CS577: Introduction to Algorithms

Discussion 1: Discrete Review

Solving Recurrences

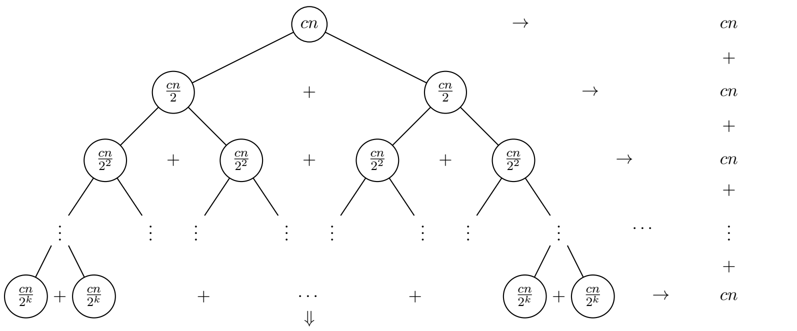
$$T(n) \leq 2 \cdot T\left(\frac{n}{2}\right) + cn; T(1) \leq c$$

Unrolling / unwinding:

$$\begin{aligned}
T(n) &\leq 2T\left(\frac{n}{2}\right) + cn \\
&\leq 2\left(2T\left(\frac{n}{4}\right) + c\frac{n}{2}\right) + cn \\
&\leq 2\left(2\left(2T\left(\frac{n}{2^3}\right) + c\frac{n}{2^2}\right) + c\frac{n}{2}\right) + cn \\
&\vdots \\
&\leq 2^k T\left(\frac{n}{2^k}\right) + kcn \\
&= nT(1) + cn\log(n), \text{ using (??)} \\
&= cn + cn\log n \\
&= O(n\log(n))
\end{aligned}$$

$$\begin{aligned}
& 1 = \frac{n}{2^k} \\
\iff & 2^k = n \\
\iff & k = \log_2(n) \qquad (1)
\end{aligned}$$

Recursion Tree:



$$\left(\sum_{i=0}^{k} 2^i \cdot \frac{cn}{2^i}\right) = (k+1)cn = (\text{tree height} + 1) \cdot cn \quad \begin{aligned} &= cn\lg n + cn \\ &= O(n \cdot \lg n)\end{aligned}$$

Induction

We want to show that a predicate $P(n)$ holds for $n = \{0, 1, 2, \ldots\}$.

1. Base Case(s): The starting point of the induction. Without a base case, there is not induction.
2. Inductive Hypothesis: We assume that $P(k)$ is true for some k. For *strong* induction, we assume that $P(k)$ is true for all $j \leq k$.
3. Inductive Step: We show that $P(k+1)$ holds, using the (2) inductive hypothesis.

So,

$$\text{Base Case(s)} \rightarrow P(0) \rightarrow P(1) \rightarrow P(2) \rightarrow P(3) \rightarrow P(4) \rightarrow \ldots$$

Program Correctness

A program/algorithm is correct if it is:

- **Sound/partial correctness/correct** (any returned value is true), and
- **Complete/termination** (returns a value for all valid inputs).

Proving correctness requires at least 2 proofs: One for soundness and one for completeness.

Iterative Code:

1. Identify and prove the loop invariants using induction.
2. Using the invariants, prove the soundness.
3. Prove the completeness: Show that the algorithm terminates for all input.

Recursive Code:

1. Prove the soundness via induction:
   (a) Show that the recursive base case is correct.
   (b) Assuming that the recursive calls return the correct value (ind hyp), show that the code returns the correct value (induction step).
2. Prove the completeness: Show that recursive calls make progress towards a base case.

