Finale

S. Halim YJ. Chang

School of Computing
National University of Singapore

CS3230 Lec13; Tue, 12 Nov 2024

Overview

Recap (plus a few extensions)

- L01+02: Asymptotics and Recurrences
- L03a: Proof of Correctness
- L03b+04a: Divide and Conquer (D&C)
- L04b+05: Randomized Algorithms
- L06+07: Dynamic Programming (DP) vs Greedy Algorithm
- L08: Amortized Analysis
- L09+10: Problem Reduction and NP-completeness
- L04+12: Lower Bound vs Linear-Time Sorting and Selection

Wrapping-Up

Recap: Asymptotics: $O, \Omega, \Theta, o, \omega$

We say	if $\exists c, c_1, c_2, n_0 > 0$ such that $\forall n \geq n_0$
$f(n) \in O(g(n))$	$0 \leq f(n) \leq c \cdot g(n)$
$f(n) \in \Omega(g(n))$	$0 \leq c \cdot g(n) \leq f(n)$
$f(n) \in \Theta(g(n))$	$0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$

We say	$\forall c > 0$, $\exists n_0 > 0$ such that $\forall n \geq n_0$
$f(n) \in o(g(n))$	$0 \leq f(n) < c \cdot g(n)$
$f(n) \in \omega(g(n))$	$0 \leq c \cdot g(n) < f(n)$

Easiest with just these **two limit tests**:

- $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \Rightarrow f(n) \in o(g(n))$
- $0 < \lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty \Rightarrow f(n) \in \Theta(g(n))$

Asymptotics exercise

Assumption: Sort n integers in range $[0..10^6]$.

- Give a sorting algorithm that runs in $\Theta(n^2)$.
- Give a sorting algorithm that runs in $\omega(n^2)$ and $o(n^3)$.
- Give a sorting algorithm that runs in $\Omega(n \log n)$ and $o(n^{1.5})$.
- Give a sorting algorithm that runs in $O(n)$.
- Give a sorting algorithm that runs in $o(n)$.

Recap: Recurrences (1)

Given $T(n) = a \cdot T(\frac{n}{b}) + f(n)$, derive a **tight** asymptotic bound.

There are a few ways to solve recurrences.
Master theorem is the easiest: Let $d = \log_b a$.

1. Case 1: $f(n) \in O(n^{d-\epsilon}) \Rightarrow T(n) \in \Theta(n^d)$.
2. Case 2: $f(n) \in \Theta(n^d \log^k n) \Rightarrow T(n) \in \Theta(n^d \log^{k+1} n)$.
3. Case 3: $f(n) \in \Omega(n^{d+\epsilon}) \Rightarrow T(n) \in \Theta(f(n))$.
   (usually regular, e.g., on $f(n)$ in this format: $c \cdot n^d \log^k n$).

Given a recurrence $T(n)$ that is amenable to it,
use **Telescoping** to derive a **tight** asymptotic bound.

Recap: Recurrences (2)

When the given recurrence $T(n)$ is 'non-standard',
we may need to use one of these two more general techniques:

We can draw the **Recursion Tree** (manually in final assessment),
e.g., by using `https://visualgo.net/en/recursion`,
and then analyze the height of the recursion tree and
the works done in each level in terms of n.
Depending on the shape of the recursion tree,
it is probably easier to show O and/or Ω separately.

We can also use **Substitution method**, where we 'guess' the answer,
then use proof by induction on the base case and inductive case.
We have to show O and/or Ω separately.
This requires intuition (to guess 'somewhat accurately') and
also perseverance (to try several times until 'the math works').

Question 1 at VisuAlgo Online Quiz

Given $T(n) = T(\frac{n}{3}) + T(\frac{2n}{3}) + n$, analyze its asymptotic bound.
PS: This is the median of medians selection with **groups of** 3.

A). $T(n) \in \Theta(n)$

B). $T(n) \in \Omega(n \log n)$

Recap: Proof of Correctness

Two types:

- For iterative algorithm, we usually use **loop invariant**.
  A condition which is TRUE at the start of EVERY iteration
  We can then use invariant to show the correctness:
  1. Initialization: The invariant is True before iteration 1
  2. Maintenance: True at iteration $x \implies$ True for iteration $x+1$
  3. Termination: True when the algorithm ends
- For recursive algorithm, we usually use **proof by induction**.
  1. Show the recursion is (trivially) correct on its base case(s).
  2. Inductive step: show that the recursive algorithm is correct, assuming that the smaller cases are all correct.

Revisiting the SS Shortest Paths (SSSP) Problem

See `https://visualgo.net/en/sssp?slide=1` for definition, and then go to Exploration mode to (re-)try Dijkstra's algorithm.

Earlier in lec03a, we have shown that (the iterative) Dijkstra's algorithm is correct using loop invariant.

