SNS COLLEGE OF TECHNOLOGY

Coimbatore-35

An Autonomous Institution

Accredited by NBA – AICTE and Accredited by NAAC – UGC with 'A++' Grade

Approved by AICTE, New Delhi & Affiliated to Anna University, Chennai

COURSE NAME : 23ITT201 DATA STRUCTURES

II YEAR/ III SEMESTER

UNIT – V SEARCHING,SORTING AND HASHING

Topic: *BINARY SEARCH*

Binary Search Greedy algorithm

Binary Search Algorithm

May only be used on a sorted array.
Eliminates one half of the elements after each comparison.

Locate the middle of the array

Compare the value at that location with the search key.

If they are equal - done!
Otherwise, decide which half of the array contains the search key.
Repeat the search on that half of the array and ignore the other half.

The search continues until the key is matched or no elements remain to be searched.

Binary Search Greedy algorithm

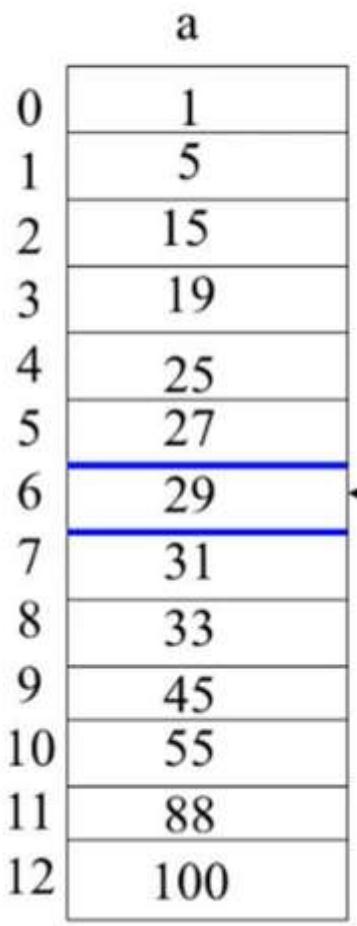
- The binary search algorithm can only be applied to an array that is sorted; furthermore, the order of sorting must be known.
- The binary search algorithm for an array sorted in ascending order is:

–if(there are more elements in the current array){
–Compare the current middle array element with key (the value searched for).
–There are three possibilities:
- 1. key == array[middle]
- the search is successful, return middle.
- 2. key < array[middle]
- binary search the current lower half of the array.
- 3. key > array[middle]
- binary search the current upper half of the array.