CS577: Introduction to Algorithms

Discussion 2: Asymptotic Analysis and Graphs

Asymptotic Bounds

Bound	Informal Notion	Formal Definition	Limit Test
$f(n) \in O(g(n))$	'$f(n) \leq g(n)$'	$O(g(n)) = \{f(n) : \exists c, n_0 > 0 \mid 0 \leq f(n) \leq cg(n)\ \forall n \geq n_0\}$	$\lim_{n\to\infty} \frac{f(n)}{g(n)} \leq d$ for $d \in \mathbb{R}_{>0}$
$f(n) \in \Omega(g(n))$	'$f(n) \geq g(n)$'	$\Omega(g(n)) = \{f(n) : \exists c, n_0 > 0 \mid 0 \leq cg(n) \leq f(n)\ \forall n \geq n_0\}$	$\lim_{n\to\infty} \frac{f(n)}{g(n)} \geq c$ for $c \in \mathbb{R}_{>0}$
$f(n) \in \Theta(g(n))$	'$f(n) = g(n)$'	$\Theta(g(n)) = \{f(n) : \exists c_1, c_2, n_0 > 0 \mid 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)\ \forall n \geq n_0\}$	$c \leq \lim_{n\to\infty} \frac{f(n)}{g(n)} \leq d$ for $c, d \in \mathbb{R}_{>0}$
$f(n) \in o(g(n))$	'$f(n) \ll g(n)$'	$o(g(n)) = \{f(n) : \forall c > 0 \exists n_0 > 0 \mid 0 \leq f(n) < cg(n)\ \forall n \geq n_0\}$	$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$
$f(n) \in \omega(g(n)$	'$f(n) \gg g(n)$'	$\omega(g(n)) = \{f(n) : \forall c > 0 \exists n_0 > 0 \mid 0 \leq cg(n) < f(n)\ \forall n \geq n_0\}$	$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$

CS577: Introduction to Algorithms

Discussion 3 and 4: Greedy

In lecture, we saw two main techniques for showing that a greedy algorithm, GREEDY, is optimal: Stays Ahead and Exchange Argument.

Stays Ahead

A *Stays Ahead* argument requires that you define a notion of time steps over which you can show by induction that GREEDY is better according to some measure $g(i)$ than any other algorithm, or an optimal algorithm for all time steps i. The opimality of GREEDY can follow immediately, or may require a proof using the property shown in the induction.

The canonical problem presented in lecture was the *Interval Scheduling Problem*, and the optimal GREEDY was the *finish first* heuristic.

Steps for a Stays Ahead argument:

1. Establish a notion of time steps. These times steps should be well-defined for any algorithm.

   Ex: For the Interval Scheduling Problem, the time steps were the index in a set of jobs S produced by an algorithm when the jobs are sorted by finish time from smallest to largest. Note that this is well-defined for any algorithm.

2. Using induction over the time steps, show that, for some measure $g(i)$, GREEDY is the same or better than any other algorithm for all time steps i.

   Ex: For the Interval Scheduling Problem, $g(i)$ was the property that the finish time of the ith job of GREEDY was less than or equal to the ith job scheduled in any other schedule.

3. Using the property shown in the induction to claim the optimality of GREEDY.

   Ex: For the Interval Scheduling Problem, if GREEDY had scheduled k items, then $g(k)$ told us that the finish time of the kth job of GREEDY was less than or equal to the kth job scheduled in any other schedule, implying that GREEDY could schedule any $k+1$ job that any other algorithm would schedule, a contradiction to GREEDY scheduling k items.

Exchange

An *exchange* argument starts from an solution S^* that may be defined as an optimal solution, or the solution of any algorithm. The solution is transformed by some sort of *exchange* to a new solution S_1 that is closer the to the solution S produced by the GREEDY heuristic and no worse than S^*. Then, by induction, we can repeat this exchange producing a sequence of equivalent solutions until we arrive at S. That is, $S^* \equiv S_1 \equiv S_2 \equiv \cdots \equiv S$ for the function we are trying to optimize for the given problem.

The canonical problem presented in lecture was a job scheduling problem of *minimizing max lateness*, and the optimal GREEDY was the *earliest deadline first* (EDF) heuristic.

Steps for a Exchange argument:

1. Consider solutions produced by GREEDY, and determine characteristic properties of those solutions that may not be shared by an arbitrary optimal solution.

   Ex: For minimizing the max lateness, the solutions produced by the EDF had no idle time and jobs were scheduled by increasing deadlines.

2. Define the exchanges need to transform any solution into a solution closer to the solution produced by GREEDY, and show that such an exchange produces a solution that is no worse.

   Ex: For minimizing the max lateness, we showed that (1) idle time could be removed from any schedule without increasing the max lateness, and (2) that in schedule (without idle time) where the jobs are not ordered by increasing deadlines, there are at least two sequential jobs where there deadlines are out of order (an inversion). These two jobs can be swapped (removing an inversion, and sorting them by their deadlines) without increasing the lateness.

