# Searching Algorithms

* Linear Search
  ↳ unsorted search list data set
  ↳ small dataset
  ↳ low memory restrictions (algorithm has low space complexity [O(1)])
  - implemented using a simple for loop

* Binary Search
  ↳ most popular search
  ↳ need sorted array as input
  ↳ not suitable for linked lists (slow middle access)
  ↳ $T(n) = O(\log_2 n)$

```
def bSearchIterative(arr, target)
    left = 0
    right = len(arr) - 1

    while left <= right:
        mid = (left + right) // 2      ← floor division

        if arr[mid] == target:
            return mid

        elif arr[mid] > target:
            right = mid - 1

        else:
            left = mid + 1

    return -1
```

* Jump Search
  ↳ for sorted arrays
  ↳ $T(n) = O(\sqrt{n})$
  ↳ Skipps ahead by fixed intervals.
  ↳ Works bdder than bSearch for linked lists
  ↳ bSearch may cause many cache misses, when accessing high latency memory.
  ↳ jump search is more cache friendly
  ↳ can be modified for partially sorted arrays
  ↳ optimum jump size is $\sqrt{n}$ (geeksforgeeks has calculation)

```
n= len(arr)
step= √n
prev= 0

      handles final iteration
while arr[min(step, n) -1] < target:
    prev=step
    step += √n
    if prev >=n:
        return -1

for i in range(prev, min(step, n)):
    if arr[i] == target:
        retur i

redur -1
```

1

Date......../......../........ No:........

* Interpolation Search
  ↳ $T(n) = O(\log(\log(n)))$
  ↳ effective for uniformly distributed arrays
  ↳ improves upon binary search by estimating probable position of target

$$\boxed{pos = low + \left(\frac{target - arr[low]}{arr[high] - arr[low]} \times (high - low)\right)}$$

```
interpolationSearch(arr, low, high, target)
    if (low<=high AND target >= arr[low] AND target <= arr[high]
        pos = low + ((target - arr[low]) / (arr[high] - arr[low]))
                                                    * (high-

        if arr[pos] == target
            return pos
        elif arr[pos] < target
            return interpolationSearch(arr, pos+1, hi, target)
        else
            retur interpolationSearch(arr, lo, pos-1, target)
```

# Sorting Algorithms

* Stable - maintains relative order of equal elements

* Insertion Sort
  - split array into 2
  - assume left half is sorted and add unsorted items to correct position in sorted array
  - start with 2nd element (take 1st element alone to be sorted part)
  - compare 1 and 2 and swap if needed
  - move to 3rd element and compare with 1 and 2 and put in correct position

```
for i = 1 to n   <- start with 1 since 0 is already sorted   [n = ~~size of array~~ number of elements]
    key = arr[i]     <- element that we are "inserting"
    j = i - 1

    while (j >= 0 && arr[j] > key)
        arr[j+1] = arr[j]
        j = j - 1

    arr[j+1] = key
```

* look for posidition of key by moving each compared element to the right along the sorted part of the array till we find the correct place

## * Shell Sort

- 1st algo to break quadratic time barrier (i.e $T(n) \leq O(n^2)$)
- AKA diminishing Increment sort
- Best → $O(n \log n)$ Avg → $O(n \log n^2)$ - Worst $O(n^2)$

```
for(int interval=n/2;interval>0; interval /=2){
    for(int i=interval; i<n ; i++){
        int temp = arr[i];
        int j;
        for (j= i; j>= interval&& arr[j-interval] >temp; j-=interv
            arr(j) = arr(j-interval]
        }
        arr[j] =temp;
    }
}
```

* A worst case, this devolves to insertion sort

## * Radix Sort

- considers the structure of keys
- groups numbers by their individual digits (radix)
- using radix as key, uses counting sort to sort them
- is a stable sort (will not work otherwise
- usually use for a fixed range
- has high auxiliary space
- $T(n) = O(d*(n+b))$

n - number of elements
d - max digits in a number
b - base of counting (10 for decimal etc.)

mx = 802

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 170 | 45 | 75 | 90 | 802 | 24 | 2 | 66 |