We have also shown that Dijkstra's algorithm can be implemented in $O((|V| + |E|) \log |V|)$ if the Priority Queue is implemented with **Binary** Heap, or in asymptotically faster $O(|E| + |V| \log |V|)$ if the Priority Queue is implemented with **Fibonacci** Heap (as $|E| \geq |V|$ in most graphs).

Question: How crucial is the non-negative weights for Dijkstra's?

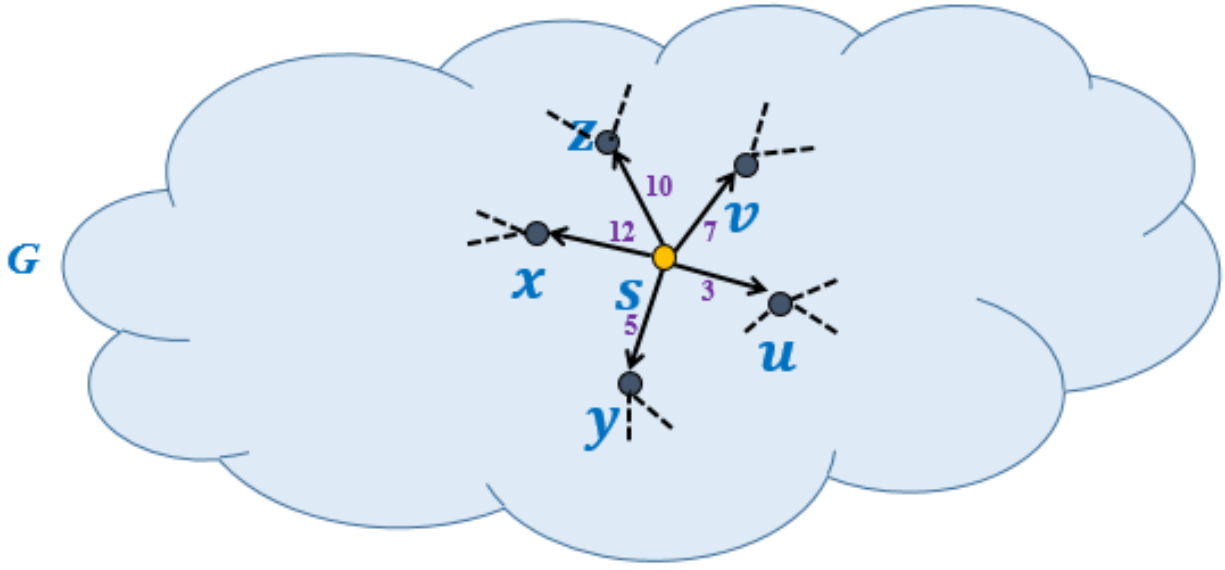
What if the edge weights are potentially negative?

There are two crucial facts that are exploited in Dijkstra's:

1. The nearest neighbor is also the vertex **nearest** to s.
2. The **optimal subpath** property.

Question: Which of these two facts get **violated** if edge weights are potentially negative?

Violation of Fact 1



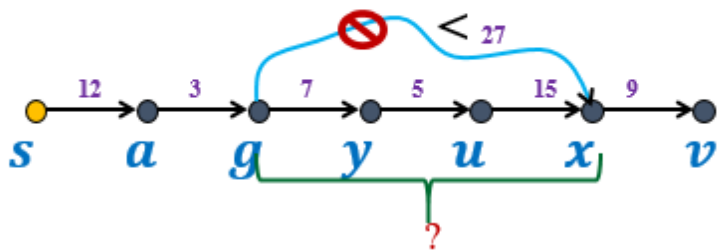
Question: Can we be certain about the shortest path distances from source s to nearest neighbor u (edge weights can be -ve)?

Violation of Fact 2 - Part 1

Consider any shortest path $P(s, v)$.

Lemma: Every **subpath** of a shortest path is also a shortest path.

$\delta(s, v) = 51$

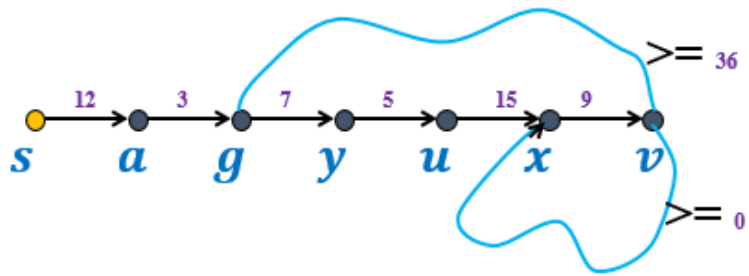


Can you prove this lemma?

Violation of Fact 2 - Part (2)

On non-negative weight graph,
the shortest path from s to x **cannot pass through** v.