3. By induction, transform any solution into a solution that is equivalent to the GREEDY solution.

   Ex: For minimizing the max lateness, the GREEDY schedule S has no idle time and the jobs are scheduling by increasing deadline. The schedule S' produced by starting from any schedule S^*, removing idle time and repeatedly doing exchanges until no inversions remain also has no idle time and the jobs are scheduling by increasing deadline. The schedule S and S' may differ on the order of jobs with the same deadlines, but this does not change the max lateness.

CS577: Introduction to Algorithms

Discussion 5 and 6: Divide and Conquer

Algorithms that fit the Divide and Conquer paradigm, like MergeSort are most often presented as recursive algorithms since the paradigm is naturally recursive. It is important to remember that recursion has no more computational power than an iteration, and that a recursive algorithm can be described iteratively, and vice-versa. For example, on the GeeksForGeeks website, you can find:

- a recursive implementation of MergeSort (`https://www.geeksforgeeks.org/merge-sort/`), and
- an iterative implementation of MergeSort (`https://www.geeksforgeeks.org/iterative-merge-sort/`).

Divide and Conquer Paradigm

1. *Divide:* Split the problem into smaller sub-problems.
2. *Conquer: (Recurse)* Solve the smaller sub-problems (usually through recursion).
3. *Combine: (Merge)* Combine the solutions to the sub-problems into a solution for the original problem.

Often a useful technique to improve the efficiency of the solution.

Divide and Conquer Correctness

Assuming a recursive algorithm:

Soundness: Typically strong induction on input size.

1. Show that the base case(s) is correct.
2. Assume that the recursive calls return the correct value.
3. Given the assumption above, show that the value return will be correct.

Completeness: Show that the recursive calls make progress towards the base case.

Divide and Conquer Runtime Analysis

Assuming recursive algorithm:

From Algorithm Definition to Recurrence:

The runtime for a recursive Divide and Conquer algorithm with input of size n is:

- $T(n) = \sum_i T(n_i) + f(n)$, where n_i is the size of the input to the ith recursive call, and $f(n)$ is the cost of the work done outside of the recurrences for a call of input size n.
- Don't forget the base cases!

Example:

Algorithm 1 DCALG

```
Input  : $A$
Output: $f(A)$
1 if $|A| = 1$ then return $g(A)$
2 $A_1$ := DCALG(Front-half of $A$)  $A_2$ := DCALG(Last 1/4 of $A$)  $A_3$ := DCALG(First 1/4 of $A$)  return $h(A_1, A_2, A_3)$
```

If $g(A)$ is constant time, and $h(A_1, A_2, A_3)$ is linear time:

$$T(n) \leq T(\frac{n}{2}) + 2T(\frac{n}{4}) + cn;\ T(1) = c$$

If $g(A)$ is linear time, and $h(A_1, A_2, A_3)$ is constant time:

$$T(n) \leq T(\frac{n}{2}) + 2T(\frac{n}{4}) + c;\ T(1) = cn$$

If $g(A)$ is quadratic time, and $h(A_1, A_2, A_3)$ is quadratic time:

$$T(n) \leq T(\frac{n}{2}) + 2T(\frac{n}{4}) + cn^2;\ T(1) = cn^2$$

Solving Recurrences: For the main techniques, see Week 1 discussion.

CS577: Introduction to Algorithms

Discussion 7 and 8: Dynamic Programming

Problems that can be effectively solved by dynamic programming are:

- Problems that can be broken up into at most a polynomial number of subproblems that can be aggregated into a final solution efficiently.
- Typically a recursive solution that has a polynomial number of unique recursive calls which allows for memoization.

Dynamic Programming Paradigm

Step 1 - Recursive Solution: Develop a recursive solution for the problem. For many dynamic program solution, the recursive solution will be inefficient with an exponential number of recursive calls, but the number of unique calls is polynomial.

E.g.: nth Fibonacci number: $f(n) = f(n-1) + f(n-2); f(2) = 1; f(1) = 1;$. An exponential number of recursive calls, but only n unique calls.

Step 2 - Memoize: Means "write it down and remember". Dynamic programs are defined recursively, but use a table/array/matrix to record the values of the recursive calls. Calculate once, record, and lookup on future calls.

