CS341: Algorithms

David Duan

Fall 2019

The study of efficient algorithms and effective algorithm design techniques. Program design with emphasis on pragmatic and mathematical aspects of program efficiency. Topics include divide and conquer algorithms, recurrences, greedy algorithms, dynamic programming, graph search and backtrack, problems without algorithms, NP-completeness and its implications.

1. Introduction

We omit the trivial definitions.

Asymptotic Notation

$$\begin{aligned}
f \in O(g) &\iff \exists c \in \mathbb{R}^+, n_0 \in \mathbb{N} : n \geq n_0 \implies f(n) \leq cg(n) \\
f \in o(g) &\iff \forall c \in \mathbb{R}^+ : \exists n_0 \in \mathbb{N} : n \geq n_0 \implies f(n) < cg(n) \\
f \in \Omega(g) &\iff g \in O(f) \\
f \in \omega(g) &\iff g \in o(f) \\
f \in \Theta(g) &\iff f \in O(g) \wedge f \in \Omega(g)
\end{aligned}$$

Limit Rules

Let $f(n), g(n)$ be positive functions. Assume that $\lim_{n\to\infty} \frac{f(n)}{g(n)} = L$ exists, then

$$f(n) \in \begin{cases} o(g(n)) & L = 0 \\ O(g(n)) & L < \infty \\ \Omega(g(n)) & L > 0 \\ \omega(g(n)) & L = \infty \end{cases}$$

2. Solving Recurrences

Consider the recurrence $T(n) = aT(n/b) + \Theta(n^c)$. At each level, the problem is divided into a sub-problems, each with problem size n/b, and requires $\Theta(n^c)$ amount of additional work to combine the solutions. There are three common ways to solve a recurrence.

Induction

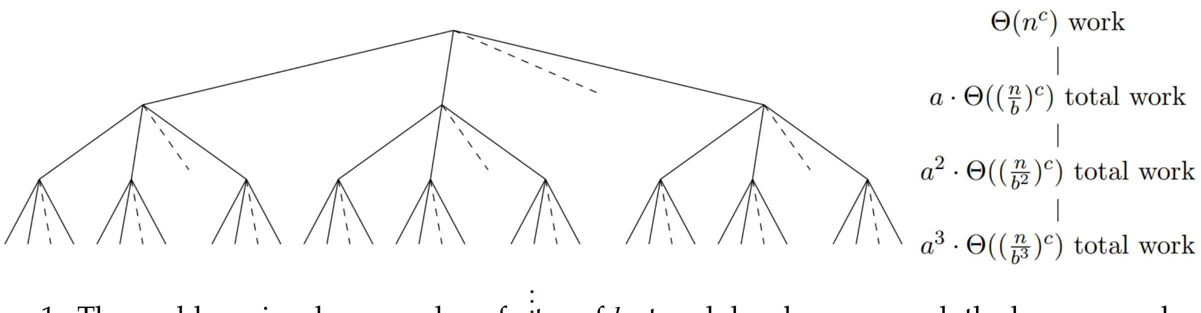
This is also called the *substitution method*.

1. Guess an (ideally tight) upper bound on $T(n)$.
2. Prove that your guess is correct using induction.

Two warnings:

1. The induction result should match exactly the guessed upper bound.
2. Guesses that are technically correct but not tight can look correct in the induction step.

Recurrence Tree


$\Theta(n^c)$ work
$a \cdot \Theta((\frac{n}{b})^c)$ total work
$a^2 \cdot \Theta((\frac{n}{b^2})^c)$ total work
$a^3 \cdot \Theta((\frac{n}{b^3})^c)$ total work

1. The problem size decreases by a factor of b at each level, so we reach the base case where $n = 1$ at level $\log_b n$. The recurrence tree thus has depth $\log_b n$.
2. The number of problems increases by a factor of a at each level, so there exists a^j sub-problems at level $j \in \{0, \ldots, \log_b n\}$. We consider the root level to be level 0.
3. Each sub-problem has problem size n/b^j so it does $\Theta((n/b^j)^c)$ work; the total work done at level j is thus $a \cdot \Theta((n/b^j)^c)$ for $j \in \{0, \ldots, \log_b n\}$.

