Internal Sort by Comparison II

Merge Sort Algorithm

This internal sorting algorithm by comparison was invented by John von Neumann in 1945 (Wikipedia). Its running time is $O(n \log n)$, better than the worst case running times of selection, insertion or bubble sort algorithms. This is stable i.e. the order of the equal input elements are preserved after sorting. It uses divide and conquer algorithmic strategy. It uses $O(n)$ extra space.

Merge Sort Algorithm (Ascending Order)

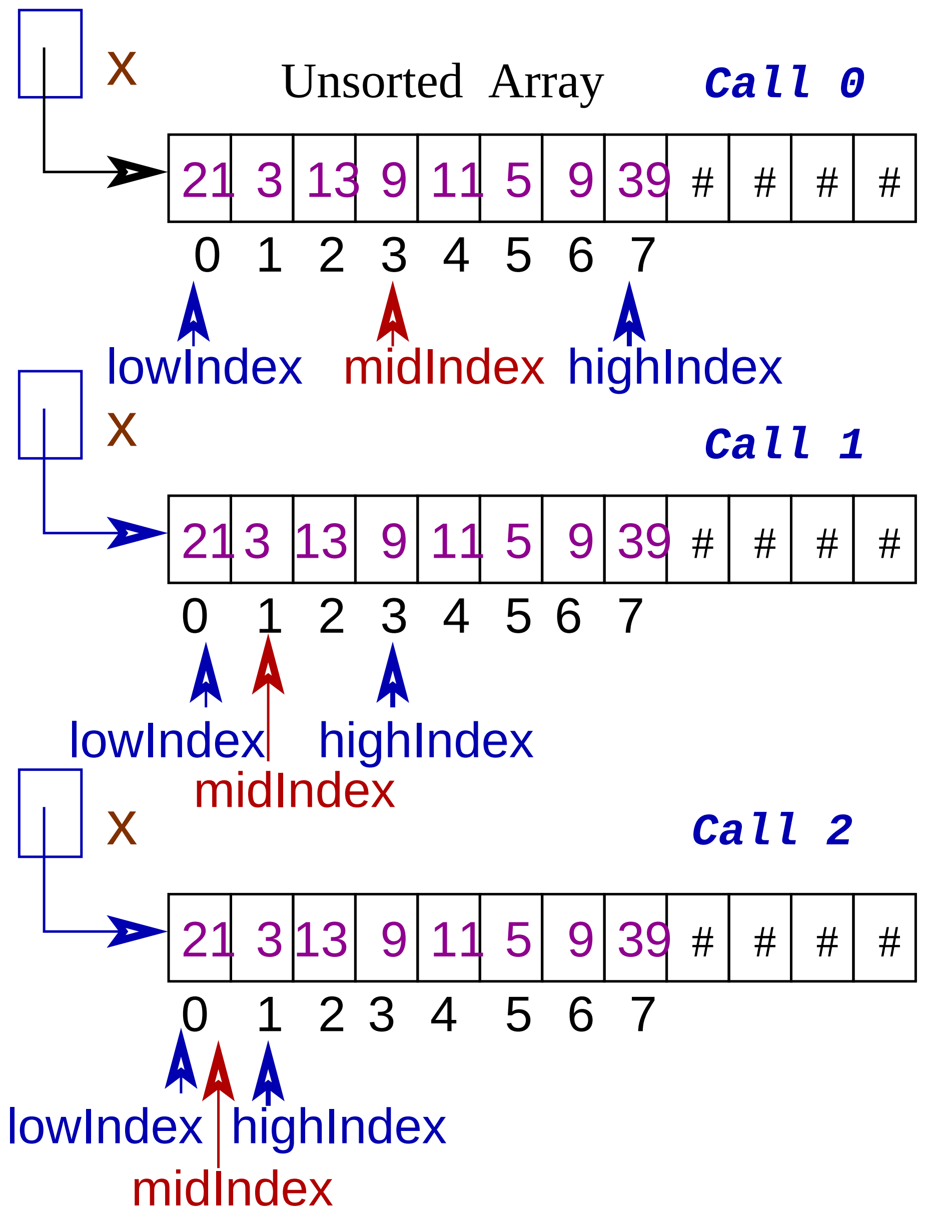
Let there there be n data elements stored in an 1-D array.

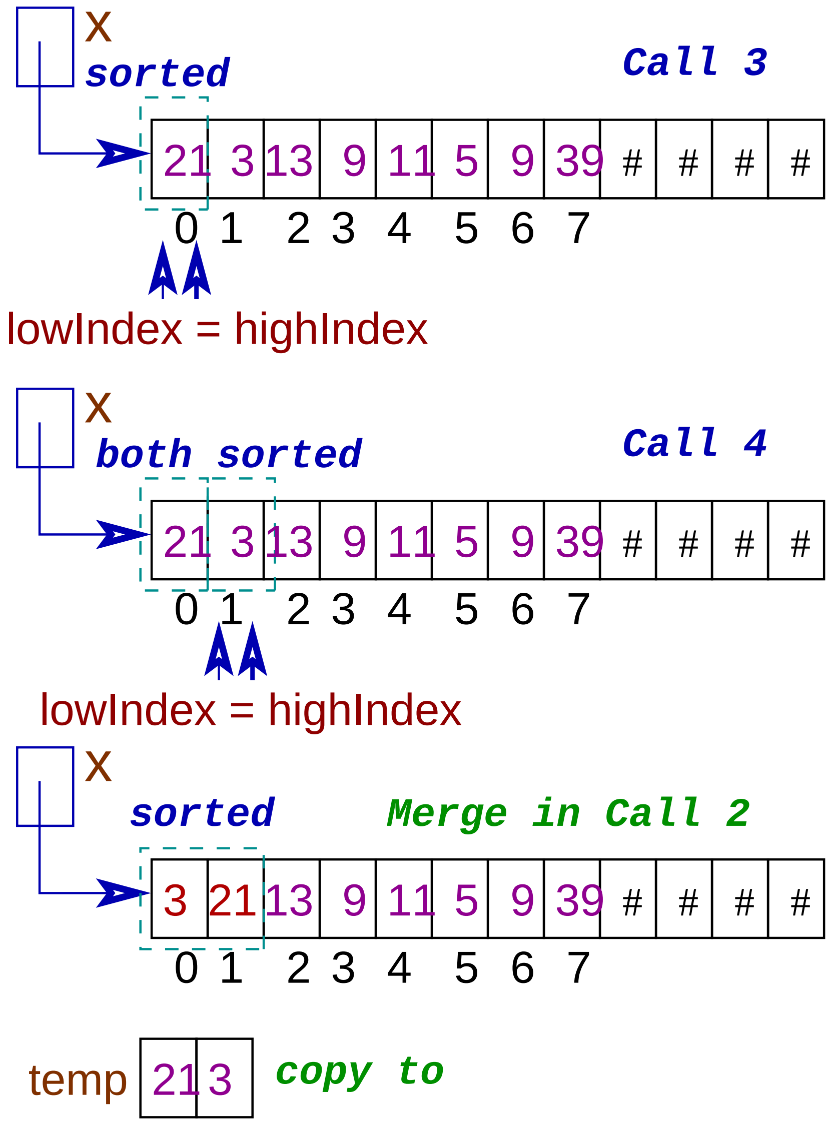
1. If $n = 1$, it is already sorted!
2. If $n > 1$,
   (a) Split the data set in the middle i.e. find out the $\texttt{mid} = \frac{\texttt{low+high}}{2}$.
   (b) Merge Sort recursively the lower half of the data (within `low` and `mid`).

