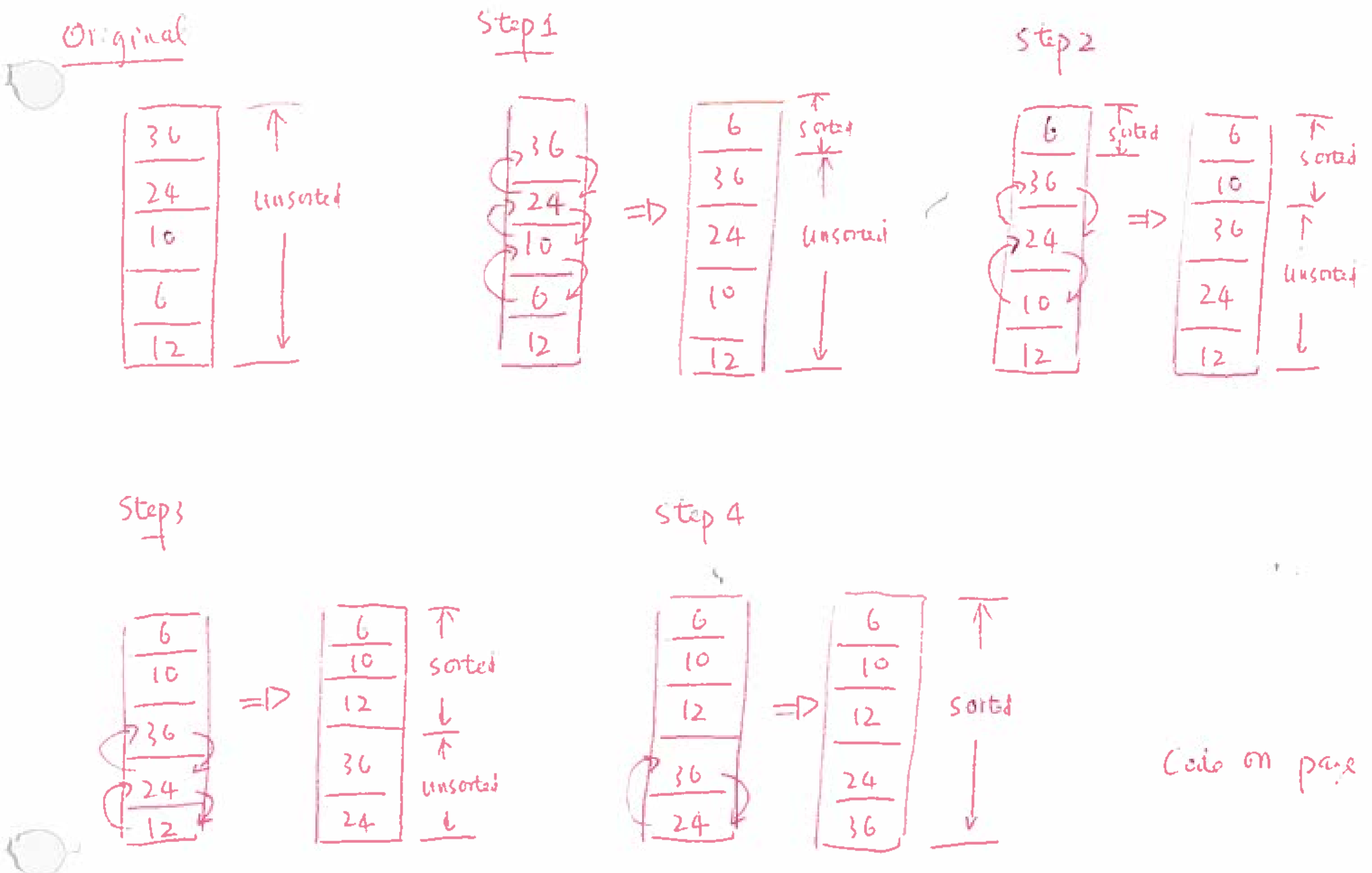
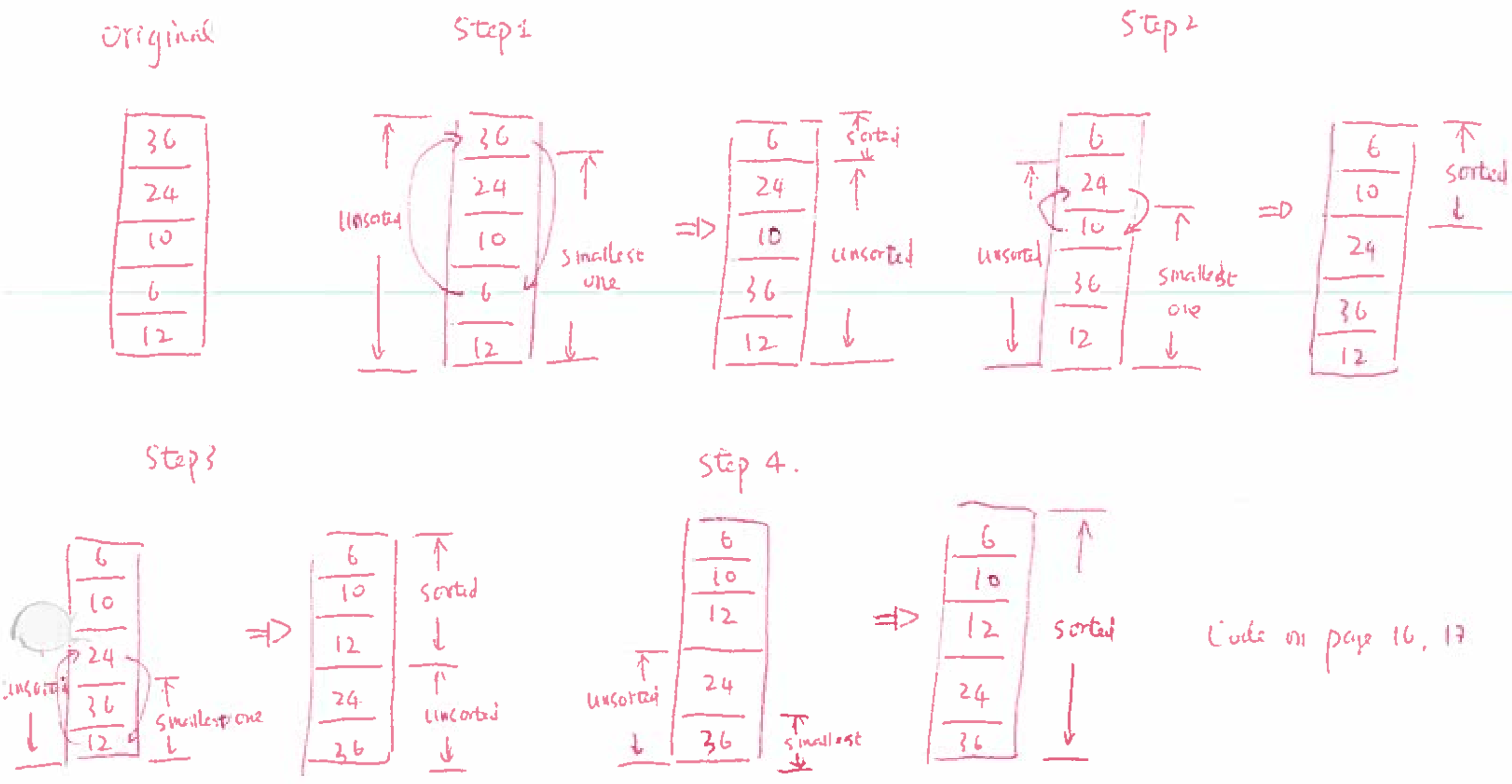
Bubble Sort (always from Bottom) Bottom-up