Master Theorem

Recall the recurrence tree has depth $\log_b n$, so we can compute the total time complexity by

$$\begin{aligned} T(n) &= \Theta(n^c) + a \cdot \Theta((n/b)^c) + a^2 \cdot \Theta((n/b^2)^c) + \cdots + a^{\log_b n}\Theta(1) \\ &= \Theta(n^c)\left(1 + \frac{a}{b^c} + \left(\frac{a}{b^c}\right)^2 + \cdots + \left(\frac{a}{b^c}\right)^{\log_b n}\right). \end{aligned}$$

- If $a/b^c = 1$, then the geometric series is $\sum_{i=0}^{\log_b n} 1$, i.e., a sum of $(\log_b n + 1)$ 1's, so $T(n) = \Theta(n^c)(1 + 1 + \cdots + 1) = (\log_b n + 1)\Theta(n^c) \in \Theta(n^c \log n)$.
- If $a/b^c < 1$, then the geometric series $1 + (a/b^c) + (a/b^c)^2 + \cdots + (a/b^c)^{\log_b n} \in \Theta(1)$ is a constant term and $T(n) = \Theta(n^c) \cdot C = \Theta(n^c)$.
- If $a/b^c > 1$, then we have an increasing geometric series that is dominated by the last term, so $T(n) = \Theta(n^c \cdot \left(a/b^c)^{\log_b n}\right) = \Theta(n^c \cdot a^{\log_b n}(1/b^{\log_b n})^c) = \Theta(a^{\log_b n})$.

3. Divide and Conquer

1. *Divide* the original problem into smaller problems.
2. *Conquer* each smaller sub-problems separately.
3. *Combined* the results back together.

Consider the following examples from class. See appendix for more.

Counting Inversions

Given an array $\mathbf{a}$ of size n, find

$$K(\mathbf{a}) = |\{i, j \in [n] : i < j \wedge a_i > a_j\}|.$$

1. Split $\mathbf{a}$ into L and R. Then $K(\mathbf{a}) = K(L) + K(R) + r$, where r counts cross inversions.
2. Recurse on L and R to compute $K(L)$ and $K(R)$.
3. Merge L and R into a sorted array C as in merge sort (compare then move the smaller one to C):
   - For each $a_j \in R$, count how many elements in L that are larger than a_j. Call it r_j.
   - Note this is precisely the size of L when a_j gets merged into C. Call it L_j.
   - Then $r = \sum_{j=n/2+1} r_j = \sum_{j=n/2+1} |L_j|$.
4. Like merge sort, this runs in $O(n \log n))$ time.

Closest Pair on a Plane

Given $(x_1, y_1), \ldots, (x_n, y_n) \in \mathbb{R}^2$, find

$$\delta = \min_{1 \le i < j \le n} \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}.$$

1. Sort by x-coordinates, split into Q and R, recurse to find δ_Q and δ_R. Let $\delta_0 = \min(\delta_Q, \delta_R)$.
2. It remains to evaluate $\delta = \min(\delta_0, \delta_1)$ where δ_1 is the min distance of any cross pairs.
   - Only interested in points within δ_0-distance from the middle line. Call the set S.
     - *Claim.* Any pair of points $(x_i, y_i) \in Q$ and $(x_j, y_j) \in R$ at distance $\le \delta_0$ from each other must satisfy $(x_i, y_i), (x_j, y_j) \in S$.
     - *Proof.* If $(x_i, y_i) \notin S$, then $\|(x_i, y_i), (x_j, y_j)\| > \delta$.
   - Computing δ_1 takes linear time (i.e., constant for each $(x^*, y^*) \in S$).
     - *Claim.* For any point $(x^*, y^*) \in S$, there can be at most 8 points $(x', y') \in S$ where $y^* \le y' \le y^* + \delta$.
     - *Proof.* The feasible area $[-\delta_0, \delta_0] \times [y^*, y^* + \delta]$ can be splitted into 8 boxes with side length $\delta/2$. Each box can only contain one point at most. Thus ≤ 8 in total.
3. Both sorting and DnC take $\Theta(n \log n)$ so $\Theta(n \log n)$ overall.

4. Greedy Algorithms

1. Break down a problem into a sequence of decisions to be made.
2. Make locally optimal decisions without worrying about later decisions.

Always Ahead Proof Strategy

Definition. Let $A = \{a_1, \ldots, a_k\}$ be the solution generated by the given algorithm. Let $O = \{o_1, \ldots, o_m\}$ be an arbitrary (or optimal) feasible solution.

Measurement. Find a measure by which greedy stays ahead of a general/optimal solution.