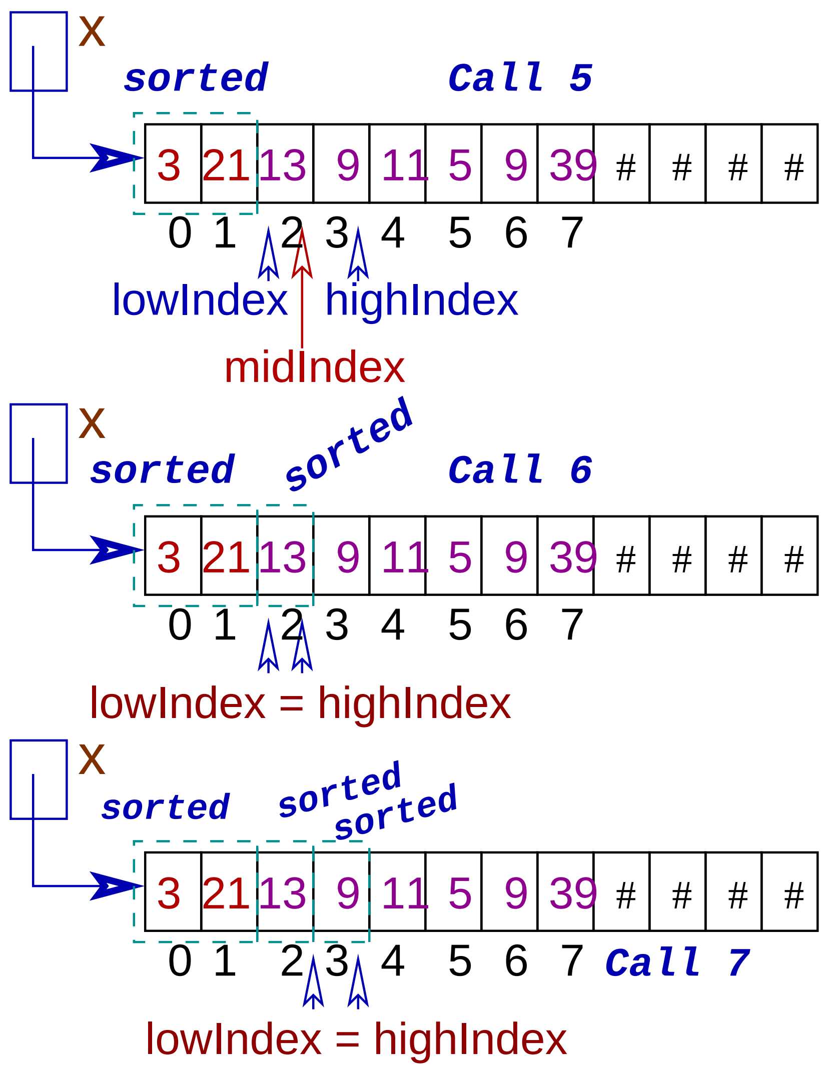
(c) Merge Sort recursively the upper half of the data (within `mid + 1` and `high`).

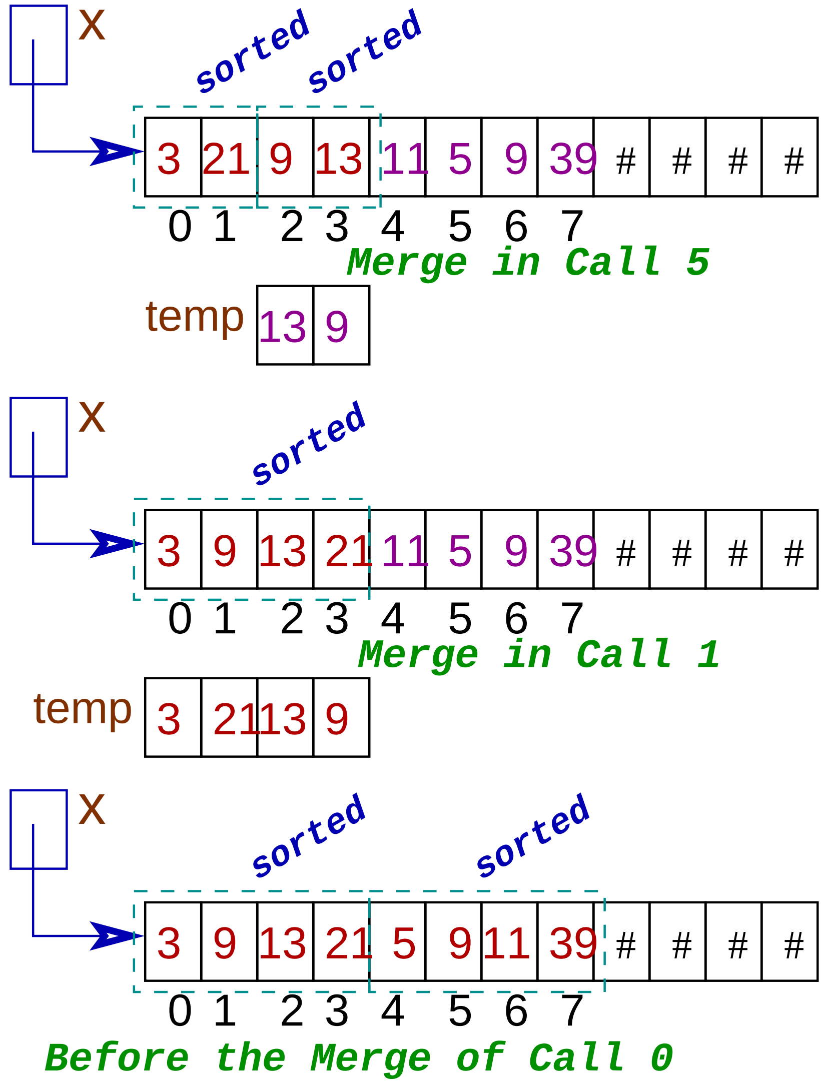
(d) Copy both the sorted halves in a second array and merge them in proper order in the original array.

