

ALGORITHMS AND LAB (CSE130)

SEARCHING

Muhammad Tariq Mahmood

tariq@koreatech.ac.kr
School of Computer Science and Engineering

Note: These notes are prepared from the following resources.

- (main text) Foundations of Algorithms, by Richard Neapolitan and Kumarss Naimipour
- Python Algorithm (파이썬 알고리즘) by Y.K. Choi (2021) (Korean)
- Introduction to the Design and Analysis of Algorithms by Anany Levitin
- Introduction to Algorithms, by By Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein
- <https://www.geeksforgeeks.org>

1 SORTING

2 COMPARISONS-BASED SORTING ALGORITHMS

- Bubble Sort
- Insertion Sort
- HeapSort

3 DIVIDE/DECREASE AND CONQUER APPROACHES FOR SORTING

- Merge Sort
- Quicksort

4 LOWER BOUNDS FOR SORTING ONLY BY COMPARISON OF KEYS

5 NON-COMPARISONS-BASED SORTING ALGORITHMS

- Counting sort
- Radix Sort

CONTENTS

➊ SEARCHING

➋ INTERPOLATION SEARCH

➌ SELECTION PROBLEM

➍ SEARCHING IN TREES

➎ FEW SEARCHING PROBLEM

- Fake-Coin Problem
- Post office location problem

SEARCHING ALGORITHMS

Searching

- Like sorting, searching is one of the most useful operation in computing.
- The problem is usually to retrieve an entire record(object) based on the value of some **key** field. For example, a record may consist of personal information, whereas the key field may be the social security number.
- Formally, the problem of searching for a key can be described as follows: Given an array S containing n keys and a key x , find an index i such that $x = S[i]$ if x equals one of the keys; if x does not equal one of the keys, report failure.
- Based on the type of search operation, these algorithms are generally classified into two categories:
 - ① **Sequential Search:** In this, the list or array is traversed sequentially and every element is checked. For example: Linear Search.
 - ② **Interval Search:** These algorithms are specifically designed for searching in sorted data-structures. These type of searching algorithms are much more efficient than Linear Search as they repeatedly target the center of the search structure and divide the search space in half. For Example: Binary Search.

SEARCHING ALGORITHMS (CONT...)

- Binary Search

```
1: procedure LOCATION(low, high)
2:   if (low > high) then
3:     return 0
4:   else
5:      $mid = \left\lfloor \frac{(low+high)}{2} \right\rfloor$ 
6:     if (x == S[mid]) then return mid
7:     else
8:       if (x < S[mid]) then
9:         return LOCATION(low, mid - 1)
10:      else
11:        return LOCATION(mid + 1, high)
12:      end if
13:    end if
14:  end if
15: end procedure
```

- Complexity: $T(n) \in \Theta(\log_2 n)$

- Elements in array should be in order

- Sequential Search

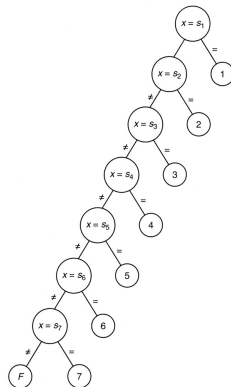
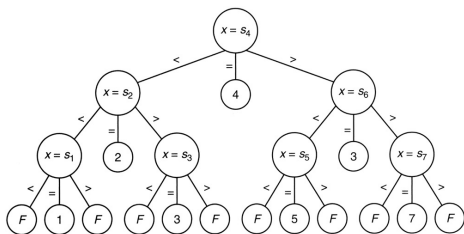
```
1: procedure SEQSEARCH(S[], x)
2:   integer location
3:   location = 1
4:   while location <= n do
5:     location = location + 1
6:     if (S[location] == x) then
7:       return location
8:     end if
9:   end while
10:  location = -1
11:  return location
12: end procedure
```

- Complexity: $T(n) \in \Theta(n)$

- Elements in array may not be in order

SEARCHING ALGORITHMS (CONT...)