$\delta(s, v) = 51$

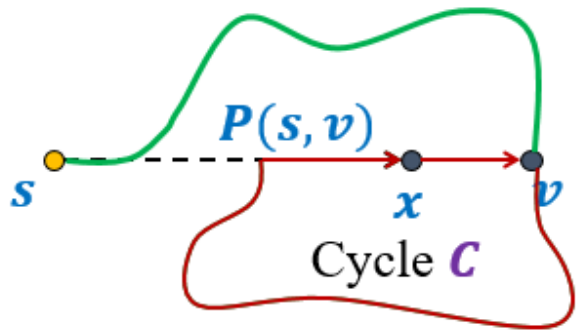


Can you prove this?

Now what if the edge weight can potentially has negative weight,
e.g., what if edge $v \to x$ above has weight ≤ -10?

Violation of Fact 2 - Part (3)

Question: When can the shortest $P(s, x)$ passes through v?
Hint: This diagram.



So, what are the implications?

Recap: D&C

Here are the usual steps for using **Divide** and Conquer (D&C) problem solving paradigm for problems that are amenable to it:

1. **Divide**: Divide/break the original problem into $a \geq 1$ smaller sub-problems of size $\frac{n}{b}$ (preferred) or $n - c$ (not preferred).
   PS: If $a = 1$, some people call this **Decrease** and Conquer, still with the same D&C abbreviation.
2. **Conquer**: Conquer/solve the sub-problems recursively.
3. **Combine**: Optionally, for $a > 1$, combine the sub-problem solutions to get the solution of the original problem.

Common algorithms: Binary Search, Merge Sort, Exponentiation (by doubling), Strassen's, Karatsuba's (tut04), etc.

Recap: Randomized Algorithms

Techniques:	linearity of expectations ($E[X+Y] = E[X] + E[Y]$), indicator random variables (very useful), union bound ($Pr[A \cup B] \leq Pr[A] + Pr[B]$), Markov inequality, principle of deferred decision, amplification of success probability.
Algorithms:	Freivalds' algorithm (Monte Carlo), (Randomized) Quick Sort (Las Vegas).
Balls&Bins:	coupon collector (probability of no empty bin), chain hashing (expected bin size).

There is a analysis of expected linear-time Quickselect in tut11.

Recap: Dynamic Programming (DP)

Key points:

- Expressing the solution recursively.
- There are only small (e.g., polynomial) number of subproblems.
- But there is a huge overlap among the subproblems.
  So the recursive algorithm may take exponential time.
  (solving the same subproblem multiple times).
- So we compute the recursion with memoization (top-down), or iteratively in a bottom-up fashion.
  This avoids wastage of computation,
  and leads to an efficient implementation.

DP Tips

The key to design a DP solution is in figuring out the **optimal substructure**, i.e., how to find the optimal solution to your problem from the optimal solution(s) to one or more subproblems.

Figure out how to index your subproblems. For example, if you have a sequence $a_0, a_1, \ldots, a_{n-1}$, trying the i-th subproblem being the answer for the sequence from $[0..i]$ is a good place to start. If there are integers in the problem upper bounded by some integer M, you probably need to index over every integer in $[0..M]$.

See if any of those subproblems overlap
(showing at least one instance of overlap case is sufficient).

If you have (a lot of) time to practice,
you can use this URL (around 450+ DP exercises).

Introducing Bellman-Ford DP algorithm

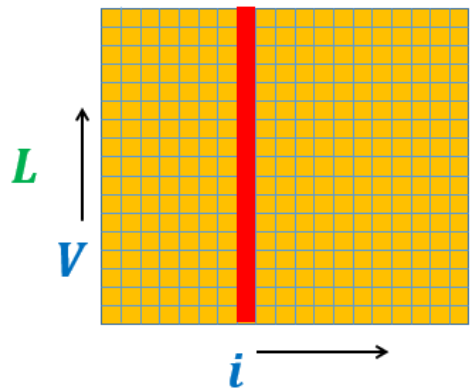
Consider once again, a shortest path $P(s, v)$.

Question: If $P(s, v)$ has i edges and (x, v) is the last edge, what can we infer about the subpath $P(s, x)$?
PS: No negative weight cycle!!

Recursive formulation for $L(v, i)$

Let $L(v, i)$ = weight of the shortest path $P(s, v)$ with $\leq i$ edges.

We aim to compute $L(v, n-1), \forall v \in V$, why?



There are two cases:

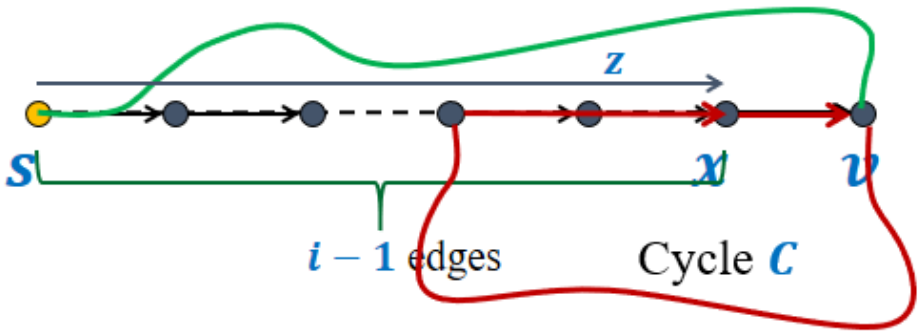
1. For $L(v, < i)$, then $L(v, i) = L(v, i-1)$.
2. $L(v, i) = \min_{(x,v) \in E} L(x, i-1) + \omega(x, v)$.

