Algorithms Design & Analysis

Unit 2: Disjoint Sets & Backtracking

Disjoint Sets

Disjoint Set Operations

Core Concept

In many computer science problems, we begin with a finite universe of n elements, represented as:

$$U = \{1, 2, 3, \ldots, n\}$$

We partition these elements into several sets $S_1, S_2, S_3, \ldots$ that are **pairwise disjoint**. This means that no element exists in more than one set:

$$S_i \cap S_j = \emptyset \quad \text{for all } i \neq j$$

Disjoint sets are managed by a data structure called the **Union-Find** (or Disjoint Set Union - DSU) structure, which supports two primary operations:

1. **Disjoint Set Union ($S_i \cup S_j$):**
   - Combines two distinct sets S_i and S_j into a single new set $S_k = S_i \cup S_j$.
   - Once combined, the original sets S_i and S_j are destroyed/replaced by S_k.
2. **Find (i):**
   - Identifies and returns the unique representative (or root ID) of the set containing element i.
   - If two elements x and y return the same representative, they belong to the same set:

$$\text{Find}(x) = \text{Find}(y) \implies x, y \in \text{same set}$$

Union and Find Algorithms (Tree Representation)

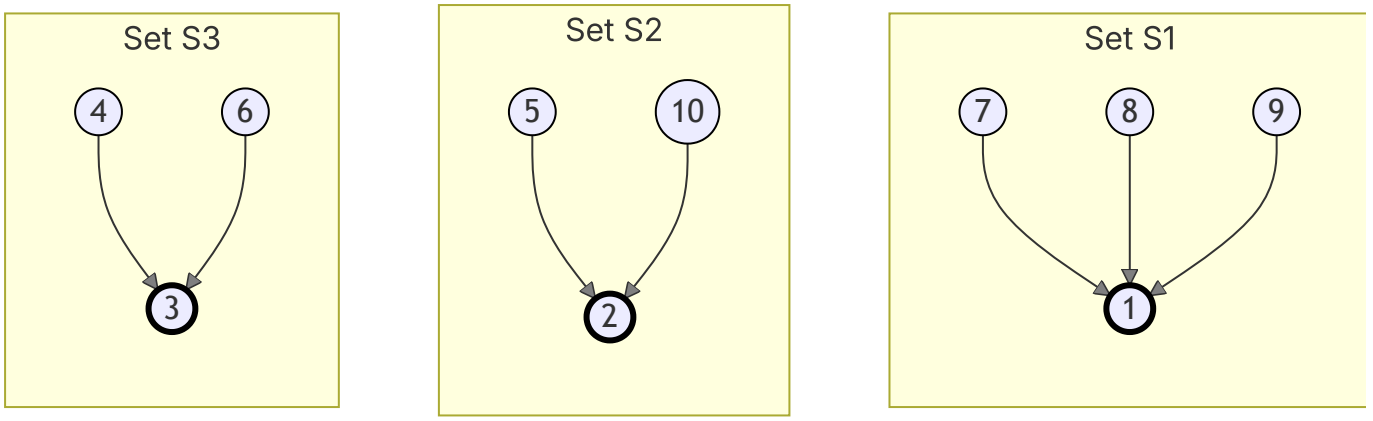
Tree-Based Data Structure

The most efficient way to implement disjoint sets is by representing each set as a tree:

- Each node in the tree points upward to its **parent**.
- The **root** of the tree has no parent (indicated by a self-loop or a parent value ≤ 0). The root acts as the **set representative**.

For example, let's represent three disjoint sets:

- $S_1 = \{1, 7, 8, 9\}$ (Root: 1)
- $S_2 = \{2, 5, 10\}$ (Root: 2)
- $S_3 = \{3, 4, 6\}$ (Root: 3)



In array representation, we use a 1-indexed array `PARENT[1..n]`:

- If `PARENT[i] = 0` (or negative), then node i is a root.
- If `PARENT[i] = p > 0`, then p is the parent of node i.

The initial state of the array for $n = 10$ elements (each element in its own singleton set):

Index `i`	1	2	3	4	5	6	7	8	9
`PARENT[i]`	0	0	0	0	0	0	0	0	0

Simple Union and Find Algorithms

Algorithm for Simple Union

This procedure links the root of one set directly as a child of the root of another set.

```
Algorithm SimpleUnion(i, j)
// i and j are the roots of two disjoint trees
{
    PARENT[i] := j;
}
```

Algorithm for Simple Find

This procedure traverses upward from node `i` along the parent links until it encounters a root node (parent value ≤ 0).

```
Algorithm SimpleFind(i)
{
    j := i;
    while (PARENT[j] > 0) do
        j := PARENT[j]; // Move up to parent
    return j; // Return root
}
```

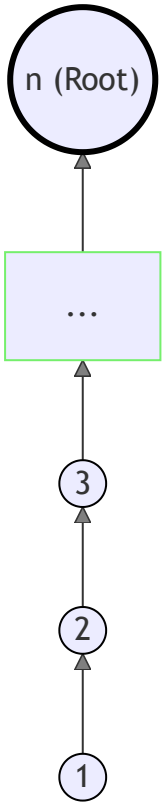
The Degenerate Tree Problem (Worst-Case Analysis)

If we use `SimpleUnion` and `SimpleFind` without checks, we can construct highly unbalanced trees.

Degenerate Trace Example:

Suppose we start with n elements in separate sets and execute the sequence of operations:

$$\text{SimpleUnion}(1,2), \text{SimpleUnion}(2,3), \ldots, \text{SimpleUnion}(n-1,n)$$



- **Result:** The tree degenerates into a linear chain of length n.
- **Cost:**
  - Performing `SimpleFind(1)` requires traversing all n nodes, taking $O(n)$ steps.
  - A sequence of m such find operations would take an inefficient $O(mn)$ **worst-case time**. If $m = n$, the total time complexity scales quadratically as $O(n^2)$.

Efficiency Improvements

To prevent tree degeneration, two optimizations are applied: **Weighted Union** and **Collapsing Find (Path Compression)**.