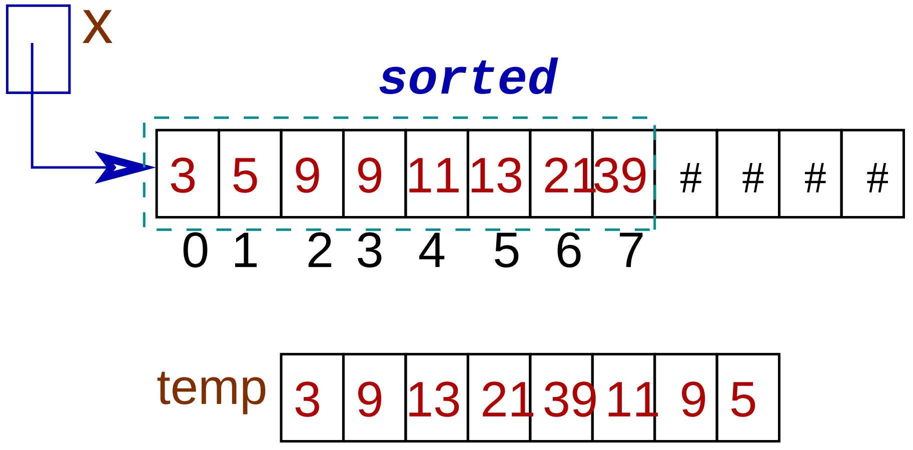
Merge Sort : An Example


x
Unsorted Array
Call 0
21 3 13 9 11 5 9 39 # # # #
0 1 2 3 4 5 6 7
lowIndex
midIndex
highIndex
x
Call 1
21 3 13 9 11 5 9 39 # # # #
0 1 2 3 4 5 6 7
lowIndex
highIndex
midIndex
x
Call 2
21 3 13 9 11 5 9 39 # # # #
0 1 2 3 4 5 6 7
lowIndex
highIndex
midIndex


x
sorted
Call 3
21 3 13 9 11 5 9 39 # # # #
0 1 2 3 4 5 6 7
lowIndex = highIndex
x
both sorted
Call 4
21 3 13 9 11 5 9 39 # # # #
0 1 2 3 4 5 6 7
lowIndex = highIndex
x
sorted
Merge in Call 2
3 21 13 9 11 5 9 39 # # # #
0 1 2 3 4 5 6 7
temp 21 3
copy to


x
sorted
Call 5
3 21 13 9 11 5 9 39 # # # #
0 1 2 3 4 5 6 7
lowIndex
highIndex
midIndex
x
sorted
sorted
Call 6
3 21 13 9 11 5 9 39 # # # #
0 1 2 3 4 5 6 7
lowIndex = highIndex
x
sorted
sorted
sorted
3 21 13 9 11 5 9 39 # # # #
0 1 2 3 4 5 6 7
Call 7
lowIndex = highIndex


X
sorted
sorted
3 21 9 13 11 5 9 39 # # # #
0 1 2 3 4 5 6 7
Merge in Call 5
temp 13 9
X
sorted
3 9 13 21 11 5 9 39 # # # #
0 1 2 3 4 5 6 7
Merge in Call 1
temp 3 21 13 9
X
sorted
sorted
3 9 13 21 5 9 11 39 # # # #
0 1 2 3 4 5 6 7
Before the Merge of Call 0



After the Merge of Call 0

C Code

```
#include <stdio.h>
#define MAXNO 100
void mergeSort(int [], int, int);
int main() // mergeSort.c
{
    int n=0, i, data[MAXNO] ;

    printf("Enter the data, terminate by Ctrl+D: ") ;
    while(scanf("%d", &data[n]) != EOF) ++n ;
    printf("\nInpute data: ") ;
    for(i=0; i<n; ++i) printf("%d ", data[i]) ;
    putchar('\n') ;
    mergeSort(data, 0, n-1) ;
    printf("Data in ascending order: ") ;
    for(i=0; i<n; ++i) printf("%d ", data[i]);
    putchar('\n');
    return 0;
}
void merge(int data[], int l, int m, int h)
{
     int temp[MAXNO], i, j, k ;

     for(i = l; i <= m; ++i) temp[i] = data[i] ;
     for(i = h, j = m+1; i > m; --i, ++j)
                         temp[i] = data[j] ;
     for(i = l, k = l, j = h; i <= j; ++k)
         if(temp[i] <= temp[j]) data[k] = temp[i++] ;
         else data[k] = temp[j--] ;
}
void mergeSort(int x[],int low,int high)
{
     if(low != high) {
        int mid = (low + high)/2 ;

     mergeSort(x, low, mid) ;
     mergeSort(x, mid+1, high) ;
     merge(x, low, mid, high) ;
 }
} // mergeSort.c
```

Analysis of Merge Sort

Let $t(n)$ be the time taken to merge sort n data. The running time $t(n)$ can be expressed by the following recurrence relation.

$$t(n) = \begin{cases} c_0, & \text{if } n = 1, \\ t(\lfloor \frac{n}{2} \rfloor) + t(\lceil \frac{n}{2} \rceil) + cn, & \text{otherwise,} \end{cases}$$

