# 1 Recursion

## Simple recursion

- **Reduction**: solve one problem using the solution to another.
- **Recursion**: a special case of reduction - reduce problem to a smaller instance of itself (self-reduction).

**Definitions**
- Problem instance of size \( n \) is reduced to one or more instances of size \( n-1 \) or less.
- For termination, problem instances of small size are solved by some other method as base cases.

Arguably the famous example of recursion. The goal is to move \( n \) disks one at a time from the first peg to the last peg.

**Tower of Hanoi**

```plaintext
Hanoi (n, src, dest, tmp):
    if (n > 0) then
        Hanoi (n-1, src, tmp, dest)
        Move disk n from src to dest
        Hanoi (n-1, tmp, dest, src)
```

## Recurrences

Suppose you have a recurrence of the form \( T(n) = rT(n/c) + f(n) \).

The master theorem gives a good asymptotic estimate of the recurrence. If the work at each level is:

- Decreasing: \( r f(n/c) = \kappa f(n) \) where \( \kappa < 1 \)
  \( T(n) = O(f(n)) \)
- Equal: \( r f(n/c) = f(n) \)
  \( T(n) = O(f(n) \log n) \)
- Increasing: \( r f(n/c) = K f(n) \) where \( K > 1 \)
  \( T(n) = O(n^\log r) \)

Some useful identities:
- Sum of integers: \( \sum_{k=1}^{n} k = \frac{n(n+1)}{2} \)
- Geometric series closed-form formula: \( \sum_{k=0}^{n} a r^k = \frac{a(1-r^{n+1})}{1-r} \)
- Logarithmic identities: \( \log(ab) = \log a + \log b, \log a/b = \log a - \log b, a^{\log b} = b^{\log a} \) for \( a, b > 1 \).

## Divide and conquer

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Runtime</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mergesort</td>
<td>( O(n \log n) )</td>
<td>( O(n \log n) )</td>
</tr>
<tr>
<td>Quicksort</td>
<td>( O(n \log^2 n) )</td>
<td>( O(n) ) if optimized</td>
</tr>
<tr>
<td>Karatsuba's algorithm</td>
<td>( O(n^{1.585}) )</td>
<td></td>
</tr>
</tbody>
</table>

We can divide and conquer multiplication like so:

\[
b_1 = 10^n b_L c_L + 10^n/2 (b_L c_R + b_R c_L) + b_R c_R.
\]

We can rewrite the equation as:

\[
b_1 = (b_L c_L + b_R c_R) (c_L x + c_R) = (b_L c_L)^2 + ((b_L + b_R) c_L + c_R) - b_L c_L - b_R c_R)
\]

Its running time is \( O(n^{1.585}) \).

## Linear time selection

The median of medians (MoM) algorithms give an element that is larger than \( 1/5 \)’s and smaller than \( 4/5 \)’s of the array elements. This is used in the linear time selection algorithm to find element of rank \( k \).

**Median-of-medians (MoM) algorithms**

```plaintext
Median-of-medians (A, x):
    sublists = \{A[j:j+i] for j in 0, 5 \ldots \text{len}(A)\}
    medians = \text{sorted(medians)}(\text{median}(\text{sublist})) for \text{sublist} in \text{sublists}
    // Base case
    if len(A) \leq 5 return sorted(A)
    // Find median of medians
    if len(medians) \leq 5 pivot = \text{sorted(medians)}(\text{medians})/2
    else pivot = \text{median-of-medians}(\text{medians}, \text{len}(medians))/2
    // Partitioning step
    low = \{i \in A | x \geq pivot\}
    high = \{i \in A | x \leq pivot\}
    k = len(low)
    if k \leq \text{len} return \text{median-of-medians}(low, x)
    else if \text{med} = \text{median-of-medians}(low, x-k)
    return med
```

## Backtracking

Backtracking is the algorithm paradigm involving guessing the solution to a single step in some multi-step process and recursing backwards if it doesn’t lead to a solution. For instance, consider the longest increasing subsequence (LIS) problem. You can either check all possible subsequences:

**Pseudocode: LIS - Naive enumeration**

```plaintext
alg LIS naive(A[1..n]):
    max = 0
    for each subsequence B of A do
        if B is increasing and \( |B| > \text{max} \)
            max = |B|
    return max
```

On the other hand, we don’t need to generate every subsequence. we only need to generate the subsequences that are increasing:

**Pseudocode: LIS - Backtracking**

```plaintext
LIS Smaller(A[1..n], x):
    if n = 0 then return 0
    max = \text{LIS Smaller}(A[1..n-1], x)
    if A[n] \leq x then
        return max
    max = \text{max}(max, 1 + \text{LIS Smaller}(A[1..(n-1)], A[n]))
    return max
```
### Dynamic programming

Dynamic programming (DP) is the algorithm paradigm involving the computation of a recursive backtracking algorithm iteratively to avoid the recomputation of any particular subproblem.

