Quick sort: Divide and Conquer algo

→ picks an element as pivot and partitions the given array around picked pivot.

Diff versions of Quick sort

→ Always pick first element as pivot
→ last as pivot
→ median
→ pick random element as pivot

key process - partition()

put x (pivot) in ~~torg~~ correct position & smaller elements before x and larger ele after x. done in linear time

```
#include <bits/stdc++.h>
using namespace std;
void swap (int *a, int *b)
{
    int t = *a;
    *a = *b;
    *b = t;
}
int partition (int arr[], int low, int high)
{
    i = low - 1;
    j = low;
int pivot = arr[high];
for (j = low; j <= high-1 ; j++)
{
```

```
    if (arr[j] < pivot)
    {
        i = i+1;
        swap(&arr[i], &arr[j])
    }
}
swap(&arr[i+1], &arr[high])
return (i+1);
}

void quicksort(int arr[], int low, int high)
{
  if (low < high)
  {
    int pi = partition(arr, low, high);
    quicksort(arr, low, pi-1);
    quicksort(arr, pi+1, high);
  }
}
void printarray(int arr[], int size)
{
    int i;
    for (i=0; i<size; i++)
        cout << arr[i] << " ";
    cout << endl;
}
int main()
{
    int arr[] = {10, 7, 8, 9, 1, 5};
    int n = sizeof(arr) / sizeof(arr[0]);
    quicksort(arr, 0, n-1);
```

printarray (arr, n);

return 0;

}.

Time complexity: $T(n) = T(n_1) + T(n_2) + c(n)$

Best case: pivot is middle element

$T(n) = 2T\left(\frac{n}{2}\right) + \underbrace{\theta(n)}_{\text{partition}}$

$= 2\left[2T\left(\frac{n}{4}\right) + \theta\left(\frac{n}{2}\right)\right] + \theta(n)$

$= 2\left[2^2\left(2T\left(\frac{n}{8}\right) + \theta\left(\frac{n}{4}\right)\right)\right] + \theta(n) + \theta(n)$

$= 2^3 T\left(\frac{n}{2^3}\right) + 3\theta(n)$

$= 2^k T\left(\frac{n}{2^k}\right) + k\theta(n)$

Let $2^k = n$

$\Rightarrow k = \log_2 2^k$

$k = \log_2 n$

$= O(n \log n)$

⇒ all elements are same.

Worst case: Already sorted array

$T(n) = T(0) + T(n-1) + c(n)$

$= 0 + T(0) + T(n-2) + c(n-1) + c(n)$

$= c(n + n-1 + n-2 + \dots)$

$= c\left(\frac{n(n+1)}{2}\right)$

MS (A, 1, 10)

1. 1 < 10 True
2. mid = $\left(\frac{1+10}{2}\right)$ = 5
3. MS (A, 1, 5)
    1. 1 < 5 True
    2. Mid = $\frac{1+5}{2}$ = 3.
    3. MS (A, 1, 3)
        1. 1 < 3 True.
        2. mid = $\frac{1+2}{2}$ = 1
        3. MS (A, 1, 2)
            1. 1 < 2 True.
            2. mid = $\frac{1+2}{2}$ = 1.
            3. MS (A, 1, 1)
                1. 1 < 1 False.
            4. MS (A, 2, 2)
                1. 2 < 2 False
            5. Merge (A, 1, 1, 2)
        4. MS (2+1, 3)
            1. 3 < 3 False
        5. Merge (A, 1, 2, 3)
    4. MS (A, 4, 5)
        1. 4 < 5 True
        2. mid = $\left(\frac{4+5}{2}\right)$ = 4.
        3. MS (A, 4, 4)
            1. 4 < 4 False
        4. MS (A, 5, 5)
            1. 5 < 5 False
        5. Merge (A, 4, 4, 5)

Mergesort (A, low, high)

1. if low < high
2. mid = $\frac{low + high}{2}$.
3. Mergesort (A, low, mid)
4. Mergesort (A, mid+1, high)
5. Merge (A, low, mid, high).

n_1 = mid - low + 1; — 1

n_2 = high - mid. — 1

For i = 1 to n_1

L(i) = arr[low + i - 1] — n_1 times

For j = 1 to n_2.