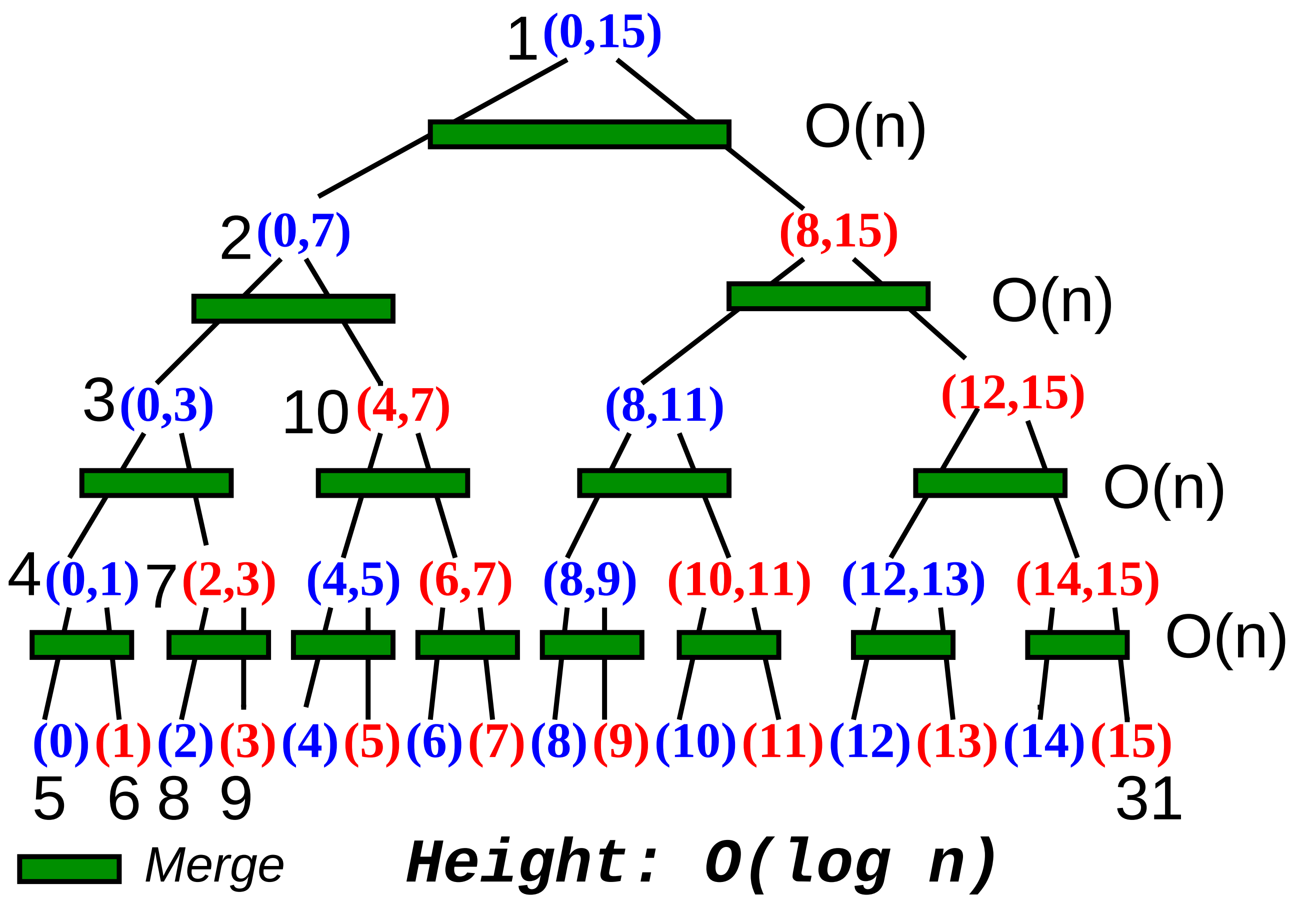
where cn is the time to merge n data.

Solution for t(n)

$$
\begin{aligned}
t(n) &= 2t\left(\frac{n}{2}\right) + cn, \ //n = 2^k \\
&= 2\left(2t\left(\frac{n}{4}\right) + c\frac{n}{2}\right) + cn \\
&= 2^2 t\left(\frac{n}{4}\right) + cn + cn \\
& \quad \ldots \\
&= 2^{\log_2 n} c_0 + \overbrace{\mathbf{cn} + \cdots + \mathbf{cn}}^{\log_2 n} \\
&= c_0 n + \overbrace{\mathbf{cn} + \cdots + \mathbf{cn}}^{\log_2 n} = \Theta(n \log n)
\end{aligned}
$$

$$\boxed{\text{Running time of merge sort is } \Theta(n \log n)}$$

Call Sequence & Indices : Merge Sort



Extra Space

- An extra array of size at most n is required to merge the data.
- The depth of recursive call is $O(\log n)$.
- So, $O(n) + O(\log n) = O(n)$ extra space is required.

Quick Sort Algorithm

This internal sorting algorithm by comparison was designed by C. A. R. Hoare. Its running time is $O(n^2)$, but on the average it is $O(n \log n)$. In actual implementation its performance is better than merge sort. But it is not a stable algorithm.

Quick Sort Algorithm (ascending order)

Let there there be n data elements stored in an 1-D array, `a[l` $\cdots$ `h]`.

1. If $l = h(n = 1)$, it is already sorted!
2. If $n > 1$,
   (a) Take a *pivot* element (may be `a[l]`, but not necessarily).
   (b) Partition the data in three parts: *left part*, *mid-data* and *right part*.

Pivot element is the *mid-data*. No data in the *left part* has a *key* larger than the *mid-data* and No data in the *right part* has a *key* smaller than the *mid-data*.

(c) Sort *left* and *right* parts recursively.

Note

After the partition there will be no data movement from the *left part* to the *right part*.

Partitioning the Data

1. Select a data element p as the pivot element. For a simple implementation we may choose the `a[l]` as the pivot.
2. Transfer all the elements that are less than or equal to p, to the lower index side of the array and also transfer elements greater than or equal to p, to the higher index side.

Partition Algorithm

```
partition(a, l, h)
p $\leftarrow$ a[l]
i $\leftarrow$ l
j $\leftarrow$ h + 1
while TRUE do
    do i $\leftarrow$ i + 1 while i $<$ h + 1 and a[i] $<$ p
    do j $\leftarrow$ j $-$ 1 while a[j] $>$ p
    if (i $<$ j) a[i] $\leftrightarrow$ a[j]
    else return j
endWhile
```

Quick Sort Algorithm

```
quickSort(a, l, h)
if (l < h)
    p ← partition(a, l, h)
    a[l] ↔ a[p]
    quicksort(a, l, p-1)
    quicksort(a, p+1, h)
```

Partition Algorithm

Variation of the partition algorithm (Cormen et al):

```
partition(a, l, h)
p ← a[l]
i ← l − 1
j ← h + 1
while TRUE do
    do j ← j − 1 while a[j] > p
    do i ← i + 1 while a[i] < p
    if (i < j) a[i] ↔ a[j]
    else return j
endWhile
```

Note

In this case the data is partitioned into two parts. Data in the *left part* are smaller than the pivot and the data in the *right part* are larger than or equal to the pivot.