Code on page 19, 20

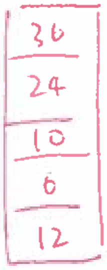
Selection Sort bottom-up



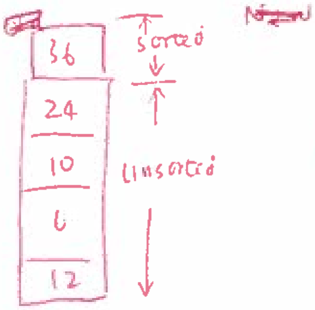
Code on page 16, 17

Insertion Sort (using only one array) Top-down

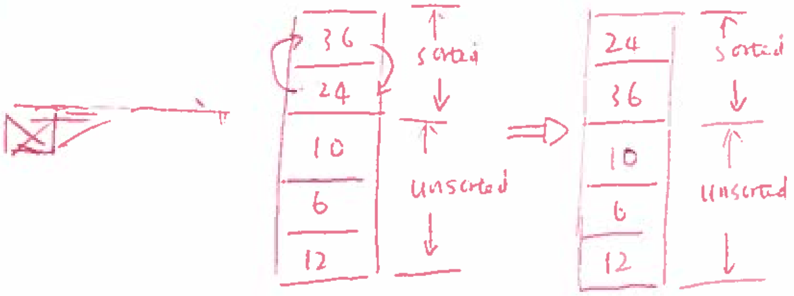
Original



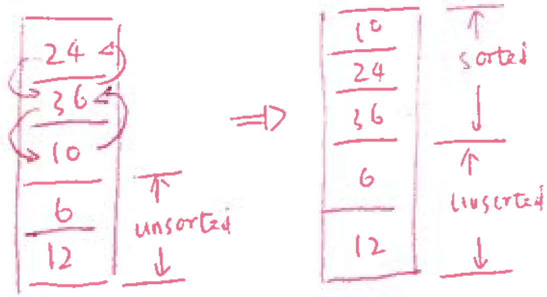
Step 1



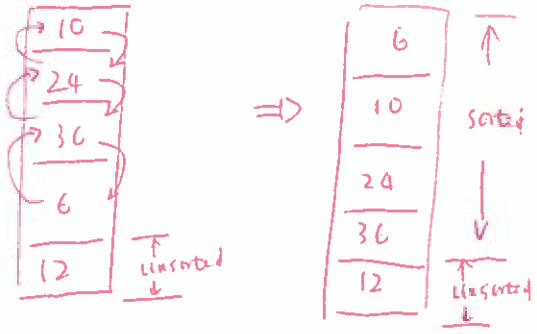
Step 2



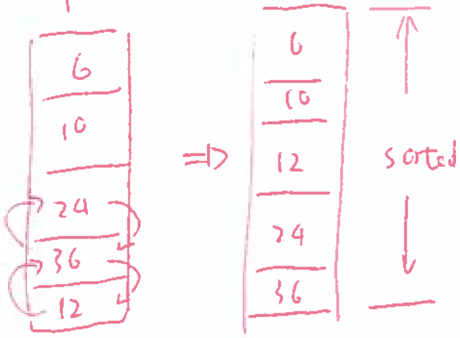
Step 3



Step 4



Step 5



Code on page 26, 27.

```
                      size_t          size_t
qsort ( void * base,  ~~int~~ num,    ~~int~~ width,  int (*Compare)( const void * x,
                                                                      Const void *y))
```

- base → Start of target array
- num → array size
- width → element size in bytes
- Compare → Comparison function
- x → pointer to the key for the search
- y → pointer to the array element to be compared with the key.

Example:

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

int compare ( const void *arg1, const void *arg2);

void main( int argc, char** argv)
{
    int i;
    argv++;    // Eliminate argv[0] from sort
    argc--;

    qsort( (void *) argv, (size_t) argc, sizeof( char *), compare);

    for(i=0; i<argc ; i++)
        Cout << argv[i];
    Cout << endl;
}
```

```
int Values []= {40, 30, 20, 10, 5};
int Comp-int( ... )
{ ... }

int main()
{ int n;
  qsort(values, 5, sizeof (int), comp_int);
  for(int i=0; i<5; i++)
      Cout << Values[i] << endl;
}
```

```
int Comp-int (const void *a, const void *b)
{
    if ( *(int *) a > *(int *) b)
        return 1
    else if (*(int *)a < *(int *) b)
        return -1;
    else return 0.
}
```

```
int compare (const void *arg1, const void *arg2)
{
    return _stricmp( *(char **) arg1, *(char **) arg2);
}
```

See the definition on next page.