R[j] = arr[mid + j] — n_2 times.

i = 1;
j = 1;
For k = low to high

if L[i] < R[j]

A[k] = L[i]

i = i + 1

else

A[k] = R[j]

j = j + 1

} n times.

linear search

Time Complexity - O(n)

Binary search: Half-interval search.

search a sorted array by repeatedly dividing it into halves

Algo:

1. compare x with middle element.
2. If x matches middle we return mid index.
3. Else if x > mid then x lies in right subarray after mid. we recu right half.
4. Else (x is smaller) recu in right half.

```
def binarysearch (arr, l, r, x):
    if r >= l:
        mid = l + (r-1) // 2
        if arr[mid] == x:
            return mid
        elif arr[mid] > x:
            return binarysearch (arr, l, mid-1, x)
        else:
            return binarysearch (arr, mid+1, r, x)
    else:
        return -1

arr = [2, 3, 4, 10, 40]
x = 10
result = binary search (arr, 0, len(arr)-1, x)
```

Time Complexity

$T(n) = T(\frac{n}{2}) + c$

$\theta(\log n)$

$T(n) = T(\frac{n}{4}) + 2c$

$T(n) = T(\frac{n}{2^k}) + kc$

$= T(\frac{n}{2^k}) + c \log_2 n$ $\quad 2^k = n$

$T(n) = \theta(\log n)$

Dijkstra's shortest path algo:

Given a graph & source vertex.
shortest path from source to all vertices in the graph.

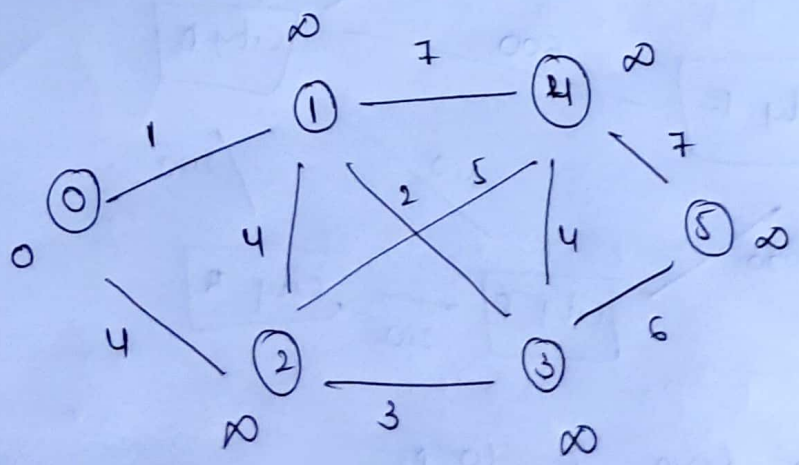
single source shortest path Algo.
we have to find shortest path from source to all other vertices.

Algo:

1. Maintain a set of processed nodes.
2. Assign all nodes with distance value = ∞.
   except source Node (0) ← we start with min
3. Repeat
   ① pick min vertex which is not already processed
   ② include selected node in processed set.
   ③ update all adj node distances.

   if (new distance < old dist) then update
   skip.

Ex:

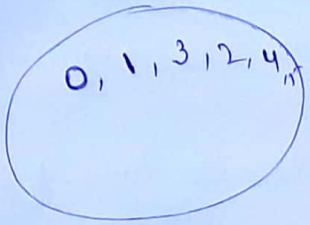


value array:

0	1	2	3	4	5
0	∞	∞	∞	∞	∞

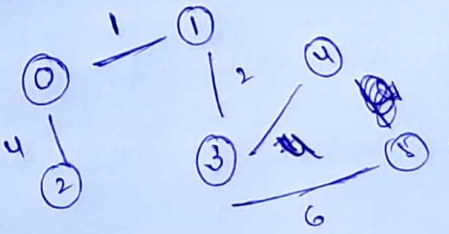
We need to do for n-1 vertices only one is source.

SET



0	1	2	3	4	5
0	~~∞~~ 4	~~∞~~ 1	∞ 3	~~∞~~ 8 7	∞ 9

All adj vertices to min value will be released other than processed vertices.



if $(C(u) + wt(u,v) < C(v))$

$C(v) = C(u) + wt(u,v)$

1st

Parent array

0	1	2	3	4	5
-1	~~-1~~ 0	~~-1~~ 0	~~-1~~ 1	~~-1~~ 3 3	~~-1~~ 3

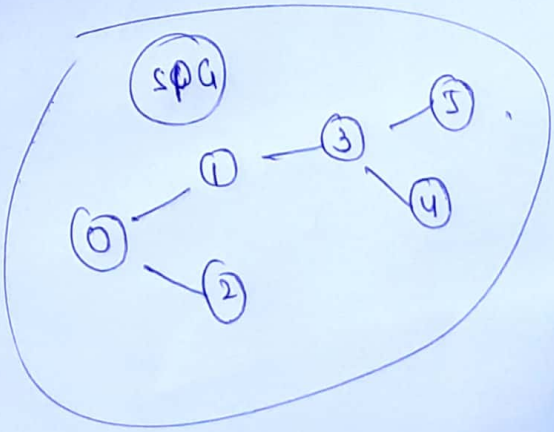
value

0	1	2	3	4	5
0	∞ 1	∞ 4	∞ 3	~~∞~~ 8 7	~~∞~~ 9

Processed array.