- A decision tree with every deterministic searching algorithm can be associated.
- The decision tree corresponding to Binary Search when searching seven keys



- The decision tree corresponding to Sequential Search when searching seven keys.
- Lower Bounds for Worst-Case: $\Theta(\lg n)$. Recall that worst-case time complexity for Binary Search is $\lceil \lg n \rceil + 1$. Can we improve on this performance?

INTERPOLATION SEARCH

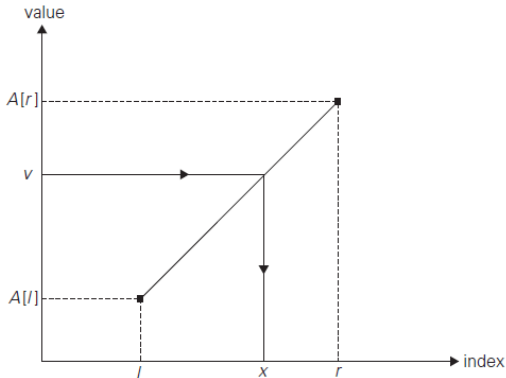
Interpolation Search

- Unlike binary search, which always compares a search key with the middle value of a given sorted array, interpolation search takes into account the **value** of the search key in order to find the array's element to be compared with the search key.
- So the array size is not divided into equal two halves, instead, array is decreased through **variable-size-decrease** mechanism
- Index x computation in interpolation search.

$$x = low + \left\lfloor \frac{v - A[low]}{A[high] - A[low]} \times (high - low) \right\rfloor$$

- For example, if $S[1] = 4$ and $S[10] = 97$, and we were searching for $v = 25$,

$$x = 1 + \left\lfloor \frac{25 - 4}{97 - 4} \times (10 - 1) \right\rfloor = 3$$



INTERPOLATION SEARCH (CONT...)

- Algorithm

```
1: procedure INTERPOLATIONSEARCH(S, x)
2:   low = 1; n=high = len(S); i = 0;
3:   if (S[low] ≤ x ≤ S[high]) then
4:     while (low ≤ high and i == 0) do
5:       denominator = S[high]-S[low]
6:       if (denominator == 0) then, mid = low
7:       else
8:         mid = low + [((x - S[low]) * (high - low)) /
denominator]
9:       end if
10:      if (x==S[mid]) then, i = mid;
11:      else
12:        if (x < S[mid]) then, high = mid- 1;
13:        else low = mid + 1;
14:      end if
15:    end if
16:  end while
17:  end if
18:  return i
19: end procedure
```

- Average case complexity for Interpolation Search
 $\Theta(\lg(\lg n))$

- Consider the input [1, 2, 3, 4, 5, 6, 7, 8, 9, 100] and x = 10, then

$$mid = 1 + \lfloor \frac{10 - 1}{100 - 1} \times (10 - 1) \rfloor = 1$$

- In this case, Interpolation Search will take time $\Theta(n)$
- A variation of Interpolation Search called **Robust Interpolation Search**

$$\begin{aligned} gap &= \lfloor (high - low + 1)^{\frac{1}{2}} \rfloor \\ mid &= \min(high - gap, \min(mid, low + gap)) \\ mid &= \min(10 - 3, \max(1, 1 + 3)) = 4 \end{aligned}$$

SELECTION PROBLEM

Selection Problem

- **Selection Problem:** The selection problem is the problem of finding the k th smallest element (rank of an element) in a list of n numbers
- This number is called the k th **order statistic**.
- The **rank** of an element is its position (index) when the set is sorted.
- The **minimum** is of rank 1 and the **maximum** is of rank n .
- Consider the set: $\{5, 7, 2, 10, 8, 15, 21, 37, 41\}$. The rank of each number is its position in the sorted order.