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| 2 | 2 | 3 | 4 | 4 | 7 | 7 | 8 | 8 | 8 |

| | | | | | | |
|---|---|---|---|---|---|---|
| | | 2 | 24 | | | 66 |

```
ind gedMax(int arr[], ind n){
    int mx = arr[0];
    for(int i=1; i<n; i++){
        if (arr[i] > mx) {
            mx = arr[i];
        }
    redurn mx;
}

void countS(int arr[], ind n, int place) {
    int outpud[n];
    int count[10] = {0};

    for(int i=0; i<n; i++){
        count[(arr[i]/place)% 10]++; }

    for(int i=1; i<10; i++){
        cound[i] += count[i-1] }

    for (int i=n-1; i>=0; i--){
        outpud[count[(arr[i]/place)% 10] - 1] = arr[i];
        count[(arr[i]/place) % 10]--; }

    for(int i=0; i<n; i++){
        arr[i] = output[i]; }
}
void radixS(int arr[], int n){
    int m = gedMax(arr, n);

    for(int place=1; m/place >0; place *= 10){
        coundS(arr, n, place);
    }
}
```

# Hashing

* Have average case time complexity = O(1)
  Worst case is still = O(n)

* 2 main types
  ↳ Static hashing
  ↳ Dynamic hashing

* Hashing has 3 main parts
  1) Key: input given to the hash function
  2) Hash function: takes key and returns index of an element in hash table called hash index
  3) Hash table: data structures that maps keys to values.

* Hash table is usually an array of structs that includes a char array to store the key ~~as~~ followed by any other required data types

* Various implementations of hashing
  ↳ hash list
  ↳ hash trees

* Ideal Hashing
  ↳ Each key is converd into a seperate index and stored in the home bucket
  ↳ No collisions

* But collisions do occur so we consider 2 things.
  1) Choose a good hash function to minimize collissions
  2) Handle overflow efficiently

RICHARD

* Clustering - hash function groups multiple key value pairs together
  - not in same bucket but many nearby buckets are filled
* Collisions - hash function evealutes completely different keys and returns the same hash index there by storing them in the same bucket.
  ↳ pigeonhole principle

* Good hash functions avoids above 2 conditions as much as possible

* Why can't we use n-sized hash table?
  ↳ high memory usage
  ↳ need a perfect hash function that maps each unique key to a unique hash index
  ↳ very hard to do for dynamic data (Data is stored and deleted)

* Characteristics of hash functions
  - Efficiency
  - Uniform distribution (avoid clustering)
  - Determinism (same input key always gives same hash index)
  - Minimized collisions

## Hash Functions

1) Middle of Square Hash function
  * Take input key x
  * Square the key ($x^2$)
  * Choose a specific number of digits from the middle of $x^2$
  * Use %m on extracted digits to fit to table (m = table size)
  - Used for small hash tables with integer keys.

3) Multiplicative Method

* $h(x) = floor(table size \times (key \times A \% 1))$
  $= \lfloor (m \cdot (k \cdot A \% 1)) \rfloor$
* A is a constant (usually $\underbrace{0.618}_{\text{Knuth's recommendation}}$ is chosen) $[0 < A < 1]$.
* Better distribution than division method.
* Slower than division

4) Folding Hash Function

1) Seperate ~~the~~ key into smaller fixed-size parts
2) Combine parts.
   ↳ XOR, ~~AND~~ or Add together
3) Mod by table size.

5) Digit Analysis

* Assumes that distribution of keys is known in advance
  ↳ Analyze digits in key that have skewed distribution
  ↳ Eliminate skewed digits
  ↳ Use remaining digits

6) Division Method

* Simplest and most widely used
* $h(x) = x \% m$ (m = table size)
* need to choose good m
* Generally: a prime number that is not close to a power of 2.

* For a hash table of size b, a given bucket will have approximately $2^{32}/b$ integers (when trying to hash all integers)
* If divisor is even, even ints go into even buckets and odd into odd buckets.
* If divisor is odd, ~~even, odd~~ both will hash into any home.

+ Biased distribution occurs if we use a divisor that is a multiple of prime numbers.
+ Ideally choose a large prime for divisor
+ Or choose divisor such that it has no prime factors smaller than 20.

+ Criterion of a Hash Table
  - Key density = $n/T$ (n - number of keys in table, T - number of distinct possible keys)
  - loading density / factor
    $\alpha = n/(sb)$ (s - number of slots)
    (b - number of buckets)

## Overflow Handling

+ When a collision occurs we get an overflow
+ 2 Solutions
  1) Search for an empty bucket (open addressing)
     - linear probing (linear open addressing)
     - Quadratic Probing
     - Random Probing
  2) Each bucket has a list of all keys indexed to it
     - Array ~~linear li~~ linear list
     - Chaining

+ Open addressing - all elements stored directly into hash table

+ ~~Random~~ Quadratic probing => $h'(k,i) = (h(k) + i^2) \% m$
+ Random probing => $h'(k) = (h(k) + r[i]) \% m$

i = number of collisions for given k.