Quick Sort Algorithm

```
quickSort(a, l, h)
if (l < h)
    p ← partition(a, l, h)
    quicksort(a, l, p)
    quicksort(a, p+1, h)
```

Another Partition Algorithm

```
partition(a, l, h)
p ← a[h]
pI ← l
for i = l to h − 1
    if a[i] < p
        a[i] ↔ a[pI]
        pI ← pI + 1
endFor
a[pI] ↔ a[h]
return pI // from Wikipedia
```

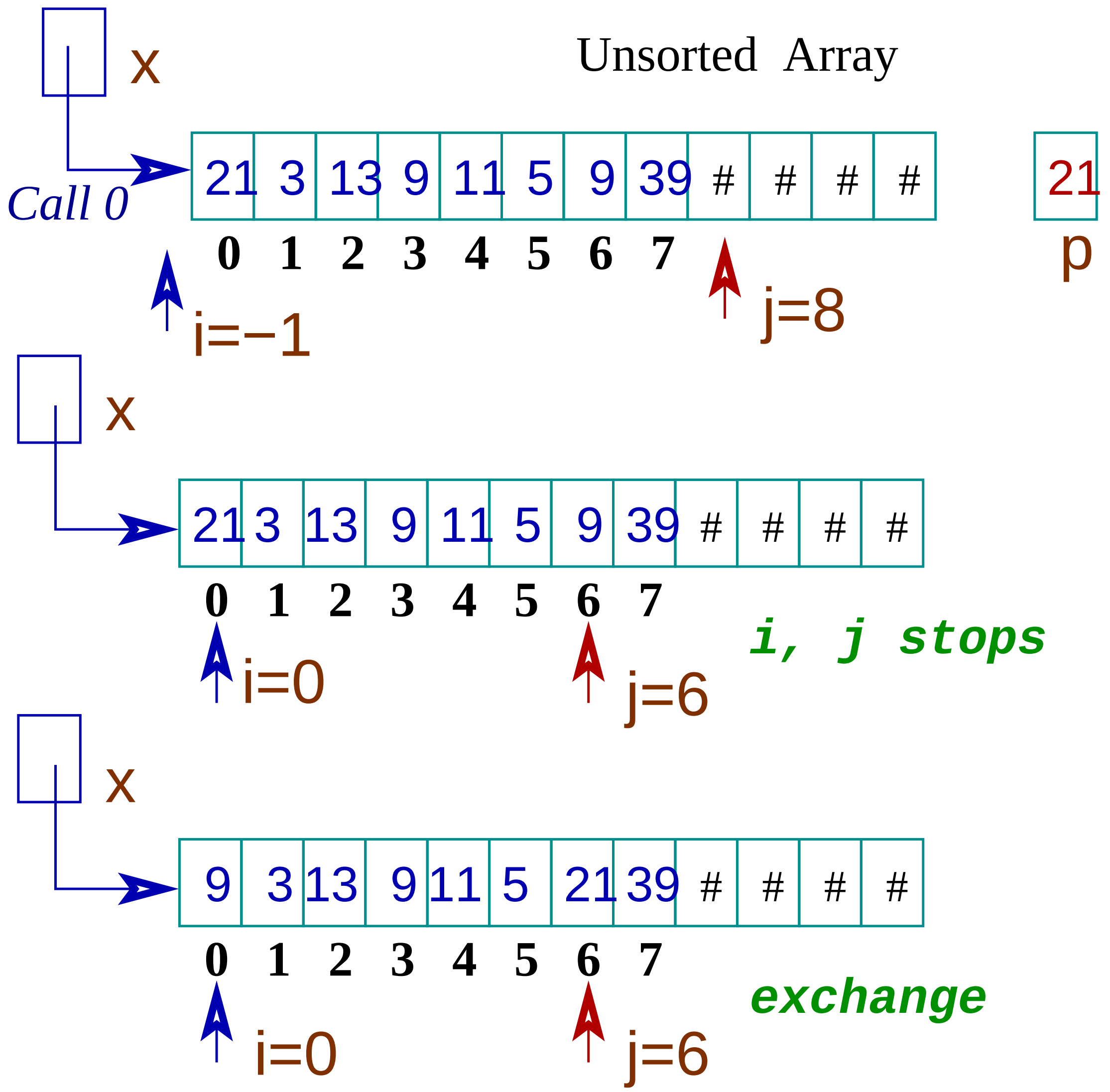
Note

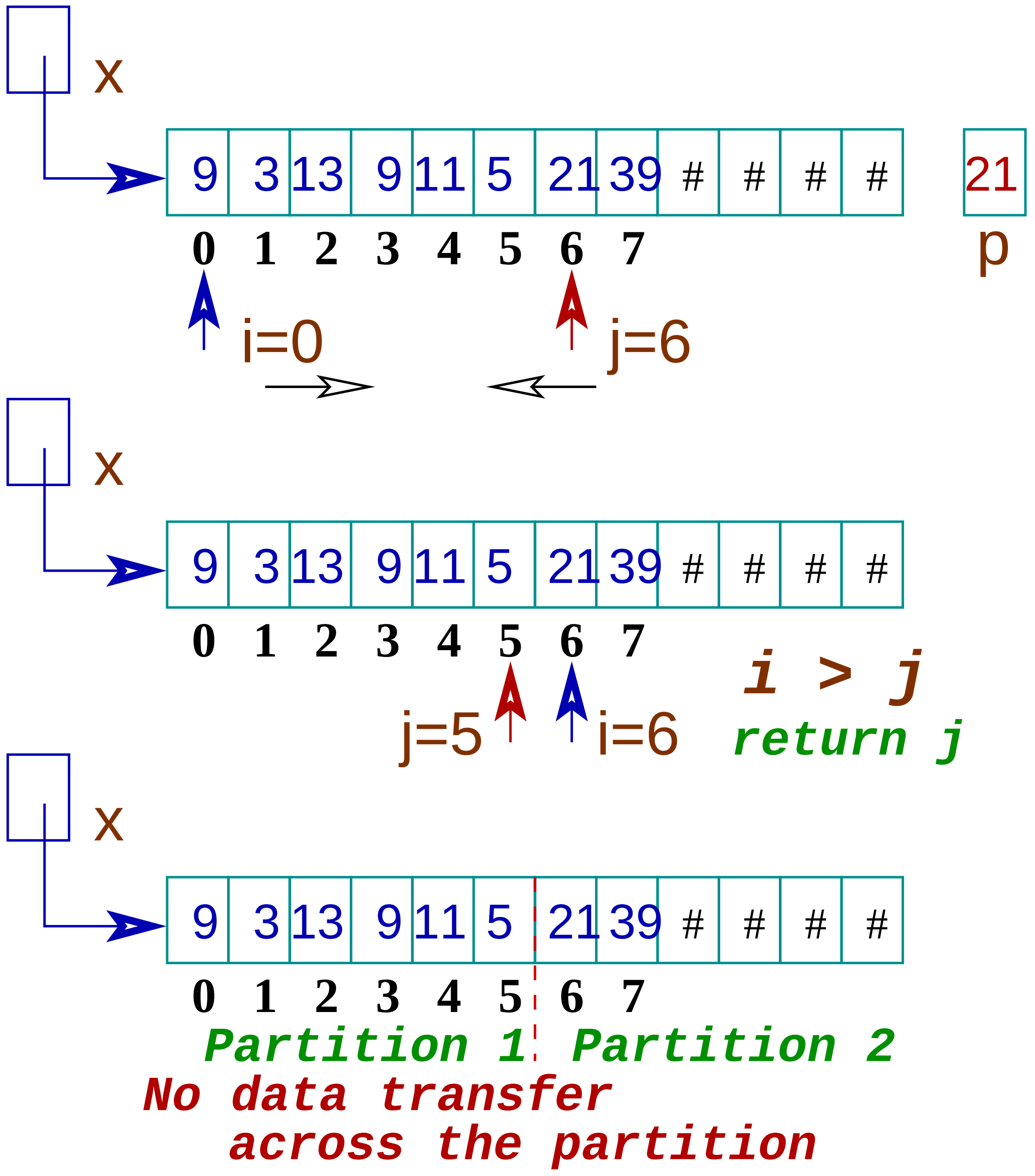
All data to the left of the `pI`th index are less than the value of the pivot element. Finally `pI` is the partition index where we place the pivot element.

