Searching Elements in an Array:

Linear and Binary Search

Searching

- Check if a given element (called *key*) occurs in the array.
  - Example: array of student records; *rollno* can be the key.
- Two methods to be discussed:
  - If the array elements are unsorted.
    - Linear search
  - If the array elements are sorted.
    - Binary search

Linear Search

Basic Concept

- **Basic idea:**
  - Start at the beginning of the array.
  - Inspect elements one by one to see if it matches the key.
- **Time complexity:**
  - A measure of how long an algorithm takes to run.
  - If there are n elements in the array:
    - *Best case:* match found in first element (1 search operation)
    - *Worst case:* no match found, or match found in the last element (n search operations)
    - *Average case:* (n + 1) / 2 search operations

```
#include <stdio.h>

int linear_search (int a[], int size, int key)
{
   for (int i=0; i<size; i++)
     if  (a[i] == key)  return i;
   return -1;
}

int main()
{
   int x[]={12,-3,78,67,6,50,19,10}, val;
   printf ("\nEnter number to search: ");
     scanf ("%d", &val);
   printf ("\nValue returned: %d \n", linear_search (x,8,val);
}
```

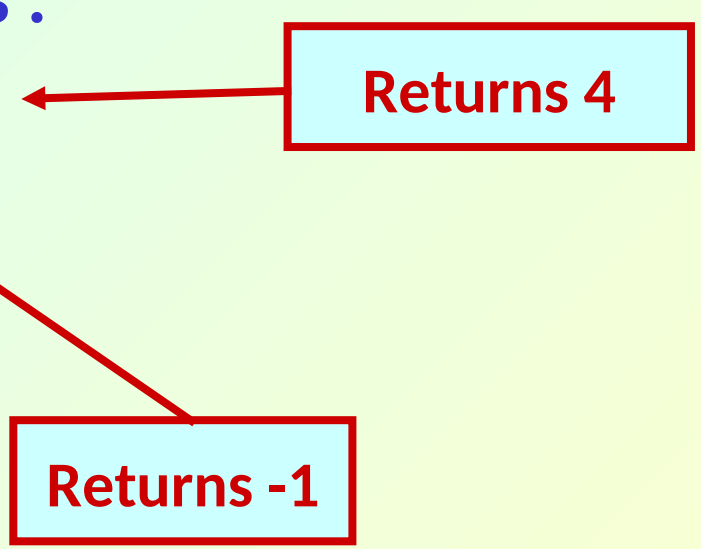
- What does the function `linear_search` do?
  - It searches the array for the number to be searched element by element.
  - If a match is found, it returns the array index.
  - If not found, it returns -1.

Contd.

```
int x[]= {12, -3, 78, 67, 6, 50, 19, 10};
```

- Trace the following calls :

```
search (x, 8, 6) ;
search (x, 8, 5) ;
```



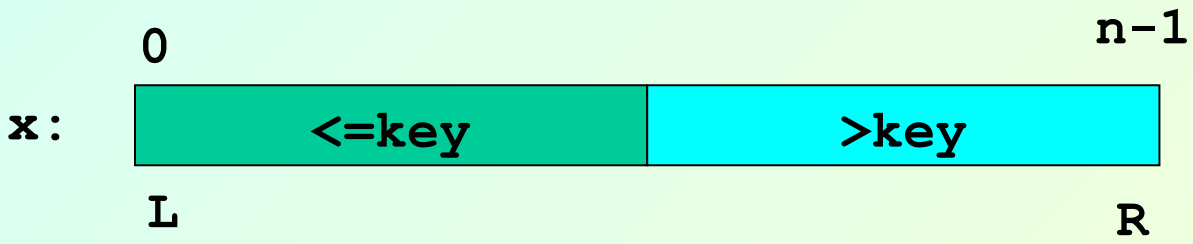
Binary Search

Basic Concept

- Binary search works if the array is *sorted*.
  - Look for the target in the middle.
  - If you don't find it, you can ignore half of the array, and repeat the process with the other half.
- In every step, we reduce the number of elements to search by half.

The Basic Strategy

- **What we want?**
  - Find split between values larger and smaller than *key*:



  - Situation while searching:
    - Initially L and R contains the indices of first and last elements.
  - Look at the element at index *[(L+R)/2]*.
    - Move L or R to the middle depending on the outcome of test.

