Elementary Sorting

Total Order :

is a binary relation $\leq$, that satisfies :

i) Antisymmetry : if $v \leq w$ and $w \leq v$, then $v = w$

ii) Transitivity : if $v \leq w$ and $w \leq x$, then $v \leq x$

iii) Totality : either $v \leq w$ or $w \leq v$ or both

eg: 1) Standard order of natural numbers
2) Alphabetic order of strings

Selection Sort :

Inplace : yes
Stable : No [long distance swaps]

- In iteration i, find index $\overset{\min}{\wedge}$ of smallest remaining entry
- swap (a[i], a[min])

```
sort (a[], n)
{
    for (i = 0 to n-1)
    {
        min = i;
        for (j = i+1 to n-1)
        {
            if (a[j] < a[min])
                min = j;
        }
        swap (a[i], a[min]);
    }
}
```

(*) No. of Compares $= (n-1) + (n-2) + \cdots + 1 + 0$

$\approx n^2/2$

No. of exchanges $\approx n^2/2$

(*) Running time insensitive to input : Quadratic time, even if input is sorted.

(*) Data movement is minimal: Linear number of exchanges

Insertion Sort :

(*) Inplace : yes
Stable : yes

• In iteration i, swap a[i] with each larger entry to its left.

```
sort (a[], n)
{
    for (i = 0 to n-1)
    {
        for (j = i to 1)
        {
            if (a[j] < a[j-1])
                swap (a[j], a[j-1]);
            else
                break;
        }
    }
}
```

(*) To sort a randomly ordered array with distinct keys, insertion sort uses $\approx \frac{1}{4} N^2$ compares and $\approx \frac{1}{4} N^2$ swaps on average.

(*) Insertion sort is twice as fast of Selection Sort on average.

(*) Best Case : If input array is sorted, insertion sort makes (n-1) compares and 0 swaps.

(*) Worst Case: If array is reversely sorted, (and no duplicate) insertion sort makes $\approx \frac{1}{2} N^2$ compares and $\approx \frac{1}{2} N^2$ swaps.

Inversion : An inversion is a pair of keys that are out of order.

Partially Sorted Array :

An array is partially sorted, if the number of inversion is $\leq cn$ [i.e, $O(n)$]

eg: i) A subarray (randomly ordered) of size 10 appended to a sorted subarray of size n

2) An array of size n with only 10 entries out of place

↳ Partially sorted array often occurs in practical applications.

(*) Proposition :

For partially sorted arrays, insertion sort runs in linear time.

Proof : We know for insertion sort,

No. of comparisons = No. of swaps + $(n-1)$

and, if the array is partially sorted, i.e, no of inversions $\leq cn$, then, No. of

and, No. of exchanges = No. of inversions

So if array is partially sorted, (i.e, no. of inversion = $O(n)$)

No. of exchanges = $O(n)$

and, No. of comparisons = $O(n)$

$\therefore$ Time = $O(n)$

Shell Sort :-

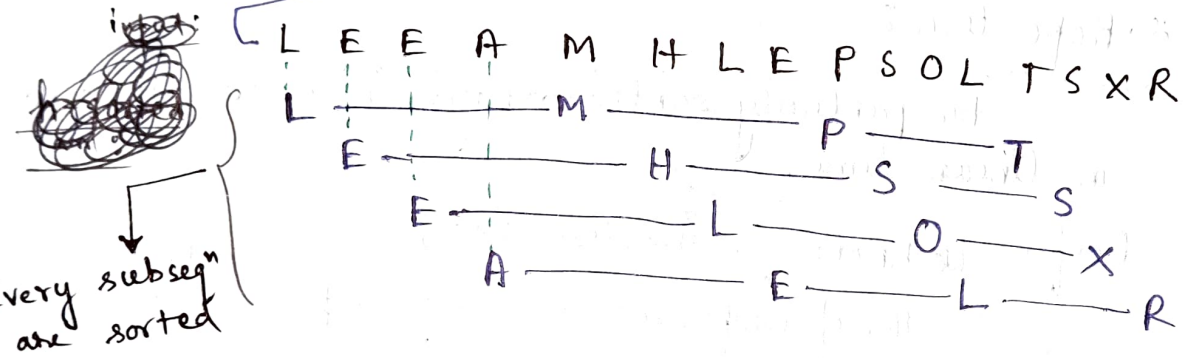
Inplace : yes
Stable : no [due to long distance swaps in h-sorting]

Idea: Insertion sort is slow, because it moves entries one position at a time, even when we know they need to move long distance.
Use h-sorting to move elements more than 1 position at a time.

h-sorted array :

An h-sorted array is h interleaved sorted subsequences

eg: h = 4 → It is 4-sorted



Shell Sort [Invented by Shell in 1959] :

(*) h sort array for decreasing sequence of values of h.

input: S H E L L S O R T E X A M P L E

13-sort : P H E L L S O R T E X A M S L E

4-sort: L H E A M H L E P S O L T S X R

1-sort: A E E E H L L L M O P R S S T X

Idea: Each of these sorts can be done with only a few swaps given that the previous ones happened.

h-sorting: Doing insertion sort skipping h elements.

```
hSort (arr[], n, h)
{
    for (i = h ; i<n ; i++)
    {
        for (j = i ; j >= h && a[j] < a[j-h] ;
        {          j = -h)
            swap(a[j], a[j-h]);
        }
    }
}
```

↳ Instead of going 1 back, we go h unit back.

We use insertion sort while h sorting in shell sort:

because,

i) if h is big: Subarray is small, so insertion sort is efficient.

ii) when h is small: because we have done previous h-sorts for larger h value, the array is partially sorted, so insertion sort is fast here.

Intuition behind Shell Sort:

(*) **Theorem:** A g-sorted array remains g-sorted, after h-sorting it.

Proof:

↳ So if we sort an array by 7-sorting, then 3-sorting, then 1-sorting, then, after 3 sorting the array is 3 sorted and 7 sorted

⇒ This property reduces the no. of inversions in each h-sortings.

↳ This is the intuition of shell sort.

(*) **Another Problem:** What sequence of h should we use?

i) **Power of 2:** 1, 2, 4, 8, 16, 32, ...

⇒ Doesn't work well, because it winds up not comparing elements in even positions with odd positions, until 1-sorting

ii) **Shell's idea:** $(2^{\text{something}} - 1)$: 1, 3, 7, 15, 31, 63, ...

⇒ may be

iii) **Knuth's idea:** $(3x + 1)$: 1, 4, 13, 40, ...

⇒ OK ; Ezz to compute

iv) **Sedgewick's idea:**

(*) finding best increment problem is research problem, not yet resolved

Shell Sort using $h = \{3x+1\}$:-

```
hSort (arr[], n, h)        → using insertion sort
{
    ============
}

ShellSort (arr[], n)
{
    h = 1;
    while (h < n/3)            or, while (3h+1 < n)
        h = 3*h + 1;           1, 4, 13, 40, ...., (3x+1)

    for( ; h >= 1 ; h = h/3)
        hSort (arr, n, h);
}
```

Analysis :

* **Worst Case:** No. of comparisons $= O(n^{3/2})$ for $(3x+1)$ increment

Property: No. of comparisons used by Shell Sort with the $(3x+1)$ increments is at most by

$$\left(\begin{array}{c}\text{a small}\\ \text{multiple of } n\end{array}\right) \times \left(\begin{array}{c}\text{No. of increments}\\ \text{used}\end{array}\right)$$

* In practice, no. of comparisons of shellshort $< 2.5 \times n \log n$,

but Accurate Model has not yet been discovered!

Why are we interested in Shell sort :

- Useful in practice :
  - Fast unless array size is huge.
  - Tiny, fixed footprint for code (used in embedded systems).
  - Hardware sort prototype.

Shuffling :-

i) Shuffle Sort:

- Generate a random number for each element of the array,
- Sort the array, according to those generated random num

eg: arr[] = { 5, 1, 7, 2, 9, 8 }

Generate random values for each arr[i] :

arr =	5	1	7	2	9	8
random val =	69	500	10	69	4	2

Sort w.r.t random values :

arr	8	9	7	5	2	1
random val	2	4	10	69	69	500

Shuffled array:

8	9	7	5	2	1

Proposition : Shuffle sort produces uniformly random permutation of the input array, provides no duplicate value.

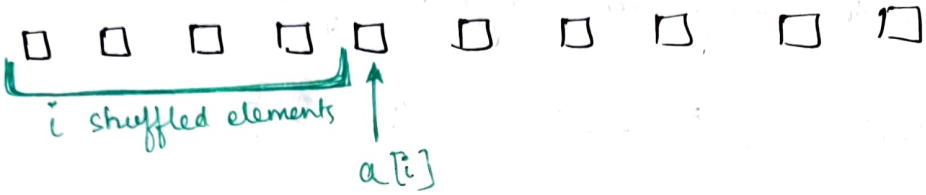
↑ assuming the generated values uniformly at random

Drawback: Sorting takes much time to shuffle.

(ii) Knuth Shuffling:

- In iteration i, pick integer r in $[0, i]$ range uniformly at random
- Swap $(a[i], a[r])$

→ Similar to insertion sort, at i-th iteration



Time Complexity: $O(n)$

Common bugs: i) Select r in $[0, \cancel{n-1}\ i-1]$ range in each iteration

ii) .

Merge Sort

Inplace : No
Stable : yes

- Merge Sort and quick sort are used in practical system sorts :

Merge Sort : 1) Java Sort for objects
2) Perl, C++ Stable Sort, Python stable Sort, Firefox Javascript

Quick Sort : 1) Java Sort for primitive types
2) C qsort, Visual C++, Unix, Python, Matlab, Chrome Javascript

- Quick sort was honored as one of the top 10 algorithms of 20-th century in science and engineering.

Merge Sort, Basic Plan : (Inventor : John von Neumann)

- Divide the array into 2 halves
- Recursively sort each halves
- Merge the two sorted halves

---

assert statements :

- are used for debugging

eg: assert isSorted(arr, l, h);

if isSorted(---) returns false then this line throws exception

- By default assertion statements are disabled, so the code is efficient in production.
- We can enable assertion statements at runtime, by specific commands.

(*) Important Optimizations :~

1) For merge() algorithm, do not create the auxiliary array in merge() or mergeSort() recursive functions. Allocate the aux[] once, and pass it to the recursive functions.
If you allocate the aux[] array in any recursive function it will cause performance loss.

2) Merge Sort has too much recursive overhead for tiny subarrays.
Cut off to insertion sort for ≤ 7 items.

