Topic 1- algorithms

Sub-topic 1- Euclid's algorithm for GCD

- **Algorithm 2:** $Euclid(x, y)$: (when $x > y \geq 0$)
  - if $y = 0$ return x
  - If $y = 1$ return 1
  - return $Euclid(y, x \bmod y)$

Runtime can be analyzed using the potential method

In this case, define potential s_i = x + y (the sum of the inputs at iteration i). This strictly decreases each iteration. More specifically, each iteration is less than or equal to ¾ the potential of the previous iteration. So, the runtime is O(log(x + y)) = O(n), linear on the size of x, where n is the number of bits required to represent x in binary.

Sub-topic 2- Divide and Conquer

General form:

1. Divide the problem into smaller subproblems
2. Solve each problem recursively
3. Combine the solutions of the subproblems in a "meaningful" way

Use the Master Theorem to analyze runtime:

Story: Divide-and-conquer algorithm breaks a problem of size n into:

- k smaller problems
- each one of size n/b
- with cost of $O(n^d)$ to combine the results together

Formally: Consider the recurrence relation $T(n) = kT(n/b) + O(n^d)$, when $k, b > 1$. Then:

$$T(n) = \begin{cases} O(n^d) & \text{if } (k/b^d) < 1 \\ O(n^d \log n) & \text{if } (k/b^d) = 1 \\ O(n^{\log_b k}) & \text{if } (k/b^d) > 1 \end{cases}$$

Example: integer multiplication. Reduce from O(n^2) to:

- **Input:** N_1 and N_2, two n-digit numbers (assume n is a power of 2)
- Split N_1 and N_2 into $n/2$ low-order digits & $n/2$ high-order digits:
  - $N_1 = a \cdot 10^{n/2} + b$
  - $N_2 = c \cdot 10^{n/2} + d$

	←$n/2$ digits→	←$n/2$ digits→
N_1	a	b
N_2	c	d

- Compute $N_1 \times N_2 = a \times c \cdot 10^n + (a \times d + b \times c) \cdot 10^{n/2} + b \times d$
  - $m_1 = (a + b) \times (c + d)$ time: $O(n) + T(n/2)$
  - $m_2 = a \times c$ time: $T(n/2)$
  - $m_3 = b \times d$ time: $T(n/2)$
  - **Return:** $m_2 \cdot 10^n + (m_1 - m_2 - m_3) \cdot 10^{n/2} + m_3$. time: $O(n)$
- $T(n)$ = time to multiply two n-digit numbers
  - $T(n) = 3T(n/2) + O(n) \Longrightarrow k = 3, b = 2 \Longrightarrow$
    $T(n) = O(n^{\log_2 3}) = O(n^{1.585})$.

Sub-topic 3- Dynamic Programming

Break a complex problem into smaller, easier subproblems subject to:

1. Principle of optimality- substructure of an optimal structure is itself optimal
2. Overlapping subproblems

Efficiency: top-down recursive (exponential) vs. top-down memoization (polynomial) vs. bottom-up table (polynomial)

Example:

- Let $LCS(i, j)$ denote the length of a longest common subsequence of $X[1..i]$ and $Y[1..j]$.
- Then:

$$LCS(i,j) = \begin{cases} 0 & i = 0 \text{ or } j = 0 \\ 1 + LCS(i-1, j-1) & X[i] = Y[j] \\ \max\begin{Bmatrix} LCS(i-1, j), \\ LCS(i, j-1) \end{Bmatrix} & X[i] \neq Y[j] \end{cases}$$

Sub-topic 4- Greedy algorithms

Kruskal's algorithm- finds a minimum spanning tree of a graph by sorting edges by weight, and keep adding the cheapest edge that doesn't create a cycle

Topic 2- computability- which problems are solvable by a computer?

DFA- deterministic finite automaton; only move right, move through the entire string, and see if you end on an accepting state or not. Will halt for every input.

Turing machine- can move right or left, with only one accept/reject state. You also overwrite the character you're looking at. Might loop forever.

A language is a set of strings.
All the strings in a language must be of finite length, but a language can contain infinite strings.

A program M decides for language A if given string x as input:

- if x is an element of A, M accepts x
- if x is not an element of A, M rejects x

In this case, M is called a decider for A, and A is called a decidable language. M always halts.
The language of M is L(M) = {x: M accepts x}

M recognizes A if:

- if x is an element of A, M accepts x
- if x is not an element of A, M either rejects or loops on x

In this case, M is called a recognizer, and may not always halt.

<M> = the source code of machine M; is a string. Has a finite length.

There are a countably infinite number of Turing machines, but an uncountably infinite number of languages; since each Turing machine decides at most one language, there are undecidable languages.

Some problem A is said to be Turing reducible to another problem B given that if B is decidable, then A is decidable.
We can prove the halting problem
L_HALT = {(<M>, x): machine M halts on input x}
or the barber problem
L_BARBER = {<M>: machine M does not accept <M>}
is undecidable by testing whether a hypothetical decider for the languages is an element of the language, and thus show a contradiction.
Other undecidable problems can be proven to be undecidable by showing that an already proven undecidable problem reduces to it. Example:
L_ACC = {(<M>, x): machine M accepts input x}
Is undecidable because the barber problem reduces to it; i.e if there existed a decider for L_ACC, we could construct a decider for the undecidable L_BARBER
L_ACC reduces to L_HALT.