E.g.: For the nth Fibonacci number, we can use a 1-D table to record the values from 1 to n. That would take the complexity down to a linear number of calculations from an exponential number of calls.

Important Notes:

- The memoization table for a dynamic program can any number of dimensions. We will see mostly 1-D and 2-D solutions, but there is nothing preventing 3-D, 4-D, or more.
- Determining the dimensions of memoization table: This will usually correspond to the explicit parameters of your recursion solution. In other words, the parameters that characterize the unique recursive calls, i.e., the degrees of freedom.
- There is always a context for the values stored in the table: You should always be able to describe the semantics of the value in a cell of the memoization table.
- For some problems, the table of values may represent a data structure like a tree.

Presenting Your Dynamic Program Solution

For dynamic programs, we do not want to be pseudocode. Rather, we want you to describe the details of solution within the context of the program.

Definitions: Describe all the definitions and preprocessing needed for the dynamic programming solution.

E.g. Weighted Interval Scheduling (WIS):

- Preprocessing: Sort the input by finish time.
- Definitions: For a given job at index j, let $i_j < j$ be the largest index such that $f_i \leq s_j$.

Matrix/table description: Describe the dimensions of the matrix, the contents of a cell, and any cells that can be initialized.

E.g. WIS: A 1D array M, where $M[j]$ is the maximum value of a compatible schedule for the first j items in the sorted input. Initialize $M[1] = v_1$.

Bellman Equation: The recursive definition for populating a cell in the table.

E.g WIS: $M[j] = \max\{M[j-1], M[i_j] + v_j\}$.

How to populate: Describe the order in which the cells of the matrix will be populated.

E.g. WIS: Populate from 2 to n.

Solution: Describe where to find the solution in the matrix, or how to calculate the solution.

E.g. WIS: The maximum value of a compatible schedule for the n jobs is found at $M[n]$.

Dynamic Program Correctness

Soundness: (Strong) induction over the order of the population of cells of the solution matrix:

1. Show that the base case(s) is correct.
2. Assume that the recursive calls return the correct value.
3. Given the assumption above, show that the Bellman equation will calculate the correct value.

Completeness: Usually immediate from the definition of the dynamic program.

Dynamic Program Runtime Analysis

- Preprocessing time, and
- Time to populate the matrix: Number of cells $\times$ Time to calculate Bellman equation, and
- Time to calculate the final solution.

E.g. WIS: $O(n \log n)$

- Preprocessing: $O(n \log n)$
- Populate the matrix: $O(n) \times O(\log n) = O(n \log n)$
- Final solution: $O(1)$

CS577: Introduction to Algorithms

Discussion 9 and 10: Network Flow

Flow Network

- A directed graph $G = (V, E)$, where $|V| = n$ and $|E| = m$.
  - Each edge $e \in E$ has a *capacity* $c_e \geq 0$.
  - A *source* node $s \in V$, and a *sink* node $t \in V$.
  - All other nodes ($V \setminus \{s, t\}$) are internal nodes.
  - Let $C = \sum_{e \text{ out of } s} c_e$.
- *Flow function*: $f : E \to R^+$, where $f(e)$ is the flow across edge e.
  - Valid flow function conditions:
    - i *Capacity*: For each $e \in E, 0 \leq f(e) \leq c_e$.
    - ii *Conservation*: For each $v \in V \backslash \{s, t\}$, $\sum_{e \text{ into } v} f(e) = f^{\text{in}}(v) = f^{\text{out}}(v) = \sum_{e \text{ out of } v} f(e)$.
- The flow starts at s and exits at t.
- Flow value $v(f) = f^{\text{out}}(s) = f^{\text{in}}(t)$.

An $s - t$ Cut

- A Cut: Partition of V into sets (A, B) with $s \in A$ and $t \in B$.
- Cut capacity: $c(A, B) = \sum_{e \text{ out of } A} c_e$.

Max-Flow = Min-Cut

- *Max-Flow*: Given a flow network G, the max-flow is the flow function f that maximizes $v(f)$.
- *Min-Cut*: Given a flow network G, the min-cut is the $s-t$ cut (A, B) that minimizes $c(A, B)$.
- For a flow network G, let f^* be the maximum flow function, and let (A^*, B^*) be the minimum cut, then $v(f^*) = c(A^*, B^*)$.