Weighting Rule for Union

- **Concept:** When merging two trees, always attach the root of the tree with **fewer nodes** as a child of the root of the tree with **more nodes**.
- **Implementation:** To avoid using extra memory, we store the size of the tree as a **negative number** inside the root's `PARENT` field:

$$\text{If } i \text{ is a root, } \text{PARENT}[i] = -(\text{number of nodes in the tree})$$

Weighted Union Pseudocode

```
Algorithm WeightedUnion(i, j)
// i and j are the roots of two disjoint trees.
// PARENT[i] and PARENT[j] contain negative sizes.
{
    temp := PARENT[i] + PARENT[j]; // Combined negative count

    if (PARENT[i] > PARENT[j]) then
    {
        // Tree i has fewer nodes than Tree j (since, e.g., -2 > -5)
        PARENT[i] := j;   // Make j the parent of i
        PARENT[j] := temp; // Update size of j
    }
    else
    {
        // Tree j has fewer or equal nodes than Tree i
        PARENT[j] := i;   // Make i the parent of j
        PARENT[i] := temp; // Update size of i
    }
}
```

Mathematical Proof (Lemma 2.3)

> **Theorem:** Let T be a tree with n nodes created by the `WeightedUnion` algorithm. The depth of any node in T is at most $\lfloor \log_2 n \rfloor + 1$.

Proof by Mathematical Induction:

- **Base Case ($n = 1$):**

- A tree with 1 node has depth 1.
- Formula: $\lfloor \log_2 1 \rfloor + 1 = 0 + 1 = 1$. The theorem holds.

- **Inductive Step:**
  - Assume the theorem holds for all trees with size $< n$ nodes.
  - Let tree T (size n) be created by joining two trees T_1 (size n_1) and T_2 (size n_2) using `WeightedUnion`. Let $n_1 + n_2 = n$.
  - Without loss of generality, assume $n_1 \geq n_2$. According to the weighting rule, T_2 is attached under the root of T_1.
  - Let's analyze the new depths of the nodes in T:
    1. **Nodes originating from T_1:** Their depths do not change. By induction hypothesis:

$$\text{Depth} \leq \lfloor \log_2 n_1 \rfloor + 1 \leq \lfloor \log_2 n \rfloor + 1$$

    2. **Nodes originating from T_2:** Their depths increase by exactly 1 because they have a new parent link to the root of T_1.

$$\text{New Depth} = \text{Old Depth in } T_2 + 1 \leq (\lfloor \log_2 n_2 \rfloor + 1) + 1 = \lfloor \log_2 n_2 \rfloor + 2$$

  - Since $n_1 \geq n_2$ and $n_1 + n_2 = n$, we know that:

$$2n_2 \leq n_1 + n_2 = n \implies n_2 \leq \frac{n}{2}$$

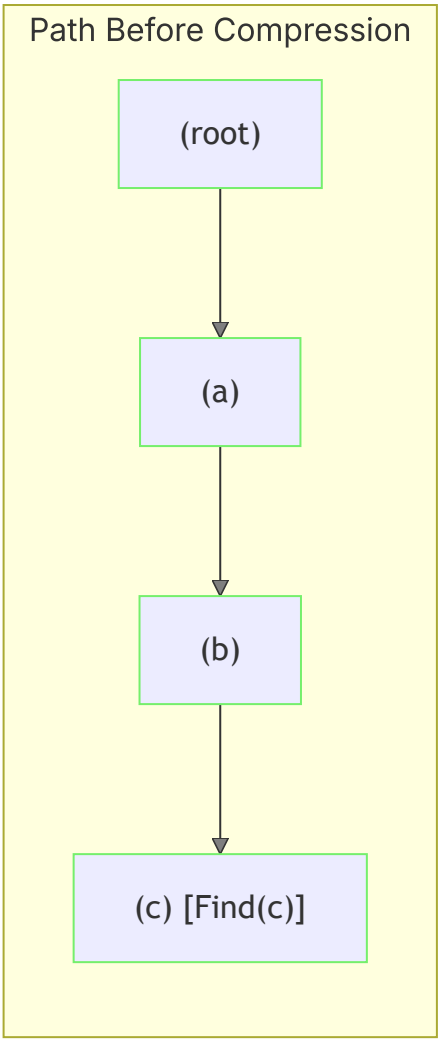
  - Substitute this inequality back:

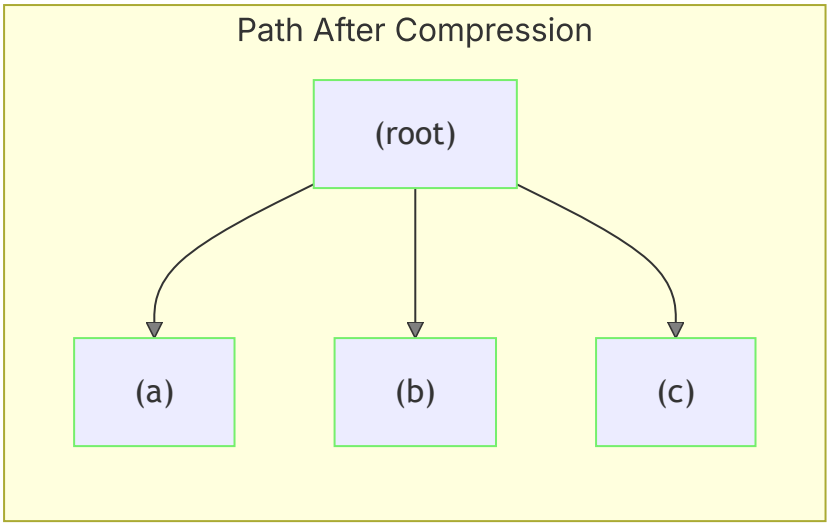
$$\text{New Depth} \leq \lfloor \log_2(n/2) \rfloor + 2 = \lfloor \log_2 n - 1 \rfloor + 2 = \lfloor \log_2 n \rfloor + 1$$

  - The theorem holds for a tree of size n. By induction, the proof is complete.
- **Impact:** Since the tree height is bounded by $O(\log n)$, the worst-case time for a single `Find` drops to $O(\log n)$.

Collapsing Rule for Find (Path Compression)

- **Concept:** During a `Find(i)` operation, we traverse up to the root. Once the root r is identified, we make another pass up the tree and change the parent pointer of **every node on the path** to point directly to r.





Collapsing Find Pseudocode

```
Algorithm CollapsingFind(i)
// Finds the root of the tree containing element i and compresses the path
{
    // Pass 1: Find the root r
    r := i;
    while (PARENT[r] > 0) do
        r := PARENT[r];

    // Pass 2: Collapse path
    curr := i;
    while (curr != r) do
    {
        parent_node := PARENT[curr];
        PARENT[curr] := r; // Redirect parent directly to root
        curr := parent_node; // Move to next node in original path
    }
    return r;
}
```

Asymptotic Bound of Combined Operations (Tarjan's Lemma)

When we combine both `WeightedUnion` and `CollapsingFind`, the tree structure becomes incredibly flat.