– }
– the search is not successful; return –1

Binary Search

Binary Search Example

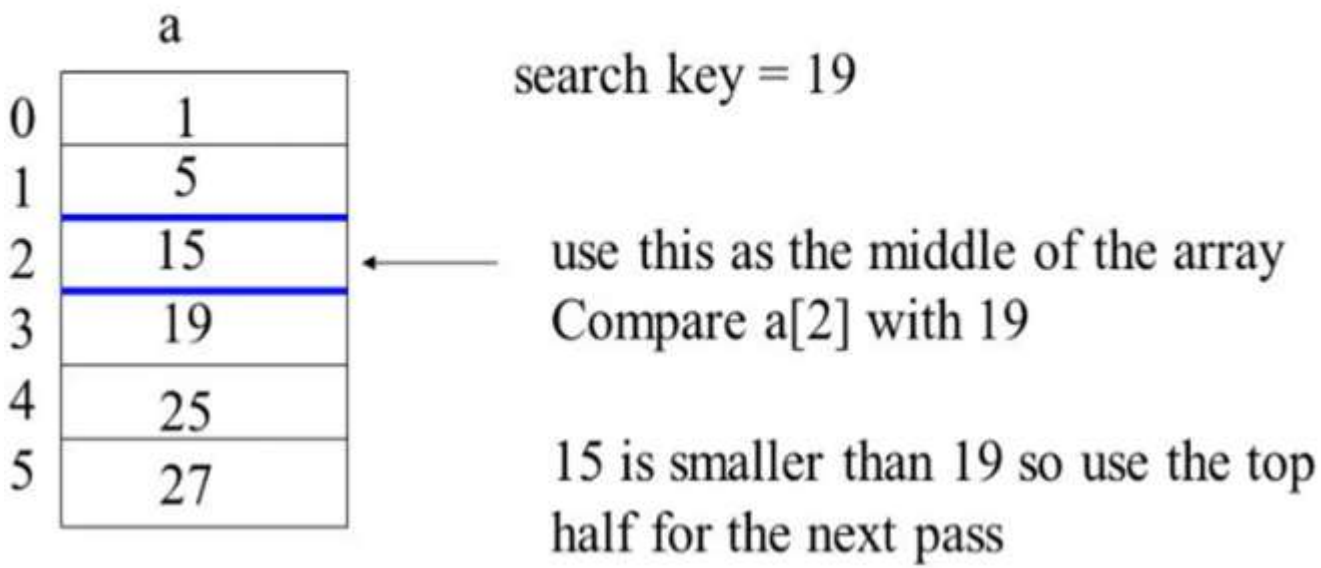


search key = 19

middle of the array
compare a[6] and 19
19 is smaller than 29 so the next
search will use the lower half of the array