Note: Case 2 assumes optimal substructure property of $L(v, i)$.

Prove the optimal substructure of $L(v, i)$

Proof by contradiction: Suppose the path corresponding to $L(x, i-1)$ passes through v, but we have $L(x, i-1) < z$.

Question: What can we say about $L(x, i-1)$ in this case?
Hint: This similar diagram and non-negative edge weights.



Bellman-Ford implementation

```
let L[v, 0] = inf for all v in V, except L[s, 0] = 0
for i = 1 to n-1 do
  for each v in V do // the next two loops sum to |E|
    L[v, i] = L[v, i-1] // at least from column i-1
    for each (x, v) in E do
      L[v, i] = min(L[v, i], L[x, i-1] + w(x, v))
```

The cell $L(v, i)$ in our 2-D DP table stores the weight of the shortest path from s to v with $\leq i$ edges. Proof: It is the implementation of the recurrence earlier.

Theorem: Given a directed graph $G = (V, E)$ with $\omega : E \to R$ and $s \in V$, **if there is no negative cycle**, then we can compute the shortest paths from s to all vertices reachable from s in G in $O(|V| \cdot |E|)$ time and using $O(|V|^2)$ space.

Bellman-Ford (real) implementation

We can reduce the space complexity to $O(|V|)$,
by realizing that we only need the last column at each iteration.

```
let L[v] = inf for all v in V, except L[s] = 0
for i = 1 to n-1 do
  // for each v in V do // the next two loops sum to |E|
    // L[v, i] = L[v, i-1] is now implied
    // for each (x, v) in E do
      // L[v] = min(L[v], L[x] + w(x, v))
  for each edge (u, v) in E do // clearly |E|
    L[v] = min(L[v], L[u] + w(u, v))
```

See this algorithm live at `https://visualgo.net/en/sssp`.

Other remarks of SSSP

There is a further improvement to Bellman-Ford algorithm, called the Bellman-Ford-Moore algorithm.

The **Single-Source** Shortest Paths (SSSP) problem and the SSSP algorithms that have been presented so far (e.g., Dijkstra's (Greedy) algorithm for graphs with non-negative weights and Bellman-Ford (DP) algorithm for graphs with no negative weight cycle) only works if there is only one source vertex.

There is a variant of Shortest Paths problem that considers shortest paths between all pairs of vertices in a directed weighted graph. This variant is called the **All-Pairs** Shortest Paths (APSP) problem. An algorithm to solve APSP is $O(|V|^3)$ Floyd-Warshall algorithm, which interestingly, is also a DP algorithm.

Recap: Greedy Algorithm Paradigm

1. Cast the problem where we have to make a **greedy choice** and are left with **one subproblem** to solve.
2. Prove, via exchange argument, that there is always an optimal solution to the original problem that makes the greedy choice, so the greedy choice is safe.
3. Use **optimal substructure** to show that we can combine an optimal solution to the subproblem with the greedy choice to get an optimal solution to the original problem.

Revisiting the Min Spanning Tree (MST) Problem

See `https://visualgo.net/en/mst?slide=1` for definition, and then go to Exploration mode to try Prim's algorithm.

For the analysis, we assume that all edge weights are distinct.

Optimal substructure of MST

Use any MST T shown by VisuAlgo on any valid test case.

Given an MST T of a graph $G = (V, E)$, if we remove any edge $(u, v) \in T$, then T is partitioned into two subtrees T_1 and T_2.

Theorem: The subtree T_1 is also an MST of $G_1 = (V_1, E_1)$, the subgraph of G induced by the vertices of T_1 where V_1 is the set of vertices of T_1 and $E_1 = \{(x, y) \in E : x, y \in V1\}$.
Note: We can define similarly for T_2.
Can you prove this?

Use DP?

We can use Dynamic Programming (DP)...

The DP algorithm would search for which edge (u, v) to add, and then recurse on the two subproblems T_1 and T_2 (avoiding overlapping subproblems).
Can you show the overlapping subproblems?