Let m be the number of `Find` operations and n be the number of elements. The total worst-case time $T(m, n)$ required to process an intermixed sequence of $m \geq n$ `Find`s and $n - 1$ `Union`s satisfies:

$$T(m, n) = \Theta(m \cdot \alpha(m, n))$$

Where $\alpha(m, n)$ is the **Inverse Ackermann Function**.

- Ackermann's function $A(p, q)$ grows incredibly fast:

$$A(4, 2) \approx 2^{2^{2^{65536}}}$$

- Because the function grows so rapidly, its inverse $\alpha(m, n)$ grows extremely slowly.
- For all practical inputs in computer science:

$$\alpha(m, n) \leq 4$$

- Therefore, the average time per operation is practically **constant** $O(1)$, making Union-Find one of the most efficient data structures in computer science.

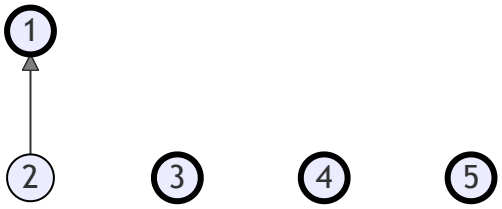
Practical Application: Equivalence Relations

The Union-Find data structure is standard for computing **equivalence classes** online. Given n variables $\{x_1, \ldots, x_n\}$ and a series of equivalence relations (such as $x_a \equiv x_b$), we group them into classes.

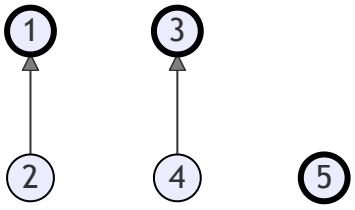
Trace Example:

Let $n = 5$ variables, initially in singleton sets:
`PARENT = [-1, -1, -1, -1, -1]` (Using -1 to represent tree size of 1).

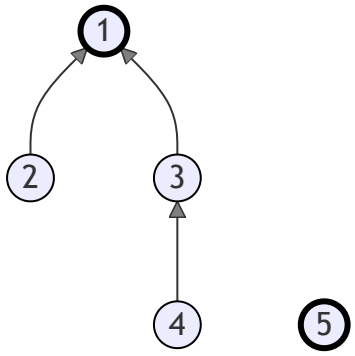
1. **Relation `1 = 2`:**
   - `root1 = CollapsingFind(1) = 1`
   - `root2 = CollapsingFind(2) = 2`
   - Since `root1 != root2`, perform `WeightedUnion(1, 2)`.
   - Size of 1 is 1, size of 2 is 1 (both -1). Let's make 2 the child of 1.
   - `PARENT[2] = 1`, `PARENT[1] = -2`.



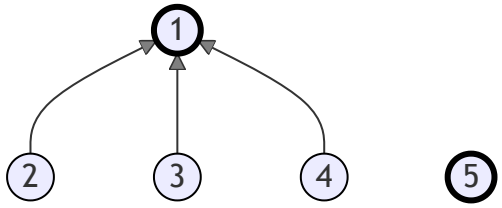
2. **Relation `3 = 4`:**
    - Perform `WeightedUnion(3, 4)`.
    - `PARENT[4] = 3`, `PARENT[3] = -2`.



3. **Relation `1 = 3`:**
    - `root1 = CollapsingFind(1) = 1` (size 2, `PARENT[1] = -2`)
    - `root2 = CollapsingFind(3) = 3` (size 2, `PARENT[3] = -2`)
    - Since size is equal, make 3 the child of 1.
    - `PARENT[3] := 1`, `PARENT[1] := -2 + (-2) = -4`.
    - Array state: `PARENT = [-4, 1, 1, 3, -1]`.



4. **Query `CollapsingFind(4)`:**
    - Path: $4 \rightarrow 3 \rightarrow 1$ (root).
    - Path collapse changes `PARENT[4]` to point directly to `1`.
    - Array state becomes: `PARENT = [-4, 1, 1, 1, -1]`.



Priority Queues: Heaps and Heapsort

Priority Queues

Core Concept

A **Priority Queue** is an abstract data type that maintains a set of elements, where each element is associated with a "priority." In a standard queue, elements are processed in a First-In-First-Out (FIFO) manner. In a priority queue:

- Elements are extracted based on their priority (e.g., the element with the highest/lowest priority is served first).
- The two primary operations are:
  1. **Insertion:** Add a new element with a given priority to the queue.
  2. **Extraction:** Retrieve and remove the element with the highest (or lowest) priority.

Motivation (Comparison of Implementations)

To understand the efficiency of heaps, consider alternative data structures for implementing a priority queue of size n:

Implementation Scheme	Insertion Complexity	Finding / Deleting Max Complexity
Unsorted Array / List	$O(1)$ (Append to the end)	$O(n)$ (Must scan the entire array)
Sorted Array / List	$O(n)$ (Must shift elements to keep sorted)	$O(1)$ (Remove from the end)
Binary Search Tree (Balanced)	$O(\log n)$	$O(\log n)$
Binary Heap	$O(\log n)$	$O(\log n)$

A Binary Heap provides a highly efficient balance: both insertion and deletion run in logarithmic time, and it requires no extra pointers, unlike tree-based node representations.

Heaps

Definition

A **Heap** is a complete binary tree represented sequentially in memory (typically in an array) that satisfies the **Heap Property**.

- **Complete Binary Tree:** A binary tree in which all levels are completely filled except

possibly the last level, which is filled from left to right.

- **Max-Heap Property:** The key (value) at any parent node is greater than or equal to the keys of its children:

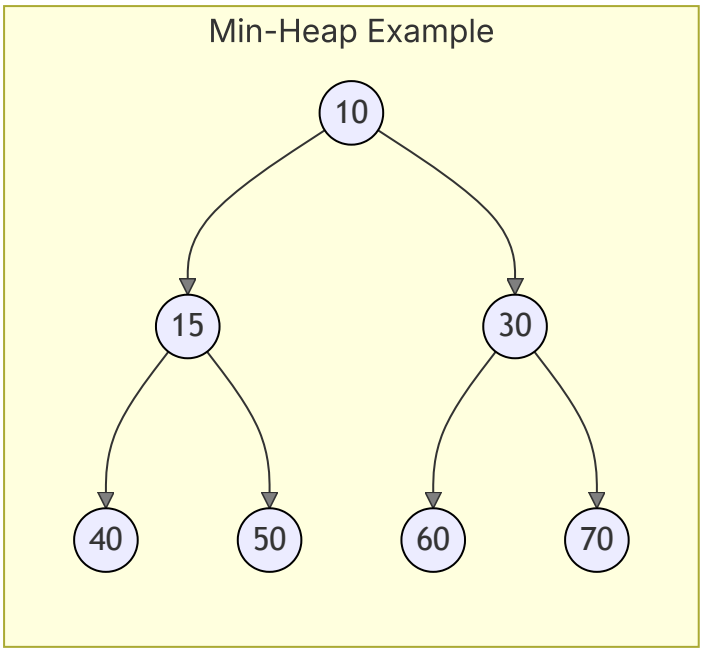
$$\text{Key}(\text{Parent}) \geq \text{Key}(\text{Child})$$

