nputational Intractability

Recall: an algorithm is efficient if it has a polynomial running time.
Certain problems are extremely hard and cannot be solved by efficient algorithms.
We do **not** know any polynomial time algorithms for these problems, and we **cannot** prove that no polynomial time-algorithm exists.
A large class of these problems has been characterized and has been proven to be equivalent in the following sense: a polynomial-time algorithm for any one of them would imply the existence of a polynomial time algorithm for all of them. These problems are known as the NP-Complete problems.

omial-Time Reduction: is the basic technique that we will use to explore the space of itationally hard problems. Using reduction, we can formally express statements like, em X is at least as hard as problem Y."

Definition of Reduction: Let X and Y be two problems.
$Y \leq_P X$ (meaning Y can be reduced to X in polynomial time)
if and only if an arbitrary instance of problem Y can be solved using a polynomial number of standard computational steps, plus a polynomial number of calls to a *black box* which solves X i.e. how many times you call the black box
Explanation of the definition: To solve an instance of Y, you can do polynomial amount of work (regular kind of algorithm to create an instance of X) but you are able to call a black box that can solve instances of X.

- The black box is sometimes called the *oracle* -- is not a realistic model of computation.
  - The oracle submits the question and receives the answer
    - Question must be asked in a "yes" or "no" format
      - What we call a *decision version* of a problem
        - Instead of returning the complete solution we simply return whether a solution exists or not

'If and only if' here doesn't mean you need to do an opposite reduction from X to Y. It means if you have a reduction R from instance I_y of problem Y to instance I_x of problem X, you need to argue for both direction of the following statement:

$$Y(I_Y) = true \leftrightarrow X(R(I_Y)) = true$$

(in other words,
1) Assume there exists an instance of Y, and we transform that instance using a reduction algorithm R. Show that this transformed into an instance of X.
2) Assume we have an instance of Y that is already built using R, and that it is also an instance of X. Show that it is also an instance of Y.

Other notes about $Y \leq_p X$

- This means Y is polynomially-reducible to X
- Also means X is at least as hard as Y
- Also means Y can be solved using a polynomial number of steps plus a computational number of calls to X's black box.

Lemma 8.9: If $Z \leq_p Y$, and $Y \leq_p X$, then $Z \leq_p X$

- Once again this intuitively makes sense since the time it takes to reduce Z to X would be the time taken to reduce Z to Y compounded with the time taken to reduce Y to X.
  - i.e. suppose you can reduce Z to Y using the function $f(|Z|)$ and that you can reduce Y to X using the function $g(|Y|)$. Therefore, to reduce Z to X you could simply call $f(g(|Y|))$

P, NP, and NP-Complete

$\mathcal{P}$ – A class of problems which can be solved in polynomial time. The set of all decision problems which are solvable by an algorithm[1] whose worst case running time is bounded by some polynomial function of the input size
Unfortunately, not all problems are solvable by a polynomial time algorithm. Therefore, they are not in class P. For example, Independent Set, Vertex Cover, and 3-SAT. There are many problems that are not known to be in P.

$\mathcal{NP}$ –The problems which can be *verified* using a polynomial time algorithm

- Verifiable means that given a certificate (i.e. a solution), we could certify (verify) that the certificate is correct in polynomial time

Lemma 8.10: $\mathcal{P} \subseteq \mathcal{NP}$

- Any problem which can be solved in polynomial time can be verified in polynomial time.

Lemma 8.11: Is there a problem in $\mathcal{NP}$ that does not belong to $\mathcal{P}$? Does $\mathcal{P} = \mathcal{NP}$?

- We don't know
- General belief is that $\mathcal{P} \neq \mathcal{NP}$ – though there is no actual evidence to support this
- $\mathcal{NP}$ = Non-Deterministic Polynomial (Does NOT stand for Non-Polynomial!)

In the absence of progress on the P VS NP question, people turned to a realted question but more approachable question: **what are the hardest problems in NP?**
$\mathcal{NP}$-Complete–A sub-class of $\mathcal{NP}$, the Hardest problems in $\mathcal{NP}$.