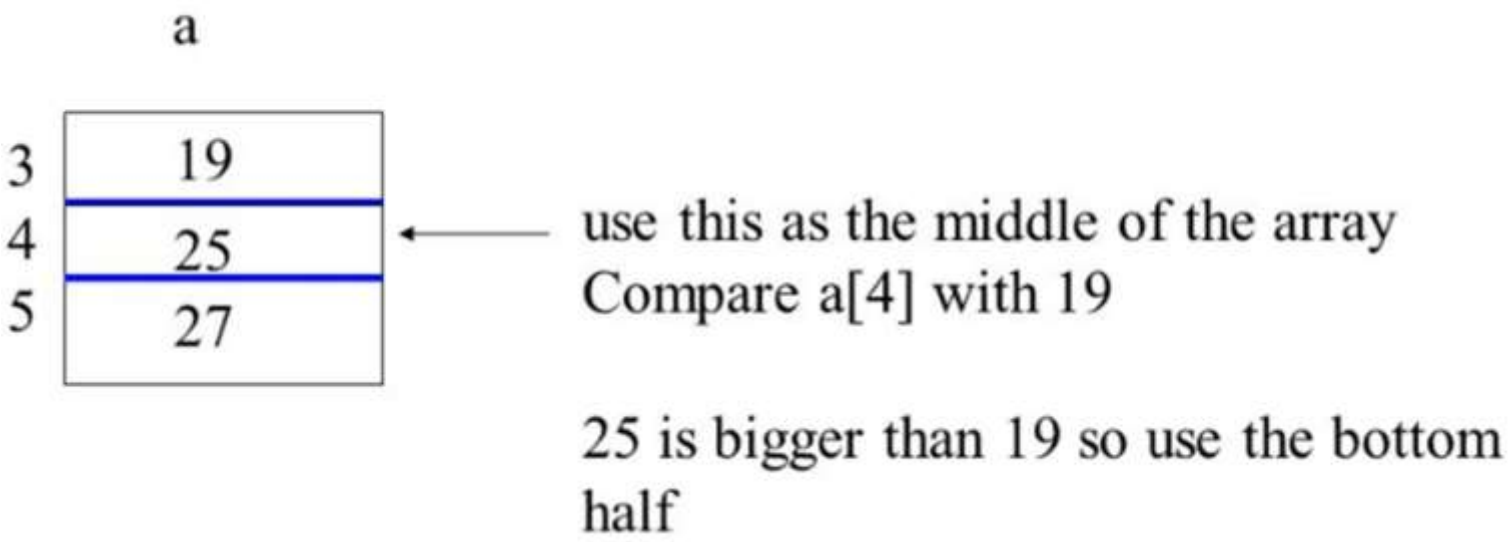
Binary Search

Binary Search Pass 2



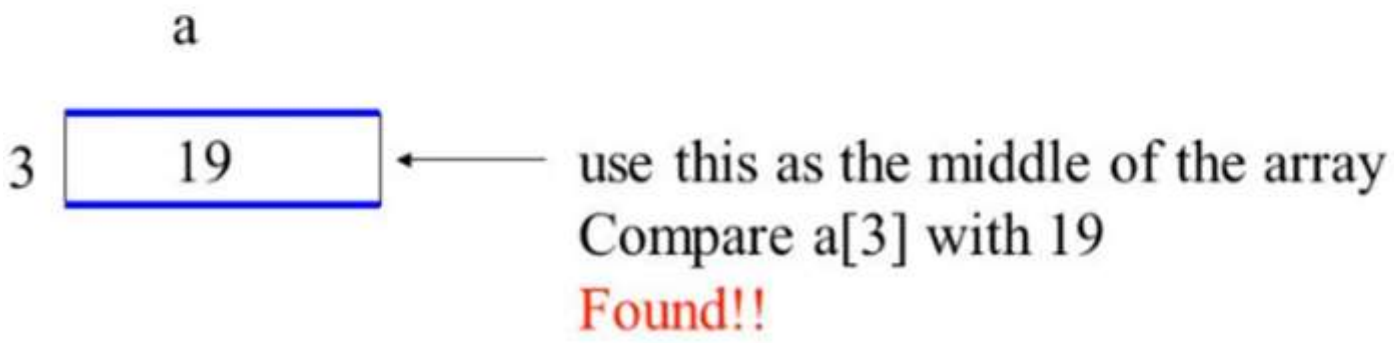
Binary Search Pass 3

search key = 19

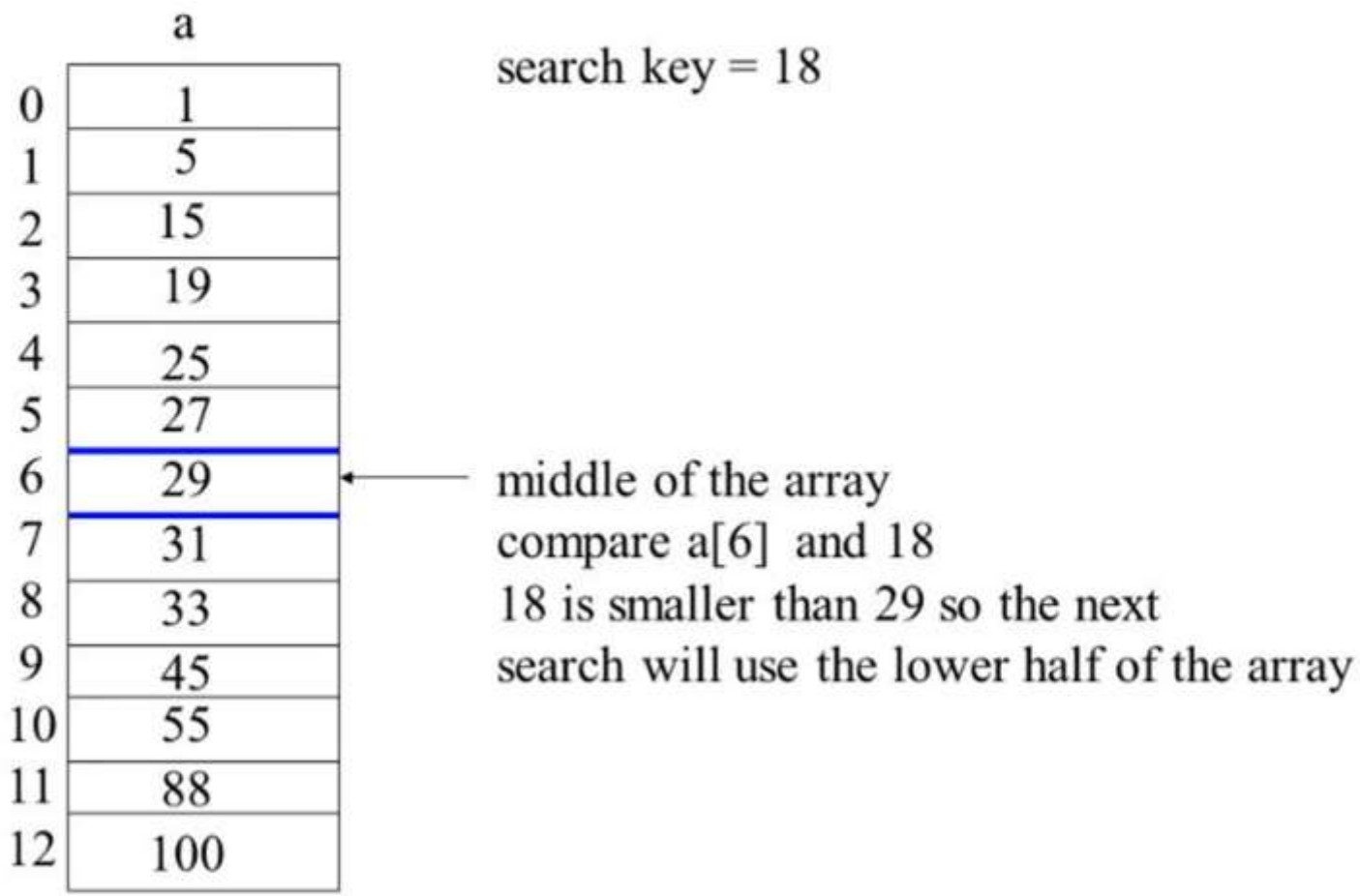


Binary Search Pass 4

search key = 19