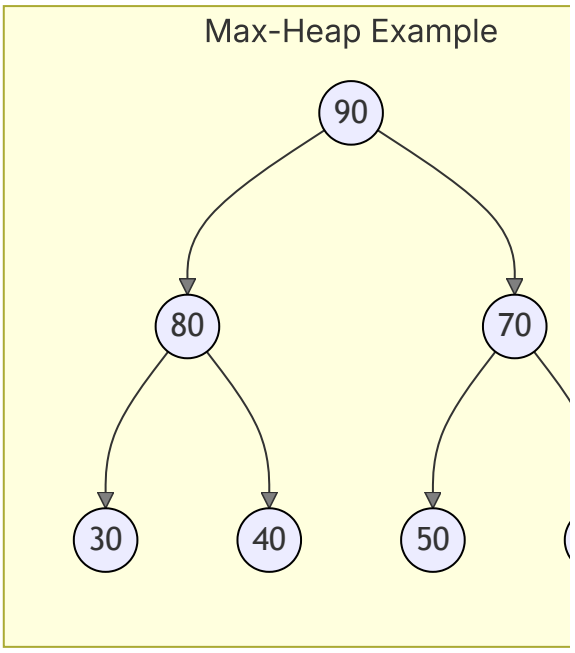
Consequently, the maximum element in a Max-Heap is always at the **root**.

- **Min-Heap Property:** The key at any parent node is less than or equal to the keys of its children:

$$\text{Key}(\text{Parent}) \leq \text{Key}(\text{Child})$$

Consequently, the minimum element in a Min-Heap is always at the **root**.

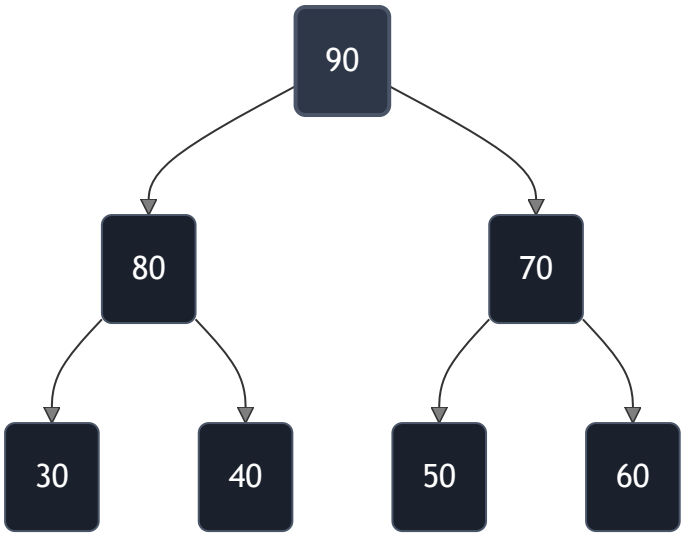




Array Representation of Complete Binary Trees

Because a heap is a complete binary tree, we can map it directly into a 1-indexed array $A[1..n]$ without using child/parent pointers. For any element at index i:

- **Parent of i:** $\lfloor i/2 \rfloor$ (for $i > 1$)
- **Left Child of i:** $2i$ (if $2i \leq n$)
- **Right Child of i:** $2i + 1$ (if $2i + 1 \leq n$)



Heap Operations

Insertion (Up-Heap / Sift-Up)

To insert an element:

1. Append the element at the end of the array (maintaining the complete binary tree property).
2. Compare the new element with its parent. If the new element is larger than its parent (in a Max-Heap), swap them.
3. Repeat this process (moving upward) until the heap property is satisfied or the element becomes the root.

Insert Pseudocode

```
Algorithm InsertMaxHeap(A, n, item)
// Inserts item into a Max-Heap A of size n.
// n is incremented before or during this operation.
{
    n := n + 1;
    i := n;
    while (i > 1 and item > A[floor(i/2)]) do
    {
        A[i] := A[floor(i/2)]; // Move parent down
        i := floor(i/2);       // Move pointer up
    }
    A[i] := item; // Place item in its correct position
}
```

- **Time Complexity:** $O(\log n)$ (since the maximum height of the tree is $\log_2 n$).

Deletion and Adjustment (Down-Heap / Sift-Down)

To remove the maximum element (the root):

1. Copy the element at the root $A[1]$ (which is the maximum).
2. Replace the root with the last element of the array $A[n]$ and decrement the heap size n.
3. Run the **Adjust** procedure to push the root element down to its correct position by comparing it with its children and swapping with the larger child.

Adjust Pseudocode

```
Algorithm Adjust(A, i, n)
// Adjusts the binary tree A[i..n] to satisfy the Max-Heap property.
// The left and right subtrees of node i are assumed to be heaps.
{
    j := 2 * i; // Left child of i
    item := A[i];
    while (j <= n) do
    {
        // Find the larger child of parent i
        if (j < n and A[j] < A[j+1]) then
            j := j + 1; // j is now the right child

        // Compare parent item with the larger child
        if (item >= A[j]) then
            break; // Heap property satisfied
        else
        {
            A[floor(j/2)] := A[j]; // Move child up
            j := 2 * j;            // Move down to next level
        }
    }
    A[floor(j/2)] := item; // Place item in final position
}
```

- **Time Complexity:** $O(\log n)$ (sinking a node from root to leaf takes at most $\log_2 n$ steps).

Heap Creation (HEAPIFY)

There are two ways to build a heap from an arbitrary array $A[1..n]$:

Method 1: Repeated Insertion

- **Process:** Start with an empty heap and call `InsertMaxHeap` for each of the n elements.

- **Worst-Case Cost:** If elements are inserted in ascending order, each element rises to the root, taking $\sum \log i = \Theta(n \log n)$ time.