<i>rank</i>	1	2	3	4	5	6	7	8	9
<i>items</i>	2	5	7	8	10	15	21	37	41

- **selection problem** can be stated as: Given a set A of n distinct numbers and an integer k . return the element from the set A of rank k .

SELECTION PROBLEM (CONT...)

- Finding the Largest Key

- Problem: Find the largest key in the array S of size n .
- Inputs: positive integer n , array of keys S indexed from 1 to n .
- Outputs: variable $large$, whose value is the largest key in S
 - 1: **procedure** FINDLARGEST (S)
 - 2: **return** $large=S[1]$
 - 3: **for** ($i = 2; i \leq n; i++$) **do**
 - 4: **if** ($S[i] > large$) **then** $large = S[i];$
 - 5: **end if**
 - 6: **end for**
 - 7: **return** $large$
 - 8: **end procedure**
- Complexity $T(n) = n - 1$

- Finding Both the Smallest and Largest Keys

- Problem: Find the smallest and largest keys in an array S of size n .
- Inputs: positive integer n , array of keys S indexed from 1 to n .
- Outputs: variables $small$ and $large$, whose values are the smallest and largest keys in S .
 - 1: **procedure** FINDSL(S)
 - 2: $small=large=S[1]$
 - 3: **for** ($i = 2; i \leq n; i++$) **do**
 - 4: **if** ($S[i] > large$) **then** $large = S[i];$
 - 5: **end if**
 - 6: **if** ($small > S[i]$) **then** $small = S[i];$
 - 7: **end if**
 - 8: **end for**
 - 9: **return** [$large, small$]
 - 10: **end procedure**
- Complexity $T(n) = 2(n - 1)$

Can we improve on this performance?

SELECTION PROBLEM (CONT...)

- **Paring Keys:** The trick is to pair the keys and find which key in each pair is smaller.
- We can then find the smallest of all the smaller keys with about $n/2$ comparisons and the largest of all the larger keys with about $n/2$ comparisons.
- Complexity

$$T(n) = \begin{cases} \frac{3n}{2} - 2 & n \text{ is even} \\ \frac{3n}{2} - \frac{2}{3} & n \text{ is odd} \end{cases}$$

- Algorithm

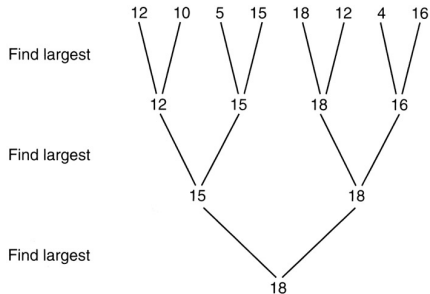
```
1: procedure FINDSLPK(S)
2:   if (S[1] < S[2]) then small = S[1]; large = S[2];
3:   else, small = S[2]; large = S[1];
4:   end if
5:   for (i = 3; i <= n - 1; i = i+2) do
6:     if (S[i] < S[i + 1])
7:       if (S[i] < small) then, small = S[i];
8:     end if
9:     if (S[i+1] > large) then large = S[i+1];
10:    end if
11:   else
12:     if (S[i+1] < small) then, small = S[i+1];
13:   end if
14:     if (S[i] > large) then , large = S[i];
15:   end if
16:   end if
17:   end for
18:   return [large, small]
19: end procedure
```

SELECTION PROBLEM (CONT...)

Finding the Second-Largest Key

- **Tournament selection** is a method of selecting an individual from a group of people based on the pairing technique.
- A **tournament tree** is a complete binary tree reflecting results of a “**knockout tournament**”: its leaves represent n players entering the tournament, and each internal node represents a winner of a match played by the players represented by the node's children.
- The winner of the tournament is represented by the root of the tree., the loser in that round is not necessarily the second largest

- To find the second-largest key, we can keep track of all the keys that lose to the largest key,



SELECTION PROBLEM (CONT...)