3) ~~Stop the merging~~
Don't merge, if already sorted :
the biggest item in first half $\leq$ the smallest item in 2nd half

4) For Experts: Eliminate the copy to auxiliary array by switching the role of the input and auxiliary array in each recursive call.

```
merge (arr[], low, mid, high, aux[])
{
    if (low >= high)
        return;

    mid = low + (high - low)/2;

    if (arr[mid] <= arr[mid+1])        → (3) Already Sorted
        return;

    for (i = low to high)
        aux[i] = arr[i];

    i = low, j = mid+1; k = low
    while (i <= mid && j <= high)
    {
        if (arr[i] < arr[j])
            aux[k++] = arr[i++];
        else
            aux[k++] = arr[j++];
    }
    while (i <= mid)  aux[k++] = arr[i++];
    while (j <= high) aux[k++] = arr[j++];

    for (i = low to high)
        arr[i] = aux[i];
}

mergeSort (arr[], low, high, aux[])
{
    if (low >= high)
        return;
    mid = low + (high - low)/2;
    mergeSort (arr, low, mid, aux);
    mergeSort (arr, mid+1, high, aux);
    merge (arr, low, mid, high, aux);
}

Sort (arr[], n)
{
    int aux[n];                        → (1) Don't allocate aux[] in recursive function
    mergeSort (arr, 0, n-1, aux);
}
```

(2) Improvement:

```
if (low + CUTOFF >= high)
    return InsertionSort
        (arr, low, high);
```

Analysis :~

Running time estimates :

1) Laptop executes 10^8 operation / second

2) Super computer executes 10^{12} operations / second

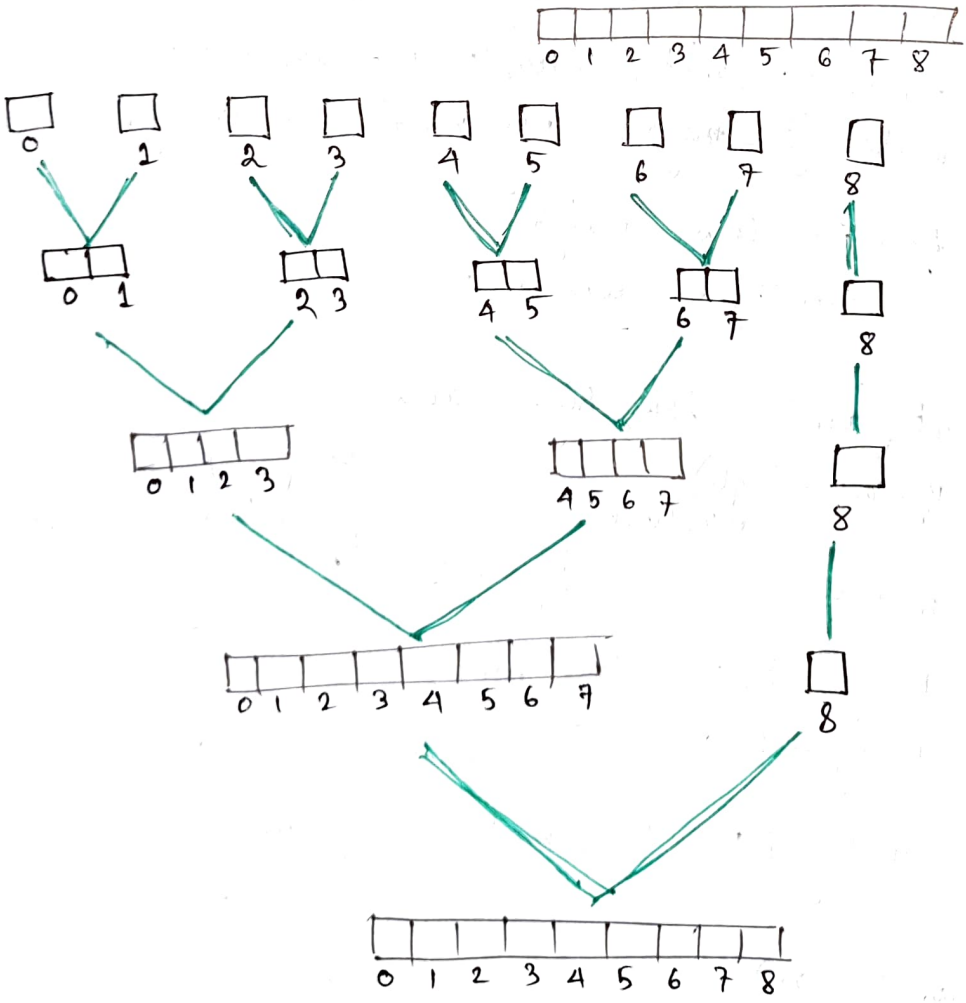
* No. of compares $\leq n \log n$

* No. of array accesses $\leq 6 \times n \log n$

Time Complexity = $\Theta(n \log n)$

Space = $\Theta(n)$

Bottom Up Merge Sort :~



```
void BottomUpMergeSort (arr[], n)
{
    int aux[n];
    for (subArrSize = 1; subArrSize < n; subArrSize = 2 * subArrSize)
    {
        for (low = 0; low + subArrSize < n; low += 2 * subArrSize)
        {
            mid = low + subArrSize - 1;
            high = min (mid + subArrSize, n-1);
            merge (arr, low, mid, high, aux);
        }
    }
}
```

For last sub array without any pair to be merged with (if any)

(*) A sorting algorithm is in-place, if it uses $O(\log n)$ space.

eg: insertion, selection, Shell Sort .....

Complexity of Sorting :~

- **Computational Complexity** : Framework to study efficiency of algorithms for solving a particular problem X.
- **Model of Computation** : Allowable operations (eg: for sorting : ~~compare, exchange~~ Decision tree)
- **Cost Model** : Operation Counts.
- **Upper bound** : Cost guarantee provided by some algorithm for X.
- **Lower bound** : Proven limit on cost guarantee of all algorithms for X. ⇒ No algorithm can solve the problem X better than the lower bound

(*) **Optimal algorithm** :

Algorithm with best possible cost guarantee for X, i.e, lower bound ≈ upper bound

(eg: Merge Sort)

---

eg: for Sorting :

Model of Computation : Decision tree

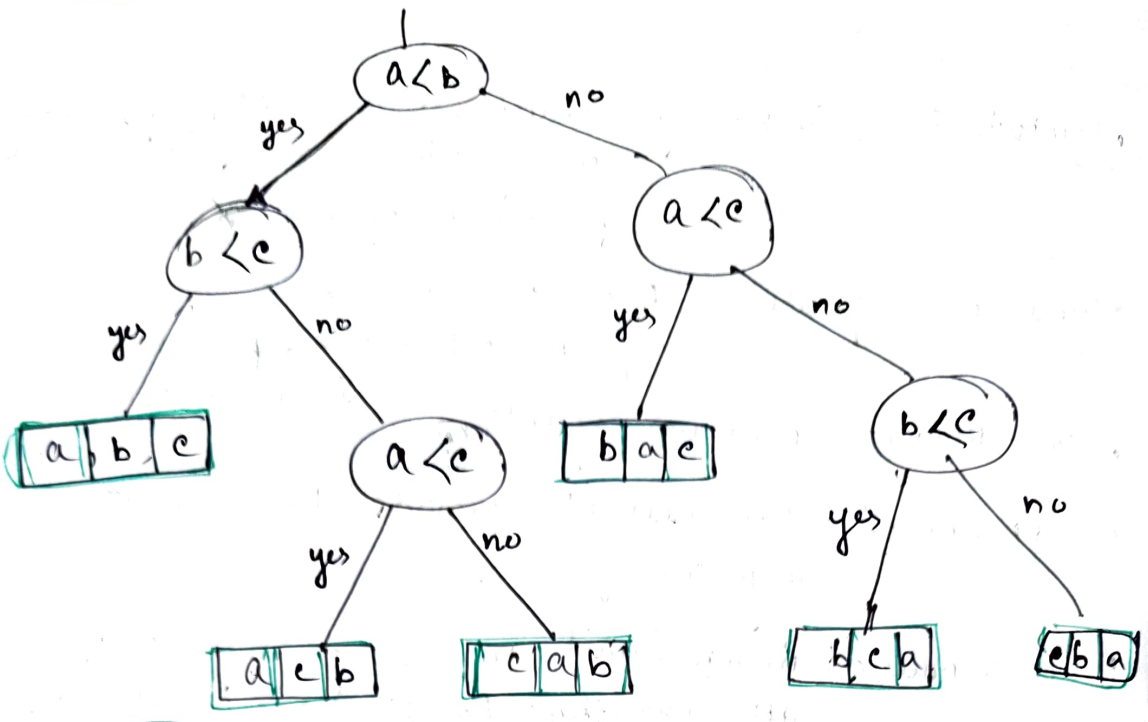
Cost model : # compares

Upper Bound : ≈ $n \log n$ for merge sort

Lower Bound : ≈ $n \log n$ for sorting

Optimal algorithm : depends on algorithm

Decision tree for 3 distinct a, b, c:



- Height of the tree = Worst case number of comparisons

→ Each leaf is a permutation of a, b, c

i.e, there are $(n!)$ leaves in the decision tree.

Proposition: Any compare based sorting algorithm must use at least $\log_2(n!) \approx n \log_2 n$ compares in the worst case. (for distinct elements)

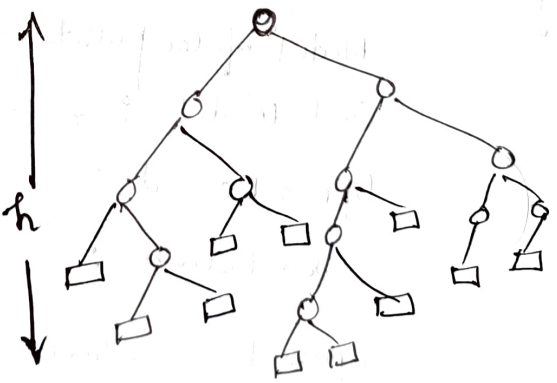
Proof:

let, h = height of decision tree

Now, Binary tree of height h has atmost 2^h leaves.

Again, $n!$ different ordering

$\Rightarrow$ at least $n!$ leaves



Decision tree for n elements

$\therefore 2^h \leq$ No. of leaves $\leq n!$

or, $2^h \leq n! \Rightarrow h \leq \log_2 n! \approx n \log n$

$\therefore$ Worst case no. of comparisons $= h \approx n \log_2 n$

(*) So, Lower Bound of sorting algo. = $\Omega(n \log n)$

in worst case

- Lowerbound may not hold if the algorithm has information about:
  - i) The initial ordering of the input
  - ii) The distribution of key values
  - iii) The representation of keys

i) Partially ordered arrays: depending on the initial order of the input, we may not need $n \log n$ compares: insertion sort requires only $(n-1)$ compares if input array is sorted.

ii) Duplicate keys: Depending on the input distribution of duplicates, we may not need $n \log n$ compares: 3-way quicksort.

iii) Digital Properties of keys:
We can use digit/character compares instead of key compares for numbers and strings:
Radix Sort.

Stability :-

Stable Sort : A stable sort preserves the relative order of items with equal keys.

Stable Sort	Unstable Sort
Insertion Sort, Merge sort	Selection, Shell, Quick Sort

(*) Also we need to carefully check code ('<' vs '≤')

Proposition : If there is long distance swaps, then it is not stable.