Method 2: Heapify (Bottom-Up Method)

- **Process:** Treat the array directly as a complete binary tree. Note that all leaf nodes at indices $\lfloor n/2 \rfloor + 1$ to n are already valid heaps. Therefore, we call `Adjust` starting from the parent of the last leaf (index $\lfloor n/2 \rfloor$) down to the root (index 1).

Heapify Pseudocode

```
Algorithm Heapify(A, n)
// Converts an arbitrary array A[1..n] into a Max-Heap.
{
    for i := floor(n/2) downto 1 do
        Adjust(A, i, n);
}
```

Mathematical Proof of $O(n)$ Heapify Complexity

A common question is why `Heapify` runs in $O(n)$ rather than $O(n \log n)$.

1. The maximum number of nodes at height h in a complete binary tree of size n is:

$$\text{Nodes at height } h \leq \left\lceil \frac{n}{2^{h+1}} \right\rceil$$

2. The time required to run `Adjust` on a node at height h is proportional to its height, $O(h)$ (since it can swap down at most h times).
3. The total work $T(n)$ for the entire heap construction is:

$$T(n) = \sum_{h=0}^{\lfloor \log n \rfloor} h \cdot \left(\frac{n}{2^{h+1}} \right) = \frac{n}{2} \sum_{h=0}^{\lfloor \log n \rfloor} \frac{h}{2^h}$$

4. For an infinite sum:

$$\sum_{h=0}^{\infty} \frac{h}{2^h} = \frac{1/2}{(1 - 1/2)^2} = 2$$

5. Substituting this result back into our equation:

$$T(n) \leq \frac{n}{2} \cdot 2 = n$$

- Therefore, the worst-case time complexity of `Heapify` is $O(n)$.

Heapsort

Heapsort Algorithm

Heapsort utilizes the Max-Heap structure to sort an array in-place.

1. Build a Max-Heap from the array $A[1..n]$ using `Heapify` in $O(n)$ time.
2. The maximum element is now at $A[1]$. Swap $A[1]$ with the last element of the heap $A[n]$.
3. Reduce the heap size by 1 and run `Adjust(A, 1, n-1)` to restore the heap property.
4. Repeat this process until all elements are sorted.

```
Algorithm HeapSort(A, n)
// Sorts the array A[1..n] in non-decreasing order.
{
    Heapify(A, n); // Step 1: Build Max-Heap
    for i := n downto 2 do // Step 2-4: Extract max and adjust
    {
        Swap(A[1], A[i]);    // Move maximum to sorted section
        Adjust(A, 1, i - 1); // Adjust remaining elements
    }
}
```

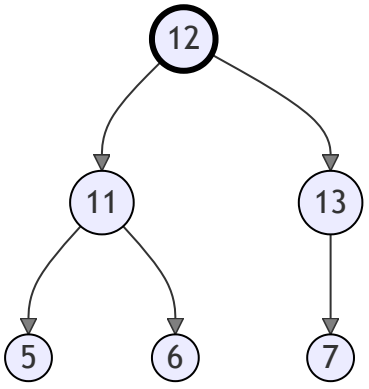
Step-by-Step Numerical Trace of Heapsort

Let's sort the array:

$$A = [12, 11, 13, 5, 6, 7] \quad (n = 6)$$

Phase 1: Build Max-Heap (`Heapify`)

We start with the unsorted tree:

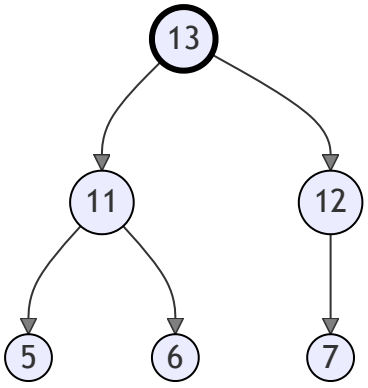


- Leaves are at indices: $4, 5, 6$ (values $5, 6, 7$).
- Non-leaves are at indices: $\lfloor 6/2 \rfloor = 3$ down to 1.

1. **Adjust at index 3 ($A[3] = 13$):**

- Children: Left child $A[6] = 7$.
- $13 \geq 7 \implies$ No change.

2. **Adjust at index 2 ($A[2] = 11$):**
   - Children: $A[4] = 5$, $A[5] = 6$. Larger child is 6.
   - $11 \geq 6 \implies$ No change.
3. **Adjust at index 1 ($A[1] = 12$):**
   - Children: $A[2] = 11$, $A[3] = 13$. Larger child is 13.
   - Since $12 < 13$, swap $A[1]$ and $A[3]$.
   - Tree becomes:



- Node 12 is pushed down to index 3. Children of index 3: $A[6] = 7$.
- Since $12 \geq 7$, adjustments are complete.
- **Max-Heap Array:** $A = [13, 11, 12, 5, 6, 7]$

Phase 2: Sort Loop (Extraction and Adjust)

1. **Iteration 1 ($i = 6$):**
   - Swap $A[1] \leftrightarrow A[6] \implies$ Swap $13 \leftrightarrow 7$.
   - Array: $[7, 11, 12, 5, 6 \mid \mathbf{13}]$ (Sorted section: $[13]$).
   - Call `Adjust(A, 1, 5)` to fix the root 7:
     - Children of index 1: $A[2] = 11$, $A[3] = 12$. Larger is 12. Swap $7 \leftrightarrow 12$.
     - Node 7 is now at index 3. Child is $A[6]$ (ignored as it's sorted).
     - Array becomes: $[12, 11, 7, 5, 6 \mid 13]$
2. **Iteration 2 ($i = 5$):**
   - Swap $A[1] \leftrightarrow A[5] \implies$ Swap $12 \leftrightarrow 6$.
   - Array: $[6, 11, 7, 5 \mid \mathbf{12, 13}]$ (Sorted section: $[12, 13]$).
   - Call `Adjust(A, 1, 4)` to fix root 6:
     - Children of 1: $A[2] = 11$, $A[3] = 7$. Larger is 11. Swap $6 \leftrightarrow 11$.
     - Node 6 is now at index 2. Child is $A[4] = 5$. Since $6 \geq 5 \implies$ stop.
     - Array becomes: $[11, 6, 7, 5 \mid 12, 13]$
3. **Iteration 3 ($i = 4$):**
   - Swap $A[1] \leftrightarrow A[4] \implies$ Swap $11 \leftrightarrow 5$.

