Divide and Conquer

High Level Approach:
1. Break a problem into subproblems
2. Recursively solve these subproblems
3. Combine the results

Master Theorem: Useful if solving a size-n problem requires recursively solving a size- \( n^d \) subproblems and \( O(n^d) \) work to prepare/combine the results from the subproblems. In recurrence form:

\[
T(n) = \begin{cases} 
O(n^d) & \text{if } d \geq \log_2(n) \\
O(n^d \log n) & \text{if } d = \log_2(n) \\
O(n^{1+\varepsilon}(a)) & \text{if } d < \log_2(n) 
\end{cases}
\]

MergeSort

Main Idea: Break down the list into several sublists until each sublist consists of a single element. Merge the sublists to get the sorted list.

\[
def \text{mergesort}(a[1...n]): \\
\text{if } (n > 1):\n\text{return merge(mergesort(a[1...n/2]), mergesort(a[n/2+1...n]))} \\
\text{else: return a}
\]

Recurrence: \( T(n) = 2T(n/2) + O(n) \). 
Runtime: \( T(n) = O(n \log n) \).

Fast Fourier Transform

Goal: Given a degree \( d \) polynomial \( P(x) = p_0 + p_1 x + \ldots + p_d x^d \) evaluate \( P \) on the \( n \)th roots of unity, \( \omega_1, \omega_2, \ldots, \omega_n^{-1} \), where \( \omega_n = e^{2 \pi i / n} \) (\( n \) is the smallest power of 2 greater than or equal to \( d + 1 \)).

Main Idea: Let \( E(x) = p_0 + p_1 x + p_2 x^2 + \ldots \) and \( O(x) = p_1 + p_2 x + p_3 x^2 + \ldots \). We can represent \( P(x) = E(x^2) + xO(x^2) \). The problem reduces to evaluating two degree \( n/2 \) polynomials, \( E \) and \( O \), on the \( n/2 \) roots of unity and combining the results.

Runtime: \( O(n \log n) \).

Matrix View: We speed up this computation (where \( n \) is a power of 2):

\[
\begin{bmatrix}
P(1) \\
P(\omega_n) \\
P(\omega_n^2) \\
\vdots \\
P(\omega_n^{n/2-1}) \\
\end{bmatrix} = 
\begin{bmatrix}
1 & 1 & \cdots & 1 \\
1 & \omega_n & \omega_n^2 & \cdots & \omega_n^{n/2-1} \\
1 & \omega_n^2 & \omega_n^4 & \cdots & \omega_n^{n/2} \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
1 & \omega_n^{n/2-1} & \omega_n^{n/2} & \cdots & \omega_n^{n-1} \\
\end{bmatrix} \cdot 
\begin{bmatrix}
p_0 \\
p_1 \\
p_2 \\
\vdots \\
p_{n/2-1} \\
\end{bmatrix}
\]

Inversion Formula: \( M_n(\omega)^{-1} = \frac{1}{n} M_n(\omega^{-1}) \)

Graph Traversals (DFS & BFS)

<table>
<thead>
<tr>
<th>Pseudocode</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>DFS</td>
<td>- Runtime: ( O(</td>
</tr>
<tr>
<td>BFS</td>
<td>- Runtime: ( O(</td>
</tr>
</tbody>
</table>

Types of Edges

Tree Edge: part of the DFS forest
Forward Edge: leads to a non-child descendant in the tree
Back Edge: leads to an ancestor in the tree
Cross Edge: leads to a node that’s neither descendant nor ancestor in the tree

Shortest Paths

Dijkstra’s Pseudocode

\[
def \text{dijkstra}(G, s, G) : \\
\text{for all } u \in V : \\
\quad \text{dist}(u) = \text{infinity} \\
\quad \text{prev}(u) = \text{nil} \\
\text{dist}(s) = 0 \\
\text{while } H \text{ is not empty:} \\
\quad \text{u = deletemin}(H) \\
\quad \text{prev}(u) = \text{nil} \\
\quad \text{dist}(u) = \text{infinity} \\
\quad \text{for all edges } (u, v) \in E : \\
\quad \quad \text{if dist}(v) > \text{dist}(u) + \text{l}(u, v) \\
\quad \quad \quad \text{dist}(v) = \text{dist}(u) + \text{l}(u, v) \\
\quad \quad \text{prev}(v) = u \\
\quad \text{decreasekey}(H, v) \\
\]