Finding Medians

- Of particular interest in statistics is the **median**. If n is odd then the median is defined to be element of rank $(n + 1)/2$. When n is even, there are two choices: $n/2$ and $(n + 1)/2$.
- **Medians** are useful as measures of the central tendency of a set especially when the distribution of values is highly skewed.
- The **selection problem** can be easily solved by simply sorting the numbers of A and returning $A[k]$. Sorting algorithms have complexity $\Theta(n \log(n))$
- The **selection problem** can be solve in linear time $\Theta(n)$ by using the **sieve technique**
- **sieve technique** calls partition algorithm once in an iteration.
 - ▶ Sieve technique is a special case of divide-and-conquer.
 - ▶ In divide-and-conquer approach the problem is divided into a small number of smaller subproblems, which are then solved recursively.
 - ▶ The sieve technique is a special case, where the number of subproblems is just 1.
 - ▶ This technique can be used to find the rank of any element in an array (selection and medians).

SELECTION PROBLEM (CONT...)

Algorithm

- **Problem:** Find the k th smallest key of n keys.
- **Inputs:** the integer k , and an array of keys S indexed from 1 to n .
- **Outputs:** The k th smallest element (return the element of the rank k)

```
1: procedure SELECTIONRANK( $S, kRank, low, high$ )
2:   if ( $high == low$ ) then
3:     return  $S[low]$ 
4:   else
5:      $pp = partition(low, high, S)$ 
6:     if ( $kRank == pp$ ) then return  $S[pp]$ 
7:     else
8:       if ( $kRank < pp$ ) then
9:         selectionRank( $S, kRank, low, pp - 1$ )
10:      else
11:        selectionRank( $S, kRank, pp + 1, high, S$ )
12:      end if
13:    end if
14:  end if
15:  return  $S[kRank]$ 
16: end procedure
```

SELECTION PROBLEM (CONT...)

Partition Algorithm

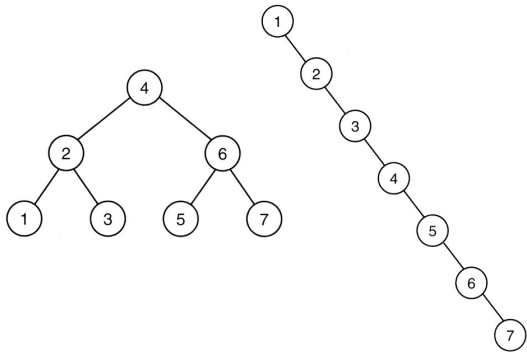
- **Problem:** Partition the array S .
- **Inputs:** two indices, low and high, and the subarray of S indexed from low to high.
- **Outputs:** pp , the pivot point for the subarray indexed from low to high.
- **Pseudo-code**
 - 1: **procedure** PARTITION($A, low, high$)
 - 2: $i = 0$ $j = low$, $pp = 0$
 - 3: $pivot = A[randomnumber(1...n)]$
 - 4: **for** ($i = low + 1; i \leq high; i++$) **do**
 - 5: **if** ($A[i] < pivot$) **then**
 - 6: $j = j + 1$
 - 7: $exchange(A[i] \text{ and } A[j])$
 - 8: **end if**
 - 9: **end for**
 - 10: $exchange(A[low] \text{ and } A[j])$
 - 11: **return** j
 - 12: **end procedure**
- Partition method is the most important function in quick sort.
- It rearranges the elements of the array so that all the elements to the left of the pivot are smaller than the pivot and all the elements to the right are greater than the pivot.
- The pivot element is selected randomly

$$T(n) = \sum_{i=2}^n 1 = n - 1$$

SEARCHING IN TREES

Searching in Trees

- By **static searching** we mean a process in which the records are all added to the file at one time and there is no need to add or delete records later.
- **dynamic searching**, which means that records are frequently added and deleted.
- **Binary search trees (BST)** are considered appropriate data structure for searching
- A binary search tree is a binary tree of items, such that
 - 1 Each node contains one key.
 - 2 The keys in the left subtree of a given node are less than or equal to the key in that node.
 - 3 The keys in the right subtree of a given node are greater than or equal to the key in that node.