Iterative Version

```
#include <stdio.h>

int bin_search (int a[], int size, int key)
{
   int  L, R, mid;
   L = 0;  R = size - 1;

   while  (L <= R)  {
     mid = (L + R) / 2;
     if  (a[mid] < key)  L = mid + 1;
     else if (a[mid] > key)  R = mid -1;
          else  return mid;   /* FOUND AT INDEX mid */
   }

  return  -1;     /* NOT FOUND */
}
```

```
int main()
{
   int x[]={10,20,30,40,50,60,70,80}, val;

   printf (”\nEnter number to search: ”);
     scanf (”%d”, &val);

   printf (”\nValue returned: %d \n”, bin_search (x,8,val);
}
```

Recursive Version

```
#include <stdio.h>

int bin_search (int a[], int L, int R, int key)
{
   int  mid;

   if  (R < L)  return -1;     /* NOT FOUND */
   mid = (L + R) / 2;
   if  (a[mid] < key)  return (bin_search(a,mid+1,R,key));
   else if (a[mid] > key) return (bin_search(a,L,mid-1,key));
        else  return mid;   /* FOUND AT INDEX mid */
}
```

```
int main()
{
   int x[]={10,20,30,40,50,60,70,80}, val;

   printf ("\nEnter number to search: ");
     scanf ("%d", &val);

   printf ("\nValue returned: %d \n", bin_search (x,0,7,val);
}
```

Is it worth the trouble ?

- Suppose that the array `x` has 1000 elements.
- Ordinary search
  - If *key* is present in *x*, it would require 500 comparisons on the average.
- Binary search
  - After 1st compare, left with 500 elements.
  - After 2nd compare, left with 250 elements.
  - After 3rd compare, left with 125 elements.
  - After at most 10 steps, you are done.

Time Complexity

- If there are n elements in the array.
  - Number of searches required in the worst case: $\log_2 n$
- For n = 64 (say).
  - Initially, list size = 64.
  - After first compare, list size = 32.
  - After second compare, list size = 16.
  - After third compare, list size = 8.
  - …….
  - After sixth compare, list size = 1.

$2^k = n$, where k is the number of steps.

$k = \log_2 n$

$\log_2 64 = 6$

$\log_2 1024 = 10$

Sorting

The Basic Problem

- Given an array

  `x[0], x[1], ... , x[size-1]`

- reorder entries so that

  `x[0]` ≤ `x[1]` ≤ `...` ≤ `x[size-1]`

  - List is in non-decreasing order.

- We can also sort a list of elements in non-increasing order.