Max-Flow / Min-Cut Algorithms

Residual Graph: Given a flow network G and a flow f on G, we define the residual graph G_f:

- Same nodes as G.
- For edge (u, v) in E:
  - Add edge (u, v) with capacity $c_e - f(e)$.
  - Add edge (v, u) with capacity $f(e)$.

Ford-Fulkerson Method: $O(mC)$

- Initialize $f(e) = 0$ for all edges.
- While G_f contains an augmenting path P:
  - Update flow f by $\textsc{bottleneck}(P, G_f)$ along P.

Other algorithms:

- Scaled Ford-Fulkerson: $O(m^2 \log C)$.
- Fewest Edges Augmenting Path [BFS] (Edmonds-Karp): $O(m^2 n)$.
- Dinitz 1970: $O\left(\min\left\{n^{\frac{2}{3}}, m^{\frac{1}{2}}\right\} m\right)$.
- Preflow-Push 1974/1986: $O(n^3)$.
- Best: Orlin 2013: $O(mn)$

Network Flow Reductions

Many problems can be *reduced*[1] efficiently to a network flow whose solution (max-flow/min-cut) and the resulting flow function solves the original problem.

Reduction to max-flow problems:

- How can the problem be encoded in a graph?
- Source/sink: Are they naturally in the graph encoding, or do additional nodes and edges have to be added?
- For each edge: What is the direction? Is it bi-directional? What is the capacity?

[1]A reduction is a transformation of any instance of one problem $\mathcal{P}$ into an instance of another problem $\mathcal{Q}$, where the solution to the instance of $\mathcal{Q}$ gives a solution to the original instance of $\mathcal{P}$.

Solutions via max-flow reductions:

- Drawing the flow network can be helpful, but a solution must describe all the details of the reduction to a network flow graph.
- Bullet points are a great way to enumerate all the steps of the reduction.
- An efficient solutions requires:
  - An efficient reduction to a network flow graph (the size of the graph cannot have more than a polynomial number of nodes and edges as a function of the original problems input size).
  - Taking the network flow solution and deriving the solutions to the original problem must also be efficient.
- The running time of a network flow solution is determined by:
  - the time to construct the network flow graph for a given an instance of the original problem,
  - the time to solve a the max-flow using one of the known max-flow algorithms, and
  - the time to determine the solution to the original problem based on the max-flow solution.

Network Flow Extensions

The network flow extensions of *node demands* and *capacity lower bounds* can make for more natural reductions. Note that it does not add anymore computational power to the model. That is, a problem with a reduction to a network flow graph with demands and lower bounds has a reduction to a standard network flow problem.

Flow network with node demands:

- Each node has a demand d_v:
  - if $d_v < 0$: a source that demands $f^{\text{in}}(v) - f^{\text{out}}(v) = d_v$.
  - if $d_v = 0$: internal node ($f^{\text{in}}(v) - f^{\text{out}}(v) = 0$).
  - if $d_v > 0$: a sink that demands $f^{\text{in}}(v) - f^{\text{out}}(v) = d_v$.
- S is the set of sources ($d_v < 0$).
- T is the set of sinks ($d_v > 0$).
- Flow conditions:
  - i Capacity: For each $e \in E$, $0 \leq f(e) \leq c_e$.
  - ii Conservation: For each $v \in V$, $f^{\text{in}}(v) - f^{\text{out}}(v) = d_v$.

- Goal is *feasibility*: Does there exist a flow that satisfies the conditions?
  - If there is a feasible flow, then $\sum_{v \in V} d_v = 0$.

(1) Reduction to max-flow from flow network with node demands:

- Super source s^*: Edges from s^* to all $v \in S$ with $d_V < 0$ with capacity $-d_v$.
- Super sink t^*: Edges from all $v \in T$ with $d_V > 0$ with capacity d_v to t^*.
- Maximum flow of $D = \sum_{v:d_v>0 \in V} d_v = \sum_{v:d_v<0 \in V} -d_v$ in G' shows feasibility.