- The drawback of binary search trees is that when keys are dynamically added and deleted, there is no guarantee that the resulting tree will be balanced

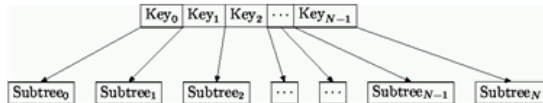
SEARCHING IN TREES (CONT...)

- In a very dynamic environment, it would be better if the tree never became unbalanced in the first place.
- Algorithms for adding and deleting nodes while maintaining a balanced binary tree were developed in 1962 by two Russian mathematicians, G. M. Adel'son-Velskii and E. M. Landis. (For this reason, balanced binary trees are often called AVL trees.)
- In such balanced binary trees, insertion, deletion, and searching are guaranteed to be $\Theta(\lg n)$.
- In 1972, R. Bayer and E. M. McCreight developed an improvement over binary search trees called **B-trees**.
- In **B-trees**, nodes may contain many keys instead of only one. Parent node may have more than two children nodes, so the trees are not binary.
- When keys are added to or deleted from a B-tree, all leaves are guaranteed to remain at the same level, which is even better than maintaining balance.
- See animation at : <https://www.cs.usfca.edu/galles/visualization/BTree.html>

Node in a Binary Search Tree



Node in a B-Tree

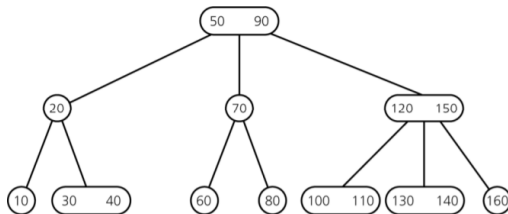


SEARCHING IN TREES (CONT...)

- B-trees actually represent a class of trees, of which the simplest is a 2-3 tree.

- A 2-3 tree is a tree with the following properties:

- ▶ Each node contains one or two keys.
- ▶ If a nonleaf contains one key, it has two children, whereas if it contains two keys, it has three children.
- ▶ The keys in the left subtree of a given node are less than or equal to the key stored at that node.
- ▶ The keys in the right subtree of a given node are greater than or equal to the key stored at that node.
- ▶ If a node contains two keys, the keys in the middle subtree of the node are greater than or equal to the left key and less than or equal to the right key.
- ▶ All leaves are at the same level.



- The worst case time-complexity of operations such as search, insertion and deletion is $O(\log(n))$ as the height of a 2-3 tree is $O(\log(n))$.

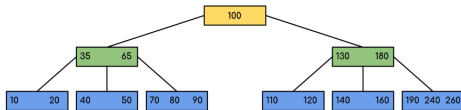
SEARCHING IN TREES (CONT...)

- **Search Operation in B-Tree:** Searching a B-Tree is similar to searching a binary tree. The algorithm is similar and goes with recursion. At each level, the search is optimised as if the key value is not present in the range of parent then the key is present in another branch. If we reach a leaf node and don't find the desired key then it will display NULL.

- Let the key to be searched be x .

- **Base cases:**

- ▶ If BTree is empty, return False (key cannot be found in the tree).
- ▶ If current node contains data value which is equal to x , return True.
- ▶ If we reach the leaf-node and it doesn't contain the required key value x , return False.



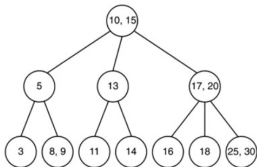
- **Recursive cases:**

- ▶ If $x < \text{currentNode.leftVal}$, we explore the left subtree of the current node.
- ▶ Else if $\text{currentNode.leftVal} < x < \text{currentNode.rightVal}$, we explore the middle subtree of the current node.
- ▶ Else if $x > \text{currentNode.rightVal}$, we explore the right subtree of the current node.