- Array: $[5, 6, 7 \mid \mathbf{11, 12, 13}]$
- Call `Adjust(A, 1, 3)` to fix root 5:
  - Children of 1: $A[2] = 6$, $A[3] = 7$. Larger is 7. Swap $5 \leftrightarrow 7$.
  - Array becomes: $[7, 6, 5 \mid 11, 12, 13]$

4. **Iteration 4 ($i = 3$):**
   - Swap $A[1] \leftrightarrow A[3] \implies$ Swap $7 \leftrightarrow 5$.
   - Array: $[5, 6 \mid \mathbf{7, 11, 12, 13}]$
   - Call `Adjust(A, 1, 2)` to fix root 5:
     - Children of 1: $A[2] = 6$. Swap $5 \leftrightarrow 6$.
     - Array becomes: $[6, 5 \mid 7, 11, 12, 13]$
5. **Iteration 5 ($i = 2$):**
   - Swap $A[1] \leftrightarrow A[2] \implies$ Swap $6 \leftrightarrow 5$.
   - Array: $[5 \mid \mathbf{6, 7, 11, 12, 13}]$
   - Loop ends.

Final Sorted Array: $A = [5, 6, 7, 11, 12, 13]$

Complexity Summary

- **Time Complexity:**
  - **Worst-Case:** $\Theta(n \log n)$ (Heapify is $O(n)$, loop runs $n - 1$ times doing adjustments of cost $O(\log n)$).
  - **Best-Case:** $\Theta(n \log n)$.
  - **Average-Case:** $\Theta(n \log n)$.
- **Space Complexity:** $O(1)$ auxiliary space. Heapsort is an **in-place** sorting algorithm.

Backtracking

General Method

Core Concept

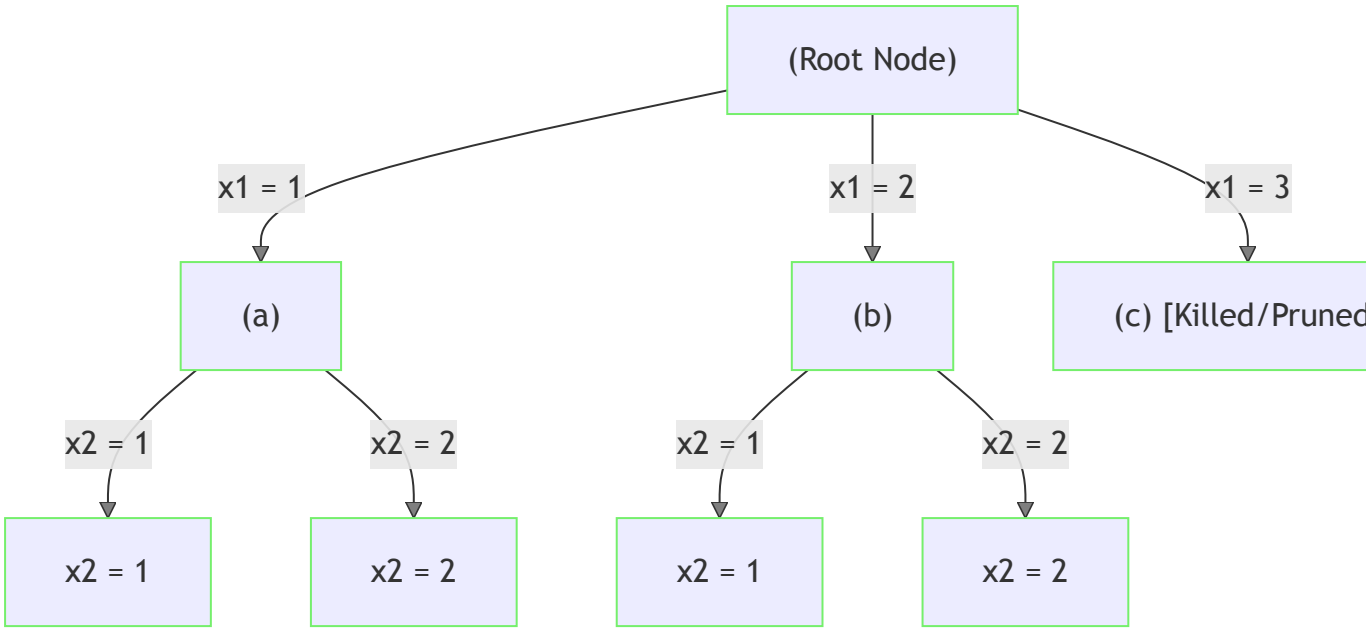
Backtracking is a systematic, depth-first search strategy used for finding solutions to problems where the solution can be expressed as an n-tuple:

$$(x_1, x_2, \ldots, x_n)$$

where each variable x_i is chosen from a finite set S_i.

Unlike brute-force search, which generates all possible $m = |S_1| \times |S_2| \times \cdots \times |S_n|$ combinations to evaluate, backtracking builds the solution tuple one component at a

time, checking constraints at each step.



State Space Tree Terminology

To conceptualize the search space, backtracking structures it as a tree called the **State Space Tree**:

1. **Problem State:** Any node in the tree that represents a partial or full state of the variable assignments.
2. **Solution Space:** The set of all possible tuples defined by the explicit constraints (the total leaf nodes of the unpruned tree).
3. **Solution State:** Any state in which the path from the root defines a valid tuple in the solution space.
4. **Answer State:** A solution state that satisfies all **implicit constraints** (represents a valid final solution to the problem).
5. **Live Node:** A node that has been generated, but its children have not yet been fully expanded.
6. **E-Node (Expanding Node):** The currently active live node whose children are being generated.
7. **Dead Node:** A node that has been generated and either:
   - Has had all its children expanded.
   - Has been determined by a bounding function to have no chance of leading to an answer node. It is pruned immediately.

Explicit vs. Implicit Constraints

- **Explicit Constraints:** Rules that restrict each variable x_i to take values from a specific set (e.g., $x_i \in \{0, 1\}$, or $1 \leq x_i \leq n$). These define the dimensions of the state space tree.
- **Implicit Constraints:** Rules that determine how the variables must relate to one another to satisfy the problem criteria (e.g., no two queens can attack each other, or the sum of chosen elements must equal M).

General Control Abstraction

Backtracking uses a recursive template to perform the depth-first traversal of the state space tree:

```
Algorithm Backtrack(k)
// k is the index of the variable currently being decided (x[k])
{
    // Generate all candidate values for x[k]
    for (each x[k] in T(x[1], x[2], ..., x[k-1])) do
    {
        if (Bound(x[1], ..., x[k]) = true) then
        {
            // If it is a full solution, record/print it
            if (Solution(x[1], ..., x[k]) = true) then
                Print(x[1], ..., x[k]);
            else
                Backtrack(k + 1); // Recurse for the next variable
        }
    }
}
```

Applications

The n-Queens Problem

Problem Definition

The problem is to place n non-attacking queens on an $n \times n$ chessboard so that no two queens share the same row, column, or diagonal.