- A set of hard problems such that all other $\mathcal{NP}$ problems can be reduced to these in polynomial time.
- A problem X is $\mathcal{NP}$-Complete if and only if
  - $X \in \mathcal{NP}$
    - Can be verified in polynomial time
  - For all $Y \in \mathcal{NP}, Y \leq_p X$
    - i.e. Every problem in $\mathcal{NP}$ can be reduced to X -- X is very expressive. If you have a problem $Y \in \mathcal{NP}$, then you can solve it using a black box for problem X.

Lemma 8.12: Suppose X is an $\mathcal{NP}$-Complete problem. Then X is solvable in polynomial time if and only if $\mathcal{P} = \mathcal{NP}$.
Proof:
→ : Suppose $\mathcal{P} = \mathcal{NP}$, then X can be solved in polynomial time since it belongs to $\mathcal{NP}$ (recall that any $\mathcal{NP}$ problem is reducible in polynomial time to any given $\mathcal{NP} - complet$ problem)
← : Suppose X is solvable in polynomial time. Given that X is also an $\mathcal{NP}$-Complete problem, take an arbitrary problem Y ∈ $\mathcal{NP}$, then by definition Y $\leq_P$ X. This means that Y can also be solved in polynomial time -- as stated by lemma 8.1. ■

Note :
$\mathcal{NP}$-Hard problems are at least as hard as $\mathcal{NP}$ problems

- Do not have to be $\mathcal{NP}$
- Do not have to be decision problems.

e.g., $\mathcal{NP}$ problems has to be decision problems: (yes/no) answer. The independent Set algorithm finds the largest IS is not in $\mathcal{NP}$ (i.e., taking a graph without a target and find the largest IS -- the output is a number). If you solve this problem, then you can solve the decision version (i.e., if you can answer the question: "what is the largest IS?, you can answer the question "is there an IS of size k?"

How to show that a problem $X \in \mathcal{NP}$-Complete?

- show that X ∈ $\mathcal{NP}$
- Choose a known $\mathcal{NP}$-Complete problem Y and show that $Y \leq_P X$

For example, we have to use one known $\mathcal{NP}$-Complete problem such as 3-SAT and we want to use it to show that the problem X is $\mathcal{NP}$-Complete. We know that every problem in $\mathcal{NP}$ can be reduced in polynomial time to 3-SAT. The 3-SAT is $\mathcal{NP}$-complete. If $3 - SAT \leq_P X$, then by transitivity all $\mathcal{NP}$ problems can be reduced to X.
Consequence of Lemma 8.12: If there is any problem in $\mathcal{NP}$ that cannot be solved in polynomial time, then no $\mathcal{NP}$-Complete problem can be solved in polynomial time massive effort has been invested to come up with a polynomial time algorithm for $\mathcal{NP}$-Complete problems and everybody has failed so far. , Independent Set, Vertex Cover, and 3-SAT are all $\mathcal{NP}$-Complete problems.
If a person worked on SAT trying to find a polynomial time algorithm and failed and another person worked on Independent Set trying to find a polynomial time algorithm and failed. The result is two separate effort provides a combined evidence that $\mathcal{P} \neq \mathcal{NP}$.
If you have a polynomial time algorithm for one $\mathcal{NP}$-Complete problem, you have a polynomial time algorithm for all $\mathcal{NP}$-Complete problems. So, either every $\mathcal{NP}$-Complete problem can be solved in polynomial time or none can.
Need to show the greedy solution is no more than a factor of 2 from the optimal solution ($T \leq 2T^*$)
Therefore…

- Let T = The makespan of our greedy solution
- Let T^* = The makespan of the optimal solution
  - We know that the optimal makespan must be at least:
    $$T^* \geq \frac{1}{m}\sum_j t_j$$
    Since the value of the makespan equals the max value over all machines.
  - However, this formula is not very useful if you have one extremely long job in relation to all the shorter jobs.
    - Need a different lower bound for T^* to reflect this however
    - In this case the optimal makespan is at least
      $$T^* \geq \max_j t_j$$
- Note: We need the two formulas mentioned above for our proof