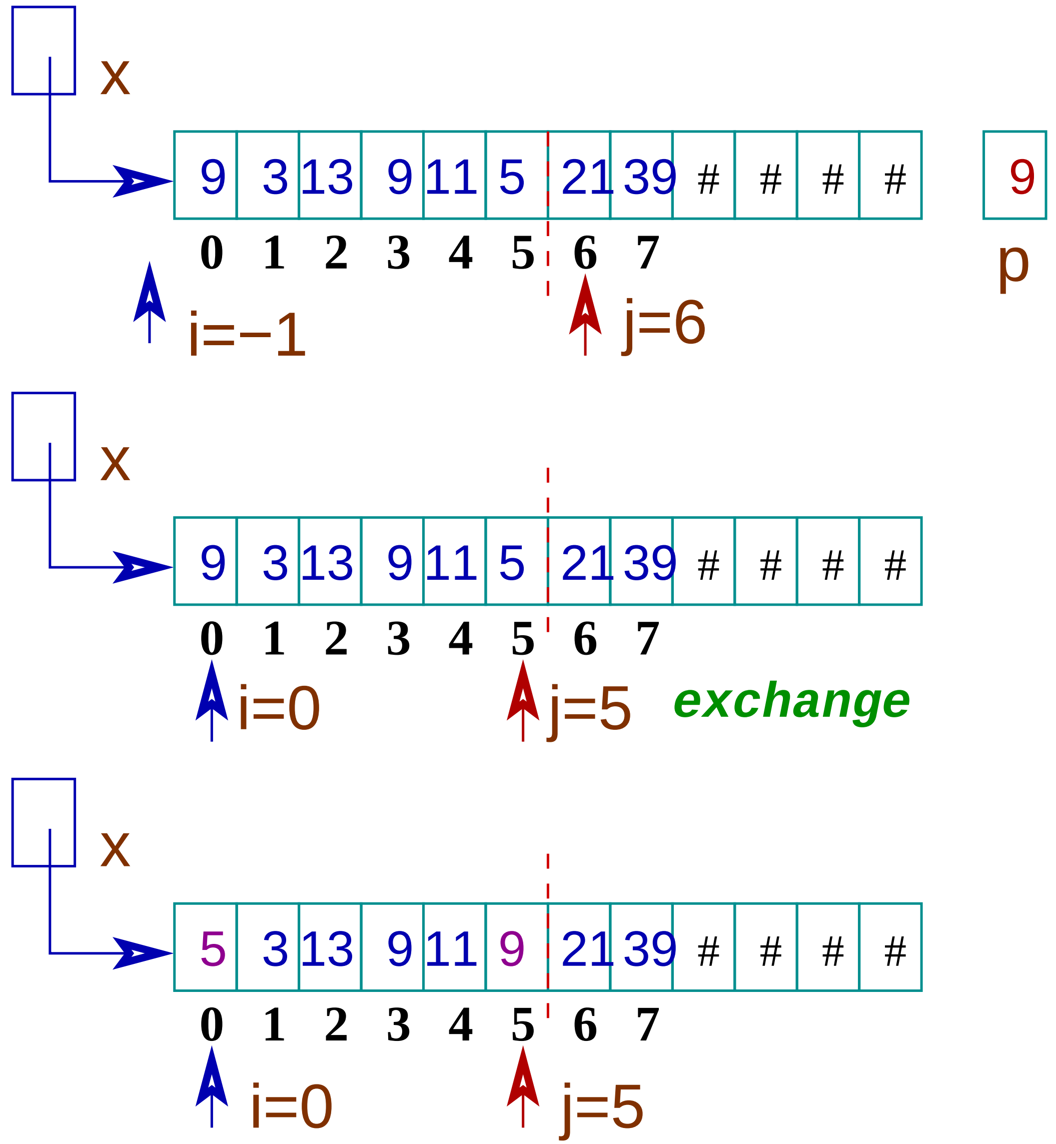
Quick Sort Algorithm

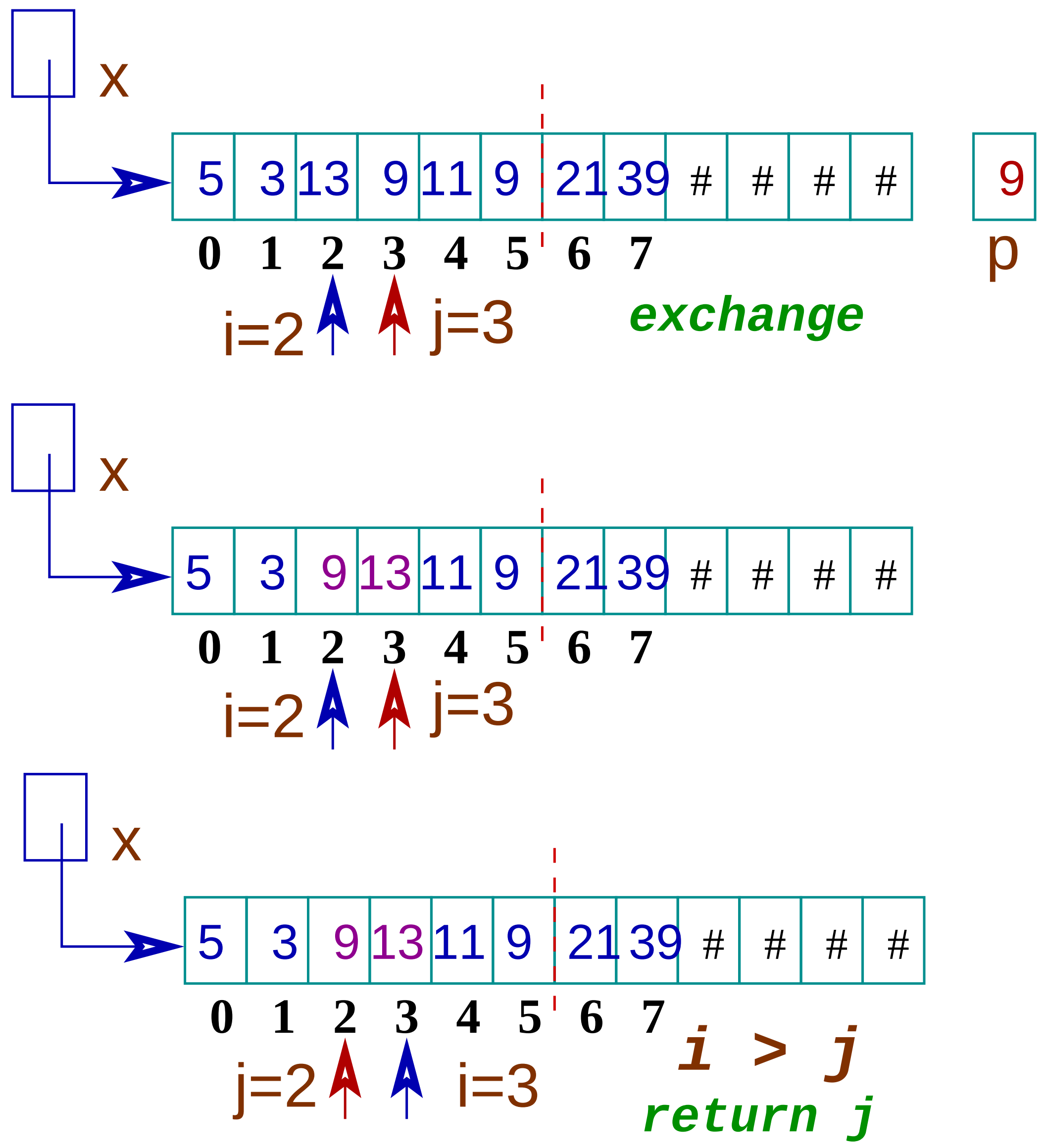
```
quickSort(a, l, h)
if (l < h)
    p ← partition(a, l, h)
    quicksort(a, l, p-1)
    quicksort(a, p+1, h)
```

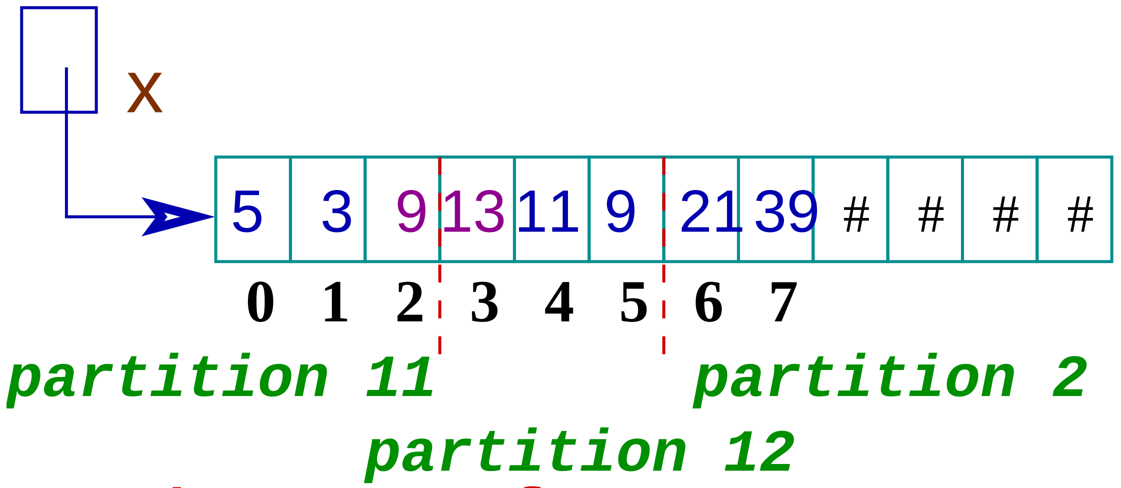
Quick Sort : An Example


Unsorted Array
x
Call 0
21 3 13 9 11 5 9 39 # # # #
0 1 2 3 4 5 6 7
i=−1
j=8
21
p
x
21 3 13 9 11 5 9 39 # # # #
0 1 2 3 4 5 6 7
i=0
j=6
i, j stops
x
9 3 13 9 11 5 21 39 # # # #
0 1 2 3 4 5 6 7
i=0
j=6
exchange


x
9 3 13 9 11 5 21 39 # # # #
0 1 2 3 4 5 6 7
21
p
i=0
j=6
x
9 3 13 9 11 5 21 39 # # # #
0 1 2 3 4 5 6 7
i > j
return j
j=5
i=6
x
9 3 13 9 11 5 21 39 # # # #
0 1 2 3 4 5 6 7
Partition 1
Partition 2
No data transfer
across the partition


x
9 3 13 9 11 5 21 39 # # # #
0 1 2 3 4 5 6 7
9
p
i=−1
j=6
x
9 3 13 9 11 5 21 39 # # # #
0 1 2 3 4 5 6 7
i=0
j=5
exchange
x
5 3 13 9 11 9 21 39 # # # #
0 1 2 3 4 5 6 7
i=0
j=5


x
5 3 13 9 11 9 21 39 # # # #
0 1 2 3 4 5 6 7
9
p
i=2
j=3
exchange
x
5 3 9 13 11 9 21 39 # # # #
0 1 2 3 4 5 6 7
i=2
j=3
x
5 3 9 13 11 9 21 39 # # # #
0 1 2 3 4 5 6 7
j=2
i=3
i > j
return j



