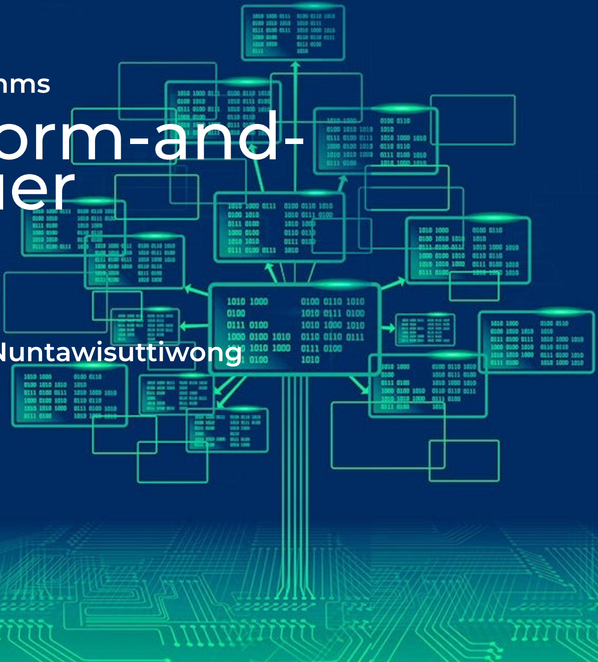
CPE 231 Algorithms

Transform-and-Conquer



Dr. Taweechai Nuntawisuttiwong

Contents

Transform-and-Conquer

Transform-and-Conquer

Definition

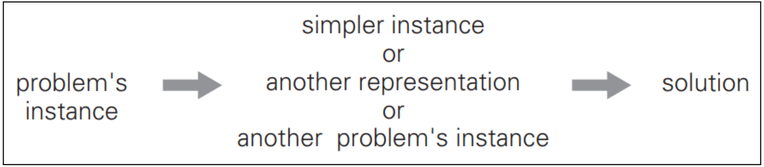
Transform-and-Conquer is a powerful algorithm design paradigm that involves transforming a problem into a different version of itself, which is easier to solve. The transformed problem is then solved, and the solution is mapped back to the original problem.

Process

1. **Transformation Stage**: Modify the problem instance to a simpler or more convenient form.
2. **Conquering Stage**: Solve the transformed problem efficiently.

Types of Transform-and-Conquer Strategies

1. **Instance Simplification**: The problem is transformed into a simpler version of itself. This might involve reducing the size of the input, removing redundancies, or sorting data.
2. **Representation Change**: The problem instance is represented in a different form that is more amenable to solution.
3. **Problem Reduction**: The problem is transformed into a different problem altogether, one for which a known algorithm exists.



Why Transform-and-Conquer?

- By transforming a problem, we often reduce the time complexity and make the problem more tractable.
- This approach can be applied to a wide variety of problems across different domains, from sorting and searching to optimization and geometry.
- The ability to change the problem's structure provides flexibility in approaching difficult problems and finding innovative solutions.

Presorting

Presorting

Definition

Presorting is the process of sorting data before performing other operations.

- Simplifies many algorithmic problems.
- Leads to more efficient algorithms by reducing problem complexity.
- **Common Applications**: Element uniqueness, mode computation, searching, and geometric algorithms.

Element Uniqueness

- Determine if all elements in an array are unique.
- Brute-force Solution:
  - Compare each element with every other element.
  - Time Complexity:

ALGORITHM *UniqueElements*$(A[0..n-1])$
    //Determines whether all the elements in a given array are distinct
    //Input: An array $A[0..n-1]$
    //Output: Returns "true" if all the elements in A are distinct
    //   and "false" otherwise
    for $i \leftarrow 0$ **to** $n-2$ **do**
        for $j \leftarrow i+1$ **to** $n-1$ **do**
            if $A[i] = A[j]$ **return false**
    return true

Element Uniqueness

Presorting Solution:

1. Sort the array.
2. Compare only adjacent elements.

ALGORITHM *PresortElementUniqueness*($A[0..n-1]$)

//Solves the element uniqueness problem by sorting the array first
//Input: An array $A[0..n-1]$ of orderable elements
//Output: Returns "true" if A has no equal elements, "false" otherwise
sort the array A
for $i \leftarrow 0$ **to** $n-2$ **do**
    if $A[i] = A[i+1]$ **return false**
return true

Complexity: $T(n) = T_{sort}(n) + T_{scan}(n)$

Computing a Mode

- Find the mode (most frequent element) in an array.
- Brute-force Solution:
  1. **Initialize Frequency Count**: Create an auxiliary list or a dictionary to keep track of the frequency of each distinct element in the input list.
  2. **Traverse the Input List**: For each element in the list
     - Check if the element is already in the auxiliary list.
     - **If the element is found**: Increment its frequency count.
     - **If the element is not found**: Add the element to the auxiliary list with an initial frequency count of 1.
  3. **Determine the Mode**: After populating the frequency list, traverse it to find the element with the highest frequency. This element is the mode of the list.