#### Longest increasing subsequence

The longest increasing subsequence problem asks for the length of a longest increasing subsequence in an unordered sequence, where the sequence is assumed to be given as an array. The recurrence can be written as:

\[
LIS(i, j) = \begin{cases} 
0 & \text{if } i = 0 \\
LIS(i - 1, j) & \text{if } A[i] \geq A[j] \\
\max \{LIS(i - 1, j), 1 + LIS(i - 1, i)\} & \text{else}
\end{cases}
\]

**Pseudocode: LIS - DP**

### Edit distance

The edit distance problem asks how many edits we need to make to a sequence for it to become another one. The recurrence is given as:

\[
Opt(i, j) = \begin{cases} 
\alpha_{x[i], y[j]} + \text{Opt}(i - 1, j - 1) & \text{if } x[i] = y[j] \\
\delta + \text{Opt}(i - 1, j) & \text{if } x[i] \neq y[j] \land y[j] \in V \\
\delta + \text{Opt}(i, j - 1) & \text{if } x[i] \neq y[j] \land x[i] \in V
\end{cases}
\]

**Base cases:** \(Opt(0, 0) = \delta \cdot i\) and \(Opt(0, j) = \delta \cdot j\)

**Pseudocode: Edit distance - DP**

### 2 Graph algorithms

#### Graph basics

A graph is defined by a tuple \(G = (V, E)\) and we typically define \(n = |V|\) and \(m = |E|\). We define \((u, v)\) as the edge from \(u\) to \(v\). Graphs can be represented as adjacency lists or adjacency matrices though the former is more commonly used.

- **path**: sequence of distinct vertices \(v_1, v_2, \ldots, v_k\) such that \(v_i, v_{i+1} \in E\) for \(1 \leq i \leq k - 1\). The length of the path is \(k - 1\) (the number of edges in the path). Note: a single vertex \(u\) is a path of length 0.
- **cycle**: sequence of distinct vertices \(v_1, v_2, \ldots, v_k\) such that \((v_i, v_{i+1}) \in E\) for \(1 \leq i \leq k - 1\) and \((v_k, v_1) \in E\). A single vertex is not a cycle according to this definition. Caveat: Sometimes people use the term cycle to also allow vertices to be repeated; we will use the term tour.
- A vertex \(u\) is connected to \(v\) if there is a path from \(u\) to \(v\).
- The connected component of \(u\), \(\text{con}(u)\), is the set of all vertices connected to \(u\).
- A vertex \(u\) can reach \(v\) if there is a path from \(u\) to \(v\). Alternatively \(v\) can be reached from \(u\). Let \(\text{rch}(u)\) be the set of all vertices reachable from \(u\).
Directed acyclic graphs

Directed acyclic graphs (DAGs) have an intrinsic ordering of the vertices that enables dynamic programming algorithms to be used on them. A topological ordering of a DAG \( G = (V, E) \) is an ordering \( \prec \) on \( V \) such that if \( (u, v) \in E \), then \( u \prec v \).

**Kahn's algorithm**

\[
\text{Kahn}(G(V, E), w):
\begin{align*}
\text{toposort} & \leftarrow \emptyset \\
\text{for } v \in V: \\
\text{in}(v) & \leftarrow \{ u \mid (u, v) \in E \} \\
\text{while } v \in V \text{ that has } \text{in}(v) = 0: \\
\text{Add } v \text{ to end of toposort} \\
\text{Remove } v \text{ from } V \\
\text{for } v \in u \to v \in E: \\
\text{in}(v) & \leftarrow \text{in}(v) - 1 \\
\text{return } \text{toposort}
\end{align*}
\]

Running time: \( O(n + m) \)

- A dag may have multiple topological sorts.
- A topological sort can be computed by DFS, in particular by listing the vertices in decreasing post-order visit.