Proof : for long distance swaps, equal items can move past each other.

eg:

val	5	1	5	2
	a	b	c	d
	0	1	2	3

→ Sorting by val

suppose, we swap a[0] and a[3] as 5 > 2

then,

2	1	5	5
d	b	c	a

But for stability $\frac{5}{a}$ must be before $\frac{5}{c}$

So, if there is long distance swaps, then equal items can move past each other. i.e, not stable.

1) Stability: Insertion Sort:

Yes, because we never move equal item past each other.

2) Stability: Selection Sort:

No, due to long distance swaps

3) Stability: Shell Sort:

No, due to long distance swaps in h sorting for $h > 1$

4) Stability: Merge Sort: Yes

⇒ Merge Sort is stable, as long as the merge() function is stable.

⇒ Merge() operation is stable, if for equal keys, it takes from the left subarray

5) Stability: Quick Sort: No, due to long distance swaps

Quick Sort

Inplace: yes (in average case)

Stable: No.

Inventor:

Sir Charles Antony Richard Hoare (won Turing award)

Basic Plan :

- Shuffle the array
- Partition so that, for some j,
  - entry $a[j]$ is in place
  - No larger entry to the left of j
  - No smaller entry to the right of j
- Sort each piece recursively.

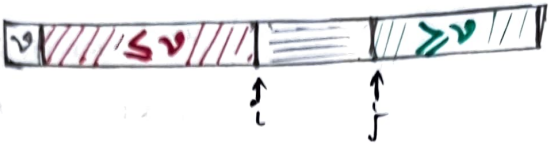
Partition :

Returns index of the entry, which is placed in its correct position,

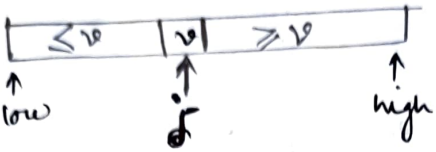
```
int Partition (low, high, a[])
{   i = low, j = high + 1;        → Select a[low] element as pivot
    while (true)
    {   while (a[++i] <= a[low])   → find item on left to swap
            if (i == high)
                break;

        while (a[low] <= a[--j])   → find item on right to swap
            if (j == low)
                break;

        if (i >= j)                → Check if pointers cross
            break;

        swap (a[i], a[j]);
    }
    swap (a[low], a[j]);           → Swap with partitioning item
    return j;                      → Return index of item now known to be in place
}
```

Initially :

Loop Invariant :



Finally :



Quick Sort :-

```
QuickSort (low, high, a[])
{
    if (low ≥ high)
        return;

    j = Partition (low, high, a);

    QuickSort (low, j-1, a);
    QuickSort (j+1, high, a);
}

Sort (a[])
{
    Shuffle (a);

    QuickSort (0, n-1, a);
}
```

Implementation details:

(i) Partitioning is inplace.

(ii) (j == low) checking in Partitioning is redundant, but (i == high) is necessary.

(*) (iii) Random Shuffling is needed for Performance Guarantee

(iv) Equal keys: When duplicates are present, it is (counter intuitive) better to stop on keys equal to the partitioning item's key.

Running Time:

(*) Practically Quick Sort is faster than Merge Sort.

Best Case: No. of comparisons $\approx n \log n$

Worst Case: No. of comparisons $\approx \frac{1}{2} n^2$ $(n + (n-1) + \cdots + 1)$

(*) Average Case: No. of comparisons $\approx 1.39 n \log n$

No. of swaps $\approx \frac{1}{3} n \log n$

Proof: Let, C_n = No. of comparisons for n items

Recurrence:

$$C_n = (n+1) + \sum_{k=0}^{n-1} \left(\frac{C_k + C_{N-1-k}}{n} \right)$$

(left piece, right piece)

$(n+1)$ → For partition()

$\frac{C_k + C_{N-1-k}}{n}$ → Partitioning Probability

and, $C_0 = C_1 = 0$

or, $n C_n = n(n+1) + 2 \sum_{k=0}^{n-1} (C_k + C_{n-1-k})$

$= n(n+1) + 2 \sum_{k=0}^{n-1} C_k$

~~Subtracting (n-1) C_{n-1}~~

Now, $n C_n = n(n+1) + 2(C_0 + C_1 + \cdots + C_{n-1})$ ——— (1)

Putting $n \to n-1$,

$(n-1) C_{n-1} = n(n-1) + 2(C_0 + C_1 + \cdots + C_{n-2})$ ——— (2)

(1) − (2) ⇒

$n C_n - (n-1) C_{n-1} = 2n + 2 C_{n-1}$

or, $n C_n = 2n + (n+1) C_{n-1}$

or, $\frac{C_n}{n+1} = \frac{2}{n+1} + \frac{C_{n-1}}{n}$

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

$= \frac{2}{n+1} + \frac{2}{n} + \left(\frac{2}{n-1} + \frac{C_{n-3}}{n-2} \right)$

$= \frac{2}{n+1} + \frac{2}{n} + \frac{2}{n-1} + \cdots + \frac{2}{3} + \frac{C_1}{1}$

$= \frac{2}{n+1} + \frac{2}{n} + \cdots + \frac{2}{3} + \frac{0}{1}$

$= 2 \left\{ \frac{1}{3} + \frac{1}{4} + \frac{1}{5} + \cdots + \frac{1}{n+1} \right\}$

or, $C_n = 2(n+1) \times \left\{ \frac{1}{3} + \frac{1}{4} + \cdots + \frac{1}{n+1} \right\}$

$= 2(n+1) \times \sum_{k=3}^{n+1} \frac{1}{k}$

for large n,

$C_n \approx 2(n+1) \times \int_3^{n+1} \frac{1}{x}$

$= 2(n+1) \left[\log x \right]_3^{n+1}$

$$\text{or } C_n \simeq 2(n+1)\log_e n$$

$$\simeq 2(n+1) \times \frac{\log_2 n}{\log_2 e}$$

$$\simeq 1.39\,(n+1)\log n$$

$$\therefore C_n \simeq 1.39\ n\log n$$

(*) With a random shuffle at the beginning, the expected no. of comparisons is concentrated around this value $1.39\ n\log n$.

(*) After shuffling randomly, the probability of worst case (sorted order) is less than the probability that your computer is struck by a lightning bolt.

(*) Faster than Merge Sort:

No. of comparisons of quick sort is 39% more than merge sort. But the copying of elements to auxiliary array inside merge() function has more overhead.

In practice, quicksort is much faster than merge sort because of less data movement.

Space Complexity :

Best Case : $O(\log n)$
Worst Case : $O(n)$
Average Case : $O(\log n)$

} Depends on size of call stack

(*) Quick Sort is inplace in average case.

(*) Important Practical Optimizations :~

(i) Insertion Sort for small subarrays :

Cutoff to insertion sort for $\approx$ 10 items.

```
if ((high - low + 1) ≤ cutoff)
    return insertionSort (low, high, arr);
```

→ Improves running time by 20%

(ii) Median of Sample :

- Best choice of pivot element = median
- Estimate true median by taking median of 3 samples.

Inside partition :

medianInd = Get Median Index (a[low], a[(low+high)/2], a[high]);

swap (a[low], a[medianInd]);

then do partitioning w.r.t a[low].

Selection / Quick Select :~

QuickSelect (low, high, K, arr) = Returns the K-th largest element, by placing all smallest K elements in a[0...(K-1)]

Application:

1) Order statistics
2) Find the "top K" elements

Invented by: Hoare

Quick Select:

```
int QuickSelect(low, high, K, a[])
{   if (low == high)
        return a[low];
    p = Partition (low, high, arr);
    if (p == K)
        return a[p];
    else if (p < K)
        return QuickSelect (p+1, high, K, arr);
    else
        return QuickSelect (low, p-1, K, arr);
}

int Select (arr[], K)
{   Shuffle(arr);
    return QuickSelect (0, n-1, K, a);
}
```

→ Important for performance

↳ After calling this function, smallest K elements will be in a[0...(K-1)]

Analysis:

Average Case: No. of compares $\approx 3.4n$

Worst Case: No. of compares $\approx \frac{1}{2}n^2$

Average Case Proof:

Intuitively: each partitioning step splits array approximately in half.

So No. of compares $= n + \frac{n}{2} + \frac{n}{4} + \cdots + 1$

$\approx n\{1 + \frac{1}{2} + \frac{1}{4} + \cdots\} + 1$

$\approx 2n + 1 \approx 2n$

Formal analysis:

$$C_n = 2n + 2k \ln\left(\frac{n}{k}\right) + 2(n-k) \ln\left(\frac{n}{n-k}\right)$$

Solving it, $\boxed{C_n = (2 + 2\ln 2)\, n}$

(*) In 1993, Blum, Floyd, Pratt, Rivest, Tarjan, found a compare based selection algorithm whose worst case running time is linear.

↳ But the constants are too high, and also complex implementation, so in practice QuickSelect() is used for selection

Quick Sort for input with many duplicate keys:

- Merge Sort's no. of comparisons is always within $\frac{1}{2} n \log n$ to $n \log n$, irrespective of duplicate keys.

- For Quick Sort:

  Algorithm goes quadratic, unless partitioning stops on equal keys! (If there are so many duplicates)

- In 1990, a C user found this defect in qsort() system sort function.

eg: a[] = | (2,a) | (2,b) | (2,c) | (2,d) | (2,e) | (2,f) |

pivot

↳ First field is key

(1) Partitioning without stopping at equal key (Previous Code):

after partitioning,

a[] = | 2,a | 2,b | 2,c | 2,d | 2,e | 2,f |

pivot → Not equally partitioned

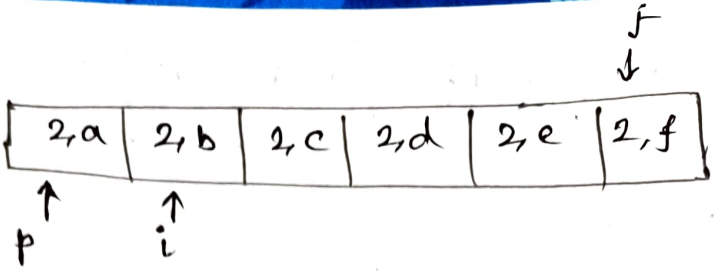
(2) Partioning with stoping at equal keys:

Modified code:

```
while (true)
{   while (a[++i] < a[low])
        if (i == high) break;
    while (a[low] < a[--j])
        if (i == high) break;
```

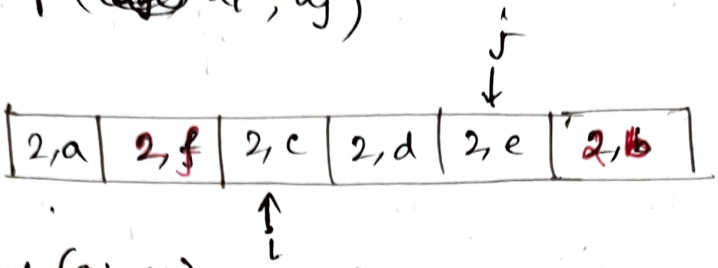
→ We stop when key is equal to pivot

Step 1:



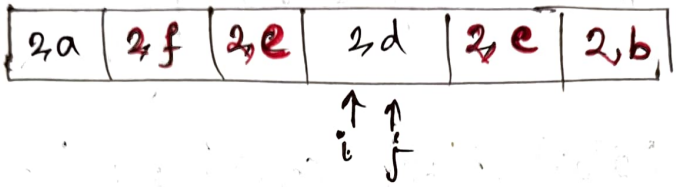
swap (ai , aj)

Step 2:



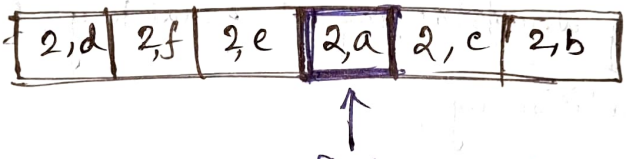
swap (ai, aj)

Step 3:



swap (Pivot, aj)

Final:



→ Next QuickSort() on each halves will be called

Almost Equally Partitioned

(*) So, if there are so many duplicates, stopping at equal key, will make the time complexity $\Theta(n \log n)$ on average.