For minimization problems, we need to show the greedy solution is no more than a factor of k from the optimal solution ($T \leq kT^*$). I.e., $\frac{T}{T^*} \leq k$ -we tried to minimize but we were not good enough (T was bigger).
For maximization problems, we need to show the greedy solution is no less than a factor of $\frac{1}{k}$ from the optimal solution ($T \geq \frac{T^*}{k}$) . I.e., $\frac{T^*}{T} \leq k$ – we tried to maximize but our solution wasn't quite big.
Def: Given a set of *inequalities* that represent *constraints*, our **goal** is to minimize or maximize a *certain quantity* represented by an equation
The basic procedure in Linear programming involves

- Graph the region bounded by the inequalities
- Find vertices that are corner points in the feasible region
- Plug the values of the corner points into the equation that you are trying to minimize or maximize

Linear Programming:
Definition: Given a $M \times N$ matrix A and vectors $b \in \mathbb{R}^m$ and $c \in \mathbb{R}^n$, find a vector $x \in \mathbb{R}^n$ to solve the following optimization problem:

$$\min(c^t x \; such \; that \; x \geq 0 \; and \; Ax \geq b)$$

$$e.g., A = \begin{bmatrix} 1 & 2 \\ 2 & 1 \end{bmatrix} \; x = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \; b = \begin{bmatrix} 6 \\ 6 \end{bmatrix} c = \begin{bmatrix} 1.5 \\ 1 \end{bmatrix}$$

- Note:
  - $c^t x$ is called objective function
  - $Ax \geq b$ represents the constraints
  - $c^t x$ (*inner product* or *dot product* of the transpose of c^t and x).

SPACE

So far, we have considered **time** as a computational resource. We defined an algorithm to be efficient if it's running time is polynomial with respect to the size of the input. Now, we will introduce a class of problems defined by treating **space** as the fundamental computational resource.

- **PSPACE:** The set of all computational problems that can be solved by an algorithm with polynomial space complexity (i.e., an algorithm that uses an amount of space that is polynomial in the size of its input).

- Lemma 9.1: $\mathcal{P} \subseteq PSPACE$
  - In polynomial time, an algorithm can consume only a polynomial amount of space (i.e., O(1) space per time unit).

- **e.g., Binary counter:** count from 0 to 2^n-1 in binary.
- Algorithm: use n bit odometer
- The algorithm runs for an **exponential amount of time**, then halts.
- In the process, it uses only a **polynomial amount of space**.
- Space can be reused during a computation in ways that time, by definition, cannot.

- Lemma 9.2: 3-SAT is in $PSPACE$.
- **Pf:**
  - enumerate all 2^n possible truth assignments using a counter.
  - The values in the counter corresponds to truth assignments, i.e., x_i has the truth value of bit i.
  - For each truth assignment, we only need a polynomial amount of space to plug each assignment into the set of clauses and see if it satisfies them.

A *Turing Machine* (TM) is an abstract model of computation invented by Alan Turing in 1936.

A Turing Machine can do everything that a real computer can do

A TM has a **control** that contains a **state** and **transitions**. It has also a **tape** that has an end on the left side but infinite on the right side.

The input is initially written on the tape but the TM can also write onto the tape (it's a read/write tape).

The TM has a read/write head which is located over some particular character on the input tape. In a single move, the TM can choose to write or not, and can move left or right. The TM moves on the tape but it's bidirectional.

Key Features:

1. Contains an **infinite tape**-contains an input string and is blank everywhere
2. Has a **read/write head**
   - After reading a character the read/write head must move left or right
   - Special case—won't move if the head is at the end of the tape
3. Has **a control**—contains a program specified by a finite number of instructions (state + transitions) - the control represent the program in modern computers.
   - Note: The control is an example of a *finite automaton* (a state transition without a tape)

Note that the TM starts from a **start state**, continue computing until reaches an **accept state** or **reject state**. If the machine does not enter an accepting state or reject state, it will go for ever (never halting).

Σ—is the input alphabet. The input alphabet is the set of symbols that can appear in the input. E.g. {0, 1}

Q— is the set of possible states that the TM can go through. E.g. $\{q_0, q_1, \dots, q_n, q_{accept}, q_{reject}\}$

Γ—is the tape alphabet. The tape alphabet is the set of symbols that can be written and read on tape. So, Γ=ΣU{⊔}. The tape alphabet includes the blank symbol, ⊔.