Flow network with node demands and capacity lower bounds:

- Node with demands as described above.
- For each edge e, define a lower bound ℓ_e, where $0 \leq \ell_e \leq c_e$.

  i Capacity: For each $e \in E$, $\ell_e \leq f(e) \leq c_e$.

  ii Conservation: For each $v \in V$, $f^{\text{in}}(v) - f^{\text{out}}(v) = d_v$.
- Goal is *feasibility*: Does there exist a flow that satisfies the conditions?

(2) Reduction to flow network with node demands from flow network with node demands and capacity lower bounds:

- Consider an f_0 that sets all edge flows to ℓ_e: $L_v = f_0^{\text{in}}(v) - f_0^{\text{out}}(v)$.
  - if $L_v = d_v$: Condition is satisfied.
  - if $L_v \neq d_v$: Imbalance.
- ¡2-¿ For G':
  - Each edge e, $c'_e = c_e - \ell_e$ and $\ell_e = 0$.
  - Each node v, $d'_v = d_v - L_v$.

Then, using the (1) reduction, you can obtain a max-flow network that you can solve to check for feasibility.

CS577: Introduction to Algorithms

Discussion 11 and 12: Intractability

Polynomial-Time Reductions

$Y \leq_p X$

- Consider any instance of problem Y.
- Assume we have a black-box solver for problem X.
- Efficiently transform an instance of problem Y into a polynomial number of instances of X that we solve via black-box solver for problem X, and aggregate the solutions efficiently to solve Y.

Corollaries from polynomial time reductions:

- Suppose $Y \leq_p X$. If X is solvable in polynomial time, then Y can be solved in polynomial time.
- Suppose $Y \leq_p X$. If Y cannot be solved in polynomial time, then X cannot be solved in polynomial time.
- Suppose $Z \leq_p Y$ and $Y \leq_p X$. Then, $Z \leq_p X$.

Intractability Definitions

Decision Problems

- Binary output: yes / no answer.
- Our complexity definitions assume decision problem versions of the problems.
- No less powerful: we can go between decision and optimization version of problems.

Easy vs Hard Problems

- *Easy Problems*: problems that can be solved by efficient algorithms.
- *Hard Problems*: problems for which we do not know how to solve efficiently.

Input Formalization

- Let s be a binary string that encodes the input.
- $|s|$ is the length of s, i.e., the # of bits in s.

Complexity class: P

- Polynomial run-time: Algorithm A has a *polynomial run-time* if run-time is $O(\mathsf{poly}(|s|))$ in the worst-case, where $\mathsf{poly}(\cdot)$ is a polynomial function.
- P is the set of all problems for which there exists an algorithm A that solves the problem with polynomial run-time.

Efficient Certification: Certifier $B(s,t)$ for a problem P:

- s is an input instance of P.
- t is a certificate; a proof that s is a yes-instance.
- Efficient: For every s, we have $s \in P$ iff there exists a t, $|t| \leq \mathsf{poly}(|s|)$, for which $B(s,t)$ returns yes.

Complexity class: NP

- Set of all problems for which there exists an efficient certifier $B(s,t)$.
- I.e., the set of all problems for which it is efficient to verify a potential solution.
- **N**on-deterministic, **P**olynomial time: can be solved in polynomial time by testing every certificate (t) simultaneously (non-deterministic).

Complexity class: NP-Hard: Problem X is NP-Hard if:

- For all $Y \in \mathsf{NP}$, $Y \leq_p X$.
- NP-Hard problem may or may not be in NP.

Complexity class: NP-Complete: Problem X is NP-Complete if:

- For all $Y \in \mathsf{NP}$, $Y \leq_p X$.
- X is in NP.

Showing that Problem X is NP-Complete: Cook Karp Reduction

Step 1: Prove that $X \in \mathsf{NP}$.

(a) Define a certificate (t) for X.

(b) Define an efficient certifier (algorithm) $B(s,t)$ for X and t as defined in (a).

Step 2: Choose a problem $Y \in \mathsf{NP}$-Complete.

- Y must be a problem that is known to be NP-Complete.
- It will be used to show that $Y \leq_p X$ in step 3:
  - Since $Y \in \mathsf{NP}$-Complete, then all NP problems $\leq_p Y$.
  - Therefore, showing $Y \leq_p X \implies$ all NP problems $\leq_p X \implies X \in \mathsf{NP}$-hard.

Step 3: $Y \leq_p X$ ($X \in \mathsf{NP}$-hard).

