Div Conq

A typical Divide and Conquer algorithm solves a problem using the following three steps:

1. Divide the problem into a number of subproblems that are smaller instances of the same problem.
2. Conquer the subproblems by solving them recursively.
   - If the subproblem sizes are small enough, however, just solve the subproblems in a straightforward manner.
3. Combine the solutions to the subproblems into the solution for the original problem.

Divide and Conquer should be used when same subproblems are not evaluated many times. Otherwise Dynamic Programming or Memoization should be used.

Master Theorem

Theorem 1.

Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the non-negative integers by the recurrence

$$T(n) = aT(n/b) + f(n).$$

Then, $T(n)$ can be bounded asymptotically as follows.

1. *If $f(n) = O\left(n^{\log_b a - \epsilon}\right)$ for some constant $\epsilon > 0$, then $T(n) = \Theta\left(n^{\log_b a}\right)$.*
2. *If $f(n) = \Theta\left(n^{\log_b a}\right)$, then $T(n) = \Theta\left(n^{\log_b a} \lg n\right)$.*
3. *If $f(n) = \Omega\left(n^{\log_b a + \epsilon}\right)$ for some constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$.*

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2T(n/2) + \Theta(n) & \text{otherwise.} \end{cases}$$

Max sum Div & Conq = This recurrence has the solution $T(n) = \Theta(n \lg n)$.

Large Int Mul = $T(n) = 3T\left(\frac{n}{2}\right) + O(n) = O\left(n^{\log_2 3}\right) = O\left(n^{1.59}\right)$

Strassen Matrix Mul = $T(n) = 7T\left(\frac{n}{2}\right) + 18\left(\frac{n^2}{4}\right)$

$= \Theta\left(n^{\log_2 7}\right) \approx \Theta\left(n^{2.807}\right)$

$$T(n) = \begin{cases} 2T(n/2) + O(n) & \text{if } n > 3 \\ O(1) \text{ otherwise.} \end{cases}$$

Closest Pair = Thus, $T(n) = \Theta(n \lg n)$.

Solving DP

1. Set up a recurrence relating a solution to a larger instance to solutions of some smaller instances.
2. Solve smaller instance once.
3. Record solutions in a table.
4. Extract solution to the initial instance from that table.

Binomial Coeff. $= \Theta(nk)$.

	0	1	2	$\dots$	$k-1$	k
0	1					
1	1	1				
2	1	2	1			
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
k	1	$\dots$	$\dots$	$\dots$	$\dots$	1
$n-1$	1				$\binom{n-1}{k-1}$	$\binom{n-1}{k}$
n	1					$\binom{n}{k}$

Matrix Chain Mul. $= O\left(n^2\right)$.

Let $T(i,j)$ be the minimum number of $\times$'s used in multiplying $\mathbf{M}_i \cdot \mathbf{M}_{i+1} \cdots \mathbf{M}_j$, where i, j.

$$T(i,j) = \begin{cases} 0 & \text{if } i = j \\ \min_{k=i}^{j-1} \{T(i,k) + T(k+1,j) + p_{i-1} \times p_k \times p_j\} & \text{otherwise.} \end{cases}$$

0/1 Knapsack

Let $V[i,j]$ be the maximum value of object i, for $1 \leq i \leq n$, that can be fitted in a knapsack with capacity j.

$$V[i,j] = \begin{cases} \max\{V[i-1,j], V[i-1,j-w_i] + v_i\} & \text{if } j \geq w_i \\ V[i-1,j] & \text{if } j < w_i \end{cases}$$

Initial conditions: $V[0,j] = 0$ and $V[i,0] = 0$.

LCS

- Let $X = \langle x_1, x_2, \dots, x_m \rangle$, $Y = \langle y_1, y_2, \dots, y_n \rangle$. Let $Z = \langle z_1, z_2, \dots, z_k \rangle$ be an LCS of X and Y.
- If $x_m = y_n$, then $z_k = x_m = y_n$ and z_{k-1} is an LCS of $X[1 \dots m-1]$ and $Y[1 \dots n-1]$.
- If $x_m \neq y_n$, then $z_k \neq x_m$ implies that Z is an LCS of $X[1 \dots m-1]$ and $Y[1 \dots n]$. item If $x_m \neq y_n$, then $z_k \neq y_n$ implies that Z is an LCS of $X[1 \dots m]$ and $Y[1 \dots n-1]$.

$$LCS(m,n) = \begin{cases} LCS(m-1,n-1) + x_m & \text{if } x_m = y_n \\ \max\{LCS(m,n-1), LCS(m-1,n)\} & \text{otherwise} \end{cases}$$

Greedy

1. Determine the optimal substructure of the problem.
2. Develop a recursive solution.
3. Show that if we make the greedy choice, then only one subproblem remains.
4. Prove that it is always safe to make the greedy choice. (Steps 3 and 4 can occur in either order.)
5. Develop a recursive algorithm that implements the greedy strategy.
6. Convert the recursive algorithm to an iterative algorithm.