No data transfer across partition

C Code

```
#include <stdio.h>
#define MAXNO 100
void quickSort(int [], int, int);
int main()
{
    int n=0, i, data[MAXNO] ;

    printf("Enter the data, terminate by Ctrl+D: ") ;
    while(scanf("%d", &data[n]) != EOF) ++n ;
    printf("\nInpute data: ") ;
    for(i=0; i<n; ++i) printf("%d ", data[i]) ;
    putchar('\n') ;
    quickSort(data, 0, n-1) ;
    printf("Data in ascending order: ") ;
    for(i=0; i<n; ++i) printf("%d ", data[i]);
    putchar('\n');
    return 0;
}
#define EXCH(X,Y,Z) ((Z)=(X), (X)=(Y), (Y)=(Z))
int partition(int data[], int low, int high)
{
    int pvt=data[low], i=low-1, j=high+1 ;
    while(1) {
          int temp;
          do --j ; while (data[j] > pvt) ;
          do ++i ; while (data[i] < pvt) ;
```

```
            if(i < j) EXCH(data[i], data[j], temp);
            else return j ;
      }
}
void quickSort(int data[], int low, int high) {
     if(low < high) {
        int partIndex ;

        partIndex=partition(data,low,high);
        quickSort(data, low, partIndex) ;
        quickSort(data, partIndex+1, high) ;
     }
} // quickSort.c
```