- Karp Reduction: For an arbitrary instance s_Y of Y, show how to construct, in polynomial time, an instance s_X of X such that s_y is a yes iff s_x is a yes.
  Steps:
  (a) Provide efficient reduction.
  (b) Prove $\Rightarrow$: if s_Y is a yes, s_X is a yes.
  (c) Prove $\Leftarrow$: if s_X is a yes, then s_Y had to have been a yes.

CS577: Introduction to Algorithms

Discussion 13: Randomization

Randomized Algorithms

- Algorithm flips a coin to make some decisions.
- Non-Deterministic: simultaneously considers multiple algorithms weighted by the probability distribution.

Types of Randomized Algorithms:

- *Monte Carlo*: With probability p returns the correct answer:
  - Run multiple times to boost the probability of correct answer.
  - Provide an approximation guarantee in expectation.
- *Las Vegas*: Has a run-time that is polynomial in expectation. It will always returns the correct solution, or informs about failure.
- *Atlantic City*: Probabilistic run-time and correctness.

Probability Definitions

Probability Space:

- *Sample space* Ω of all possible outcomes.
  - Can be infinite, but we will focus on finite.
  - Ex: 6-sided fair die (D6): $\Omega = \{1, 2, 3, 4, 5, 6\}$.
- *Probability mass*: each $i \in \Omega$ has a nonnegative probability mass: $1 \geq p(i) \geq 0$.
- Total probability mass is 1: $\sum_{i\in\Omega} p(i) = 1$.

Probability Event:

- An event ε is a set of outcomes of Ω.
- $\Pr[\varepsilon] = \sum_{i\in\varepsilon} p(i)$.
- Note: $\Pr[\bar{\varepsilon}] = 1 - \Pr[\varepsilon]$

Conditional Probability: Probability of ε given $\mathcal{F}$.

$$\Pr[\varepsilon|\mathcal{F}] = \frac{\Pr[\varepsilon \cap \mathcal{F}]}{\Pr[\mathcal{F}]}$$

Independent Events:

- Events ε and $\mathcal{F}$ are independent if $\Pr[\varepsilon|\mathcal{F}] = \Pr[\varepsilon]$ and $\Pr[\mathcal{F}|\varepsilon] = \Pr[\mathcal{F}]$.
- This implies $\Pr[\varepsilon \cap \mathcal{F}] = \Pr[\varepsilon] \cdot \Pr[\mathcal{F}]$.
- Generalization: Say $\varepsilon_1, \varepsilon_2, \ldots, \varepsilon_n$ are independent.

$$\Pr\left[\bigcap_{i=1}^{n} \varepsilon_i\right] = \prod_{i=1}^{n} \Pr[\varepsilon_i]$$

Union Bound:

$$\Pr\left[\bigcup_{i=1}^{n} \varepsilon_i\right] \leq \sum_{i=1}^{n} \Pr[\varepsilon_i],$$

where equality *only* if events are mutually exclusive.

Random Variables:

- Technical: Given a probability space, a random variable X is a function from the sample space to the natural (finite – real if infinite) numbers, such that, for number j, $X^{-1}(j)$ is the set of all sample points taking the value j is an event.
- Informally: A random variable X takes on a value that depends on a random process.
- Ex: $\Pr[X = 1] = 1/6$, where X is a toss of a 6-sided die.

Expectation Definitions

Expected Value:

- "Weighted average value"
- $\mathbb{E}[X] = \sum_{j=0}^{\infty} j \cdot \Pr[X = j]$
- Ex: $\mathbb{E}[X] = \sum_{i=1}^{6} \frac{1}{6} \cdot i = 3.5$, where X is a toss of a 6-sided fair die.

Expectation Properties: Let X and Y be random variables, and a be a constant.

- Linearity of expectation:
  - $\mathbb{E}[X+Y] = \mathbb{E}[X] + \mathbb{E}[Y]$
  - $\mathbb{E}[aX] = a\,\mathbb{E}[X]$
- If X and Y are independent, $\mathbb{E}[XY] = \mathbb{E}[X]\,\mathbb{E}[Y]$.

Guarantee in Expectation: An algorithm that returns a solution that has a r approximation ratio in expectation:

$$\forall I, \mathbb{E}[\textsc{alg}(I)] \leq r \cdot \textsc{opt}(I) + \eta$$