Greedy Shortcuts

1. Form the optimization problem so that, after a choice is made and there is only one subproblem left to be solved.
2. Prove that there is always an optimal solution to the original problem that makes the greedy choice, so that the greedy choice is always safe.
3. Demonstrate that, having made the greedy choice, what remains is a subproblem with the property that if we combine an optimal solution to the subproblem with the greedy choice we have made, we arrive at an optimal solution to the original problem.

When to use greedy

No way works all the time, but the greedy-choice property and optimal substructure are the two key ingredients.

- If we can demonstrate that the problem has these properties, then we are well on the way to developing a greedy algorithm for it.
- Beneath every greedy algorithm, there is almost always a more cumbersome dynamic-programming solution.

DFS

Theorem 6 (Parenthesis Theorem).

In any depth-first search of a (directed or undirected) graph $G = (V,E)$, for any two vertices u and v, exactly one of the following three conditions holds:

- *the interval $[u.d, u.f]$ and $[v.d, v.f]$ are entirely disjoint, and neither u nor v is a descendant of the other in the depth-first forest,*
- *the interval $[u.d, u.f]$ is contained entirely within the interval $[v.d, v.f]$, and u is a descendant of v in a depth-first tree, or*
- *the interval $[v.d, v.f]$ is contained entirely within the interval $[u.d, u.f]$, and v is a descendant of u in a depth-first tree, or*

DFS edges

- Based on this spanning tree, the edges of the original graph can be divided into three classes
  1. *Tree edges*, the edge leading to new vertices during DFS.
  2. *Forward edges*, which point from a node of the tree to one of its descendants
  3. *Back edges*, which point from a node to one of its ancestors
  4. *Cross edges*, which do neither
- Discover Time and Finish Time are used for determining the type of edge.
- For each travel, discover time, $v.d$, and finish time, $v.f$ are saved in the vertices.

Theorem 7.

For any edge $e = (u,v)$, e can be classified by the parenthesis structure.

1. *If $[u.d, u.f]$ and $[v.d, v.f]$ are entirely disjoint, then neither u nor v are ancestor. Therefore, (u,v) is cross edge.*
2. *If $v.d < u.d < u.f < v.f$, then v is an ancestor of u. Therefore, (u,v) is back edge.*
3. *If $u.d < v.d < v.f < u.f$, then u is an ancestor of v. Therefore, (u,v) is tree edge or forward edge.*

Topological Sort

```
procedure TOPOLOGICAL-SORT(G)
    Call DFS(G) to compute finishing times v.f for each vertex v.
    Sorted each vertex by finishing time in decreasing order.
    return the linked list of vertices.
end procedure
```

Strongly Connected Components

```
procedure STRONGLY-CONNECTED-COMPONENTS(G)
    call DFS(G) to compute finishing times u.f for each vertex u.
    compute $G^T$.
    call DFS($G^T$), but in the main loop of DFS, consider the vertices
in order of decreasing u.f.
    output the vertices of each tree in the depth-first forest formed in
line 3 as a separate strongly connected component
end procedure
```

Biconnected Components

Vertex v in DFS tree can be classified into 2 types:

1. v is root of DFS tree: If v has child at least 2, then v is an articulation point.
2. v is an internal vertex of DFS tree. Consider the following 3 cases:
   1. There is back edge to the ancestor of v. Vertex v is not an articulation point.
   2. There is back edge to the vertex v itself. Vertex v is an articulation point.
   3. There is no back edge. Vertex v is an articulation point.

- Using the parenthesis property of DFS to find the back edge.
  - If v first discovered before u, then $v.d < u.d$.
- Let $v.low$ be the minimum discovery time of v's ancestor.

$$v.low = \begin{cases} w.d & \text{when } (u,w) \text{ is a back edge and} \\ & u \text{ is a vertex in subtree rooted at } v, \\ v.d & \text{otherwise.} \end{cases}$$

Kruskal MST

```
procedure KRUSKAL(G, w)
    $T = \phi, M = \phi$
    Sort the edges of G.E into nondecreasing order by weight w
    for all v in V do
        $M = M \cup \{v\}$
    end for
    while $|M| > 1$ do
        Choose minimal $\{u, v\}$.
        if(u and v is not in the same set of M)
            $M = M - \{\{u\}, \{v\}\} \cup \{\{u, v\}\}$
            $T = T \cup \{\{u, v\}\}$
    end while
end procedure
```