Induction. Prove greedy stays ahead by showing that the partial solutions constructed by greedy are always just as good as the initial segments of your other solution, based on the measure you selected, i.e., for all indices $r \leq k$, prove by induction that $f(a_1, \ldots, a_r)[\leq, \geq]f(o_1, \ldots, o_r)$. Your induction step should use your algorithm as the argument.

Conclusion. Since greedy stays ahead of the other solution with respect to the measure you selected, the final output is optimal.

Always Ahead Proof Example

Given n pairs of start and finish time (s_i, f_i) , find a maximum non-overlapping subset of intervals.

Solution. Sort by finish time then go through the list, add (s_i, f_i) if and only if s_i is later than the last finish time.

Correctness. Let $A = \{i_1, \ldots, i_k\}$ be the intervals selected by our greedy algorithm, in the order which they were added. Let $O = \{j_1, \ldots, j_m\}$ be the optimal solution, order by finish times. The measure is number of non-overlapping intervals selected, i.e., size of $|A|$ and $|O|$. We want to show that for all $r \leq k$, $f(i_r) \leq f(j_r)$.

- If $r = 1$, we select the job with the earliest finish time, so $f(i_1) \leq f(j_1)$.
- For $t > 1$, assume the statement is true for $t - 1$ and we prove for t. By IH, $f(i_{t-1}) \leq f(j_{t-1})$, so any jobs that are valid to add to the optimal solution are certainly valid to add to our greedy algorithm, so $f(i_t) \leq f(j_t)$.
- Thus, for all $r \leq k$, $f(i_r) \leq f(j_r)$. In particular, $f(i_k) \leq f(j_k)$.

If A is not optimal, then it must be the case that $m > k$, so there is a job j_{k+1} in O that is not in A. This job must start after O's kth job which finishes at $f(j_k)$ and hence after $f(i_k)$, by our previous result. But then this job would have been compatible with all jobs in A, so our greedy algorithm would have added it. Contradiction. Thus A is optimal. □

Exchange Proof Strategy

Definition. Let $A = \{a_1, \ldots, a_k\}$ be the solution generated by the given algorithm. Let $O = \{o_1, \ldots, o_m\}$ be an arbitrary (or optimal) feasible solution.

Compare. Assume the arbitrary/optimal solution is not the same as the greedy solution, then

1. There is an element of O that is not in A or an element of A that is not in O, or
2. There are two consecutive elements in O in a different order than they are in A.

Exchange. Swap the elements in question in O, and argue that you have a solution that is no worse than before. Then argue that if you continue swapping, you can eliminate all differences between O and A in a polynomial number of steps without worsening the quality of the solution.

Conclusion. Thus, the greedy solution produced is just as good as any arbitrary/optimal solution, and hence is optimal itself.

Exchange Proof Example

Given n tasks with processing time p_i and deadlines d_i, find the ordering of tasks that minimizes the maximum lateness of any task.

Solution. We sort the tasks in order of increasing deadlines and perform tasks in this order.

Correctness. Sort the tasks in increasing deadline, so that the greedy algorithm performs them in order. Let $A = \{1, 2, \ldots, n\}$ and O be an arbitrary ordering. Then in O that must be two consecutive tasks i, j with $d_i \leq d_j$ but j performed right before i. Swap those two tasks to obtain a new ordering O'. Then every task except i and j have the same lateness in O and in O'. The lateness of i satisfies $L_i^{(O')} \leq L_i^{(O)}$ since we do task i earlier in O'. And the lateness of j satisfies $L_i^{(O')} \leq L_i^{(O)}$ because $d_i \leq d_j$. Therefore, the maximum lateness of O' is at most that of O. This means that we can continue swapping in this way until we obtain the order A produced by the greedy algorithm, and at every step along the way we never increase the maximum lateness so the algorithm's solution is valid. □

5. Dynamic Programming

1. Breaking up a problem into smaller sub-problems.
2. Solving the sub-problems from smallest to largest.
3. Storing solutions along the way to avoid repetitions.

Class Examples

We covered the following examples in class:

- Text segmentation.
- Longest increasing subsequence.
- Longest common subsequence.
- Edit distance.
- Weighted interval scheduling.
- Optimal binary search tree.
- Binary knapsack.

Subproblems

To solve a problem using DP, you want to first come up with an appropriate subproblem to work with, then think about how to combine the results of smaller problems to solve a larger problem. A couple questions/remarks:

1. If you don't know where to start, simply reduce the problem size to $\ell = [k]$.
2. Think about how can u leverage the results of subproblems $\ell = 1, \dots, k-1$ to solve the problem for $\ell = k$.
3. Think about how to initialize the DP array/matrix, i.e., what should the answer to the problem be when $\ell = 0$?