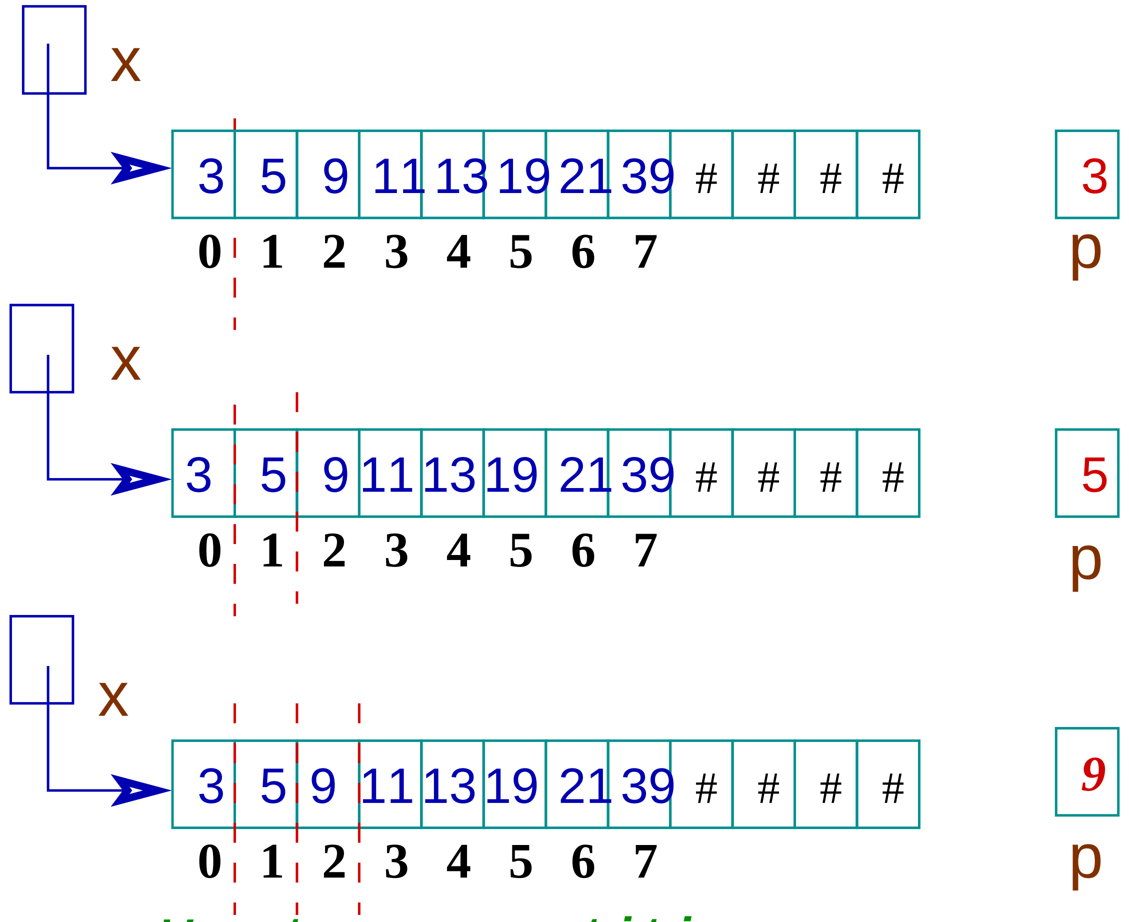
C Code

```
#include <stdio.h>
#define MAXNO 100
void quickSort(int [], int, int);
int main() // quickSort1.c
{
    int n=0, i, data[MAXNO] ;

    printf("Enter the data, terminate by Ctrl+D: ") ;
    while(scanf("%d", &data[n]) != EOF) ++n ;
    printf("\nInpute data: ") ;
    for(i=0; i<n; ++i) printf("%d ", data[i]) ;
    putchar('\n') ;
    quickSort(data, 0, n-1) ;
    printf("Data in ascending order: ") ;
    for(i=0; i<n; ++i) printf("%d ", data[i]);
    putchar('\n');
    return 0;
}
#define EXCH(X,Y,Z) ((Z)=(X), (X)=(Y), (Y)=(Z))
int partition(int data[], int low, int high)
{
    int pvt=data[high], sI=low, i, temp;
    for(i=low; i<high; ++i)
        if(data[i] < pvt){
            EXCH(data[i], data[sI], temp);
            ++sI;
        }
    EXCH(data[sI], data[high], temp);
    return sI;
}
void quickSort(int data[], int low, int high) {
     if(low < high) {
        int partIndex ;

        partIndex=partition(data,low,high);
        quickSort(data, low, partIndex-1) ;
        quickSort(data, partIndex+1, high) ;
     }
} // quickSort.c
```

Analysis of Quick Sort

The performance of quick Sort depends on the nature of the partition. In the worst case there may be only one element in one of the partitions.

The sum of the total number of decrements of index `j` and the increments of `i` can at most be $n + 2$ for n data.
The maximum number of exchanges may be $\frac{n}{2}$ and the running time of quick sort is $O(n^2)$.



Worst case partitions

Analysis of Quick Sort

Let t_n be the time taken to quick sort n data. The time t_n can be expressed by the following recurrence relation.

$$t_n = \begin{cases} O(1), & \text{if } n = 1, \\ t_{n-k} + t_k + O(n), & \text{otherwise,} \end{cases}$$

where the size of one partition is k and $O(n)$ is the time to partition n data.

Solution of t_n

If every time, $k = 1$ in one of the two partitions, then t_n is proportional to

$$(n+2) + (n+1) + \cdots + 2 = \frac{(n+4)(n+1)}{2}$$

i.e. $t_n = O(n^2)$. On an average if the partitions are of comparable sizes, then the solution for t_n is $O(n \log n)$. But the worst case running time is $O(n^2)$.

Extra Space

- The depth of call in the worst case is $O(n)$.
- The depth of call is $O(\log n)$ in the best case.
- So the extra space is $O(n)$ for the recursive implementation.