**DFS and BFS**

**Pseudocode: Explore (DFS/BFS)**

\[
\text{Explore}(G, v): \\
\begin{align*}
\text{for } i \leftarrow 1 \text{ to } n: \\
\text{Visited}[i] & \leftarrow \text{False} \\
\text{Add } u \text{ to ToExplore and to } S \\
\text{Visited}[u] & \leftarrow \text{True} \\
\text{Make tree } T \text{ with root as } u \\
\text{while } B \text{ is non-empty do} \\
\text{Remove node } x \text{ from } B \\
\text{for each edge } (x, y) \in \text{Adj}(x) \text{ do} \\
\text{If } \text{Visited}[y] = \text{False} \\
\text{Visited}[y] & \leftarrow \text{True} \\
\text{Add } y \text{ to } B, S, T \text{ (with } x \text{ as parent)}
\end{align*}
\]

Pre and post numbering aids in analyzing the graph structure. By looking at the numbering we can tell if an edge \((u, v)\) is:
- **Forward edge**: \( \text{pre}(u) < \text{pre}(v) < \text{post}(u) < \text{post}(v) \)
- **Backward edge**: \( \text{pre}(u) < \text{pre}(v) < \text{post}(u) < \text{post}(v) \)
- **Cross edge**: \( \text{pre}(u) < \text{post}(u) < \text{pre}(v) < \text{post}(v) \)

**Strongly connected components**

- Given \( G \), \( u \) is strongly connected to \( v \) if \( v \in \text{rcc}(u) \) and \( u \in \text{rcc}(v) \).
- A maximal group of vertices that are all strongly connected to one another is called a strong component.

**Metagraph**

\[
\text{Metagraph}(G(V, E)):
\begin{align*}
\text{for } i \in V: \\
\text{for } j \in V \text{ do} \\
\text{d}(i, j, 0) & \leftarrow \text{e}(i, j) \\
\text{if } (i, j) \notin E & \text{ and } k = 0 \\
& \text{d}(i, j, k) \leftarrow \infty \\
\text{else} & \text{d}(i, j, k) \leftarrow \text{d}(i, j, k - 1) + d(k, j, k - 1) \\
\text{for } k \leftarrow 0 \text{ to } n - 1 \text{ do} \\
\text{for } i \in V \text{ do} \\
\text{for } j \in V \text{ do} \\
\text{d}(i, j, k) & \leftarrow \min \{ \text{d}(i, j, k - 1), \\
& \text{d}(i, k, k - 1) + d(k, j, k - 1) \}
\end{align*}
\]

**Dijkstra's algorithm**

\[
\text{Dijkstra}(G(V, E), s):
\begin{align*}
\text{for } v \in V \text{ do} \\
\text{d}(v) & \leftarrow \infty \\
\text{X} & \leftarrow \emptyset \\
\text{d}(s, s) & \leftarrow 0 \\
\text{for } i \leftarrow 1 \text{ to } n \text{ do} \\
\text{v} & \leftarrow \arg \min_{u \in V - X} \text{d}(u) \\
\text{X} & \leftarrow X \cup \{ v \} \\
\text{for } u \in \text{Adj}(v) \text{ do} \\
\text{d}(u) & \leftarrow \min \{ (\text{d}(u), \text{d}(v) + \text{e}(v, u)) \}
\end{align*}
\]

Running time: \( O(m + n \log n) \) (if using a Fibonacci heap as the priority queue)

**Bellman-Ford algorithm**

\[
\text{Bellman-Ford}(G(V, E)):
\begin{align*}
\text{for each } v \in V \text{ do} \\
\text{d}(v) & \leftarrow \infty \\
\text{for } k \leftarrow 1 \text{ to } n - 1 \text{ do} \\
\text{for each edge } (u, v) \in \text{in}(v) \text{ do} \\
\text{d}(v) & \leftarrow \min \{ \text{d}(v), \text{d}(u) + \text{e}(u, v) \}
\end{align*}
\]

Running time: \( O(nm) \)

**Floyd-Warshall algorithm**

\[
\text{Floyd-Warshall}(G(V, E)):
\begin{align*}
\text{for } i \in V \text{ do} \\
\text{for } j \in V \text{ do} \\
\text{d}(i, j, 0) & \leftarrow \text{e}(i, j) \\
\text{if } (i, j) \notin E & \text{ and } k = 0 \\
& \text{d}(i, j, k) \leftarrow \infty \\
\text{else} & \text{d}(i, j, k) \leftarrow \text{d}(i, j, k - 1) + d(k, j, k - 1) \\
\text{for } k \leftarrow 0 \text{ to } n - 1 \text{ do} \\
\text{for } i \in V \text{ do} \\
\text{for } j \in V \text{ do} \\
\text{d}(i, j, k) & \leftarrow \min \{ \text{d}(i, j, k - 1), \\
& \text{d}(i, k, k - 1) + d(k, j, k - 1) \}
\end{align*}
\]

Then \( d(i, j, n - 1) \) will give the shortest-path distance from \( i \) to \( j \).