Consider the binary knapsack problem. We consider the first k elements and we fix the capacity of a knapsack to be w, i.e., for $k \in \{0, 1, \dots, n\}$ and for $w = \{0, 1, \dots, W\}$, we solve

$$M(k, w) = \max_{S \subseteq \{1,2,\dots,k\} : \sum_{i \in S} w_i \le w} \sum_{i \in S} v_i .$$

Base Case. For every $w \in W$ and $k \le n$, $M(0, w) = 0$ (we do not choose any item) and $M(k, 0) = 0$ (we cannot choose any item).

"Induction". For $k \ge 1$, we then have two possibilities, either $w_k > w$, in which case the item k does not fit into the knapsack and $M(k, w) = M(k-1, w)$, or $w_k \le w$, in which case $M(k, w)$ is the maximum of the optimal value $M(k-1, w)$ obtained by leaving out item k and the optimal value $v_k + M(k-1, w-w_k)$ obtained by including item k in the knapsack. Thus, for each $k = \{1, 2, \dots, n\}$, we have

$$M(k, w) = \begin{cases} M(k-1, w) & w_k > w \\ \max\{M(k-1, w), v_k + M(k-1, w-w_k)\} & w_k \le w \end{cases}$$

6. Graph Algorithms

Let $G = (V, E)$, $n = |V|$ and $m = |E|$. We omit the basic definitions.

In CS341, a path (in graph theory) is called a *simple path* and a walk (in graph theory) is called a *path*.

Graph Representation

- An *adjacency matrix* is an $n \times n$ matrix where $A[i, j] = 1$ if $v_i v_j \in E$ and 0 otherwise.
  - Space: $\Theta(n^2)$
  - List all edges: $\Theta(n^2)$
  - List all neighbours of v: $\Theta(n)$
  - Check if v_i is adjacent to v_j: $\Theta(1)$
- An *adjacent list* is a set of n linked lists, where the i-th linked list contains the neighbours of v_i.
  - Space: $\Theta(n + m)$ [1]
  - List all edges: $\Theta(n + m)$
  - List all neighbours of v: $\Theta(1 + \deg(v))$
  - Check if v_i is adjacent to v_j: $O(1 + \deg(v_i))$

1: We need $n \in \Theta(n)$ vertices to start the list; the sum of degrees of all vertices add up to $2m \in \Theta(m)$.

When a graph is sparse, the matrix representation wastes a lot of space as most of the matrix cells remain unused. We generally prefer the adjacency list representation in CS341.

Breadth-First Search

BFS finds all the vertices that can be reached from s, exploring all of the neighbour nodes at the present depth prior to moving on to the nodes at the next depth level.

Algorithm 1: BFS(G, s)

```
1  for each $v \in V(G)$ do
2      visited[$v$] ← False;
3  visited[$s$] ← True;
4  $Q$.Enqueue($s$);
5  Pre-Visit(); // Additional initialization steps.
6  while $Q$ is not empty do
7      $v$ ← $Q$.Dequeue();
8      for each $w \in G.AdjList(v)$ do
9          if !visited[$w$] then
10             Func(w); // Additional operations on $w$.
11             $Q$.Enqueue($w$);
12     Post-Visit(); // Additional operations on $v$.
13     visited[$v$] = True;
14 return;
```

Depth-First Search

DFS finds all vertices that can be reached from s, exploring as far as possible along each branch before backtracking.

Algorithm 2: DFS(G, s)

```
for each $v \in V(G)$ do
    visited[$v$] ← False;
visited[$s$] ← True;
$S$.Push($s$);
Pre-Visit(); // Additional initialization steps.
while $S$ is not empty do
    $v$ ← $S$.Pop();
    for each $w \in G.AdjList(v)$ do
        if !visited[$w$] then
            Func(w); // Additional operations on $w$.
            $S$.Push($w$);
    Post-Visit(); // Additional operations on $v$.
    visited[$v$] = True;
return;
```

Applications of BFS and DFS

We covered the following applications of BFS and DFS in class and in assignments:

- Modify BFS to solve Single-Source Shortest Path problem.
- Modify BFS to test bipartiteness of a graph.
- Modify BFS to generate a spanning tree of a graph.
- Modify DFS to generate a spanning tree of a graph.
- Use the DFS tree to identify cut vertices in a graph.
- Use the DFS tree to check if the digraph contains a dicycle.
- Use DFS for topological sorting on directed acyclic graphs.
- Use DFS to test strong connnectivity of a graph.
- Use BFS/DFS to determine if there exists an s, v-path in G with length $\le k \in \mathbb{Z}^+$.
- Use BFS/DFS to determine if there exists a cycle in G that contains $e \in E(G)$.

