Searching: Unsorted List
- Each element is compared to locate the desired element one after another starting at the head of the list.
- Worst Case Order = $O(N)$
  - desired element is at the end of the list.
- Average Case Order = $O(N/2)$ ∈ $O(N)$
  - one half of the list must be scanned on the average.
- Assumes that the probability of each element in the list being searched for is equal.

Sequential Searching on a Sorted list
- Search stops when element is located or a larger element (ascending order) is encountered.
- Worst case and average case orders are the same as the unordered list.

Probability != for all Elements
- Unequal Access Probabilities
  - Implemented when a small subset of the list elements are accessed more frequently than other elements.
  - Static Probabilities
  - Dynamic Probabilities

Simple Searching
- Internal (primary memory) searching
- External => File Search
  - (Indexes, Btrees, files, etc.)

Static Probabilities
- You know before hand which elements are going to be accessed more than others
- The contents of the list are static and the most frequently accessed elements are stored at the beginning of the list.
- Assumes that access probabilities are also static

Dynamic Probabilities
- For non-static lists or lists with dynamic probability element accesses, a dynamic element ordering scheme is required:
  - Sequential Swap Scheme
    - Move each element accessed to the start of the list if it is not within some threshold units of the head of the list.
  - Bubble Scheme
    - Swap each element accessed with the preceding element to allow elements to “bubble” to the head of the list.
  - Access Count Scheme
    - Maintain a counter for each element that is incremented anytime an element is accessed.
    - Maintain a sorted list ordered on the access counts.
Optimizing Sequential Search

```c
const int MISSING = -1;

inline int SeqSearch2 (const Item A[], Item K, int size) {
    int i;
    for (i=0;((i<size) && !(K==A[i]));i++);
    return ((i<size)?(i):(MISSING));
}
```

Problem: the two comparisons in the loop are inefficient, so we instead search sequentially down to 0 using 0 as limit test

More Efficient

```c
const int MISSING = -1;

int SeqSearch3(const Item A[], Item K, int size) {
    //i is not a compare, it is a truth
    for ( i = size -1; (!(K == A[i]) && (i)); i--);
    if ( K == A[i] )
        return ( i );
    else
        return (MISSING);
}
```

Binary Search

```
Worst Case Order = O( log₂ N )
```

Binary Search: Recursive

```c
const int MISSING = -1;

int BinarySearch ( const Item A[], Item K, int L, int R) {
    int Midpoint = (L+R) / 2 ;  //compute midpoint
    if (L>R)//If search interval is empty return -1
        return MISSING ;
    else if(K==A[Midpoint])  //successful search
        return Midpoint;
    else if ( K< A[Midpoint] ) //search upper half
        return BinarySearch(A, K, Midpoint + 1, R);
    else  //search lower half
        return BinarySearch(A, K, L, Midpoint - 1);
}
```
Binary Search Optimizations

- For small lists a sequential search will usually be faster due to the midpoint computation and comparisons.
- Minor changes to highly efficient algorithms (e.g., binary search) can have a drastic negative effect on execution.
- Changing the indexes to longints can increase execution time by a factor of 3.
- Using real division and truncating for the midpoint computation may slow execution by more than 10 times.

Variation of Binary Search

- Interpolation Search
  - Attempts to more accurately predict where the item may fall within the list. Similar to looking up telephone numbers
  - Standard Binary Search Midpoint Computation:
    \[
    \text{Midpoint} = \frac{\text{L} + \text{R}}{2}
    \]

Interpolation Search

- General Binary Search Midpoint Computation
  \[
  \text{Midpoint} = \text{L} + \frac{1}{2} \cdot \left( \frac{\text{R} - \text{L}}{} \right);
  \]
- Interpolation replaces the 1/2 (in the above formula) with an estimate of where the desired element is located in the range, based on the available values (be careful of int arithmetic):
  \[
  \text{Interp} = \text{L} + \frac{(\text{K} - \text{A}[\text{L}])}{(\text{A}[\text{R}] - \text{A}[\text{L}])} \cdot \frac{\text{R} - \text{L}}{};
  \]

Interpolation Search Example

- Assume 30K recs of SSNs in the range from 0 ... 600 00 0000
- Searching for 222 22 2222 yields an initial estimate of:
  \[
  \text{Interpolation} = \frac{(22222222 - 0)}{(600000000 - 0)} \cdot \frac{30000}{30000} = 11111
  \]

Interpolation Search Big Oh

- Worst Case Order approximately \( \mathcal{O}(\log \log N) \)
- Can be assumed to be a constant of about 5 for this example since \( \approx (\log \log 10^9) \)
- Assumes the search values are evenly distributed over the search range, (however, this is NOT true for SSNs)
- Inefficient for searching small number of elements