Example

- **Original list:**

  `10, 30, 20, 80, 70, 10, 60, 40, 70`

- **Sorted in non-decreasing order:**

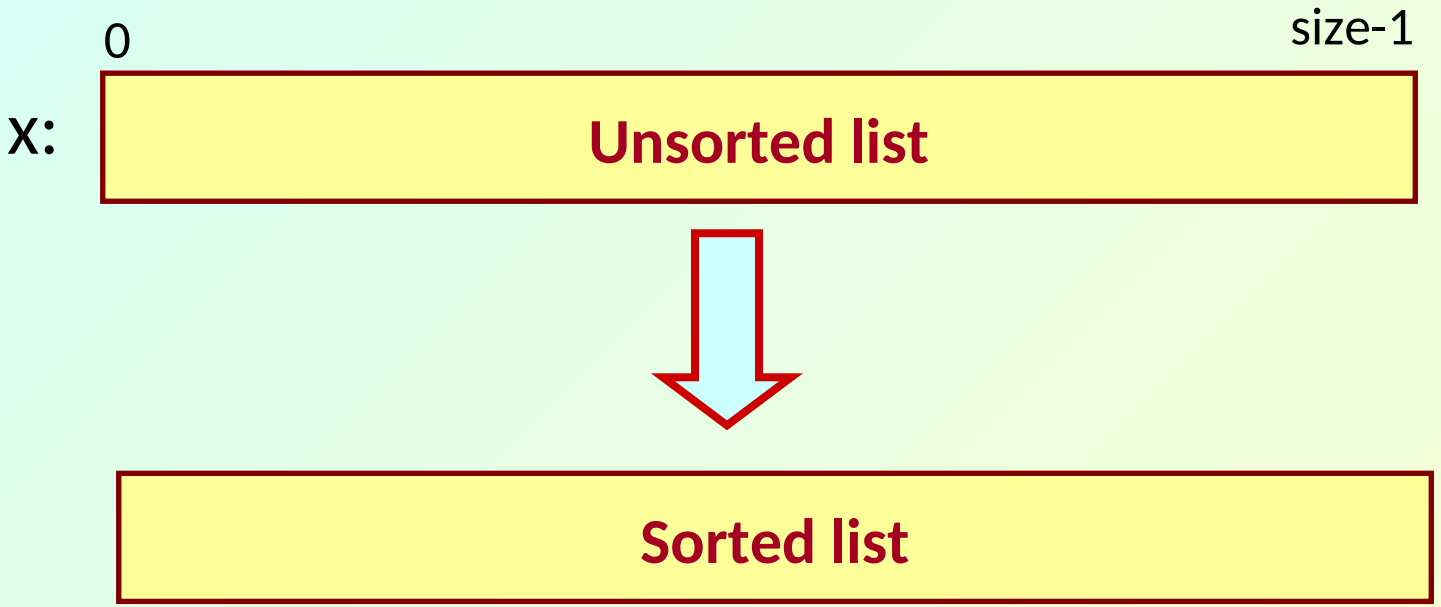
  `10, 10, 20, 30, 40, 60, 70, 70, 80`

- **Sorted in non-increasing order:**

  `80, 70, 70, 60, 40, 30, 20, 10, 10`

Sorting Problem

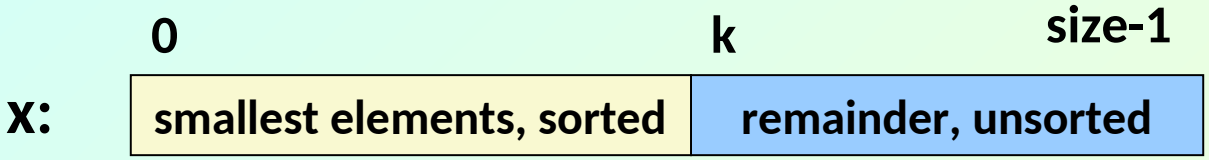
- **What we want ?**
  - **Sort the given data sorted in the specified order**



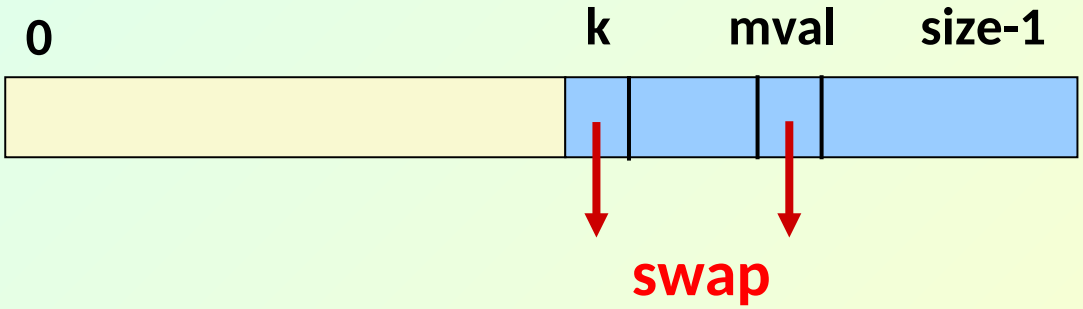
Selection Sort

How it works?

- General situation :



- Steps :
  - Find smallest element, `mval`, in `x[k..size-1]`
  - Swap smallest element with `x[k]`, then increase `k` by 1



An example worked out

PASS 1:

10	5	17	11	-3	12	Find the minimum
10	5	17	11	-3	12	Exchange with 0th
-3	5	17	11	10	12	element

PASS 2:

-3	5	17	11	10	12	Find the minumum
-3	5	17	11	10	12	Exchange with 1st
-3	5	17	11	10	12	element

PASS 3:

-3	5	17	11	10	12	Find the minumum
-3	5	17	11	10	12	Exchange with 2nd
-3	5	10	11	17	12	element

PASS 4:

-3 5 10 11 17 12	Find the minumum
-3 5 10 11 17 12	Exchange with 3rd
-3 5 10 11 17 12	element

PASS 5:

-3 5 10 11 17 12	Find the minumum
-3 5 10 11 17 12	Exchange with 4th
-3 5 10 11 12 17	element

Subproblem

```
/* Yield index of smallest element in x[k..size-1];*/

int min_loc (int x[], int k, int size)
{
    int j, pos;

    pos = k;
    for (j=k+1; j<size; j++)
        if (x[j] < x[pos])
            pos = j;
    return pos;
}
```

The main sorting function

```
/* Sort x[0..size-1] in non-decreasing order */

int sel_sort (int x[], int size)
{  int k, m;

   for (k=0; k<size-1; k++)
   {
      m = min_loc (x, k, size);
      temp = a[k];
      a[k] = a[m];
      a[m] = temp;
   }
}
```

SWAP

```
int main()
{
  int x[ ]={-45,89,-65,87,0,3,-23,19,56,21,76,-50};
  int i;
  for(i=0;i<12;i++)
      printf("%d ",x[i]);
  printf("\n");
  sel_sort(x,12);
  for(i=0;i<12;i++)
      printf("%d ",x[i]);
  printf("\n");
}
```

```
-45 89 -65 87 0 3 -23 19 56 21 76 -50
-65 -50 -45 -23 0 3 19 21 56 76 87 89
```

Analysis

- How many steps are needed to sort n items ?
  - Total number of steps proportional to n^2.
  - No. of comparisons?

    $(n-1) + (n-2) + \ldots + 1 = n(n-1)/2$

    Of the order of n^2
  - Worst Case? Best Case? Average Case?

Insertion Sort

Basic Idea

- Insert elements one at a time, and create a partial sorted list.
  - Sorted list of 2 elements, 3 elements, 4 elements, and so on.
- In general, in every iteration an element is compared with all the elements before it.
- After finding the position of insertion, space is created for it by shifting the other elements and the desired element is then inserted at the suitable position.
- This procedure is repeated for all the elements in the list.

An example worked out

PASS 1:

```
10  5  17  11  -3  12      item = 5
?  10  17  11  -3  12        compare 5 and 10
5  10  17  11  -3  12        insert 5
```

PASS 2:

```
5  10  17  11  -3  12      item = 17
5  10  17  11  -3  12        compare 17 and 10
```

PASS 3:

```
5  10  17  11  -3  12      item = 11
5  10  ?   17  -3  12        compare 11 and 17
5  10  ?   17  -3  12        compare 11 and 10
5  10  11  17  -3  12        insert 11
```

PASS 4:

5 10 11 17 -3 12	item = -3
5 10 11 ? 17 12	compare -3 and 17
5 10 ? 11 17 12	compare -3 and 11
5 ? 10 11 17 12	compare -3 and 10
? 5 10 11 17 12	compare -3 and 5
-3 5 10 11 17 12	insert -3

PASS 5:

-3 5 10 11 17 12	item = 12
-3 5 10 11 ? 17	compare 12 and 17
-3 5 10 11 ? 17	compare 12 and 11
-3 5 10 11 12 17	insert 12

Insertion Sort

```
void insert_sort (int list[], int size)
{
  int i,j,item;

  for (i=1; i<size; i++)
     {
      item = list[i] ;
      j = i - 1;
      while  ((item < list[j]) && (j >= 0))
      {
         list[j+1] = list[j];
         j--;
      }
      list[j+1] = item ;
  }
}
```

```
int main()
{
  int x[ ]={-45,89,-65,87,0,3,-23,19,56,21,76,-50};
  int i;
  for(i=0;i<12;i++)
      printf("%d ",x[i]);
  printf("\n");
  insert_sort (x,12);
  for(i=0;i<12;i++)
      printf("%d ",x[i]);
  printf("\n");
}
```

```
-45 89 -65 87 0 3 -23 19 56 21 76 -50
-65 -50 -45 -23 0 3 19 21 56 76 87 89
```

Time Complexity

- Number of comparisons and shifting:
  - Worst case?

    $1 + 2 + 3 + \ldots\ldots + (n-1) = n(n-1)/2$

  - Best case?

    $1 + 1 + \ldots\ldots$ to $(n-1)$ terms $= (n-1)$

Bubble Sort

How it works?