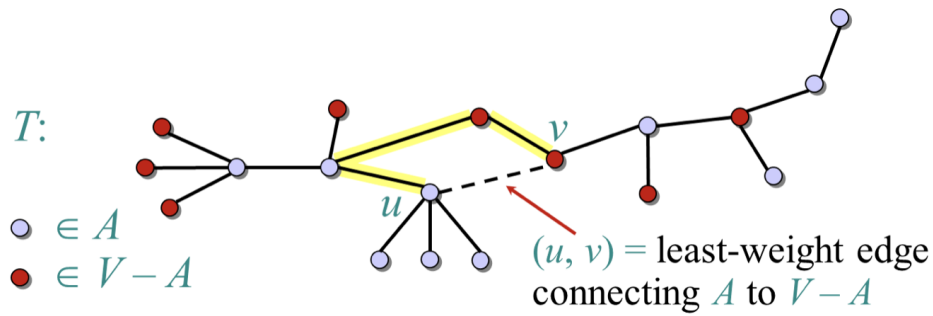
But should we use DP?

Hallmark for 'greedy' algorithms

Greedy-choice: A locally optimal choice is globally optimal.

Theorem: Let T be a MST of $G = (V, E)$, and let A be any subset of vertices. Suppose that $(u, v) \in E$ is the **least-weight** edge connecting A to $V \setminus A$. Then, $(u, v) \in T$.

Proof: Suppose $(u, v) \notin T$. We can use exchange argument.



Suppose there is an MST T that does not use (u, v). Consider the unique simple path from u to v in T. If we swap (u, v) with the first edge on this path that connects a vertex in A to a vertex in $V \setminus A$, a lighter weight (as all edge weights are distinct) spanning tree than T results. Contradiction.

Prim's algorithm implementation – with Decrease-Key

We can now implement this greedy algorithm.
Idea: Maintain $V \setminus A$ as a Priority Queue PQ.
We key each **vertex** in PQ with the weight of the **least weight** edge that connects that vertex to a vertex in A.

```
PQ = V
key[v] = inf for all v in V, except key[s] = 0
while PQ != empty
  u = PQ.extractMin()
  for each v in AL[u] // use Adjacency List
    if v in PQ and w(u, v) < key[v]
      key[v] = w(u, v) // Decrease-Key of PQ <- tedious
      pi[v] = u // the MST
```

Unless you have a ready PQ implementation with **Decrease-Key**, it is rather tedious to use Prim's implementation.

Prim's algorithm implementation – with lazy deletion

Idea: Maintain $V \setminus A$ as a Priority Queue PQ **with lazy deletion**.
We key each **edge** in PQ with the weight of the edge that connects a vertex u in A to a vertex in $V \setminus A$.
The least weight edge will be nearer to the front of PQ than the other heavier edges (that will cause cycle if added).

```
PQ.enqueue((w,s,u) for all edge (s,u) with weight w)
vis[u] = False for all u in V, except vis[s] = True
while PQ != empty
  w,u,v = PQ.extractMin()
  if vis[u] continue // (w,u,v) was an old info
  vis[u] = True
  pi[v] = u // the MST
  for each v in AL[u] // use Adjacency List
    if not vis[v]
      PQ.enqueue((w,u,v)) // no Decrease-Key!!
```

Example run of Prim's algorithm

Click here to open VisuAlgo MST page to see a sample run of Prim's algorithm implementation – with lazy deletion – on a specific test case of connected undirected **distinct**-weighted graph.

Analysis of Prim's algorithm

```
PQ.enqueue((w,s,u) for all edge (s,u) with weight w)
vis[u] = False for all u in V, except vis[s] = True
while PQ != empty
  w,u,v = PQ.extractMin() // |E| times, each O(log |E|)
  if vis[u] continue // (w,u,v) was an old info
  vis[u] = True // each vertex is processed once
  pi[v] = u
  for each v in AL[u]     // overall |E| times
    if not vis[v]
      PQ.enqueue((w,u,v)) // each O(log |E|)
```

In this Prim's implementation, each edge is enqueued to PQ once.
Once an edge is extracted out from PQ, it will never re-enter PQ.
Each enqueue and extractMin cost up to $O(\log |E|)$.
In simple graph, $|E| \in O(|V|^2)$, so $O(\log |E|) = O(\log |V|)$.
Overall, $O(|E| \log |V|)$.

Other remarks of MST

$O(|E| \log |V|)$ time complexity is Prim's algorithm implementation using Binary Min Heap Priority Queue and with lazy deletion.

As with Dijkstra's, Prim's can run in $O(|E| + |V| \log |V|)$, if the Priority Queue is implemented with Fibonacci heap... But this data structure is more complicated than Binary Heap.

There are other MST algorithms, e.g., Kruskal's (in VisuAlgo) and Boruvka's (not in VisuAlgo). Both run in $O(|E| \log |V|)$.

But, the best algorithm to date is Karger, Klein, and Tarjan [1993]. It is a **randomized** algorithm with expected $O(|V| + |E|)$ time, i.e., linear in terms of $|V|$ and $|E|$, which extends Boruvka's algorithm.

Which one to use?