Minimum Spanning Tree

Three graph theory properties concerning MST:

- Cut Property: If e is the min weight edge (no ties) in a cut $E(S, V - S)$, then e is in every MST.
- Contraction: If T is an MST for G and $e \in T$, then T/e is an MST for G/e.

- Cycle Property: If e is the max weight edges (no ties) in a cycle in G, then e is in no MST.

Two MST algorithms:

- Kruskal's Algorithm ($O(m \log m)$ using Union-Find [2]) is a greedy algorithm which finds a minimum spanning tree by finding an edge of least possible weight that connects two trees in the forest.
- Prim's Algorithm ($(O(V+E)\log V)$ is a greedy algorithm that finds an MST by keep adding the cheapest possible edge in the cut.

2: more on this in CS466

Shortest Paths with Non-Negative Weights

Consider a weighted graph G. If all weights are non-negative, we could use Dijkstra's algorithm ($O(m \log n)$) to find the shortest distance between $s \in V(G)$ and every other node in G. We keep selecting the vertex that is closest to the source among all of those that have not been visited yet and see if we could get to other vertices faster.

Algorithm 3: Dijkstra($G = (V, E), w, s$)

```
for all $v \in V$ do
    dist[$v$] $\leftarrow \infty$;
dist[$s$] = 0;
$H \leftarrow$ MakePriorityQueue($V$, dist);
while $H \neq \varnothing$ do
    $u \leftarrow$ DeleteMin($H$);
    for all $v \in G$.AdjList($u$) do
        if dist[$u$] + $w(u, v)$ < dist[$v$] then
            dist[$v$] $\leftarrow$ dist[$u$] + $w(u, v)$;
            DecreaseKey($H$, $v$, dist[$v$]);
return dist;
```

Informally [3], the algorithm works because we can never have a shortest s, v-path that goes through u which is further away from s than v, provided that $w \in \mathbb{R}^+$.

3: For a formal proof of correctness, prove the following invariant using induction: at every iteration of the while loop, the distance dist[v] from s to every vertex $v \notin H$ is correct, i.e.,

$$\text{dist}[u] = \min_{v \notin H}\{\text{dist}[v] + w(v, u)\}.$$

Shortest Paths with Negative Weights

If G contains negative dicycles, then Dijkstra's will fail its job. Instead, we will use Bellman-Ford ($O(mn)$), which is a DP algorithm.[4] We run the algorithm $n-1$ times to compute distances and one more times to detect negative dicycles. If the values change between the last two iterations, then there exists a negative dicycle.

4: The update rules for BF are:

$$d_0[v] = \begin{cases} w(s, v) & sv \in E \\ \infty & sv \notin E \end{cases}$$

$$d_i[v] = \min \begin{cases} d_{i-1}[v] \\ d_{i-1}[u] + w(u, v) & \forall u : uv \in E \end{cases}$$

Algorithm 4: Bellman-Ford($G = (V, E), w, s$)

```
for all $v \in V$ do
    if $sv \in E$ then
        $d[v] \leftarrow w(s, v)$;
    else
        $d[v] \leftarrow \infty$;
for $i = 2, 3, \dots, n-1$ do
    for all $uv \in E$ do
        if $d[u] + w(u, v) < d[v]$ then
            $d[v] \leftarrow d[u] + w(u, v)$;
return $d$;
```

All-Pairs Shortest Path

To determine the length of a shortest path between every pair of vertices in V, we want to use Floyd-Warshall $O(n^3)$, which is also a DP algorithm. Instead of limiting the number of edges that the path can use, we want to limit the set of intermediate node that those path can traverse, i.e., in the k-th iteration, we can only use $\{v_1, \dots, v_k\}$ as intermediate nodes in our paths. The key observation is the shortest path from u to v that uses only $\{v_1, \dots, v_k\}$ as intermediate nodes either goes through v_k or it does not. [5]