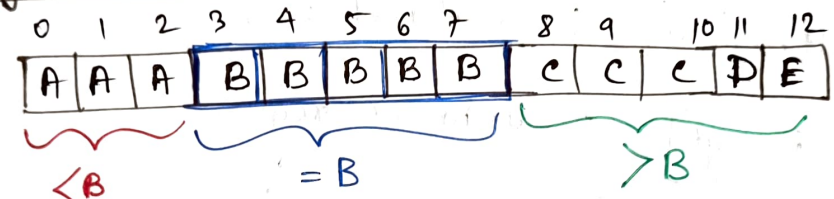
3 way partitioning / Dutch national flag :

- Dijkstra proposed this for quicksort optimization.
- For given a[], partition it w.r.t pivot v, such that all elements $< v$ are at left position, all elements equal to v are in middle, then all elements $> v$.

eg:

B	A	A	B	A	B	C	C	B	C	B	D	E

3 way partition w.r.t B :



After partitioning : Call quickSort()

Call quickSort

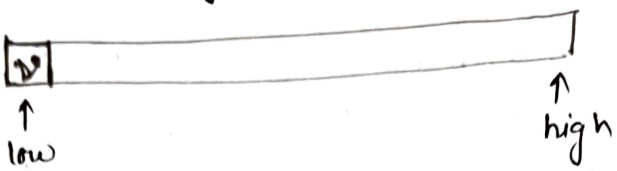
After partitioning, now call quickSort on a[0...2] and a[8...12]

⟹ This 3 way partitioning can thus improve the running time for almost linear for array with very few distinct elements.

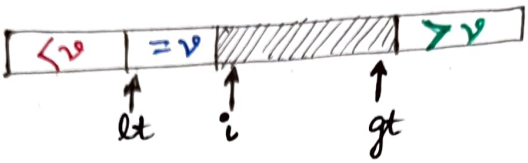
Algorithm :

let, v = Partitioning element

Initially :



Invariant :



i) $a[low \ldots (lt-1)]$: all elements $< v$

ii) $a[lt \ldots (i-1)]$: all elements $= v$

iii) $a[i \ldots gt]$: unknown region

iv) $a[(gt+1) \ldots high]$: all elements $> v$

```
3 WayPartition (low, high, v, a[])
{                                   → Pivot value
    i = low; lt = low; gt = high;
    while (i <= gt)
    {   if (a[i] == v)  i++;
        else if (a[i] < v)  swap (a[lt++], a[i++]);
        else                swap (a[i], a[gt--]);
    }
    return { lt, gt };
}
```

↓

returns the first and last index of the subarray containing all elements equal to v.

```
QuickSort (low, high, a[])
{
    if (high - low + 1 ≤ CUTOFF)
        return insertionSort (low, high, a);
    v = Median of a[low], a[(low+high)/2], a[high];
    {lt, gt} = 3WayPartition (low, high, v, a);
    QuickSort (low, lt - 1, a);
    QuickSort (gt + 1, high, a);
}
```

Proposition (By Sedgewick - Bentley, 1997):
Quicksort with 3 way partitioning is entropy - optimal.
(linear, when only a constant number of distinct keys)

System Sorts :

- In Java, Array.sort()
  - uses MergeSort for objects
  - uses QuickSort for primitive types.
  - Because for sorting primitive types, usually stability is not needed, and also more performance is needed so, quick sort is used here.

    And for Objects, usually stability is needed and since programmer is using objects, space is not a critically important consideration. So MergeSort is used for Objects.

- In c++, sort() :
  - uses combination of insertionSort(), quickSort()
  - is not stable
  - space = $O(\log n)$
- In c++, stable_sort() :
  - uses combination of insertionSort() and mergeSort()
  - is stable
  - space = $O(n)$

(*) Merge Sort and Heap Sort are optimal comparison based sorting algorithm.

Applications have diverse attributes :~

- Stable ?
- Parallel ?
- Deterministic ?
- Keys all distinct ?
- Multiple key types ?
- Linked list or arrays ?
- Large or small items ?
- Is your array randomly ordered ?
- Need guaranteed performance ?

	Inplace	Stable	Worst Case	Average Case	Best Case	Remarks
Selection Sort	✓	✗	$\frac{n^2}{2}$	$\frac{n^2}{2}$	$\frac{n^2}{2}$	n exchanges
Insertion Sort	✓	✓	$n^2/2$	$n^2/4$	n	use for small n, or partially ordered.
Shell Sort	✓	✗	?	?	?	tight code, subquadratic
Merge Sort	✗	✓	$n \log n$	$n \log n$	$n \log n$	N log N guarantee, stable
Quick Sort	✓	✗	$n^2/2$	$2 n \log n$	$n \log n$	N log N Probalistic guarantee, fastest in practice
3-way Quick Sort	✓		$n^2/2$	$2 n \log n$	n	Improves quicksort in presence of duplicate keys
LSD Radix Sort	✗	✓	$2nw$	$2nw$		w = width of string, R = Radix (No. of distinct char)
MSD Radix Sort	✗	✓	$2nw$	$n \log n$		
3 way string Quick Sort	✓	✗	$1.39 \times wn \log n$	$1.39 \times n \log n$		
Heap Sort	✓	✗	$2n \log n$	$2n \log n$		

RADIX SORT / STRING SORT

Key Indexed Counting / Counting Sort :

- Stable
- Not inplace

- It is not comparison based sorting algorithm.

Assumption : Keys are integers between 0 and R-1

Implication : Can use key as an array index. (Small keys)

Applications :
- Sort by first letter
- Sort class roster by section
- Sort phone numbers by area code
- Subroutine in a sorting algorithm (Radix Sort)

(*) **Remark :** Keys may have associated data,
⇒ we can't just count up number of keys of each value.

Algorithm :~ **Goal :** Sort an array a[] of n integers between [0, R-1]

```
sort (a[], n, R)
{   int count [R+1], aux[n];
    count[0] = 0;

    for (i = 0 to n-1)
        count [a[i]] ++;          } count frequency of each element using key as index

    for (r = 0 to R-1)
        count [r+1] += count [r];  } compute frequency cumulates (which specifies the no. of elements, that should be before it in sorted array) [i.e, destination]

    for (i = 0 to n-1)
        aux [ count [a[i]] ++ ] = a[i];   } Move items

    for (i=0 to n-1)
        a[i] = aux [i];           } Copy back
}
```

eg:

initially, a[] :

d	a	c	f	f	b	d	b	f	b	e	a
0	1	2	3	4	5	6	7	8	9	10	11

after counting frequency : count[] :

0	2	3	1	2	1	3
0	1	2	3	4	5	6
	a	b	c	d	e	f

(offset by 1)

after counting frequency cumulates : count[] :

0	2	5	6	8	9	12
0	1	2	3	4	5	6
	a	b	c	d	e	f

Now, we know, there should be count[x-1] elements before x, in the sorted array,

i.e., in the sorted array, destination index of x should be count[x-1].

aux[] :

a	a	b	b	b	c	d	d	e	f	f	f
0	1	2	3	4	5	6	7	8	9	10	11

a[] = aux[]

Running time :

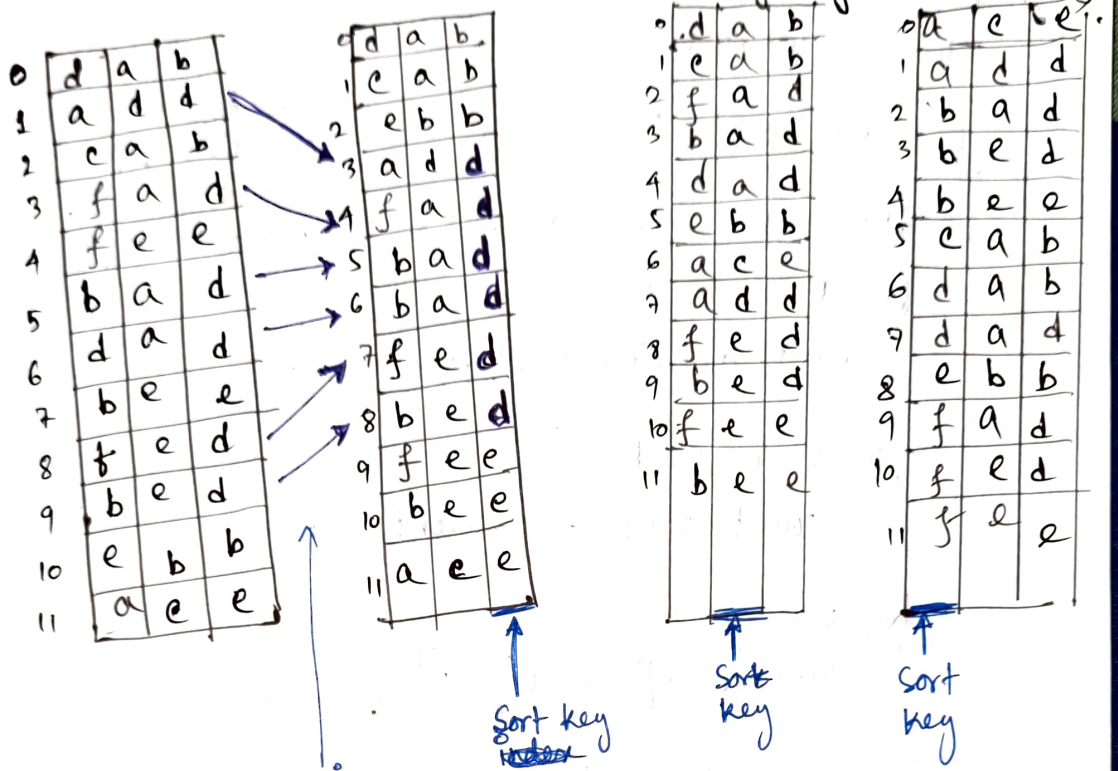
- No. of array accesses $\simeq (11n + R)$
- extra space $\Theta(n + R)$

Stable ? : Yes

① LSD Radix Sort (Least Significant Digit first Radix Sort)

(or String) (or, String)

- Consider characters from right to left.
- Stably sort using d^{th} character as the key index (using key index counting)



Proposition: LSD sorts fixed length strings in ascending order.