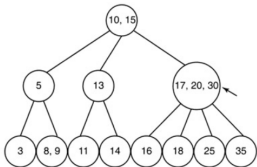
SEARCHING IN TREES (CONT...)

- The process of adding nodes to such trees.

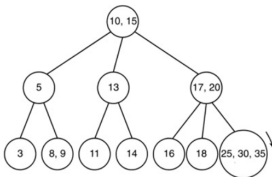
(a) A 3—2 tree



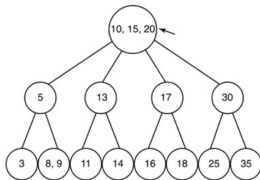
(c) If the leaf contains three keys, it breaks into two nodes and sends the middle key up to its parent.



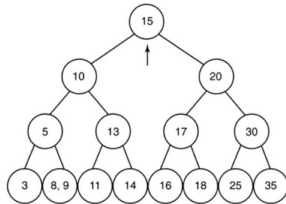
(b) 35 is added to the tree in sorted sequence in a leaf.



(d) If the parent now contains three keys, the process of breaking open and sending the middle key up repeats.




(e) Finally, if the root contains three keys, it breaks open and sends the middle key up to a new root.



- Notice that the tree remains balanced because the tree grows in depth at the root instead of at the leaves.
- B-trees are used in most modern database management systems.

FAKE-COIN PROBLEM

Problem definition

- Start with n coins, all the same except for one fake coin which is lighter than the others.
 - We have a **balance scale** which lets us compare any two piles of coins, to see if they are equal or if one pile is lighter than the other (and which pile is lighter).
 - A divide and conquer approach can be applied for searching a fake coin
- 
- The image shows a classic balance scale with a central vertical pillar and a horizontal beam. Two pans are suspended from the beam by thin wires. Each pan contains a stack of five gold coins. The scale is perfectly balanced, with the beam horizontal and the pans at equal heights. The background is a light blue gradient.
- There are three cases
 - ① If n is $n = 3k$ (i.e. $n \% 3 = 0$) : we can divide the coins into **three piles** of (k, k, k) coins each and weigh two of the piles
 - ② If n is $n = 3k + 1$ (i.e. $n \% 3 = 1$) : we can divide the coins into the piles of sizes $(k, k, \text{and } k + 1)$ $(k + 1, k + 1, \text{and } k - 1)$
 - ③ If n is $n = 3k + 2$ (i.e. $n \% 3 = 2$) :we will divide the coins into the piles of sizes $(k + 1, k + 1, \text{and } k)$

FAKE-COIN PROBLEM (CONT...)

Algorithm: Fake-Coin Problem (Pseudo-code)

- **Problem:** Determine the fake coin among the n coins given
- **Inputs:** an array of coins $C[]$ of length n . It is assumed that there is exactly one fake coin among the coins given and that the fake coin is lighter than the other coins.
- **Outputs:** return the fake coin $C[k]$ stored at k_{th} index.

• Complexity Analysis

$$T(n) = \begin{cases} T(1) = 1 & n = 1 \\ T(\lceil \frac{n}{3} \rceil) + 2 & n > 1 \end{cases}$$

• Solution

$$T(n) = \log_3(n) + 2$$

```
1: procedure FINDFAKECOIN( $n$ ,  $C[]$ )
2:   if ( $n == 1$ ) then return  $C[1]$  the coin is fake
3:   else
4:     piles  $C_1[], C_2[], C_3[]$  of sizes
        $\lceil \frac{n}{3} \rceil, \lceil \frac{n}{3} \rceil, \lceil n - 2(\frac{n}{3}) \rceil$ 
5:     if ( $W(C_1[]) == W(C_2[])$ ) then
6:        $findFakeCoin(n, C_3[])$ 
7:     else
8:       if ( $W(C_1[]) > W(C_2[])$ ) then
9:          $findFakeCoin(n, C_2[])$ 
10:      else
11:         $findFakeCoin(n, C_1[])$ 
12:      end if
13:    end if
14:  end if
15: end procedure
```

FAKE-COIN PROBLEM (CONT...)