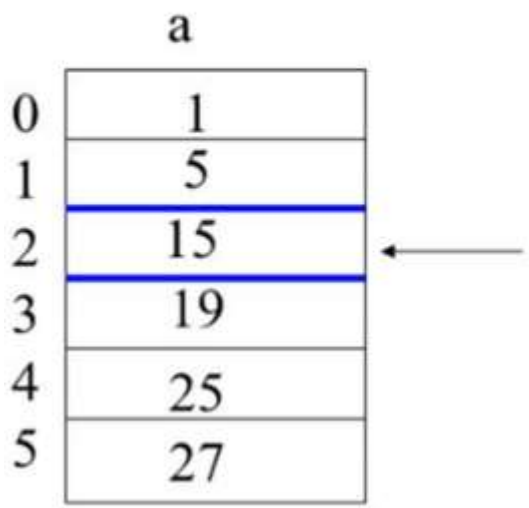


Binary Search Example



Binary Search Greedy algorithm

Binary Search Pass 2



search key = 18

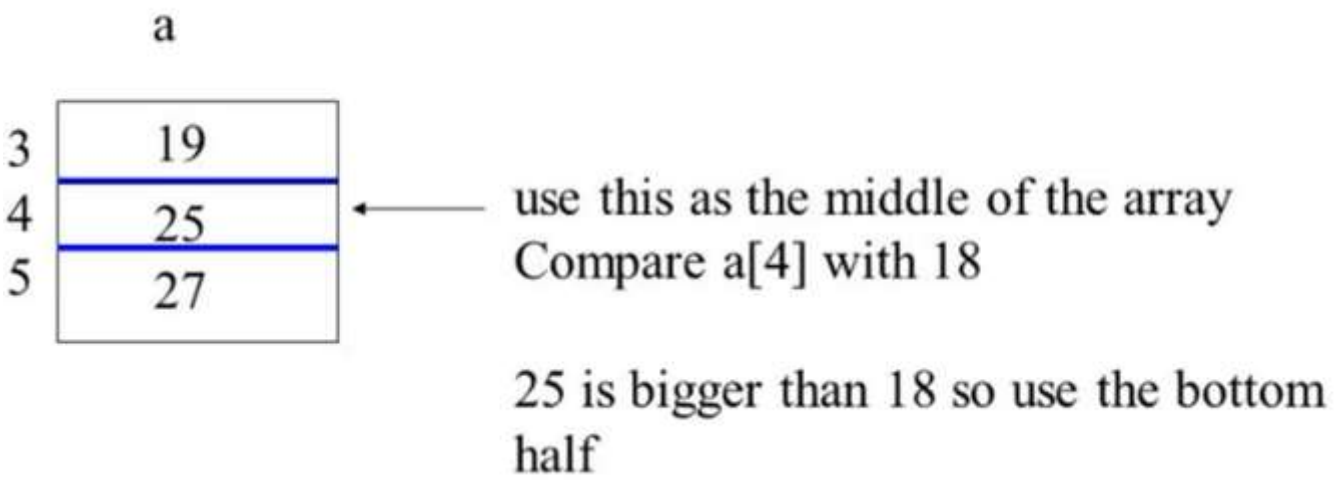
use this as the middle of the array
Compare a[2] with 18