Proof: [By induction]

Let, $P(i)$ = After i passes, strings are sorted by last i characters

Now assuming $P(i)$ is true,

during $(i+1)^{th}$ pass:

1) If two strings differ on sort key, then Key Indexed Sort (Counting Sort) puts them in proper relative order.

2) If, two strings have equal sort key, then stability keeps them in proper relative order.

$\therefore P(i) \Rightarrow P(i+1)$ and $P(1)$ is true.

$\therefore P(i)\ \forall\ i$ is true.

Pseudo Algorithm :

LSD

```
LSD_Radix_Sort (a[], n, w)        // w → fixed length w strings
{
    R = 256;                      // R → Radix
    string aux[n];
    for (d = (w-1) to 0)          // do key indexed counting for each digit from right to left
    {
        int count[R+1]; count[0] = 0
        for (i = 0 to n-1)
            count[a[i][d] + 1]++;

        for (r = 0 to (R-1))
            count[r+1] += count[r];

        for (i = 0 to n-1)
            aux[count[a[i][d]]++] = a[i];

        for (i = 0 to n-1)
            a[i] = aux[i];        // copy reference instead of whole string everywhere
    }
}
```

Analysis :

- No. of array access = $2wn$
- Extra Space = $(n+R)$

String Sorting interview question ?—

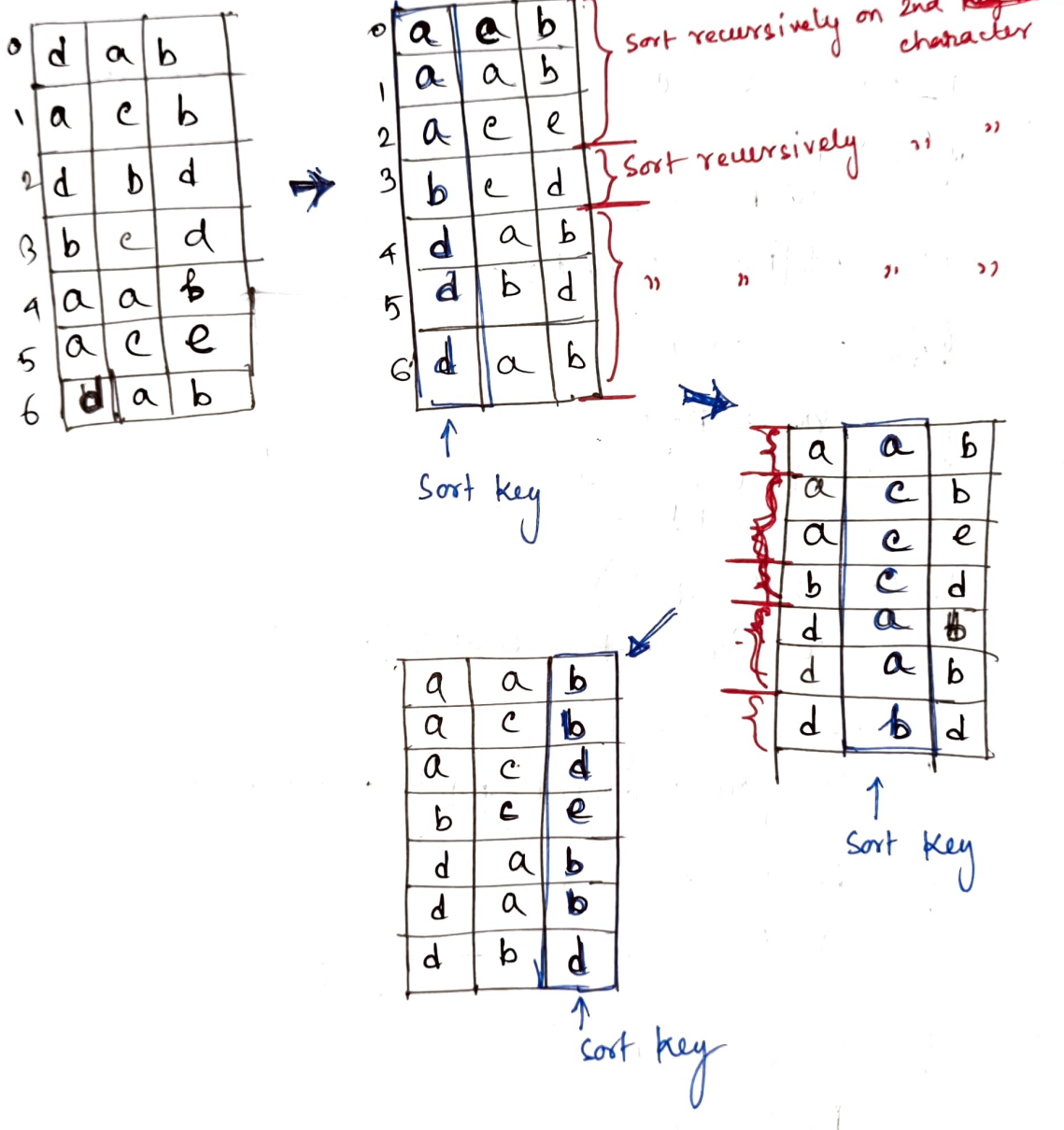
Problem : Sort one million 32 bit integers.

Ans: LSD string sort

② MSD String Sort (Most Significant Digit first)

- Partition array into R pieces according to first character (key indexed counting).
- Recursively sort all string that start with each character (key indexed counts delineate subarrays to sort).
  - Similar to Quick Sort.

eg:



for variable length strings :

Treat strings as they had an extra character at the end, which is smaller than any character.

```
int charAt (string s, int d)
{
    if (d >= s.length) return -1;
    else return s[d];
}
```

Pseudo Algorithm :

```
MSD_string_sort ( low, high, a[], aux[], d);
{
  if (low >= high)
      return;
  int count[R+1]; count[0]=0;
  for (i=low to high)
      count[a[i][d]+1]++;
  for (r=0 to (R-1))
      count[r+1] += count[r];
  for (i= low to high)
      aux[count[a[i][d]]++] = a[i];
  for (i= low to high)
      a[i] = aux[i - low];
  for (r=0 to R-1)
      MSD_string_sort (low + count[r], low + count[r+1] - 1,
                       a, aux, d+1);
}
```

Improvement: Cut off to insertion sort for small size

```
sort(a[], n)
{
    char aux[n];
    MSD_String_sort(0, n-1, a, aux, 0);
}
```

Observation:

1) We can recycle aux[], but not count[].
We need different count[] arrays local to each procedure calls.

2) • It is much too slow for small arrays, as R is greater than array size.
   • Each fn call needs its own count[].
   • ASCII (256 counts): 100 x slower than copy pass for n = 2
   • Unicode (65,536 counts): 32,000x slower for n = 2.

3) Huge no. of small subarrays because of recursion.

Analysis :~

- No. of character depends on keys.
- Can be sublinear in input size.

Worst Case time = $2nw$

Average " " = $n \log_R n$

Extra Space = $n + DR$

D = fn. call stack depth

MSD String Sort vs Quick Sort for strings :

Disadvantages of MSD String Sort :

- Accesses memory "randomly", (cache inefficient)
- Inner loop has lot of instructions
- Extra space for count[] ⟶ This is why it is not used
- Extra space for aux[]

Disadvantages of quicksort :

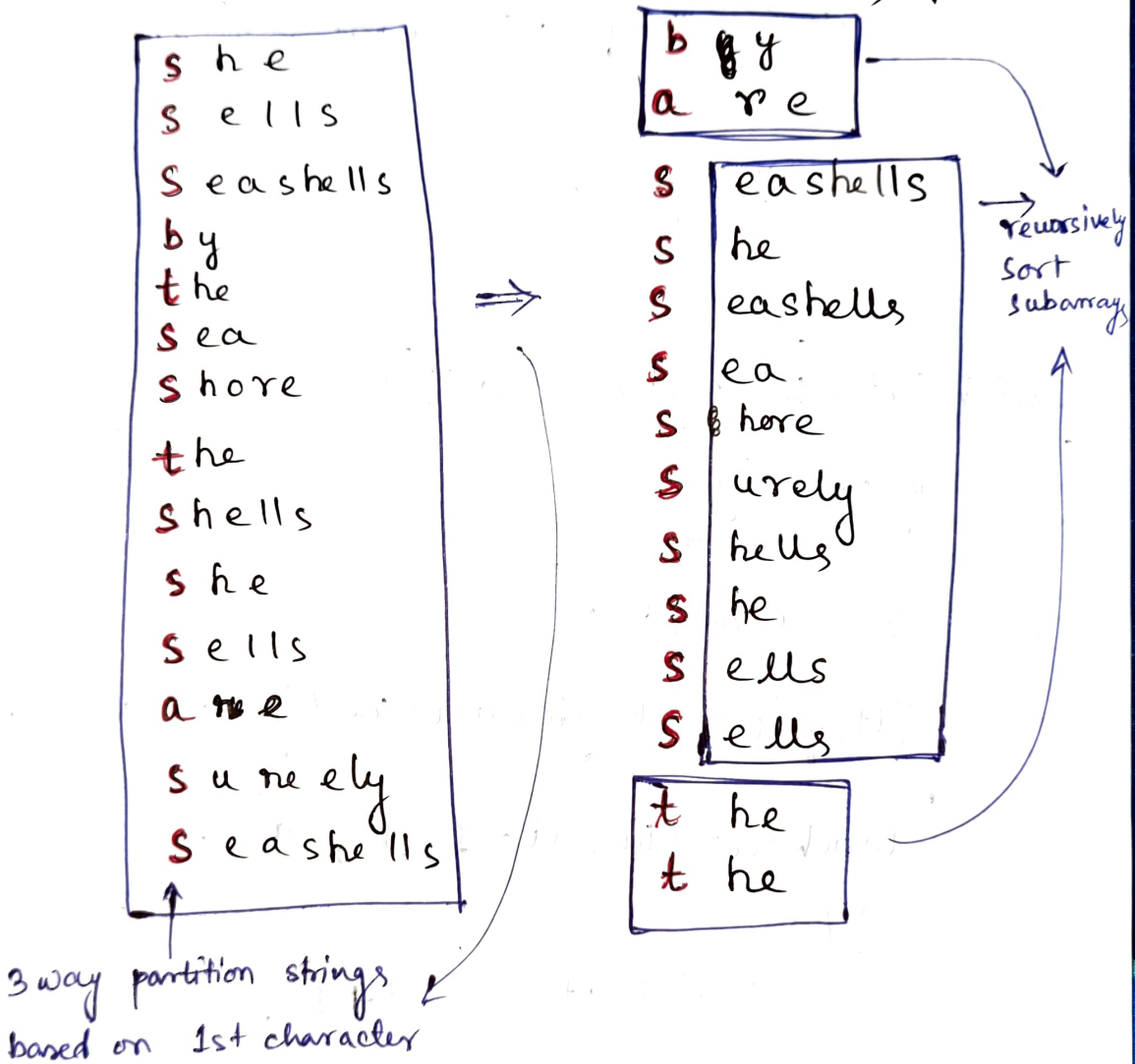
- Linearithmic no. of string compares (not linear)
- Has to rescan many characters in keys with long prefix matches.

