Sequential Search

Write a function that takes n data stored in an array of type `int` and an integer known as the key. The function sequentially searches the array for the key. If the key is present among the data in the array, it returns the corresponding array index; otherwise it returns -1.

Inductive Definition

$$S(a[], n, k) = \left\{ \begin{array}{l} 0, \quad \text{if } a[0] = k, \\ \quad -1, \quad \text{if } n = 1, \\ \quad -1, \quad \text{if } S(a+1, n-1, k) = -1, \\ \quad 1 + S(a+1, n-1, k), \quad \text{else} \end{array} \right\}, a[0] \neq k.$$

Iterative C Function

```
#define NOTFOUND -1
int seqSearch(int data[], int noOfData,
              int key){
    int i ;
    for(i=0; i<noOfData; ++i)
        if(data[i] == key) break ;
    if(i == noOfData) return NOTFOUND;
    return i;
} // seqSearchF.c
```

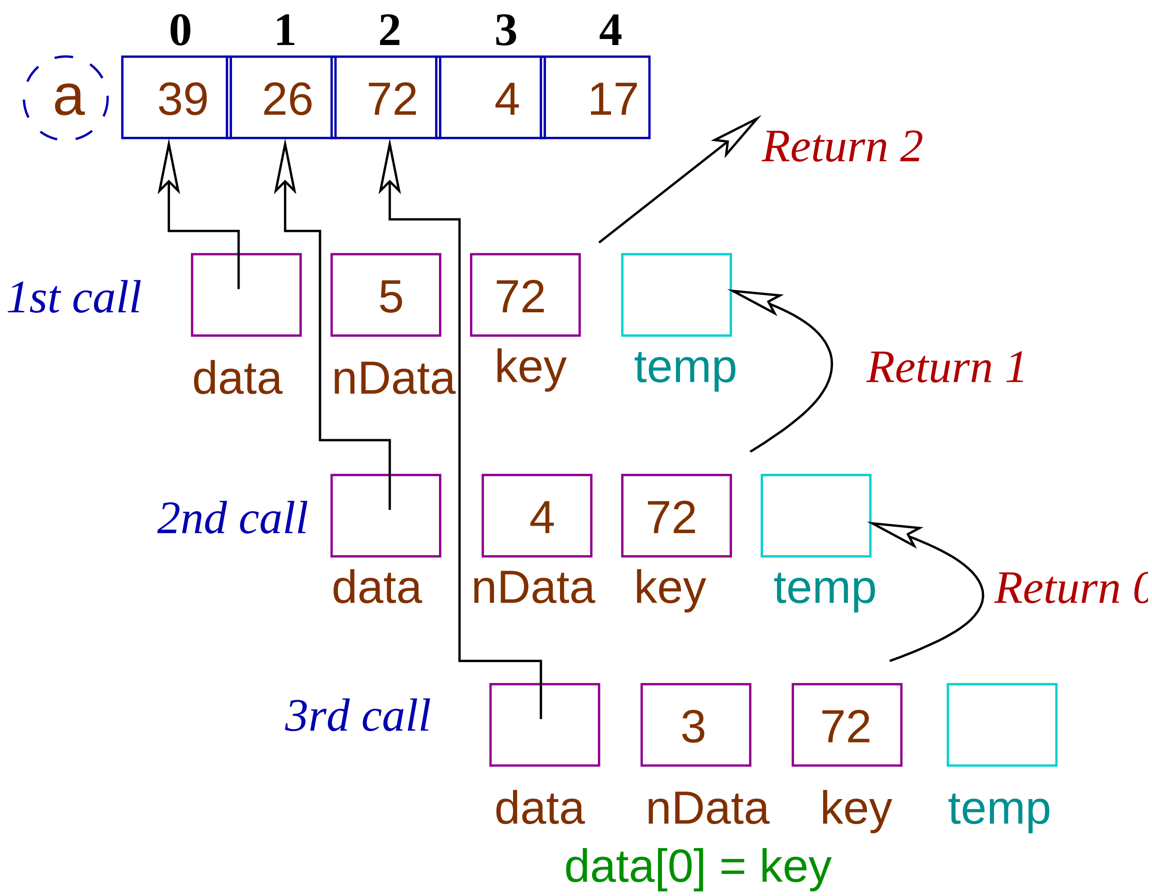
Running Time

In the best case the `key` may match with the 0^{th} element of the array and the for-loop is executed only once.

But in the worst case when either the match is with a data at the end or there is no match, the number of times the loop is executed is proportional to n, the number of data.

Recursive C Function

```
#define NOTFOUND -1
int seqSearch(int data[],int nData,int key){
    int temp ;
    if(data[0] == key) return 0 ;
    if(nData == 1) return NOTFOUND ;
    temp = seqSearch(data+1, nData-1, key) ;
    if(temp == NOTFOUND) return NOTFOUND ;
    return temp + 1 ;
} // seqSearchFR.c
```


0 1 2 3 4
a
39 26 72 4 17
Return 2
1st call
5
72
data nData key temp
Return 1
2nd call
4
72
data nData key temp
Return (
3rd call
3
72
data nData key temp
data[0] = key

Running Time

In the best case the `key` may match with the 0^{th} element of the array and there is only one call.

But in the worst case when either the match is with a data at the end or there is no match, the number of times the function is called is proportional to n, the number of data.

Extra Space Usage

The number of stack frames (activation records) used by the recursive function is also proportional to n, the space complexity is $O(n)$. Whereas the iterative function uses constant amount of extra space, the space complexity is $O(1)$.

Better Search

Search for a key can be made more efficient if the data stored in the array are sorted in some order. Let us assume that the data is stored in ascending order.

Binary Search: an Inductive Definition

$$\mathrm{bS}(a, l, h, k) = \begin{cases} l & l = h \;\&\; a[l] = k, \\ -1 & l = h \;\&\; a[l] \neq k, \\ bS(a, l, m, k) & l < h \;\&\; k \leq a[m] \\ bS(a, m+1, h, k) & l < h \;\&\; k > a[m] \end{cases}$$

where l is the low-index, h is the high index, k is the key to search and $m = \frac{l+h}{2}$.

Iterative C Function

```
int binarySearch(int data[], int l,
                 int h, int key) {
    while(l != h) {
          int m = (l+h)/2;
          if(key <= data[m]) h = m ;
          else l = m+1 ;
    }
    if(key == data[l]) return l ;
    return -1;
} // binarySearchF.c
```

Running Time

If there are n data, the while-loop is executed at most $\log_2 n + 1$ times. For a large value of n this gives a definite advantage over sequential search that executes the loop almost n times on a 'bad' data set. But then in case of binary search, sorted data is required.

Recursive C Function

```
int binarySearch(int data[], int l, int h, int key) {
    if(l == h) {
        if(data[l] == key) return l ;
        else return -1 ;
    }
    if(l < h) {
        int m = (l+h)/2 ;
        if(data[m] >= key)
            return binarySearch(data, l, m, key);
        else return binarySearch(data, m+1, h, key);
    }
} // binarySearchR.c
```

Running Time

Let there are n data. We assume that $n = 2^k$ for the ease of calculation. If t_n be the running time of recursive binary-search, then the inductive definition of t_n is

$$t_n = \begin{cases} c_0 & \text{if } n = 1 \\ t_{n/2} + c_1 & \text{if } n > 1. \end{cases}$$

The solution of this recurrence relation is $c_0 + kc_1$, so the running time of binary-search is proportional to $k = \log_2 n$.