Prim MST

```
procedure PRIM(G, w, r)
    $V_T = \{r\}$
    $d[r] = 0$
    for(all $v \in V - V_T$)
        if($\{r, v\} \in E$)
            $d[v] = w(r, v)$
        else
            $d[v] = \infty$
    while($V_T \neq V$)
        Find a vertex u such that $d[u] = \min\{d[v] \mid v \in V - V_T\}$
        $V_T = V_T \cup \{u\}$
        for(all $v \in V - V_T$)
            $d[v] = \min\{d[v], w(u, v)\}$
end procedure
```

Bellman Ford => O(V*E), doesn't work with negative cycles

Algorithm 1.

```
procedure INITIALIZE-SINGLE-SOURCE(G, s)
    for each vertex $v \in G.V$ do
        $v.d = \infty$
        $v.parent = NULL$
    end for
    $s.d = 0$
end procedure
```

Algorithm 2.

```
procedure RELAX(u, v, w)
    if $v.d > u.d + w(u, v)$ then
        $v.d = u.d + w(u, v)$
        $v.parent = u$
    end if
end procedure
```

Algorithm 4.

```
procedure BELMAN-FORD(G, w, s)
    INITIALIZE-SINGLE-SOURCE(G, s)
    for(i = 1 to |G.V| - 1)
        for(each edge $(u, v) \in G.E$)
            RELAX(u, v, w)
    for(each edge $(u, v) \in G.E$)   ▷ Check for a negative-weight cycle.
        if($v.d > u.d + w(u, v)$)
            return False
    return True
end procedure
```

Dijkstra => O(V + ElogV), doesn't work with negative edges

Floyd Warshall => O(V^3), all pair shortest path

```
procedure FLOYD-WARSHALL(W)
    n = W.rows
    D = W
    for(k = 1 to n)
        for(i = 1 to n)
            for(j = 1 to n)
                $d_{i,j} = \min\{d_{i,j}, d_{ik} + d_{kj}\}$
    return D
end procedure
```

Reduction properties

- Let Σ be a set of alphabet.
- A language over Σ is a set of strings created from symbol in Σ.
  - $B = \{0, 1\}^*$ is the set of all binary strings.
  - $\{\omega \in \Sigma^* \mid \omega \text{ is a palindrome}\}$ is a language over B.
- A *yes-instance* (or *no-instance*) is an instance whose answer is "*yes*" (or "*no*").
- An instance must be either a yes- or a no-instance.
- Examples:
  - Shortest path problem

$$\{G = (V, E), s, t\} \to \{p = \langle s, \ldots, t\rangle\}.$$

  - 0/1 Knapsack problem

$$\{\text{set } S \text{ of objects, capacity } W\} \to \{\text{the maximum value of objects}\}$$

Lemma 18.

Suppose $A, B, C \subseteq \Sigma^*$, a polynomial time reduction f from A to B, and a polynomial time reduction g from B to C. Then, the following statements hold true:

1. *If $A \leq_p B$ and $B \in \mathcal{P}$, then $A \in \mathcal{P}$.*
2. *If $A \leq_p B$ and $B \in \mathcal{NP}$, then $A \in \mathcal{NP}$.*
3. *If $A \leq_p B$ and $B \leq_p C$, then $A \leq_p C$.*

A language $L \in \Sigma^*$ is in $\mathcal{P}$ if and only if there exist a polynomial p and a deterministic Turing machine, M, such that

- $\forall x, T_M(|x|) \leq p(|x|)$.
- $\forall x, x \in L \to M(x) = 1$.
- $\forall x, x \notin L \to M(x) = 0$.

If Π is a decision problem for which $L_\Pi \in \mathcal{P}$, then there exist a polynomial time algorithm for Π.

A language $L \in \Sigma^*$ is in $\mathcal{NP}$ if and only if there exist polynomials p and q, and a deterministic Turing machine, M, such that

- $\forall x \forall y, T_M(|x|, |y|) \leq p(|x|)$.
- $\forall x \exists y, x \in L \wedge |y| \leq q(|x|) \to M(x, y) = 1$.
- $\forall x \forall y, x \notin L \wedge |y| \leq q(|x|) \to M(x, y) = 0$.

Definition 1.

A **Turing machine** (TM) is a 5-Tuple $T = (Q, \Sigma, \Gamma, q_0, \delta)$, where

- Q is a finite set of states, assumed not to contain h_a and h_r, the two **halting states**;
- Σ and Γ are finite sets, the **input** and **tape** alphabets respectively, with $\Sigma \subseteq \Gamma$; Γ is assumed not to contain Δ, the **blank** symbol;
- q_0 the initial state, is an element of Q;
- $\delta : Q \times (\Gamma \cup \{\Delta\}) \to (Q \cup \{h_a, h_r\}) \times (\Gamma \cup \{\Delta\}) \times \{R, L, S\}$ is a partial function.