③ 3-way Quick Sort : [Invented by Bentley and Sedgewick, 1997]

- Do 3 way partitioning on d-th character.
- Less overhead than R-way partitioning in MSD string sort.
- Does not re-examine characters equal to the partitioning char.

(but does re-examine characters not equal to partitioning char).

eg:



Pseudo Algorithm :

3 way partitioning w.r.t value v, based on d-th character

```
3WayPartition (low, high, v, d, a[])
{
    i = low;  lt = low;  gt = high;
    while (i ≤ gt)
    {
        if (a[i][d] == v)  i++;
        else if (a[i][d] < v)  swap (a[lt++], a[i++]);
        else  swap (a[i], a[gt--]);
    }
    return {lt, gt};
}
```

Swap references, not copy

```
QuickSort (low, high, d, a[])
{
    if (low ≥ high)
        return;
    v = a[low][d];
    {lt, gt} = 3WayPartition (low, high, v, d, a);
    QuickSort (low, lt-1, d, a);
    QuickSort (lt, gt, d+1);
    QuickSort (gt+1, high, d);
}
Sort (a[])
{
    QuickSort (0, n-1, 0, a);
}
```

(*) Note: We need to take special care for variable length strings. (append a special char at end which is smaller than all char)

Analysis:

Standard Quick Sort:

- No. of string compares on average $\approx 2n \log n$
- Costly for keys with long common prefixes (and this is a common case in practice)

3 way String Quick Sort:

- No. of character compares on average $\approx 2n \log n$
- Avoids re-comparing long common prefixes.

MSD string Sort	vs.	3 way String Quick Sort
• Cache inefficient.		• Cache friendly.
• Too much memory for count[].		• In place
• Too much overhead reinitializing count[] and aux[].		• Has short inner loop.

- **Bottom Line:** 3 way string Quick sort is a good method of choice for sorting strings.

(*) 3 way Quick Sort is asymptotically optimal.

Average No. of character compares: $1.39\, n \log n$

for, LSD radix sort, no. of array accesses $= 2nw$

Quick Select for String :-

```
int QuickSelect (low, high, d, a[], k)
{
    if (low >= high)
        return low;

    v = a[low][d];
    {lt, gt} = ThreeWayPartition (low, high, v, d, a);

    if (lt == gt == k)
        return k;
    else if (lt <= k <= gt)
        return QuickSelect (lt, gt, d+1, a, k);
    else if (k < lt)
        return QuickSelect (low, lt-1, d, arr, k);
    else // if (k > gt)
        return QuickSelect (gt+1, high, d, arr, k);
}
```

SUFFIX ARRAYS

① Keyword in context search :

Given a text of n characters, preprocess it to enable fast substring search (find all occurrences of query string context).

Applications:

- Linguistic
- Databases
- Web Search
- Word Processing

Solution : Suffix Sort :

input string: i t w a s b e s t i t w a s w

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14

Form Suffixes :

0 → i t w a s b e s t i t w a s w
1 → t w a s b e s t i t w a s w
2 → w a s b e s t i t w a s w
3 → a s b e s t i t w a s w
4 → s b e s t i t w a s w
5 → b e s t i t w a s w
6 → e s t i t w a s w
7 → s t i t w a s w
8 → t i t w a s w
9 → i t w a s w
10 → t w a s w
11 → w a s w
12 → a s w
13 → s w
14 → w

Sort Suffixes to bring repeated substrings together

3 a s b e s t
12 a s w
5 b e s t i t w a s w
6 e s t i t w a s w
0 i t w a s b e s t i t w a s w
9 i t w a s w
4 s b e s t i t w a s w
7 s t i t w a s w
13 s w
8 t i t w a s w
1 t w a s b e s t i t w a s w
10 t w a s w
14 w
2 w a s b e s t i t w a s w
11 w a s w

Search "it was" keyword

** store all suffixes using reference, instead of copying whole string

After sorting the suffixes, the repeated substrings are placed consecutively. So we can use Binary Search to find all occurrences of that substring efficiently.

Keyword in context search:

- **Preprocess:** Suffix sort the text
- **Query:** Binary search for query; scan until mismatch

(2) Longest Repeated Substring:

Given a string of n characters, find the longest repeated substring.

Applications:

- Bioinformatics
- Cryptanalysis
- Data Compression
- Visualize Repetitions in music

(i) Brute force Algorithm:

- Try all indices i and j for start of possible match.
- Compute longest common prefix (LCP) for each pair (starting from i and j).
- **Time Complexity:** $O(n^2 \times D)$, where, D = length of longest repeated substring

(ii) Sorting Soln (Suffix Sort):

1) Compute suffixes (Using Reference, not copy)
2) Suffix Sort to bring repeated substrings together
3) Check each consecutive pair of suffixes to find max length.

```
String lrs (string s, n)
{
    1) suffixes [] = Build Suffixes (s, n);
    2) String Sort (Suffixes, n);
    3) resInd = -1, resLen = 0;
       for (i= 0 to n-2)
       {
           len = Length Of Longest Common Prefix (suffixes[i], suffixes[i+1])
           if (len > resLen)
           {
               resLen = len;
               resInd = i;
           }
       }
       return suffixes[resInd] . substr (0, resLen);
}
```

Worst Case Input :

- Two copies of same substring

eg: "twins twins"

Suffixes in Sorted order:

3 compares: ins / ins twins

2 compares: ns / ns twins

1 compare: s / s twins

5 compares: twins / twins twins

4 compares: wins / wins twins

[D = length of LRS]

- Here LRS is twins,

LRS needs at least $1 + 2 + 3 + \cdots + D$

$\approx O(D^2)$ character compares.

Running time : Quadratic (or worse) in D for LRS (also for sorting)

C++ implementation of Suffix Sort using reference (without copying)

```
class Suffix
{
    const string * str;
    int startInd;
public:
    Suffix() { }
    Suffix(const string & str, int startInd)
    {   this -> str = &str;
        this -> startInd = startInd;
    }
    int size() const { return (str -> size() - startInd); }
    char operator[] (int ind) const
    {
        if (this -> startInd + ind > str -> size())
            return -1;
        else
            return (*str)[startInd + ind];
    }
    friend void swap(Suffix & a, Suffix & b);
};
void swap(Suffix &a, Suffix & b)
{   swap(a.str, b.str);
    swap(a.startInd, b.startInd);
}
```

(*) important for variable length strings

TRIE

String Symbol Table Basic API :

public class StringST<Value> :

- StringST() ⟶ Creates an empty symbol table
- void put(String key, Value val) ⟶ put (key, value) pair in the ST
- Value get(String key) ⟶ return value paired with given key
- void delete(String key) ⟶ delete key and corresponding value.

Goal: Faster than hashing and more flexible than BST.

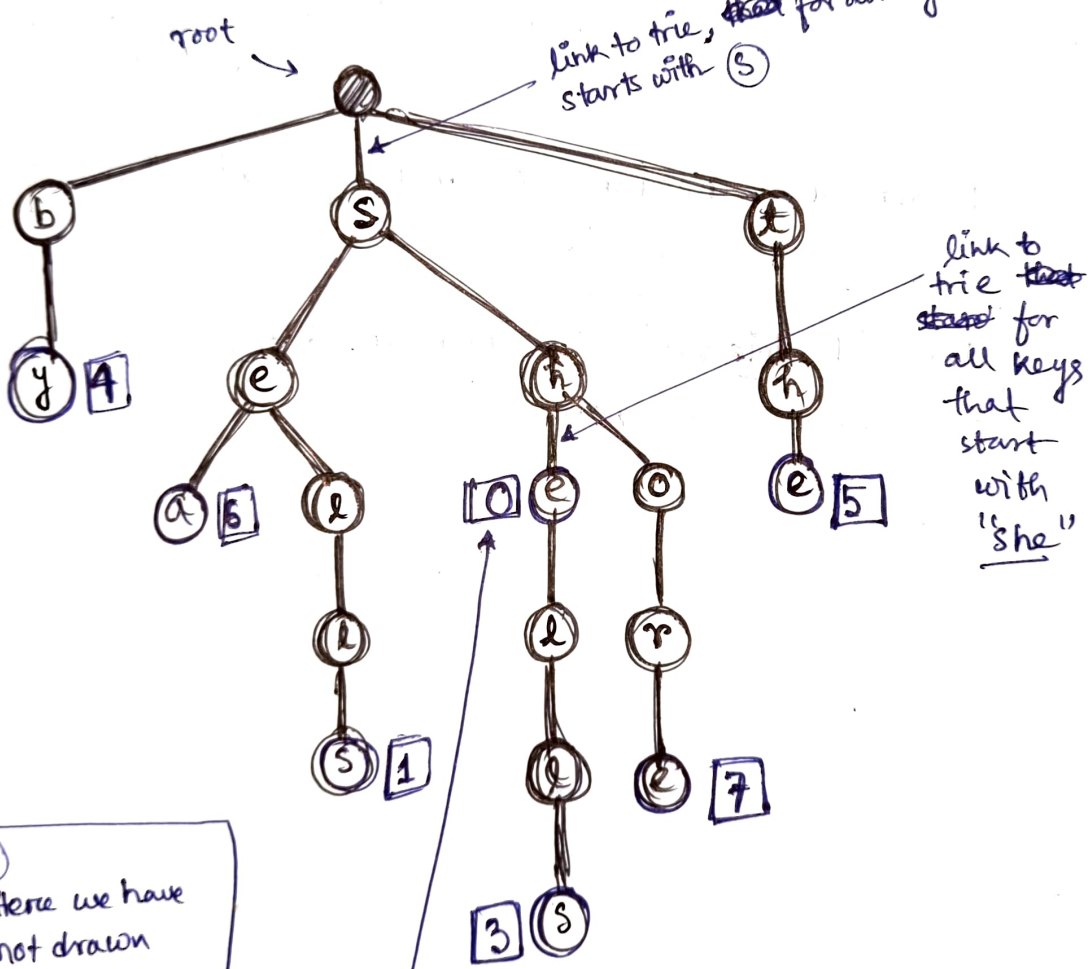
Trie [from "Retrieval", but pronounced "try"]

R-way Trie :-

- Store character in nodes (not keys).
- Each node has R children, one for each possible character. [R = Radix]
- Store values in nodes corresponding to last character of keys.

eg:

Key	by	sea	sells	she	shells	shore	the
value	4	6	1	0	3	7	5



(*) Here we have not drawn null links

value for "she" in node corresponding to last key char ('e')

Search :-

Follow links corresponding to each character in the key.

- **Search Hit:** node where search ends has an associated value
- **Search Miss:** 2 cases: 1) Reached null link
  2) Node where search ended, doesn't have associated val.