Bellman-Ford pseudocode

\[
def \text{bellman-ford}(G, l, s) : \\
\text{for each u in V in linear order:} \\
\text{dist}(u) = \text{infinity} \\
\text{while } H \text{ is not empty:} \\
\quad \text{u = deletemin}(H) \\
\quad \text{prev}(u) = \text{nil} \\
\quad \text{dist}(u) = \text{infinity} \\
\quad \text{for all edges } (u, v) \in E : \\
\quad \quad \text{if dist}(v) > \text{dist}(u) + l(u, v) \\
\quad \quad \quad \text{dist}(v) = \text{dist}(u) + l(u, v) \\
\quad \quad \text{prev}(v) = u \\
\quad \text{if } \text{dist}(v) > \text{dist}(u) + l(u, v) \\
\]

Strongly Connected Components (SCC)

To find an SCC, run DFS on \( G_{\text{Reverse}} \) to get the post-order of the graph Run DFS on \( G \) starting at the max post-order of \( G_{\text{Reverse}} \) (sink of \( G \)). When there are no more vertices to explore = one SCC. 
Runtime: \( O(|V| + |E|) \).
**Greedy Algorithms**

**High Level Approach:** Greedy algorithms make the locally optimal choice at each step. Hence, greedy algorithms work for problems where making locally optimal choices yields a global optimum.

**Minimum Spanning Trees (MST)**
- **Goal:** Given a weighted undirected graph $G = (V, E)$, find the lightest tree that connects all vertices $V$.
- **Cut Property:** The lightest edge across a cut is in some MST.
- **Main Idea:** Repeatedly add the next lightest edge that doesn’t produce a cycle.
- **Runtime:** $O(|E| \log(|V|))$.

**Prim’s Algorithm**
- **Main Idea:** On each iteration, pick the lightest edge that connects a vertex in the current subtree $S$ and a vertex outside $S$.
- **Runtime:** $O(|E| \log(|V|))$.

**Huffman Coding**
- **Goal:** Encode characters/frequencies $\{i, f_i\}$ to binary codes such that the encoding is maximal-efficient.
- **Main Idea:** At each step, find the two symbols with smallest frequency, say $i$ and $j$, and make $i$ and $j$ children of a new node $k$.
- **Performance:** Not optimal, but pretty good: $k_g \leq k_w \cdot \ln(n) + 1$, where $k_g$ is number of subsets output by greedy and $k_o$ is optimal.

**Dynamic Programming**

**High Level Approach:** Define subproblems s.t. the solution to a big problem can be easily derived from the solutions to its subproblems. Solve all subproblems from small to large, using results from previous subproblems to solve the current subproblem. Both recursion with memoization (top-down) and iteration (bottom-up) approaches exist.

**Shortest Path in a DAG**
- **Main Idea:** For every $v \in V$, define $dist(v)$ as the length of the shortest path from $s$ to $v$. Order of subproblems: topological order. Base cases: $dist(s) = 0$, and $dist(v) = \infty$ if $v \neq s$ is a source. To calculate the shortest path to $v$, we look at edges $(u, v)$ going into $v$, we use all the pre-computed $dist(u) + \text{length of the edge } (u, v)$ and find the minimum: $dist(v) = \min_{(u, v) \in E} \{dist(u) + l(u, v)\}$.

**Knapack**
- **Main Idea:** Define $f(i, u)$ as the max value achievable with a knapsack of a capacity $u$ and items $1,...,i$. The optimal solution to $f(i, u)$ has two cases: to include item $i$ or to not include it in the knapsack. If we include it, we need to subtract its capacity. Thus, the recurrence is:
  
  $$f(i, u) = \max(f(i-1, u), f(i-1, u-w_i) + v_i)$$

- **Runtime:** Storing $f(i, u)$ values in a 2-D array of $n + 1$ rows and $W + 1$ columns, the algorithm fills this array from left to right, with each entry taking $O(1)$ steps, giving total runtime of $O(nW)$.

**Tips**
1. Define an appropriate subproblem with relevant parameters, ensuring that the parameters fully determine the subproblem.
2. Given access to all previously solved subproblems, develop an appropriate relation to solve the current subproblem.
3. Test on smaller cases

**Max Flow**

**Network Flow**
- **Goal:** To find maximum flow from $s$ to $t$ in a directed graph $G$.
- **Residual graph:** A directed graph $R$ whose edge weight is denoted as residual capacity $r(u, v)$:
  - $r(u, v) = c(u, v) - f(u, v)$ if $(u, v) \in E$ and $f(u, v) < c(u, v)$
  - $r(u, v) = c(u, v) + f(v, u)$ if $(v, u) \in E$ and $f(v, u) > 0$
- **Kruskal’s Algorithm**
  - **Main Idea:** Repeatedly add the next lightest edge that doesn’t produce a cycle.
  - **Runtime:** $O(|E| \log(|V|))$.