- The sorting process proceeds in several passes.
  - In every pass we go on comparing neighboring pairs, and swap them if out of order.
  - In every pass, the largest of the elements under considering will *bubble* to the top (i.e., the right).

```
10   5   17   11   -3   12
5   10   17   11   -3   12
5   10   17   11   -3   12
5   10   11   17   -3   12
5   10   11   -3   17   12
5   10   11   -3   12   17
```

An example worked out

PASS 1:

```
10  5  17  11  -3  12
5  10  17  11  -3  12
5  10  17  11  -3  12
5  10  11  17  -3  12
5  10  11  -3  17  12
5  10  11  -3  12  17
```

PASS 2:

```
5  10  11  -3  12  17
5  10  11  -3  12  17
5  10  11  -3  12  17
5  10  -3  11  12  17
5  10  -3  11  12  17
```

PASS 3:

5 10 -3 11 12 17

5 10 -3 11 12 17

5 -3 10 11 12 17

5 -3 10 11 12 17

PASS 4:

5 -3 10 11 12 17

-3 5 10 11 12 17

-3 5 10 11 12 17

PASS 5:

-3 5 10 11 12 17

-3 5 10 11 12 17

- How the passes proceed?
  - In pass 1, we consider index 0 to n-1.
  - In pass 2, we consider index 0 to n-2.
  - In pass 3, we consider index 0 to n-3.
  - ……
  - ……
  - In pass n-1, we consider index 0 to 1.

Bubble Sort

```
void swap(int *x, int *y)
{
  int tmp = *x;
  *x = *y;
  *y = tmp;
}
```

```
void bubble_sort
          (int x[], int n)
{
  int i,j;

  for (i=n-1; i>0; i--)
    for (j=0; j<i; j++)
      if (x[j] > x[j+1])
        swap(&x[j],&x[j+1]);
}
```

```
int main()
{
  int x[ ]={-45,89,-65,87,0,3,-23,19,56,21,76,-50};
  int i;
  for(i=0;i<12;i++)
      printf("%d ",x[i]);
  printf("\n");
  bubble_sort (x,12);
  for(i=0;i<12;i++)
      printf("%d ",x[i]);
  printf("\n");
}
```

```
-45 89 -65 87 0 3 -23 19 56 21 76 -50
-65 -50 -45 -23 0 3 19 21 56 76 87 89
```

Time Complexity

- Number of comparisons :
  - Worst case?

    $1 + 2 + 3 + \ldots\ldots + (n-1) = n(n-1)/2$

  - Best case?
    - Same

- How do you make best case with `(n-1)` comparisons only?
  - By maintaining a variable `flag`, to check if there has been any swaps in a given pass.
  - If no swaps during a pass, the array is already sorted.

```
void bubble_sort (int x[], int n)
{
  int i, j, flag;

  for (i=n-1; i>0; i=flag)
  {
      flag = 0;
      for (j=0; j<i; j++)
         if (x[j] > x[j+1])
         {
            swap(&x[j],&x[j+1]);
            flag = j;
         }
  }
```

Some Efficient Sorting Algorithms

- Two of the most popular sorting algorithms are based on divide-and-conquer approach.
  - Quick sort
  - Merge sort
- Basic concept (divide-and-conquer method):