Formulation

- **Solution Vector:** An n-tuple $(x_1, x_2, \ldots, x_n)$, where x_i represents the column index of row i where the queen is placed.

- **Explicit Constraint:** $x_i \in \{1, 2, \ldots, n\}$ for all $1 \leq i \leq n$.
- **Implicit Constraints:**
  1. No two queens in the same column: $x_i \neq x_j$ for all $i \neq j$.
  2. No two queens on the same diagonal: For any two queens placed at (i, x_i) and (j, x_j), they share a diagonal if and only if:

$$|x_i - x_j| = |i - j|$$

Algorithms

```
Algorithm Place(k, col)
// Returns true if a queen can be placed in row k, column col.
// x[1..k-1] store the column positions of queens placed in previous rows.
{
    for i := 1 to k - 1 do
    {
        // Check column conflict or diagonal conflict
        if (x[i] = col or abs(x[i] - col) = abs(i - k)) then
            return false;
    }
    return true;
}

Algorithm NQueens(k, n)
// Recursive backtracking search for n-Queens
{
    for col := 1 to n do
    {
        if Place(k, col) then
        {
            x[k] := col;
            if (k = n) then
                Print(x[1..n]);
            else
                NQueens(k + 1, n);
        }
    }
}
```

Step-by-Step Trace of the 4-Queens Problem

Let's solve the 4-Queens problem ($n = 4$):

1. **Row 1:** Try $x_1 = 1$. (Board: `[1, 0, 0, 0]`)
2. **Row 2:**
   - Try $x_2 = 1$ (attacks x_1).

- Try $x_2 = 2$ (attacks x_1 diagonally).
- Try $x_2 = 3$. Valid! (Board: `[1, 3, 0, 0]`)

3. **Row 3:**
   - Try $x_3 = 1$ (attacks x_1).
   - Try $x_3 = 2$ (attacks x_2 diagonally).
   - Try $x_3 = 3$ (attacks x_2).
   - Try $x_3 = 4$ (attacks x_2 diagonally).
   - All columns fail! Backtrack to Row 2.
4. **Row 2 (Resume):**
   - Try $x_2 = 4$. Valid! (Board: `[1, 4, 0, 0]`)
5. **Row 3:**
   - Try $x_3 = 1$ (attacks x_1).
   - Try $x_3 = 2$. Valid! (Board: `[1, 4, 2, 0]`)
6. **Row 4:**
   - Try $x_4 = 1$ (attacks x_1 and x_3 diagonally).
   - Try $x_4 = 2$ (attacks x_3).
   - Try $x_4 = 3$ (attacks x_2 diagonally).
   - Try $x_4 = 4$ (attacks x_2).
   - All columns fail! Backtrack to Row 3, then Row 2, then Row 1.
7. **Row 1 (Resume):** Try $x_1 = 2$. (Board: `[2, 0, 0, 0]`)
8. **Row 2:** Try $x_2 = 4$. Valid! (Board: `[2, 4, 0, 0]`)
9. **Row 3:** Try $x_3 = 1$. Valid! (Board: `[2, 4, 1, 0]`)
10. **Row 4:** Try $x_4 = 3$. Valid! (Board: `[2, 4, 1, 3]`)
    - **Solution Found:** $(2, 4, 1, 3)$

```
Solution 1: (2, 4, 1, 3)          Solution 2: (3, 1, 4, 2)
     . Q . .                           . . Q .
     . . . Q                           Q . . .
     Q . . .                           . . . Q
     . . Q .                           . Q . .
```

Sum of Subsets Problem

Problem Definition

Given n positive numbers (weights) w_i and a target sum M, find all subsets of these weights whose sum is exactly M.

Formulation

- **Solution Vector:** A boolean tuple $(x_1, x_2, \ldots, x_n)$ where $x_i = 1$ if weight w_i is chosen, and 0 otherwise.
- **Pre-condition:** Sort the weights in non-decreasing order: $w_1 \leq w_2 \leq \cdots \leq w_n$.
- **Bounding Functions:**
  Let $s = \sum_{i=1}^{k} w_i x_i$ be the current sum. A partial solution can be pruned if:
  1. The sum exceeds the target: $s + w_{k+1} > M$.
  2. The remaining weights are insufficient to reach M: $s + \sum_{i=k+1}^{n} w_i < M$.

Algorithm

```
Algorithm SumOfSub(s, k, r)
// s is current sum, k is current index, r is remaining weight sum
{
    // Generate left child (Include w[k])
    x[k] := 1;
    if (s + w[k] = M) then
        Print(x[1..k]); // Found solution
    else if (s + w[k] + w[k+1] <= M) then
        SumOfSub(s + w[k], k + 1, r - w[k]);

    // Generate right child (Exclude w[k])
    if (s + r - w[k] >= M and s + w[k+1] <= M) then
    {
        x[k] := 0;
        SumOfSub(s, k + 1, r - w[k]);
    }
}
```

Step-by-Step Trace of Sum of Subsets

- **Weights:** $W = [5, 10, 12, 13, 15, 18]$, Target $M = 30$.
- **Initial Sum:** $s = 0$, index $k = 1$, remaining weight sum $r = 5 + 10 + 12 + 13 + 15 + 18 = 73$.

1. Try $x_1 = 1$ (Include 5): $s = 5, r = 68$.
2. Try $x_2 = 1$ (Include 10): $s = 15, r = 58$.
3. Try $x_3 = 1$ (Include 12): $s = 27, r = 46$.
   - Try $x_4 = 1$ (Include 13): $s + 13 = 40 > 30$ (Killed).
   - Try $x_4 = 0$ (Exclude 13): Remaining sum $27 + (15 + 18) = 60 \geq 30$. Recurse:
     - Try $x_5 = 1$ (Include 15): $27 + 15 = 42 > 30$ (Killed).
     - Try $x_5 = 0$ (Exclude 15): Remaining sum $27 + 18 = 45 \geq 30$. Recurse:
       - Try $x_6 = 1$ (Include 18): $27 + 18 = 45 > 30$ (Killed).

        - Try $x_6 = 0$ (Exclude 18): $27 + 0 = 27 < 30$ (Killed).
4. Backtrack to $x_3 = 0$ (Exclude 12): $s = 15$.
5. Try $x_4 = 1$ (Include 13): $s = 28$.
    - $28 + 15 > 30$ (Killed).
6. Try $x_4 = 0$ (Exclude 13): $s = 15$.
7. Try $x_5 = 1$ (Include 15): $s = 30$.
    - **Solution Found:** $\{5, 10, 15\} \implies (1, 1, 0, 0, 1, 0)$.