**Ford-Fulkerson Algorithm**
- **Main Idea:** To include it, we need to subtract its capacity. Thus, the recurrence is: $f(u, v) = \min\{c(u, v) - f(u, v), r(u, v)\}$
- **Two cases:**
  - To include item $i$: $f(i, u) = \min\{c(i, u) - f(i, u), r(i, u)\}$
  - To include item $u$: $f(u, v) = \min\{c(u, v) - f(u, v), r(u, v)\}$
- **Main Idea:** For every $v \in V$, we define $dist(v)$ as the length of the shortest path to $v$. There are two cases: to include item $i$, or to not include it.
- **Runtime:** $O(|V| \cdot |E|)$ if using BFS to find paths.

**Max-Flow Min-Cut Theorem**
- **Explanation:** Back edges in residual graph undo suboptimal flows in some edges for future iterations.
- **Runtime:** $O(|V| \cdot |E|^2)$ using BFS to find paths.

**Linear Programming**

**High Level Approach:** Define your problem as an objective function with a set of linear constraints while following the given convention, and the Simplex algorithm does the rest. If a linear program has a bounded optimum, then so does it’s dual, and the two optimum values coincide.

**Generic Linear Program:**

<table>
<thead>
<tr>
<th>primal</th>
<th>dual</th>
</tr>
</thead>
<tbody>
<tr>
<td>max $c^T x$</td>
<td>min $y^T b$</td>
</tr>
<tr>
<td>$Ax\leq b$</td>
<td>$y^T A \geq c^T$</td>
</tr>
<tr>
<td>$x \geq 0$</td>
<td>$y \geq 0$</td>
</tr>
</tbody>
</table>

**Constraint Transformations:**

<table>
<thead>
<tr>
<th>Changing Objective</th>
<th>Inequality to Equality</th>
<th>Equality to Unrestricted</th>
</tr>
</thead>
<tbody>
<tr>
<td>max $c^T x = \min -c^T x$</td>
<td>$ax \leq b \rightarrow ax + s = b$, $s \geq 0$</td>
<td>$x \in R \rightarrow x = x^+ - x^-$</td>
</tr>
<tr>
<td>min $c^T x = \max -c^T x$</td>
<td>$s \geq 0$</td>
<td>$x \geq b$</td>
</tr>
</tbody>
</table>

**Zero-Sum Games**

**Setup:** Given a 2D matrix where the rows are player A’s moves, the columns are player B’s moves, and the values are A’s reward for each move. Whichever player goes first has to announce their strategy first.

**Strategy:** Vector of probability of playing specific moves for each player. Whichever player goes first has to announce their strategy first.

**Main Idea:** Player A’s strategy is $\{x_1, x_2\}$, Player B’s strategy is $\{y_1, y_2\}$.

**Example:**

<table>
<thead>
<tr>
<th>$B_1$</th>
<th>$B_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A_1$</td>
<td>$a$</td>
</tr>
<tr>
<td>$A_2$</td>
<td>$b$</td>
</tr>
<tr>
<td>$c$</td>
<td>$d$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$y_1$</th>
<th>$y_2$</th>
</tr>
</thead>
</table>

**A’s strategy is $\{x_1, x_2\}$**
- $B_1$'s strategy is $\{y_1, y_2\}$

**Minimax Theorem:**

- **Example:**

<table>
<thead>
<tr>
<th>$B_1$</th>
<th>$B_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A_1$</td>
<td>$a$</td>
</tr>
<tr>
<td>$A_2$</td>
<td>$b$</td>
</tr>
<tr>
<td>$c$</td>
<td>$d$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$y_1$</th>
<th>$y_2$</th>
</tr>
</thead>
</table>

- $A$ picks $\{x_1, x_2\}$ to maximize the value $\min\{az_1 + bz_2, cz_1 + dx_2\}$
- $B$ picks $\{y_1, y_2\}$ to minimize the value $\max\{ay_1 + cy_2, by_1 + dy_2\}$

**Strategy:** Vector of probability of playing specific moves for each player. Whichever player goes first has to announce their strategy first.

**Main Idea:** The optimal point is at an intersection of the geometrical representation of the constraints. If the chosen vertex is not optimal, then the objective function can be improved by following an outgoing edge from this vertex.