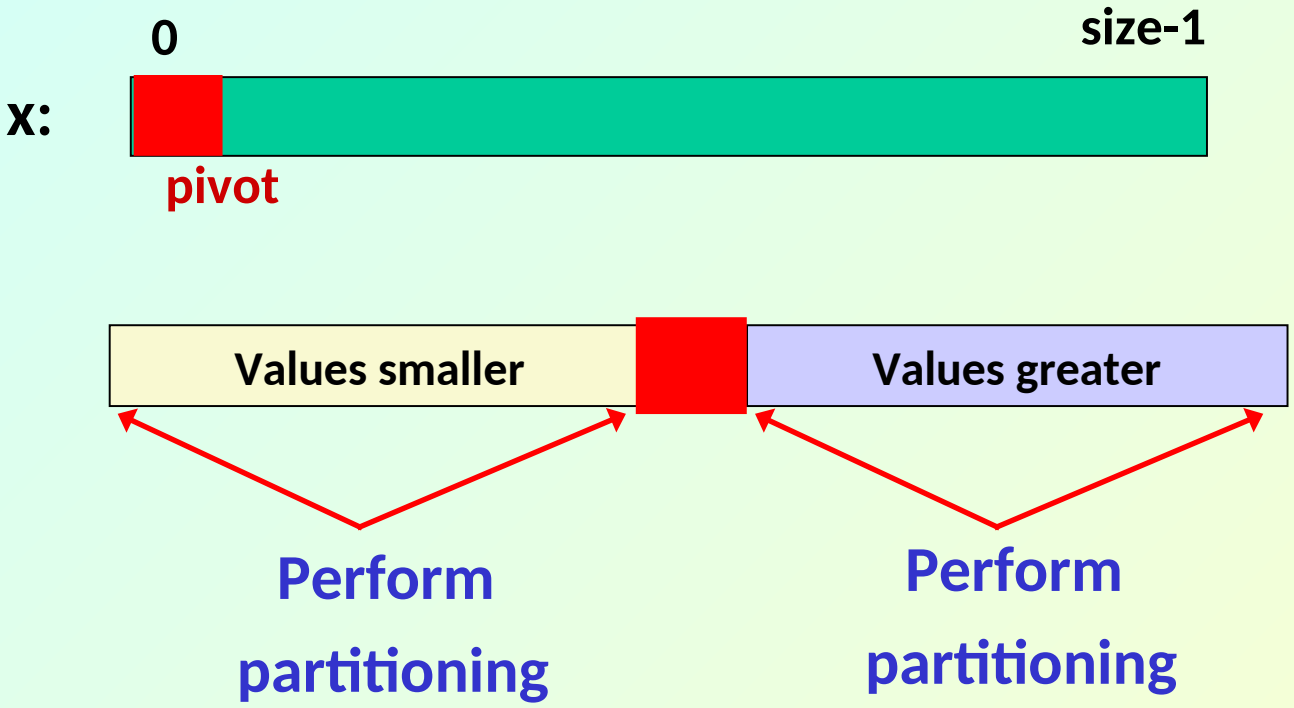
```
sort (list)
{
   if the list has greater than 1 elements
   {
      Partition the list into lowlist and highlist;
      sort (lowlist);
      sort (highlist);
      combine (lowlist, highlist);
   }
}
```

Quick Sort

How it works?

- At every step, we select a *pivot element* in the list (usually the *first* element).
  - We put the pivot element in the final position of the sorted list.
  - All the elements less than or equal to the pivot element are to the left.
  - All the elements greater than the pivot element are to the right.

Partitioning



```
void print (int x[], int low, int high)
{
   int i;

   for(i=low; i<=high; i++)
      printf(" %d", x[i]);
   printf("\n");
}
```

```
void swap (int *a, int *b)
{
  int tmp=*a;
  *a=*b;
  *b=tmp;
}
```

```
void partition (int x[], int low, int high)
{
   int i = low+1,  j = high;
   int pivot = x[low];
   if (low >= high) return;
   while (i<j)  {
      while ((x[i]<pivot) && (i<high))  i++;
      while ((x[j]>=pivot) && (j>low))  j--;
      if (i<j)  swap (&x[i], &x[j]);
   }
   if (j==high)  {
      swap (&x[j], &x[low]);
      partition (x, low, high-1);
   }
   else
      if (i==low+1)
         partition (x, low+1, high);
      else {
            swap (&x[j], &x[low]);
            partition (x, low, j-1);
            partition (x, j+1, high);
          }
}
```

```
int main (int argc, char *argv[])
{
   int x[] = {-56,23,43,-5,-3,0,123,-35,87,56,75,80};
   int i=0;
   int num;

   num = 12;   /* Number of elements */


   partition(x,0,num-1);

   printf("Sorted list: ");
   print (x,0,num-1);
}
```

Trace of Partitioning: an example

45 -56 78 90 -3 -6 123 0 -3 45 69 68

45 -56 78 90 -3 -6 123 0 -3 45 69 68

-6 -56 -3 0 -3 45 123 90 78 45 69 68

-56 -6 -3 0 -3 68 90 78 45 69 123

-3 0 -3 45 68 78 90 69

-3 0 69 78 90

Sorted list: -56 -6 -3 -3 0 45 45 68 69 78 90 123

Time Complexity

- **Worst case:**

  n^2 ==> list is already sorted

- **Average case:**

  $n \log_2 n$

- **Statistically, quick sort has been found to be one of the fastest algorithms.**