δ— is a transition function. It's a mapping based on present state, Q, and present tape alphabet, Γ, (pointed by head pointer), it will move to new state, Q, change the tape symbol, Γ, (may or may not) and move head pointer to either left or right: $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$

(1) the transition rules are deterministic, i.e., when the TM reads a particular symbol when it is at a particular state, it moves to a particular state, write a particular symbol, and moves to a particular direction (i.e., either left or right).

(2) To reduce clutter when drawing a TM, typically the transitions tat leads to the accept state are shown.

Other things to note about Turing Machine Diagrams

- The L/R which appears after the arrow denotes the direction the read/write head moves
- In the following example, 0 →⊔, R, the character "0" followed by an arrow indicates that whenever we **read** a "0" then the head will write a "⊔" to the current cell and moves one cell to the right.

Probabilistic analysis: The number of comparisons in an execution is a Random variable for an input of size n.

Run the **RandomizedQuicksort** algorithm once on an input array, S, and obtain certain number of comparisons. Run it again on the same array S and can obtain a different number of comparisons. The number of comparisons is a Random Variable and the behavior of the algorithm is non-deterministic .

A *random variable* X is a variable whose possible values are the outcomes a set of random experiments.

A random variable typically associates a number (probability) with each outcome of the sample space.

The *probability distribution* of a random variable is a list of probabilities associated with each outcome of the sample space.

Random variables can be either discrete or continuous.

The expectation of a Random variable, X, denoted $E(X)$,is the average value taken by that random variable.

- **Expectation of Discrete Random Variables:**

Let X be a discrete random variable with

- PMF (Probability Mass Function) $P_X(x)$ and
- has a support x

Then the expectation of the random variable is denoted by

$$\mu = E\{X\} = \sum_{x \in x} x P_X(x)$$

Let Y be a continuous random variable with

- PDF (Probability Density Function) $f_Y(y)$ and
- has a support y
- Then the expectation of the random variable is denoted by

$$\mu = E\{Y\} = \int_{y} y f_{Y(y)dy}$$

Quick sort (QS) is a divide-and-conquer sorting algorithm.

Input: an array of n distinct numbers (S).

Output: sorted array

Process: pick a pivot p, divide (reorder) the elements in-place such that all elements $< p$ appear before p, and all elements $> p$ appear after p, apply this to the sub-arrays in the partition, until their sizes become one, and finally conquer step is trivially since ultimately all elements will be sorted in place.

Note: In the basic QS, the pivot is chosen to be either S[0] or S[n].

We need to find: E[**The number of comparisons in an execution**]

Definition: Let $S_{(i)}$ be the i^{th} smallest element in the set (array) S.

Definition: The Random variable X_{ij} is defined as follows.

$$X_{ij} = \begin{cases} 1, & S_{(i)} \text{ is compared to } S_{(j)} \text{ in an execution of the algorithm} \\ 0, & otherwise \end{cases}$$

Note: Comparisons are conducted after the algorithm picks a pivot. For $S_{(i)}$ or $S_{(j)}$ to be compared with each other in an iteration of the algorithm, one of them has to be the pivot.

E[The number of comparisons in an execution]=
$E[\sum_{i=1}^{n} \sum_{j>i} X_{ij}]$
(i and j pairs will be enumerated once in the summation)

By linearity of expectation (expectation adds)
$\sum_{i=1}^{n} \sum_{j>i} E[X_{ij}]$
Remember $E[X] = \sum XP(X)$
(i.e., enumerate all values of X, which is 0 and 1 in this example, multiply them by some corresponding probability)

Definition: probability that $S_{(i)}$ is compared to $S_{(j)} \equiv P_{ij}$
$E[X_{ij}] = 1.P_{ij} + 0.(1 - P_{ij}) = P_{ij}$

$\sum_{i=1}^{n} \sum_{j>i} E[X_{ij}] = \sum_{i=1}^{n} \sum_{j>i} P_{ij}$