5: The update rules for FW are:

$$d_0[u, v] = \begin{cases} w(u, v) & uv \in E \\ \infty & uv \notin E \end{cases}$$

$$d_k[u, v] = \min \begin{cases} d_{k-1}[u, v_k] + d_{k-1}[v_k, v] \\ d_{k-1}[u, v] \end{cases}$$

Algorithm 5: Floyd-Warshall($G = (V, E), w$)

```
for all $u, v \in V \times V$ do
    if $uv \in E$ then
        $d_0[u, v] \leftarrow w(u, v)$;
    else
        $d_0[u, v] \leftarrow \infty$;
for $k = 1, 2, 3, \dots, n$ do
    for all $u \in V$ do
        for all $v \in V$ do
            $d_k[u, v] =$
              $\min\{d_{k-1}[u, v_k] + d_{k-1}[v_k, v], d_{k-1}[u, v]\}$;
return $d_n$;
```

7. Exhaustive Search

Backtracking

Algorithm 6: Backtrack Template

```
A ← the set of active configurations;
while $A \neq \emptyset$ do
    C ← next configuration of A;
    if C is a solution then
        return true;
    if C is not a dead end then
        Expand C to C' and add C' to A;
return false;
```

We consider the set of all possible *configurations*[6] in a tree structure and explore the tree with the following rules:

- If the current configuration is a valid solution, return it.
- If the current configuration is a dead end, that is, it will not lead to any valid solution, stop exploring this branch by ignoring this configuration.
- If the current configuration is a valid partial solution that may lead to valid full solutions, expand it and add the new configuration to the set to explore further.

6: full or partial solutions to the problem, which are represented by nodes in the configuration tree.

Branch and Bound

Algorithm 7: Branch and Bound Template

```
A ← the set of active configurations;
best ← ∞;
while $A \neq \emptyset$ do
    C ← next configuration of A;
    if C is a solution and Value(C) < best then
        best ← Value(C);
    else if C is not a dead end and
     ValueLowerBound(C) < best then
        Expand C to C' and add C' to A;
return best;
```

- `Value`(C) is the value of the current configuration when it represents a full solution.
- `ValueLowerBound`(C) is a lower bound on the value of any full solution that can be obtained by expanding configuration C.

We consider the set of all possible configurations to the problem. At each step, we

- *Branch* by splitting A into subsets $A_1, \dots, A_k$ of possible configurations, and
- *Bound* the minimum value of any solution in configuration A_i; we discard A_i if this value is smaller [7] than the optimal solution discovered so far.

7: if maximization and greater if minimization

8. Complexity Theory

P and Polynomial-Time Reduction

The class **P** [8] is the set of all decision problems [9] that can be solved by a polynomial-time algorithm, that is, an algorithm that runs in $O(n^k)$ time for any input for some constant k. Problems in **P** are said to be *tractable*.

8: stands for *polynomial* time

9: we can often turn a non-decision problem (e.g. an optimization problem or a search problem) into an analogous decision problem

The decision problem A is *polynomial-time reducible* to decision problem B, denoted $A \leq_{\mathbf{P}} B$, if a polynomial time algorithm for B can be used to make a polynomial time algorithm for A. [10]

10: This is called a *Turing reduction* and is more general. In *many-one reduction*, only a single call to the algorithm for B is allowed.

If A and B are two decision problems that satisfy $A \leq_{\mathbf{P}} B$, then $B \in \mathbf{P} \implies A \in \mathbf{P}$ and $A \notin \mathbf{P} \implies B \notin \mathbf{P}$.

If $Hard$ is a decision problem that satisfies $Hard \notin \mathbf{P}$ and B is a decision problem that satisfies $Hard \leq_{\mathbf{P}} B$, then $B \notin \mathbf{P}$. [11]

11: **The direction for polynomial-time reduction is very important!** Showing $A \leq_{\mathbf{P}} Hard$ says nothing about the complexity of problem A.

NP and Polynomial-Time Verification

A *verifier* for the decision problem X is an algorithm A that takes in as arguments an input x to problem X and an additional input (called a *potential certificate*) y and satisfies two conditions:

1. For every Yes input x to X, there is a *certificate* y that causes $A(x, y)$ to output Yes, and
2. For every No input x to X, for any possible input y the algorithm $A(x, y)$ outputs No.

NP[12] is the set of all decision problems that have polynomial-time verifiers, that is, a verifier with time complexity that is polynomial in the size of the input to the decision problem.