Heuristic design steps for solving optimization problem:

1. Identify the optimal substructure (common first step)
   Then, formulate the recurrence (the recursive equation)
2. If no overlapping substructure, just run the recursion verbatim
   e.g., Binary Search.
   otherwise, run **Dynamic Programming**.
   e.g., 0/1-Knapsack, Coin Change (tricky cases).
3. Try to find some greedy choice (or local optimal choice).
   Use exchange argument to see if it leads to a global optimal.
   If yes, give the faster and usually simpler **Greedy Algorithm**.
   e.g., Frac Knapsack, Coin Change (on some denominations).
   otherwise, stay with step 2 decision earlier.

Recap: Amortized Analysis

There are three common amortization arguments:

1. **Aggregate** method (less flexible for mixed operations)
   Steps: Compute overall cost, divide by n operations
2. **Accounting** (or Banker's) method
   Heuristics: save on frequent but cheap operations
   Use your savings on rare but expensive operations
3. **Potential** method
   Heuristics: Carefully observe the costly operation
   Is there some quantity that is 'decreasing'?

Which analysis method to choose/learn/focus on?

For some papers, we tell you to use one (to simplify grading).

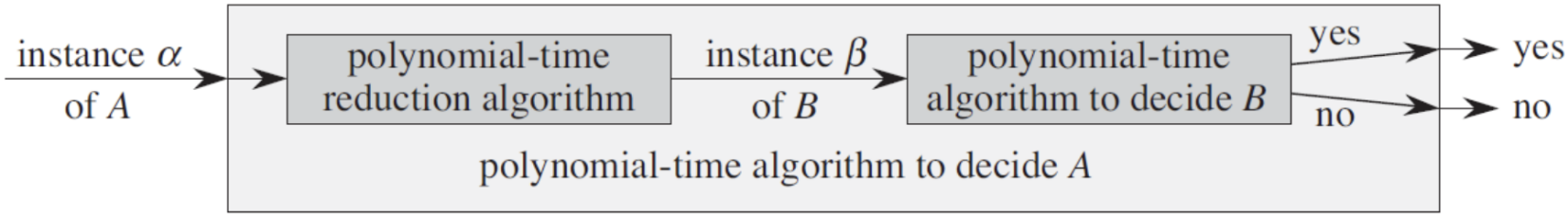
For some other papers, use the following heuristics:

Aggregate method is the easiest to use, if the analysis involves just one operation (e.g., Binary Counter with only increment).
But if there are two or more conflicting operations (insert vs delete; enqueue vs dequeue; push vs pop, etc), use the other two.

Between accounting versus potential method, they are $\approx$ equal.
Therefore, use one that you are more confident with.

Practice on one very recent past paper task in tut11!

Recap: Problem Reduction



We call this **polynomial time reduction** (or Karp's reduction) if both sub-functions above (translate A to B and translate B to A) runs in polynomial time, and we denote this process as $A \leq_p B$.

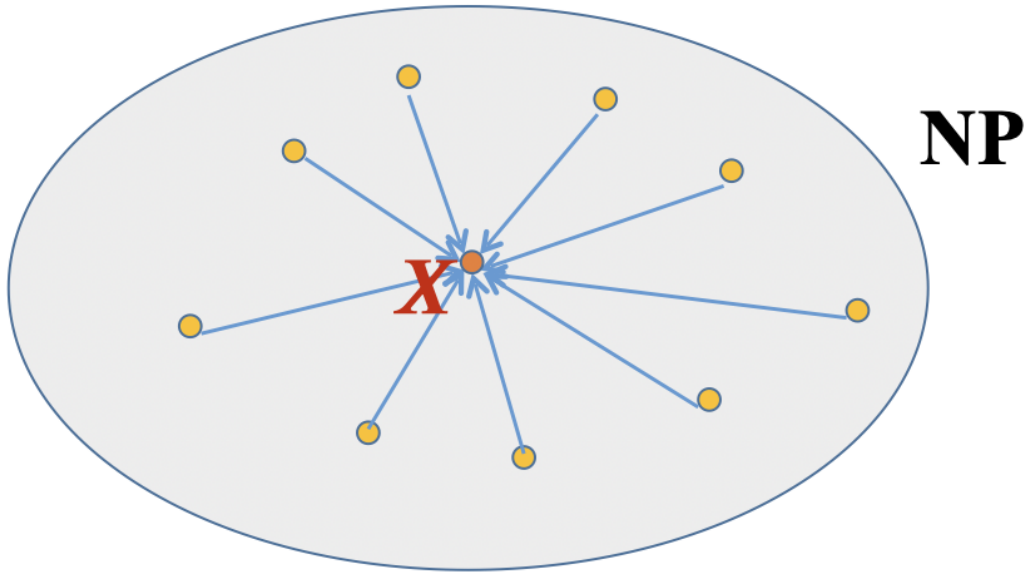
If B has a polynomial time algorithm, then so does A.

If A is 'hard', then so is B.