Computing a Mode

Example: {5, 1, 5, 7, 6, 5, 7}

Auxiliary List (Frequency Count):

After processing 5:

After processing 1:

After processing 5:

After processing 7:

After processing 6:

After processing 5:

After processing 7:

Result: Mode =

Computing a Mode

Presorting Solution:

1. Sort the array.
2. Scan for the longest sequence of identical elements.

```
ALGORITHM  PresortMode(A[0..n − 1])
    //Computes the mode of an array by sorting it first
    //Input: An array A[0..n − 1] of orderable elements
    //Output: The array's mode
    sort the array A
    i ← 0                        //current run begins at position i
    modefrequency ← 0            //highest frequency seen so far
    while i ≤ n − 1 do
        runlength ← 1;   runvalue ← A[i]
        while i + runlength ≤ n − 1 and A[i + runlength] = runvalue
            runlength ← runlength + 1
        if runlength > modefrequency
            modefrequency ← runlength;   modevalue ← runvalue
        i ← i + runlength
    return modevalue
```

Computing a Mode

Example: {5, 1, 5, 7, 6, 5, 7}

Sorted List: {1, 5, 5, 5, 6, 7, 7}

i	modefrequency	runlength	runvalue	modevalue

Result: Mode =

Computing a Mode

Complexity

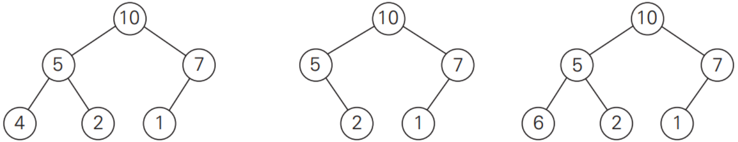
Brute-force solution:

Presorting solution:

Heaps and Heapsort

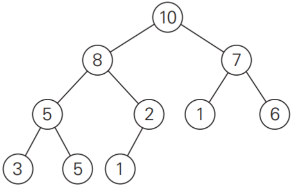
Heaps

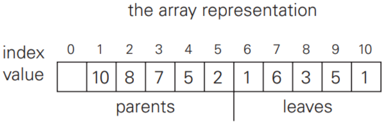
- A heap is a **binary tree** with two properties:
  1. **Shape Property**: The tree is complete, meaning all levels are full except possibly the last level, which is filled from left to right.
  2. **Heap Property**: The key in each node is greater than or equal to the keys in its children (for a max-heap).
- Heaps are used to implement priority queues, which support operations like finding, inserting, and deleting the highest-priority element efficiently.



Heap Properties

1. There exists exactly one essentially complete binary tree with n nodes. Its height is equal to $\lfloor \log 2n \rfloor$.
2. The root of a heap always contains its largest element.
3. A node of a heap considered with all its descendants is also a heap.
4. A heap can be implemented as an array by recording its elements in the topdown, left-to-right fashion.
   a. the parental node keys will be in the first $\lfloor n/2 \rfloor$ positions of the array, while the leaf keys will occupy the last $\lceil n/2 \rceil$ positions;
   b. the children of a key in the array's parental position i ($1 \leq i \leq \lfloor n/2 \rfloor$) will be in positions $2i$ and $2i + 1$, and, correspondingly, the parent of a key in position i ($2 \leq i \leq n$) will be in position $\lfloor i/2 \rfloor$.





Bottom-Up Heap Construction

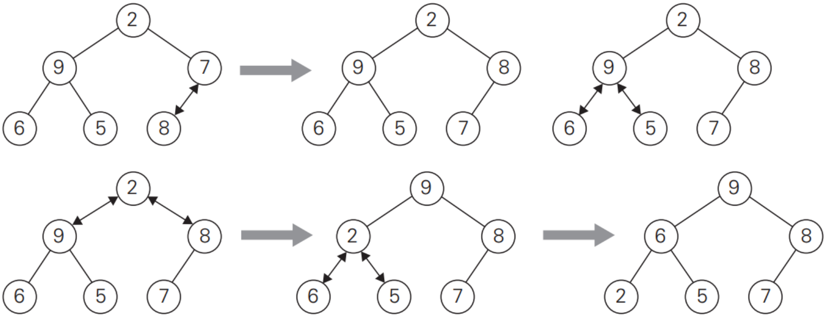
- Start with a complete binary tree.
- Convert it into a heap by **heapifying**: ensuring the heap property from the last parent node to the root.