Insertion :-

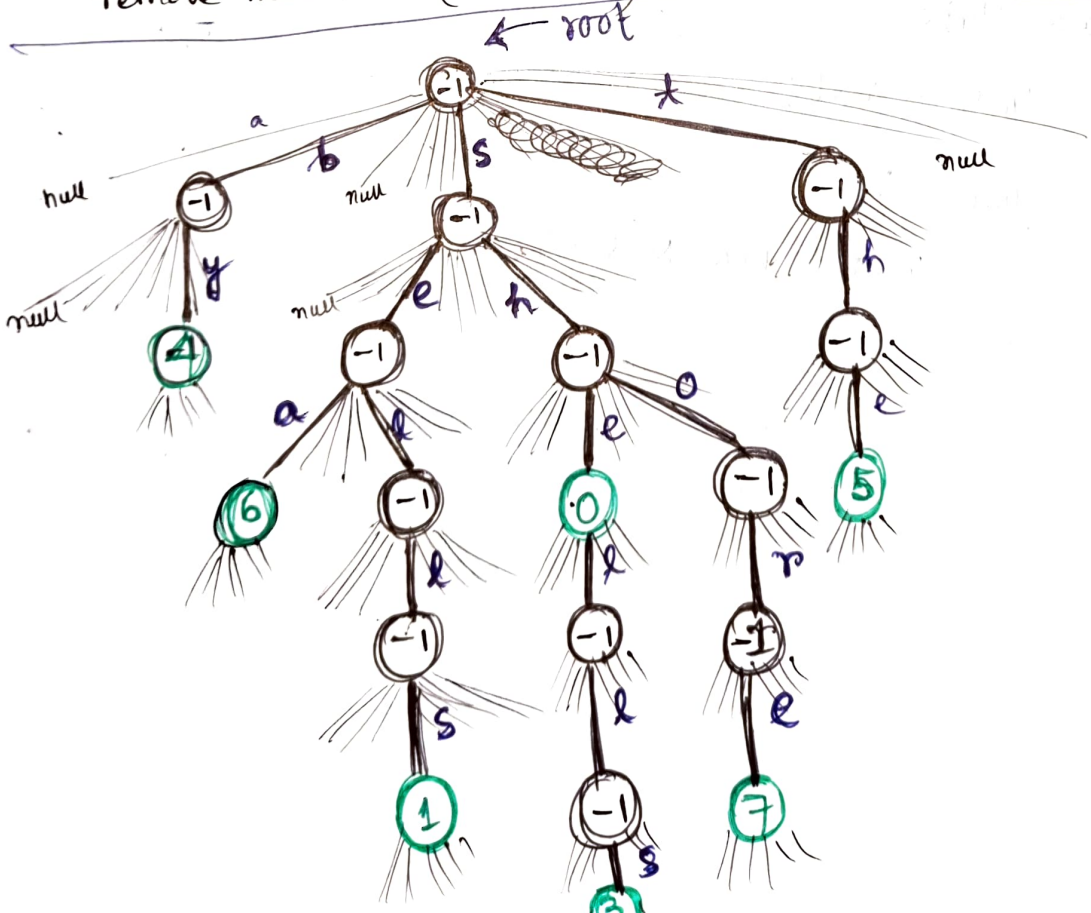
Follow links corresponding to each character in the key:

1) if null link encountered: create a new node
2) last char of the key encountered: set value in that node

Deletion :-

To delete a key value pair:

1) Find the node corresponding to key and set value to null (-1)
2) If node has null value and all null links, remove that node (and recur)



Implementation :

R = 256
↘ extended ASCII

```
Node
{   val ;
    Node *  next [R] ;
}
```

```
class Node
{   int val ;
    Node * next [R];

    Node ( val = -1)
    {   this → val = val ;
        for (i=0 to R-1)
            next[i] = NULL;
    }
};
```

```
Class Trie
{   Node * root ;

    Trie ( ):
    {   root = new Node ();
    }
};
```

Put :

```
void put (string key, val)
{
    root = put (root, key, val, 0);
}

Node * put (Node * x, string& key, int val, int d)
{
    if (x == null)  x = new Node ();
    if (d == key.size)
    {
        x->val = val;
        return x;
    }
    char c = key.charAt(d);  key[d];
    x->next[c] = put (x->next[c], key, val, d+1);
    return x;
}
```

Get :

```
int get (key)
{
    Node * x = get (root, key, 0);
    return x != null ? x->val : -1;
}
Node * get (Node * x, string& key, int d)
{
    if (x == null) return null;
    if (d == key.size) return x;
    return get (x->next[key[d]], key, d+1);
}
```

Performance :

- **Search hit :**
  - need to examine all L characters for equality.
- **Search miss :**
  - Could have mismatch on first char
  - Typical case : Examine only a few char (sublinear)
- **Space :**
  - R null links at each leaf.
  - but sublinear space possible if many short strings share common prefix.

Bottom line : fast search hit and even faster search miss, but wastes space.

Popular Interview Question :

Goal : design a data structure to perform efficient spell checking.

Soln : Build a 26 way trie (Key = word, value = bit)

↑ english language text

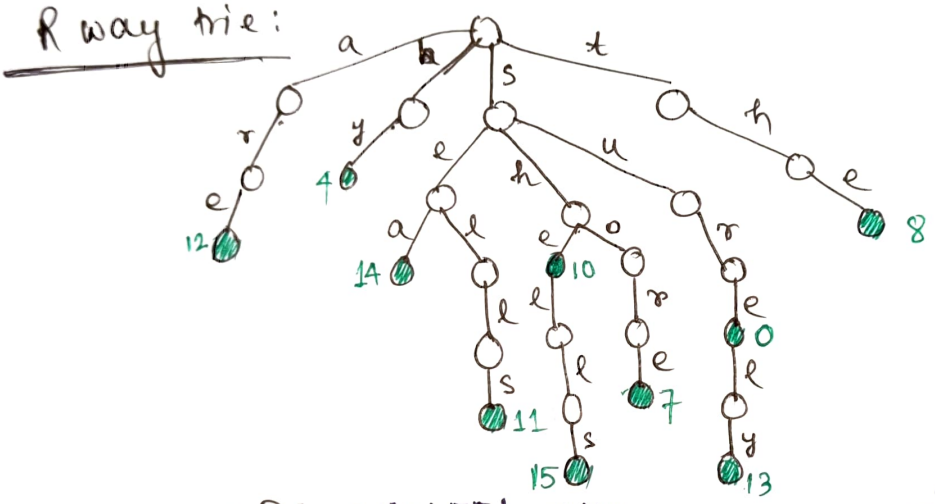
Ternary Search Trie : [Invented by Bentley and Sedgewick]

- Store characters and values in nodes (not keys)
- Each node has 3 children : 1) Smaller (left)
  2) ~~Middle~~ Equal (middle)
  3) Larger (right)
- Similar to Binary Search Tree.

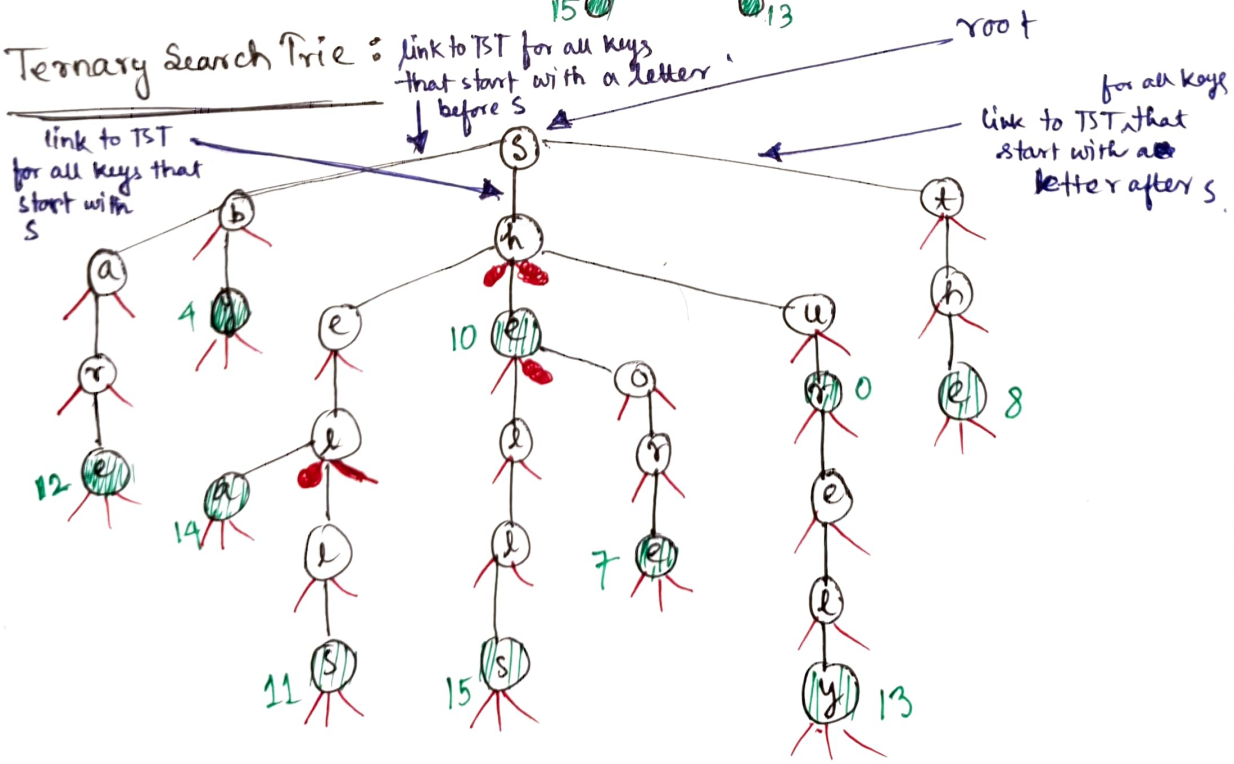
eg:

Key	are	by	sea	sells	she	shells	shore	sure	surely	the
val	12	4	14	11	10	15	7	0	13	8

R way trie:



Ternary Search Trie : link to TST for all keys that start with a letter before s



Search :-

Follow links corresponding to each character in the key:

- if less, take left link; if greater take right link
- if equal, take the middle link and move to the next character

Search hit: Node where search ends has a non null value

Search miss: Reach a null link or node, where search ends, has a null value.