Recap: P, NP, NP-hard, NP-complete

In CS3230, we learn the following classes of problems:

- **P**: ... can be **solved** in polynomial time
- **NP**: ... can be **verified** in polynomial time
- NP-**hard**: ... can be **p-time reduced from all** problems in NP

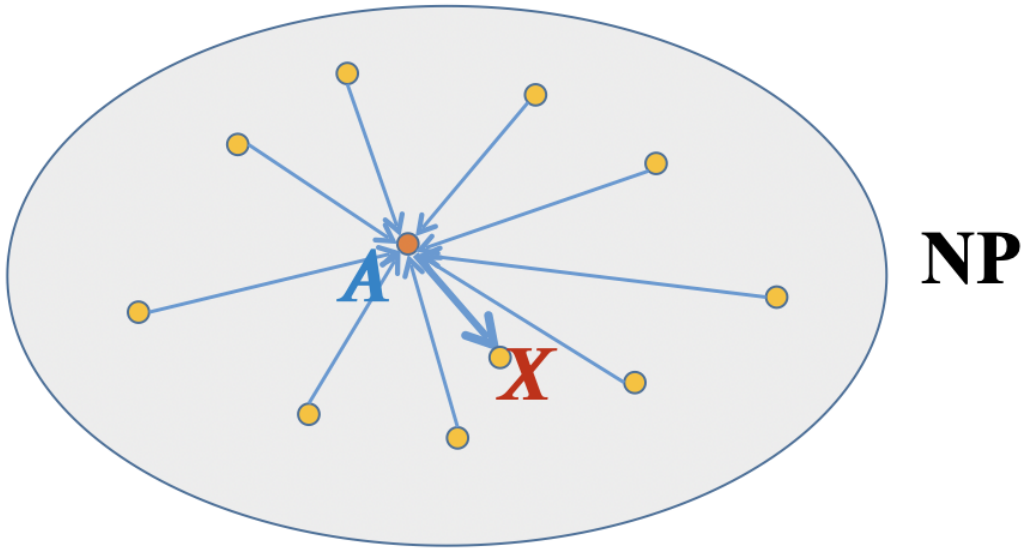


- NP-**complete**: ... is **both** in NP and is NP-hard

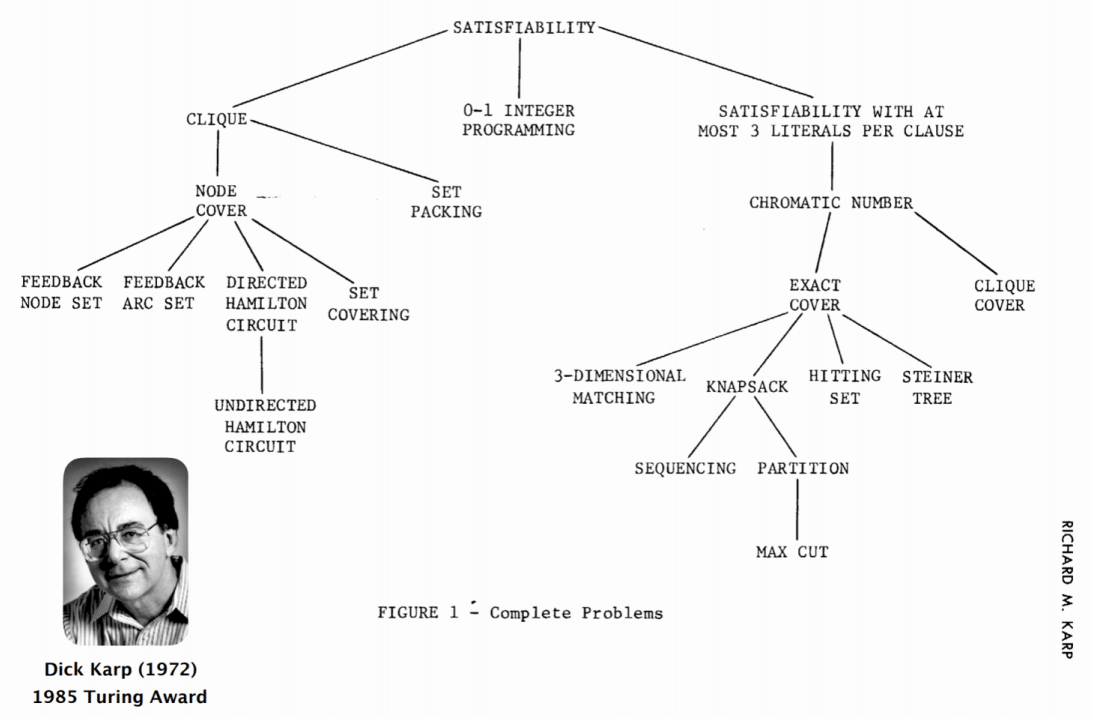
Proving NP-completeness

To prove SOMETHING is NP-complete, we need to show that:

1. Prove SOMETHING is in NP
   Verify the 'Yes' instance in polynomial time via a certificate.
   State the certificate,
   then show it verifies the 'Yes' instance in polynomial time.
2. Prove SOMETHING is NP-hard
   Show that it is as hard as any pre-existing NP-hard problem
   Show that A-PROVEN-NP-HARD-PROBLEM $\leq_p$ SOMETHING



The classic NP-complete reductions



Dick Karp (1972)
1985 Turing Award

Partially digitized at `https://visualgo.net/en/reductions`.
The compendium will grow over time :).