ALGORITHM *HeapBottomUp*$(H[1..n])$

//Constructs a heap from elements of a given array
// by the bottom-up algorithm
//Input: An array $H[1..n]$ of orderable items
//Output: A heap $H[1..n]$

for $i \leftarrow \lfloor n/2 \rfloor$ **downto** 1 **do**
  $k \leftarrow i; \quad v \leftarrow H[k]$
  $heap \leftarrow$ **false**
  while not $heap$ **and** $2 * k \leq n$ **do**
    $j \leftarrow 2 * k$
    if $j < n$ //there are two children
      if $H[j] < H[j+1]$ $j \leftarrow j+1$
    if $v \geq H[j]$
      $heap \leftarrow$ **true**
    else $H[k] \leftarrow H[j]; \quad k \leftarrow j$
  $H[k] \leftarrow v$

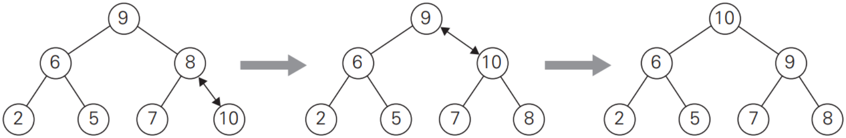
Bottom-Up Heap Construction



Efficiency:

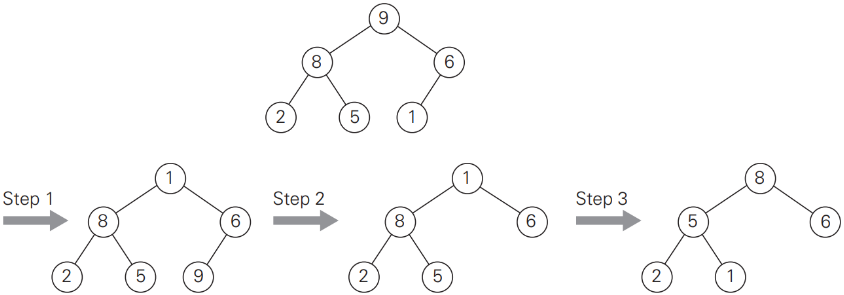
Top-Down Heap Construction

- Insert the new key to the last leaf.
- Repeatedly swap the new key with its parent until the parental dominance is satisfied.



Efficiency:

Maximum Key Deletion



1. Swap the root element with the last element in the heap.
2. Decrease the size of the heap by 1.
3. Starting from the new root, restore the heap property by "sifting down" the element to its correct position.

Heapsort

1. (heap construction): Construct a heap for a given array.
2. (maximum deletions): Apply the root-deletion operation n - 1 times to the remaining heap.

Heapsort

Example: {2, 9, 7, 6, 5, 8}

Stage 1 (heap construction)

Heapsort

Example: {2, 9, 7, 6, 5, 8}

Stage 2 (maximum deletions)

Heapsort

Efficiency

Problem Reduction

Problem Reduction

Definition

A problem-solving strategy where a complex problem is reduced to a simpler, known problem that can be solved more easily.

How it works:

1. **Identify the Problem**: Start with a complex problem that needs to be solved.
2. **Reduction**: Transform this problem into another problem that you know how to solve.
3. **Solve and Map**: Solve the reduced problem and map the solution back to the original problem.

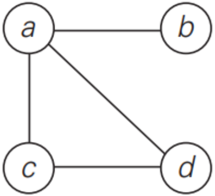
Least Common Multiple (LCM)

- Compute the LCM of two integers.
- **Brute-force Method**: Requires prime factorization and is inefficient.
- **Reduction Approach**: Use the formula

$$lcm(m,n) = \frac{m \cdot n}{gcd(m,n)}$$

Counting Paths in a Graph

- Count the number of paths of a certain length between two vertices in a graph.
- **Reduction Approach**:
  - **Adjacency Matrix**: Use the graph's adjacency matrix.
  - **Matrix Exponentiation**: The number of paths of length k between two vertices can be found using the kth power of the adjacency matrix.



$$A = \begin{array}{c c} & \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{array}{c} a \\ b \\ c \\ d \end{array} & \begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{bmatrix} \end{array} \qquad A^2 = \begin{array}{c c} & \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{array}{c} a \\ b \\ c \\ d \end{array} & \begin{bmatrix} 3 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 1 & 2 & 1 \\ 1 & 1 & 1 & 2 \end{bmatrix} \end{array}$$