Following this search path, the algorithm discovers two additional solutions:

- **Solution 2:** $\{5, 12, 13\} \implies (1, 0, 1, 1, 0, 0)$
- **Solution 3:** $\{12, 18\} \implies (0, 0, 1, 0, 0, 1)$

Graph Coloring (m-Colorability)

Problem Definition

Find all ways to color the vertices of an undirected graph $G = (V, E)$ using at most m colors such that no two adjacent vertices share the same color.

Formulation

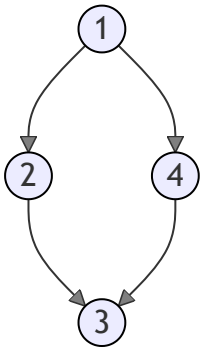
- **Representation:** Graph represented as adjacency matrix `G[1..n, 1..n]`.
- **Solution Vector:** An n-tuple $(x_1, x_2, \ldots, x_n)$ where $x_i \in \{1, 2, \ldots, m\}$ is the color of vertex i.
- **Implicit Constraint:** If `G[i, j] = 1` (edge exists), then $x_i \neq x_j$.

Algorithms

```
Algorithm NextValue(k, m)
// Assigns a valid color value to x[k]
{
    repeat
    {
        x[k] := (x[k] + 1) mod (m + 1); // Try next color
        if (x[k] = 0) then return;      // All colors tried

        // Check adjacency conflicts
        conflict := false;
        for j := 1 to k - 1 do
        {
            if (G[k, j] = 1 and x[k] = x[j]) then
            {
                conflict := true;
                break;
            }
        }
        if (not conflict) then return; // Valid color found
    } until false;
}

Algorithm GraphColoring(k, m)
// Recursively color node k
{
    repeat
    {
        NextValue(k, m); // Assign a valid color
        if (x[k] = 0) then return; // No color possible
        if (k = n) then
            Print(x[1..n]);
        else
            GraphColoring(k + 1, m);
    } until false;
}
```

Step-by-Step Trace of Graph Coloring

Let's color a cycle graph C_4 (4 vertices, edges: $(1,2), (2,3), (3,4), (4,1)$) with $m = 3$ colors.



1. **Vertex 1:** Assign color 1. ($x = [1, 0, 0, 0]$)
2. **Vertex 2:** Cannot be color 1 (adjacent to 1). Assign color 2. ($x = [1, 2, 0, 0]$)
3. **Vertex 3:** Adjacent to 2.
    - Try color 1. Valid (not connected to 1). ($x = [1, 2, 1, 0]$)
    - **Vertex 4:** Adjacent to 1 and 3.
        - Cannot be color 1 (adjacent to 1 and 3). Try color 2. Valid!
        - **Solution Found:** $(1, 2, 1, 2)$
    - Try color 3. Valid. ($x = [1, 2, 3, 0]$)
        - **Vertex 4:** Adjacent to 1 and 3.
            - Cannot be color 1 or 3. Try color 2. Valid!
            - **Solution Found:** $(1, 2, 3, 2)$

Hamiltonian Cycles

Problem Definition

Find a closed loop in a graph that visits every vertex exactly once and returns to the starting vertex.

Formulation

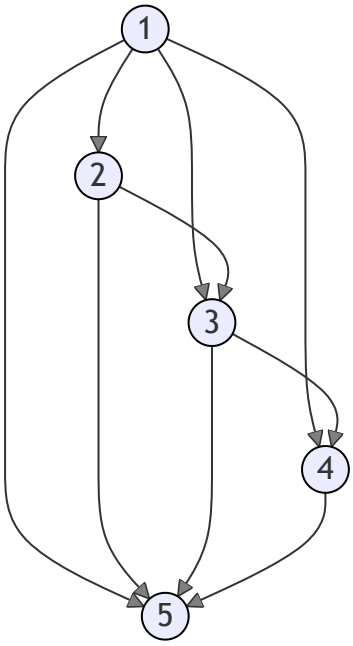
- **Solution Vector:** An n-tuple $(x_1, \ldots, x_n)$ containing the path vertices.
- **Starting Constraint:** Fix $x_1 = 1$ to remove cyclic shifts.
- **Implicit Constraints for x_k ($k > 1$):**
    1. Edge must exist: `G[x[k-1], x[k]] = 1`.
    2. Distinct vertices: $x_k \neq x_i$ for all $i < k$.
    3. Final return (for $k = n$): `G[x[n], x[1]] = 1`.

Algorithms

```
Algorithm NextValueHC(k)
{
    repeat
    {
        x[k] := (x[k] + 1) mod (n + 1); // Try next vertex
        if (x[k] = 0) then return;      // All vertices tried

        if (G[x[k-1], x[k]] = 1) then
        {
            // Check if vertex was already visited
            visited := false;
            for j := 1 to k - 1 do
                if (x[j] = x[k]) then visited := true;

            if (not visited) then
            {
                // If last vertex, check return edge to start x[1]
                if (k < n or (k = n and G[x[n], x[1]] = 1)) then
                    return; // Valid vertex
            }
        }
    } until false;
}

Algorithm Hamiltonian(k)
{
    repeat
    {
        NextValueHC(k);
        if (x[k] = 0) then return;
        if (k = n) then
            Print(x[1..n] + " " + x[1]); // Cycle completed
        else
            Hamiltonian(k + 1);
    } until false;
}
```

Step-by-Step Trace of Hamiltonian Cycles

Let's find a Hamiltonian cycle in a 5-vertex graph:



Let's assume adjacency list for Node 1: $[2, 3, 4, 5]$. Node 2: $[1, 3, 5]$, etc.

1. Start at $x_1 = 1$.
2. Try $x_2 = 2$.
3. Try $x_3 = 3$.
4. Try $x_4 = 4$.
5. Try $x_5 = 5$.
   - Check if $x_5 = 5$ has edge back to $x_1 = 1$. Yes, edge $(5, 1)$ exists.
   - **Cycle Found:** $1 \to 2 \to 3 \to 4 \to 5 \to 1$.
   - If $(5, 1)$ did not exist, the algorithm would backtrack, try $x_5 = 0$, return to x_4, and try other configurations.

> Website: https://r25.jntuh.harshrb.in
> YouTube: https://www.youtube.com/@QueryTheConcept