12: stands for *non-deterministic, polynomial* time

NP-Completeness

A decision problem X is said to be **NP**-*hard* if every problem $A \in \mathbf{NP}$ satisfies $A \leq_{\mathbf{P}} X$. A decision problem X is said to be **NP**-*complete* if $X \in \mathbf{NP}$ and X is **NP**-hard. [13]

13: Intuitively, a problem is **NP**-hard if it is at least as hard as the hardest problem in **NP**; a problem is **NP**-complete if it is at least as hard as the hardest problem in **NP** and itself is in **NP**.

Let X be any **NP**-complete problem. Then:

- If $X \in \mathbf{P}$ then $\mathbf{P} = \mathbf{NP}$ and every problem in **NP** can be solved with a polynomial-time algorithm.
- If $X \notin \mathbf{P}$, then $\mathbf{P} \neq \mathbf{NP}$ and no **NP**-complete problem can be solved with a polynomial-time algorithm.

Let A be an **NP**-complete problem and let $X \in \mathbf{NP}$ be a decision problem such that $A \leq_{\mathbf{P}} X$. Then X is **NP**-complete. A classic choice for **NP**-complete A is the 3SAT problem *.

* see appendix

A. Appendix: Introduction, Recurrence, DnC, Greedy, DP

Recurrence by Induction Proof

Use induction to find a tight upper bound for T where $T(n) = 2T(\lfloor n/2 \rfloor) + n$ and $T(1) = 1$.

First, guess an upper bound on $T(n)$: $T(n) \leq c \cdot n \log n$ for some constant c (that we will determine later) whenever $n \geq 2$. Next, prove the guess is correct using induction. The induction hypothesis is that for every $2 \leq k < n$, $T(k) \leq c \cdot k \log k$. Then

$$\begin{aligned} T(n) &= 2T(\lfloor n/2 \rfloor) + n \\ &\leq 2(c\lfloor n/2 \rfloor \log\lfloor n/2 \rfloor) + n && \text{Substitution} \\ &\leq cn\log(n/2) + n && \text{Remove floor} \\ &= cn(\log n - 1) + n \\ &= cn \log n - cn + n \\ &\leq cn \log n && \text{For } c \geq 1 \end{aligned}$$

Note the above steps work only when $n \geq 4$.[*] Thus, we fix $n \geq 4$ and show base cases $T(2)$ and $T(3)$ satisfies our induction hypothesis:

$$T(2) = 2T(1) + 2 = 4 \leq c \cdot 2 \log 2 \text{ for } c \geq 2.$$
$$T(3) = 2T(1) + 2 = 5 \leq c \cdot 3 \log 3 \text{ for } c \geq 2.$$

Hence, $T(n) \leq 2 \cdot n \log n \in O(n \log n)$ as desired. □

Divide and Conquer Example

The diameter of a tree is the length of the longest simple path between nodes of the tree. Design a DnC algorithm to compute the diameter of a rooted binary tree.

Suppose that P is the longest path in the (sub)-tree with root node Y. Let A and B be two ends of P. Then there are three cases to consider:

1. A and B are both in Y_L. In this case, $|P| = \texttt{diameter}(Y_L)$.
2. A and B are both in Y_R. In this case, $|P| = \texttt{diameter}(Y_R)$.
3. $A \in Y_L$ and $B \in Y_R$. Then P passes through node Y. The portion from A to Y has length $\texttt{depth}(Y_L) + 1$ and the portion from Y to B has length $\texttt{depth}(Y_R) + 1$, so the total length is $\text{depth}(Y_L) + \text{depth}(Y_R) + 2$.

Thus, $\texttt{diameter}(Y) = \max\{\texttt{diameter}(Y_L), \texttt{diameter}(Y_R), \texttt{depth}(Y_L) + \texttt{depth}(Y_R) + 2\}$ where $\texttt{depth}(Y) = \max\{\texttt{depth}(Y_L) + 1, \texttt{depth}(Y_R) + 1\}$.

We now show $T(n) \in \Theta(n)$. Let n denote the number of nodes in a rooted binary tree T. Define n_L and n_R naturally. Note that $n = n_L + n_R + 1$. Then $T(n) = T(n_L) + T(n_R) + \Theta(1)$ given $n > 0$ ($T(n) = 0$ for $n = 0$). It can be proved by induction that $T(n) \leq dn$ for some positive d and hence $T(n) \in O(n)$. On the other hand, there is one recursive call for each node, so $T(n) \in \Omega(n)$. Hence, $T(n) \in \Theta(n)$. □

[*] For $n = 2$ or $n = 3$, $\log\lfloor n/2 \rfloor = 0$.