- Let rewrite the code

```
1: procedure FINDFAKECOIN( $n$ ,  $C[]$ )
2:   if ( $n == 1$ ) then
3:     the coin is fake
4:     return  $C[1]$ 
5:   else
6:     divide the coins into  $b$  piles
7:      $C_m = \text{selectPile}(C_1[] \dots C_b[])$ 
8:      $\text{findFakeCoin}(n, C_m[])$ 
9:   end if
10: end procedure
```

▷ Base Case

▷ Recursive Cases

▷ compare the weights of b piles, and select the lightest

- In general case, we can write as

$$T(n) = \begin{cases} T(1) = 1 & n = 1 \\ T(\lceil \frac{n}{b} \rceil) + \underbrace{b-1}_{\text{No of comparisons for selecting pile}} & n > 1 \end{cases}$$

FAKE-COIN PROBLEM (CONT...)

- case $b=2$

$$T(n) = \begin{cases} T(1) = 1 & n = 1 \\ T(\lceil \frac{n}{2} \rceil) + 1 & n > 1 \end{cases}$$
$$T(n) \in \Theta(\log_2 n)$$

- case $b=3$

$$T(n) = \begin{cases} T(1) = 1 & n = 1 \\ T(\lceil \frac{n}{3} \rceil) + 2 & n > 1 \end{cases}$$
$$T(n) = \log_3 n + 2$$

- case $b=4$

$$T(n) = \begin{cases} T(1) = 1 & n = 1 \\ T(\lceil \frac{n}{4} \rceil) + 3 & n > 1 \end{cases}$$
$$T(n) = \log_4 n + 3$$

- case $b=n$

$$T(n) = \begin{cases} T(1) = 1 & n = 1 \\ T(\lceil \frac{n}{n} \rceil) + n - 1 & n > 1 \end{cases}$$
$$T(n) = \log_n n + n - 1$$

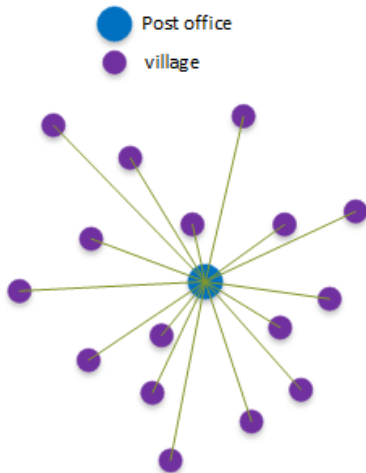
POST OFFICE LOCATION PROBLEM

Post office location problem

- Consider the two-dimensional post office location problem: given n points (villages) $V[(x_1, y_1), \dots, (x_n, y_n)]$ in the Cartesian plane, find a location $p(x, y)$ for a post office that minimizes the distance from the post office to these points (villages).

$$\minDist = \frac{1}{n} \sum_{i=1}^n d(p, V_i)$$

- There are several alternative ways to define a distance between two points $p_1(x_1, y_1)$ and $p_2(x_2, y_2)$ in the Cartesian plane.
- Euclidean distance :
 $d_E(p_1, p_2) = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$
- Manhattan distance : $d_M(p_1, p_2) = |x_2 - x_1| + |y_2 - y_1|$



SUMMARY

➊ SEARCHING

➋ INTERPOLATION SEARCH

➌ SELECTION PROBLEM

➍ SEARCHING IN TREES

➎ FEW SEARCHING PROBLEM

- Fake-Coin Problem
- Post office location problem