NP-complete problem exist ?

Circuit satisfiability problem:
[Cook and Levin , 1971]

A DAG with nodes corresponding to **AND,NOT,OR** gates and n binary inputs,

does there exist any binary input which gives output 1 ?

Why is in NP?
Certificate is an binary input that gives output 1



Note that Circuit Satisfiability reduces to CNF-SAT, and it reduces to 3-SAT. See lecture slides for problem descriptions.

How to show a problem to be NP-complete? Let X be such a problem (we wish to show X NP-complete).

- Show $X \in$ NP.
- Pick problem A already known to be NP-complete.
- Show $A \leq_P X$.

Also, see lec slides to see problems in NP but *believed* to not be NP-complete.

Extra Notes. List of NP-complete problem formulations.

- The travelling salesman problem (also called the travelling salesperson problem or TSP) asks the following question: "Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city?"
- In theoretical computer science, the circuit satisfiability problem (also known as CIRCUIT-SAT, CircuitSAT, CSAT, etc.) is the decision problem of determining whether a given

Boolean circuit has an assignment of its inputs that makes the output true. In other words, it asks whether the inputs to a given Boolean circuit can be consistently set to 1 or 0 such that the circuit outputs 1. If that is the case, the circuit is called satisfiable. Otherwise, the circuit is called unsatisfiable.

- In the maximum clique problem, the input is an undirected graph, and the output is a maximum clique in the graph. If there are multiple maximum cliques, one of them may be chosen arbitrarily.
- Vertex cover problem: Given a graph of n nodes and m edges, find its minimum size vertex cover. Vertex Cover: The vertex Cover of a graph is defined as a subset of its vertices, such for every edge in the graph, from vertex u to v, at least one of them must be a part of the vertex cover set.