15 is smaller than 18 so use the top half for the next pass

Binary Search

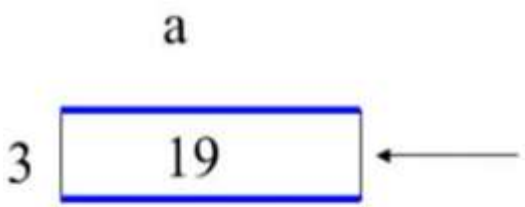
Binary Search Pass 3

search key = 18



Binary Search Pass 4

search key = 18



use this as the middle of the array
Compare a[3] with 18
Does not match and no more elements to compare.
Not Found!!

Binary Search

```
const int arraySize = 15;
int a[arraySize] = {1, 2, 9, 12, 23, 44, 45, 55, 66, 77, 87, 88, 100 };
int key = 23 , low = 0, middle, high = (arraySize - 1);

while (low <= high) {
   middle = (low + high) / 2;

   if (key == a[middle]) {
      cout << "Found element " << key << " at index " << middle << end
      break;
   }
   else if (key < a [middle])
            high = middle -1;      // search low end of array
   else
            low = middle + 1;      // search high end of array
}
```

Binary Search

Efficiency

Searching and array of 1024 elements will take at most 10 passes to find a match or determine that the element does not exist in the array.
512, 256, 128, 64, 32, 16, 8, 4, 2, 1

An array of one billion elements takes a maximum of 30 comparisons.

The bigger the array the better a binary search is as compared to a linear search

Binary Search Vs Linear Search

Comparison between Binary Search and Linear Search:

- Binary Search requires the input data to be sorted; Linear Search doesn't
- Binary Search requires an ordering comparison; Linear Search only requires equality comparisons
- Binary Search has complexity O(log n); Linear search has complexity O(n) as discussed earlier
- Binary Search requires random access to the data; Linear Search only requires sequential access (this can be very important — it means a Linear Search can stream data of arbitrary size)

Greedy algorithm

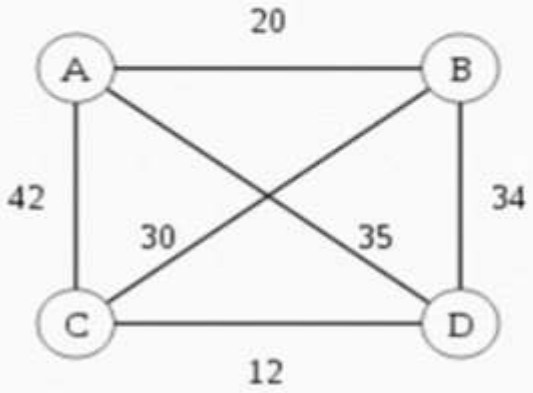
Greedy Algorithms

- A *greedy algorithm* always makes the choice that looks best at the moment
  - My everyday examples:
    - Walking to the Corner
    - Playing a bridge hand
  - The hope: a locally optimal choice will lead to a globally optimal solution
  - For some problems, it works
- Dynamic programming can be overkill; greedy algorithms tend to be easier to code