Insert :-

Implementation :~

```
class Node
{   int val;
    char c;
    Node * left, * middle, * right;
};
```

```
class TST
{   Node *root;
    TST()
    {
    }
};
```

Put :

```
void put(key, val)
{   root = put(root, 0, key, val);
}
Node * put(Node * root, int d, string &key, int val)
{   c = key[d];
    if (root == null) { root = new Node(); root->c = c; }
    if (c < root->c)  root->left = put(root->left, d, key, val);
    else if (c > root->c)  root->right = put(root->right, d, key, val);
    else if (d < key.size - 1)  root
    else
```

```
Node *
put (root, d, key, val)
{
    c = key[d];
    if (!root)
    {
        root = new Node();
        root->c = c;
    }
    if (c == root->c)
    {
        if (d < key.size - 1)
            root->mid = put(root->mid, d+1, key, val);
        else
            root->val = val;
    }
    else if (c < root->left)
        root->left = put(root->left, d, key, val);
    else // if (c > root->c)
        root->right = put(root->right, d, key, val);
    return root;
}
```

Get :

```
int get (key)
{
    Node * x = get(root, 0, key);
    return  x ≠ null ? x.val : -1;
}

Node * get (root, d, key)
{   if(!root)
        return NULL;
    c = key[d];
    if (c == root→c)
    {   if (d < key.size - 1)
            return get(root→mid, d+1, key);
        else
            return root;
    }
    else if (c < root→c)
        return get(root→left, d, key);
    else // if (c > root→c)
        return get(root→right, d, key);
}
```

TST vs Hashing :

Hashing :

- Need to examine the entire key
- Search hits and search misses cost about the same.
- Performance relies on hash function.
- Does not support ordered symbol table operations.

TST :

- Works only for strings (or digital keys)
- Only examines just enough key characters.
- Search miss may involve only a few characters.
- Supports ordered symbol table operations (plus others!)

Character Based Operations

- Trie data structure can support several useful character ~~character~~ based operations.

eg:

key	by	sea	sells	she	shells	shore	the
val	4	6	1	0	3	7	5

Prefix Match: all keys with prefix "sh" : "she", "shells", "shore"

Wild card Match: Keys that match ".he" : "she" and "the"

Longest Prefix Match: Key that is the longest prefix of "shell sort" is: "shells"

(*) We can also add other ordered symbol Table methods eg: floor() and rank().

Find all keys in sorted order : Inorder Traversal:

To iterate through all keys in sorted order:
- Do inorder traversal of trie, add keys encountered to queue.
- Maintain sequence of characters on path from root to node.

For R-way Trie:

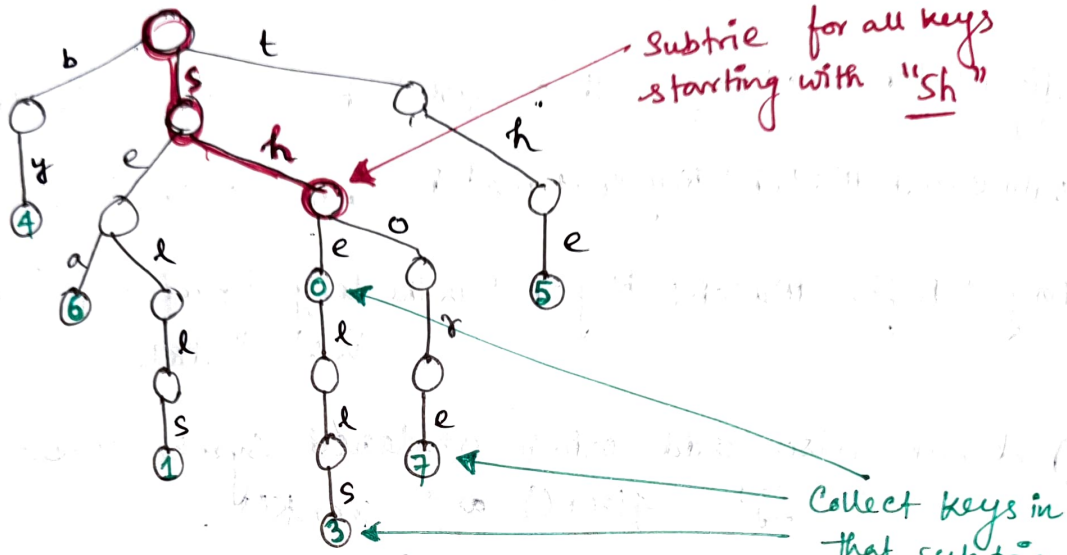
```
vector<string> getKeys()
{   vector<string> q; string prefix = "";
    inorderCollect(root, prefix, q);
    return q;
}
void inorderCollect(Node* root, string &prefix, vector<string> &q)
{   if (!root) return;
    if (root->val != -1) q.push_back(prefix);
    for (c = 0 to R-1)
    {   prefix.push_back(c);
        inorderCollect(root->next[c], prefix, q);
        prefix.pop_back();
    }
}
```

Prefix Matches :

- Find all keys in a symbol Table starting with a given prefix.

eg: Autocomplete in a cell phone, search bar, text editor or shell

eg:



To find keys with Prefix ("sh"),

i) Find subtrie for all keys begining with "sh"

ii) collect keys of that subtrie using inorder traversal

```
vector<string> getKeysWithPrefix (prefix)
{
    vector<string> q ;
    i) Node * x = get(root, 0, prefix);
    ii) inorderCollect (x, prefix, q );
    return q;
}
```

Longest Prefix Match :-

- Find longest key in symbol table, that is a prefix of the given query string.

eg: To send packet toward destination IP address, router chooses IP address in Routing table that is the longest prefix match.

"128"
"128.112"
"128.112.055"
"128.112.055.15"
"128.112.136"
"128.112.155.11"
"128.112.155.13"
"128.222"
"128.222.136"

- LongestPrefixOf ("128.112.136.11") = "128.112.136"
- LongestPrefixOf ("128.112.100.16") = "128.112"
- LongestPrefixOf ("128.166.123.45") = "128"

Note: Not the same as floor;

floor ("128.112.100.16") = "128.112.055.15"

Pseudo Algorithm:

Find longest key in symbol table that is a prefix of query string:

- Search for query string
- keep track of longest key encountered

```
string LongestPrefixOf (query)
{
    length = searchLongestKeyLength (root, query, 0, 0);
    return query.substr (0, length);
}

int searchLongestKeyLength (Node * root, const string & query,
                            int d, int length)
{
    if (!root) return length;
    if (root->val != -1)
        length = d;
    if (d == query.size())
        return length;
    return searchLongestKeyLength (root->next[query[d]],
                                   d+1, length);
}
```

2 possibilities of termination of search: either there is a mismatch, or, query string has been exhausted

Patricia trie :

[Practical algorithm to retrieve information coded in Alphanumeric]

- Remove one way-branching.
- Each node represents a sequence of characters.
- Implementation is beyond this course.

Applications :

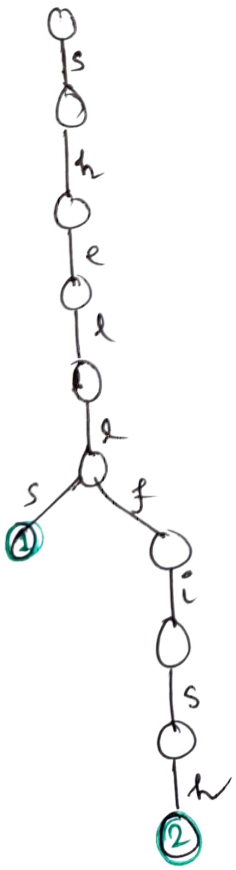
- Database search
- P2P network search
- IP routing tables : find longest prefix match
- compressed quad tree for N-body simulation.
- Efficiently storing and querying XML documents

eg :

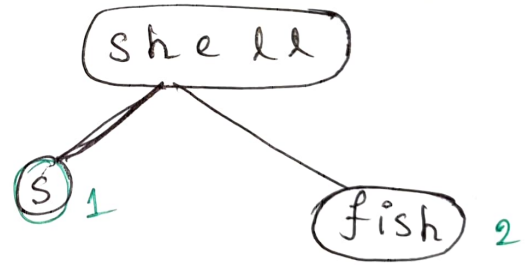
```
put ("shells", 1);
put ("shellfish", 2);
```

[Patricia trie]

Standard trie



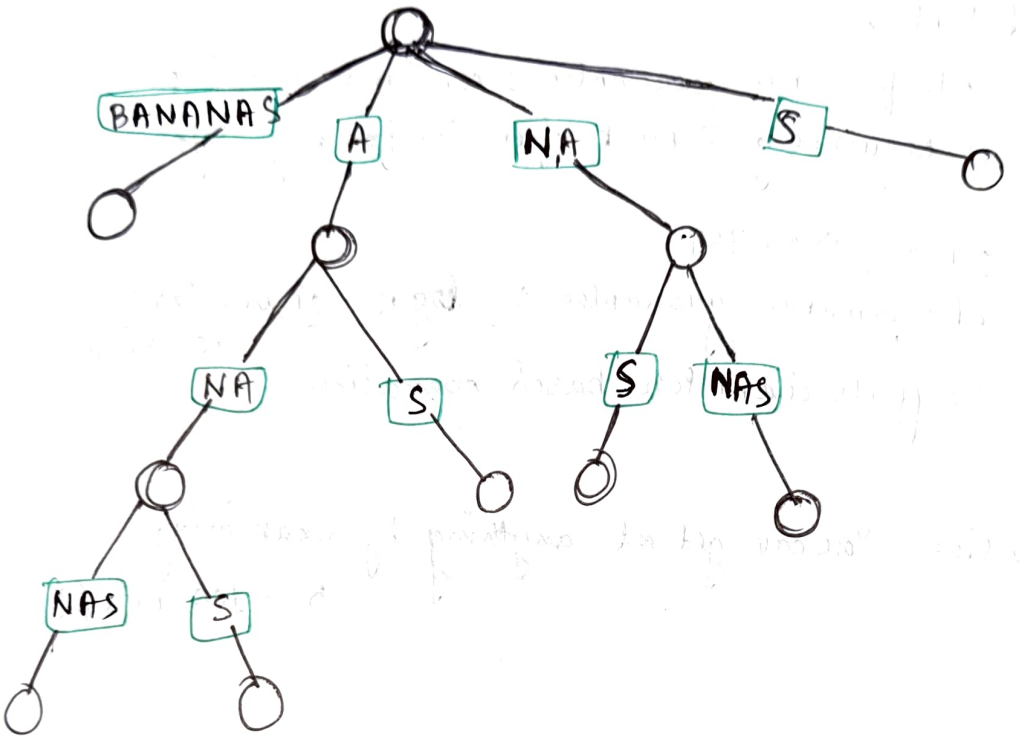
No one way branching :



Suffix tree :

- Patricia trie of suffixes of string
- Linear-time construction : beyond this course.

eg: Suffix tree for "BANANAS" :



Applications :

- Linear time : Longest repeated substring,
  Longest Common Substring,
  Longest Palindromic Substring,
  Substring Search,
  Tandem Repeats
- Computational biology database (BLAST, FASTA)

String Symbol Table Summary :~

1) Red-Black BST:

- Performance guarantee : $\log n$ key compares
- Supports ordered symbol table API

2) Hash Table:

- Performance guarantee : constant no. of probes
- Requires good hash function for key type

3) Tries: R-way trie, TST

- Performance guarantee : $\log n$ characters accessed.
- Supports character based operations.

Bottom line : You can get at anything by examining 50 - 100 bits (!! !)