If you did WA3, task 1 was the NP-complete proof from last sem.

Recap: Linear-Time Sorting

Back in lec04a, we have used Decision Tree model to show that any comparison-based sorting requires $\Omega(n \log n)$ to sort n elements by comparing them.

However, we can break this lower bound if we do not compare elements (terms and conditions apply). We learn two algorithms:

1. Counting Sort, simple and stable version
   Runs in $O(n + k)$ on small range of integers in $[0..k-1]$
2. Radix Sort, iterated stable Counting sort
   from Least Significant Digit to Most Significant Digit
   Runs in $O(d \cdot (n + k))$ if there are d digits in range $[0..k-1]$
   Runs in $O(\frac{b}{r} \cdot (n + 2^r))$ on b-bit integers in base 2^r
   Actual time complexity depends on the setup

Recap: Linear-Time Selection

We also learn Selection (Order Statistics) problem: Given an unsorted array, find the i-th smallest element in that array.

Assuming that we can only compare, i.e., we cannot use Counting and/or Radix sort, we can use:

1. The 'median of medians' selection algorithm that spend some computation time just to find a 'good pivot' x.
   By partitioning the array around x, we can design a Divide and Conquer algorithm that It runs in worst-case linear-time.
   But, this algorithm is not practical in real-life.
2. The `Quickselect` algorithm.
   We can partition the array around a random pivot.
   It runs in expected linear-time and is much practical.

You will spend some time in tut11 to analyze `Quickselect`.

Other remarks of Selection Problem

The worst-case and the expected linear-time selection algorithms are rarely used per se (but they are very good problems to be used in Design and Analysis algorithm course like this).

In practice, selection is rarely asked just once (i.e., on multiple queries) and the underlying array is unlikely to remain static (i.e., data added/deleted/updated). On such Dynamic Order Statistics problem, we probably need augmented balanced Binary Search Tree (bBST), sometimes called as the Order Statistics Tree (OST).

But take note that to build the bBST of n elements itself, we already need $\Omega(n \log n)$.

The CS3230 course ends here...

Thanks a lot for

- Giving us an opportunity to share
  the joy of algorithm design and analysis.
- For sparing your precious time listening to us.
  (especially the (regular) live attendees).

We look forward to your **official** feedback/criticism in the NUS **official student feedback** (the non-official ones at NUSmods (or NUSwhispers) are generally read with a grain of salt).

The teaching team of S1 AY24/25 will share what work and what did not work to our S2 AY24/25 colleagues, so that the quality of the course will gradually improve over time.

ADS

Some recommended courses for interested students:

- CS3233 Competitive Programming
  PS: needs minimum CodeForces rating of ≥ 1400.
- CS4231 Parallel and Distributed Algorithms
- CS4234 Optimisation Algorithms
- CS5230 Computational Complexity
- CS5234 Algorithms at Scale
- CS5330 Randomized Algorithms

We hope that we have prepared you for these courses.

Credits

Special thanks to all the TAs of S1 AY24/25, namely:

1. Dominic Berzin Chua
2. Kale Aprup Vinay
3. Tan Wei Seng
4. Stanve Avrilium Widjaja
5. Bryan Chan Kah Hoe
6. Prof Chang Yi-Jun
7. Chen Yanyu
8. NUSOne (could be grp 08)
9. NUSOne (could be grp 09)
10. NUSOne (could be grp 10)
11. Teow Hua Jun
12. Ling Yan Hao
13. Huang Xing Chen
14. Nguyen Doan Phuong Anh
15. Ramanathan Kumarappan
16. Wong Kai Jie
17. Juan Carlo Vieri
18. Alvin Yan Hong Yao
19. Teoh Jun Jie
20. Agrawal Naman

Question 3 at VisuAlgo Online Quiz

What is your expectation for final assessment, Fri, 29 Nov 2024?

A). A+, A, or at least A- :)

B). I have a feeling I am around average, so B+, B, at least B-

C). I just want to pass..., i.e., $>$ F

Acknowledgement

The slides are modified from previous editions of this course and similar course elsewhere.

List of credits: Erik D. Demaine, Charles E. Leiserson, Surender Baswana, Leong Hon Wai, Lee Wee Sun, Ken Sung, Arnab Battacharya, Diptarka Chakraborty, Steven Halim, Sanjay Jain, Chang Yi-Jun.