Greedy algorithm

Sample Usage of Greedy

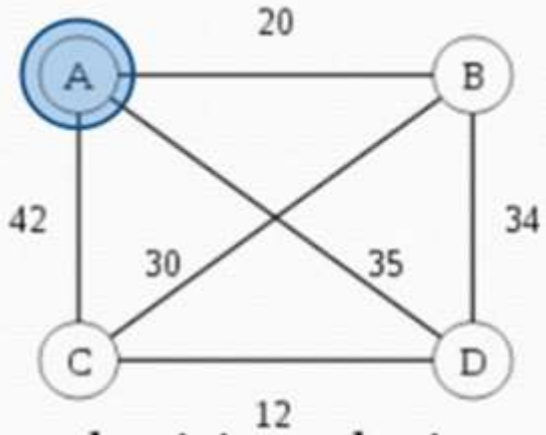
- For better explanation we use old simple problem: Travelling Salesman Problem:



Greedy algorithm

TSP

- The Problem is how to travel from city A and visit all city on the map, then back to city A again.

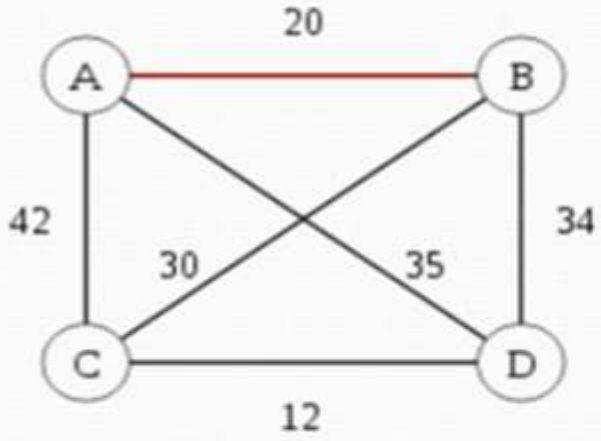


- The rules: you only visit each city once and you can't pass through any traversed path.

Greedy algorithm

Solution:

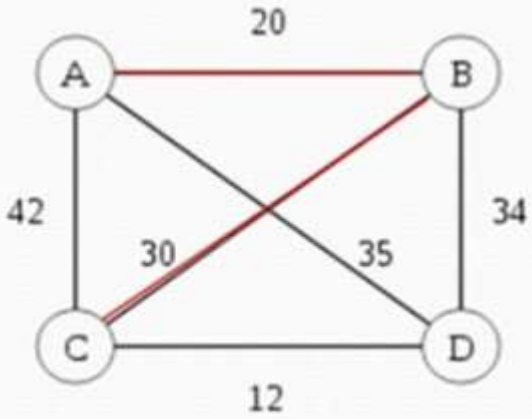
- Find the shortest path from city A(start) to any other city.

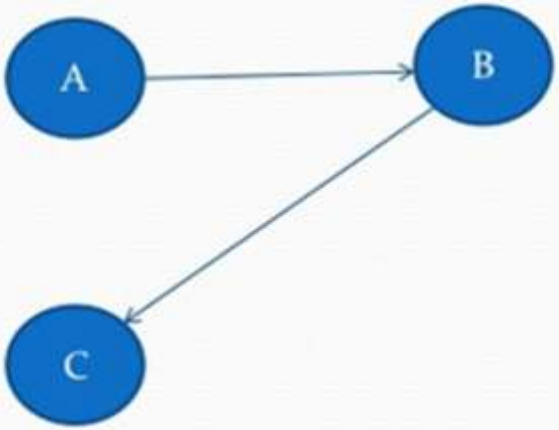


- Because the nearest city is B, so we go to B

Greedy algorithm

- From B, we find any other city but A(because A has been visited) that has nearest path. So we choose C:

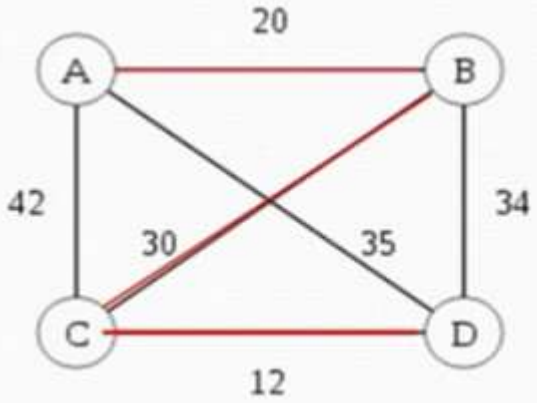


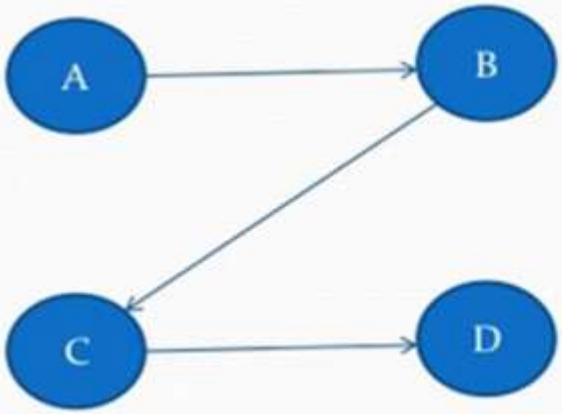


Keep tuning on...

Greedy algorithm

- From C, we look to nearest city again, but don't look for A and B, because both has been visited. So we choose D.

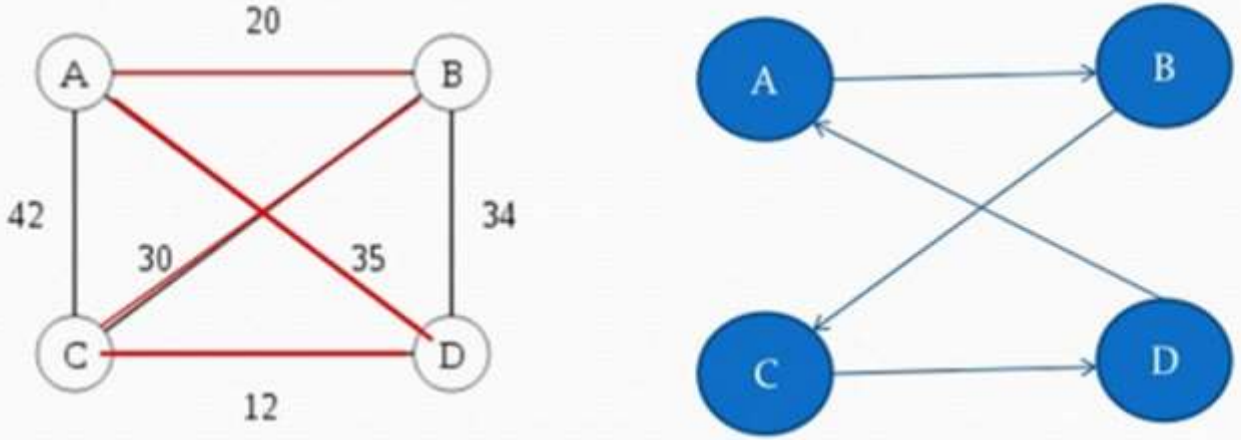




Soon end...

Greedy algorithm

- At this node(D), we can't go to any city, because all neighbor of D has been visited. We go back to first city(A).



- And that was how to solve TSP problem.

Greedy algorithm

Advantage of Greedy

- Greedy is easy to be implemented. Just search the best choice from the current state that 'reachable' (has any paths or any connections).
- In simple case, greedy often give you the best solution.

Greedy algorithm

Drawback of Greedy

- In large and complex case, greedy doesn't always give you the best solution, because it's just search and take the best choice that you can reach from the current state.
- It takes longer time than any other algorithms for big case of problem