Reduction Example

For a graph $G = (V, E)$, S is an independent set if and only if $(V - S)$ is a vertex cover.

- Let S be an independent set. Suppose $(V - S)$ is not a vertex cover. Then there exists an edge $e = (u, v)$ in G where $u, v \notin S$, i.e., $u, v \in S$. Then S is not an independent set. Contradiction.
- Let $(V - S)$ be a vertex cover. Suppose S is not an independent set, so there exists an edge $e = (u, v)$ in G where $u, v \in S$. Then $(V - S)$ is not a vertex cover. Contradiction.

B. Appendix: Graph, Exhaustive Search, Complexity

Complexity Proof Template

Show $A \in \mathbf{P}$ - Simply provide a polynomial-time algorithm that solves A.

Show $A \leq_{\mathbf{P}} B$ **(Many-One)** - Show there exists a polynomial-time algorithm F that transforms an input I_A to an input I_B to B which produces the same truth value.

Show $A \leq_{\mathbf{P}} B$ **(Turing)** - Show there exists a polynomial-time algorithm F that uses a polynomial-time algorithm that solves B as a subroutine to solve A.

Show $A \in \mathbf{NP}$ - Show $X \leq_{\mathbf{P}} A$ for some **NP**-complete problem X, then provide a polynomial-time verifier for A.

A List of NP-Complete Problems

3SAT Given a Boolean CNF formula (an AND of ORs) on n variables in which each clause has at most 3 literals, determine whether there is an assignment of T or F values to the n variables that satisfies the formula.

Clique Determine if G has a clique [*] of size at least k.

IndepSet Determine if G has an independent set [†] of size $\geq k$.

VertexCover Determine if G has a vertex cover of size $\leq k$.

SetCover Given a collection $\mathcal{S}$ of subsets of $[m]$ and $k \in \mathbb{Z}^+$, determine if there are k sets $S_1, \ldots, S_k \in \mathcal{S}$ such that $S_1 \cup \cdots \cup S_k = [m]$.

HamPath Determine if G contains a Hamiltonian path. [‡]

HamCycle Determine if G contains a Hamiltonian cycle.

DirHamPath Determine if G contains a directed Hamiltonian path.

DirHamCycle Determine if G contains a directed Hamiltonian cycle.

SubsetSum Given an array of n elements $w_1, w_2, \ldots, w_n$ and a target weight T, determine whether there is a subset $S \subseteq [n]$ such that $\sum_{i \in S} w_i = T$.

Polynomial Reduction Example

Example 1. DirHamPath $\leq_{\mathbf{P}}$ HamPath. Given the digraph $G = (V, E)$, construct $G' = (V', E')$ where for each $v \in V$, we create three vertices in V': v itself, v_{in} and v_{out}. We build E' by adding edges (v_{in}, v) and (v, v_{out}) for each $v \in V$, and for each $(u, v) \in E$ by adding the edge (u_{out}, v_{in}) in E'. With this construction, if there is a Hamiltonian path $v^{(0)}, v^{(1)}, \ldots, v^{(n)}$ in G, then the path $v_{in}^{(0)}, v^{(0)}, v_{out}^{(0)}, v_{in}^{(1)}, \ldots, v_{out}^{(n)}$ is a Hamiltonian path in G'. Also, if we have a Hamiltonian path in G', then by our construction, we must also have a Hamiltonian path in G as well. □

Example 2. HamPath $\leq_{\mathbf{P}}$ HamCycle. Let F be the algorithm that takes the input $G = (V, E)$ and creates the graph $G' = (V', E')$ where the set of vertices of the new graph $V' = V \cup \{s\}$ is obtained by adding a new vertex s and the set of edges $E' = E \cup \{(s, v) : v \in V\}$ is the original set along with an edge between every vertex in V and the new vertex x. This transformation is easily computed in polynomial time. Now, if G has a Hamiltonian path P with ends $a, b \in V$, then $sa + P + bs$ is a Hamiltonian path in G'; if G' has a Hamiltonian cycle C with edges (s, a) and (s, b), then removing these two edges results in a Hamiltonian path in G. □

[*] a set of vertices where every two vertices are adjacent

[†] a set of vertices such that no two of which are adjacent

[‡] a path (cycle) that visits every vertex in G exactly once