0	1	2	3	4	5
~~F~~ T	~~F~~ T	~~F~~ T	~~F~~ T	~~F~~ T	~~F~~ T

0	1	2	3	4	5
-1	0	0	1	3	3



Time complexity — $O(V^2)$

or

$O(E \log V)$

Adj list + min-heap.

void

Trie data structure:

It is a tree data structure that stores strings.

We can do a prefix based search.

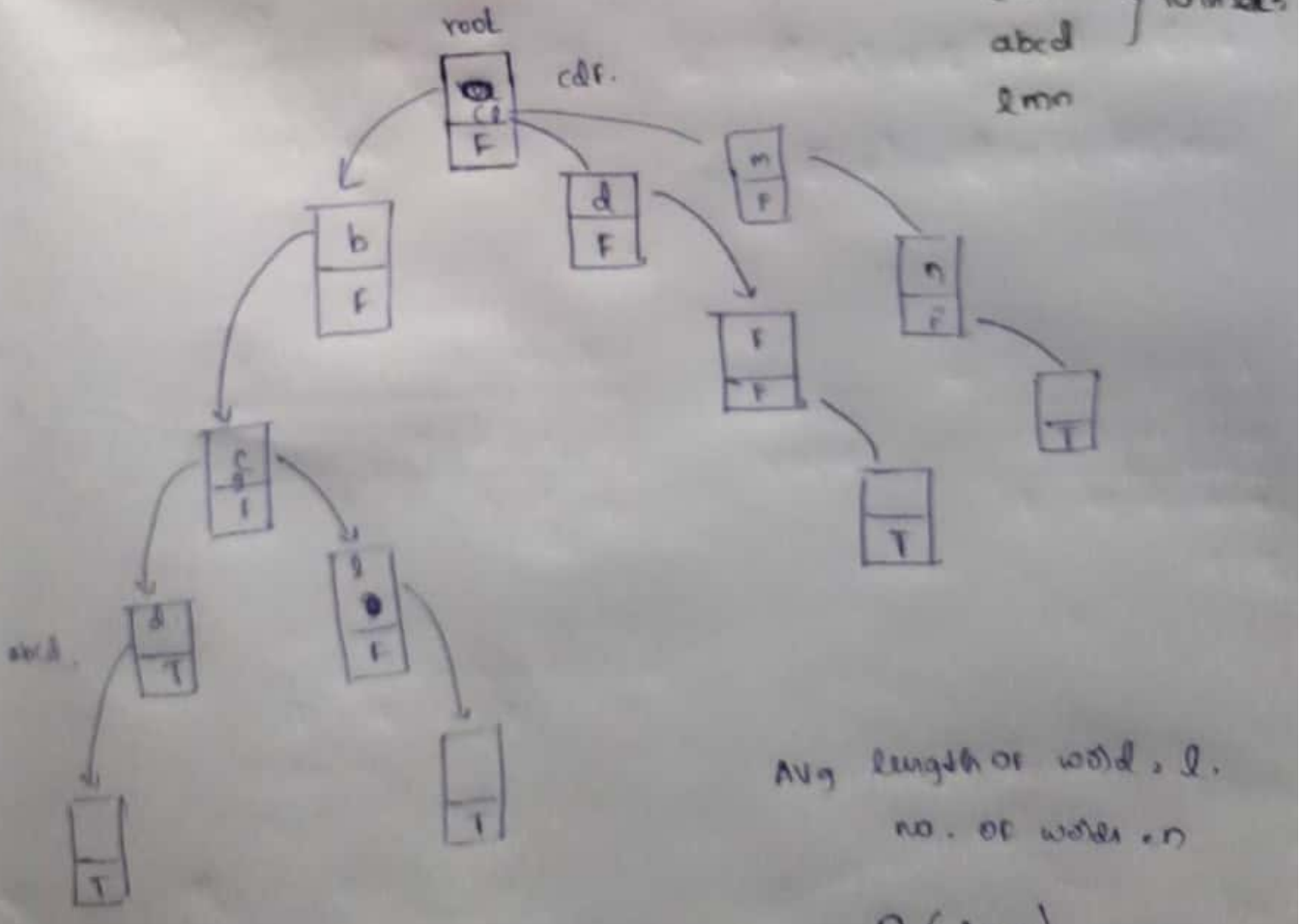
All strings with same prefix has same parent.

Insertion in trie.

```
Trie Node {
    map < character , TrieNode ) children;
    boolean endofword;
}
```

abc
abgl
cdf
abcd
lmn

} trying to insert



Avg length of word, l.
no. of words, n

O(l×n)

Search

Prefix - ab, lo

ab → True.
lo → False

whole - lmn, ab, cdf, gh.

lmn → T (endOfWord has to be True)
ab → False.
cdf → True
gh → False.

O(l)