```
qsort (nn, k, sizeof (int),
       comp_int);
```

```
~~int compareError (const Void *a, const void *b)~~
{
```

```
int _stricmp(const char * s1, Const char * s2)
```

```
                         <                           -d <0
_stricmp ( " less " ,  " lesst ");       return      0
                                                     +d >0
```

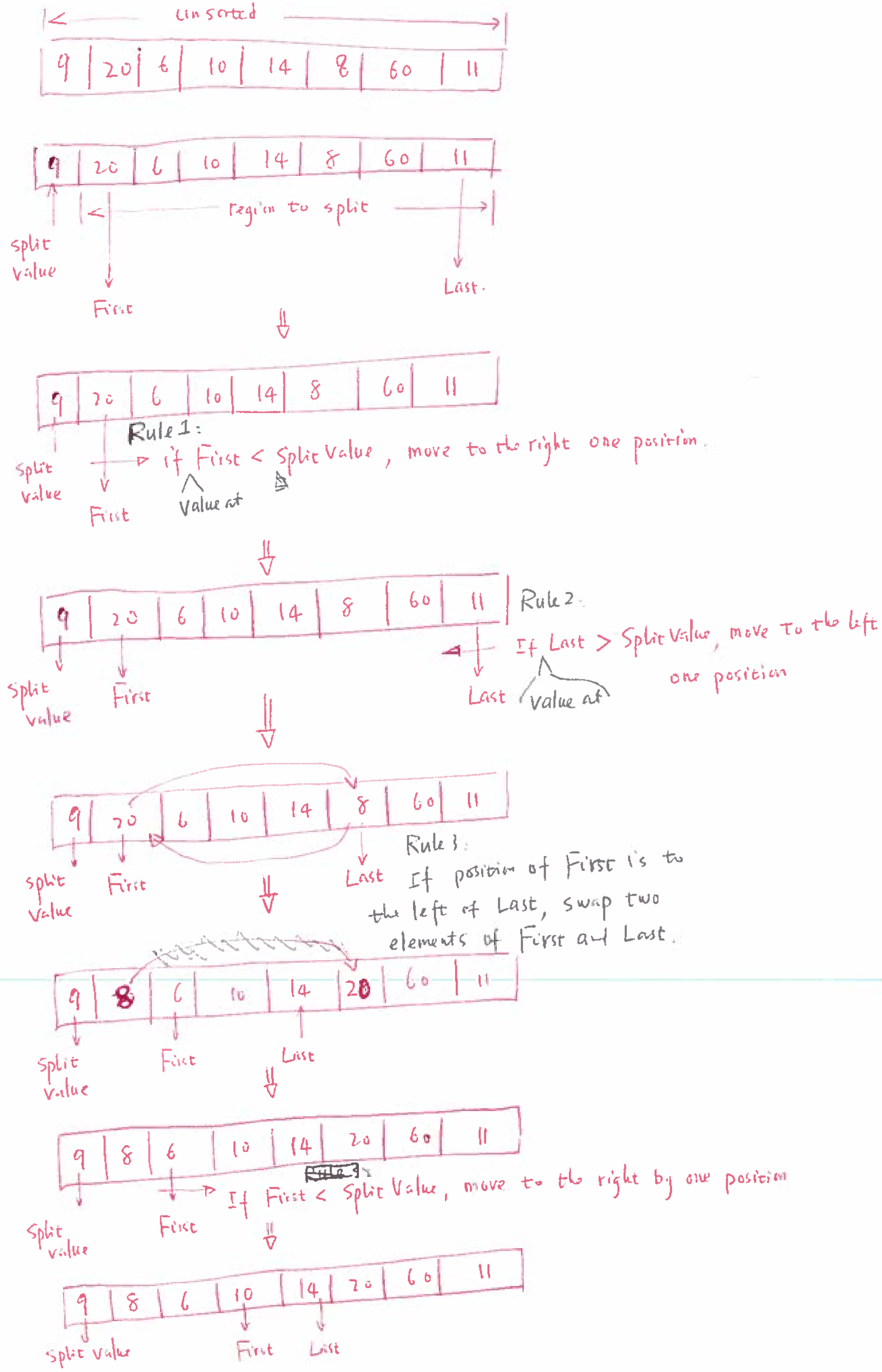
```
C:> qsort every good boy deserves favor
    boy deserves every favor good.
```

quick sort

Q1

Original

Step 1



9	20	6	10	14	8	60	11

9	20	6	10	14	8	60	11

9	20	6	10	14	8	60	11

Rule 1:
if First < Split Value, move to the right one position.

9	20	6	10	14	8	60	11

Rule 2:
If Last > Split Value, move To the left one position

9	20	6	10	14	8	60	11

Rule 3:
If position of First is to the left of Last, swap two elements of First and Last.

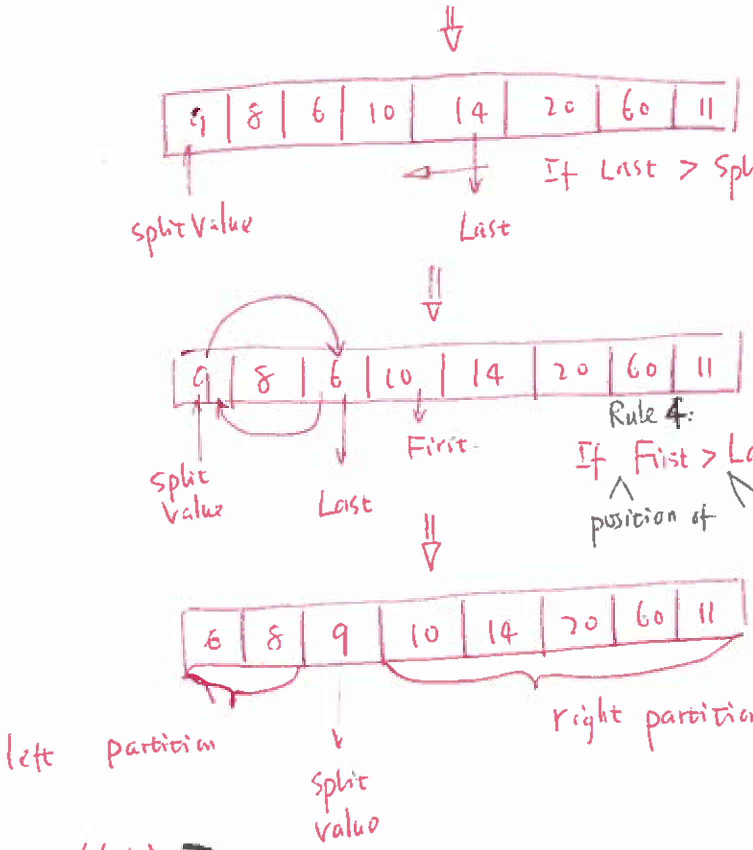
9	8	6	10	14	20	60	11

9	8	6	10	14	20	60	11

~~Rule 3~~ If First < Split Value, move to the right by one position

9	8	6	10	14	20	60	11

Q2



If Last > Split Value, move to the left by one position

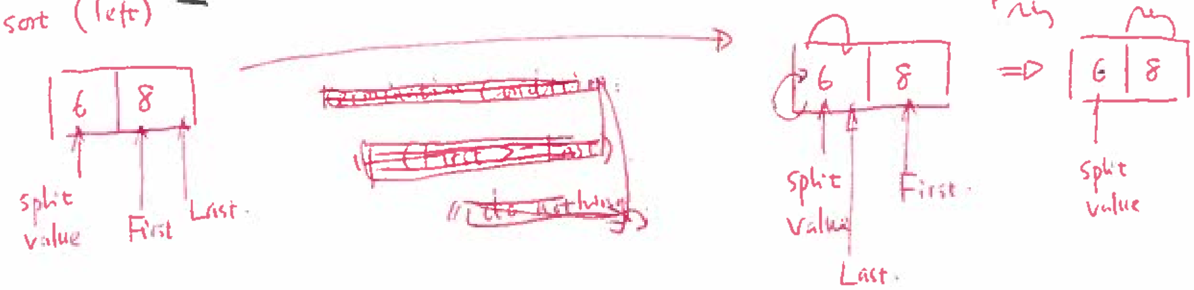
Rule 4:

If First > Last, stop ~~the~~ ~~swap~~ moving First and Last.

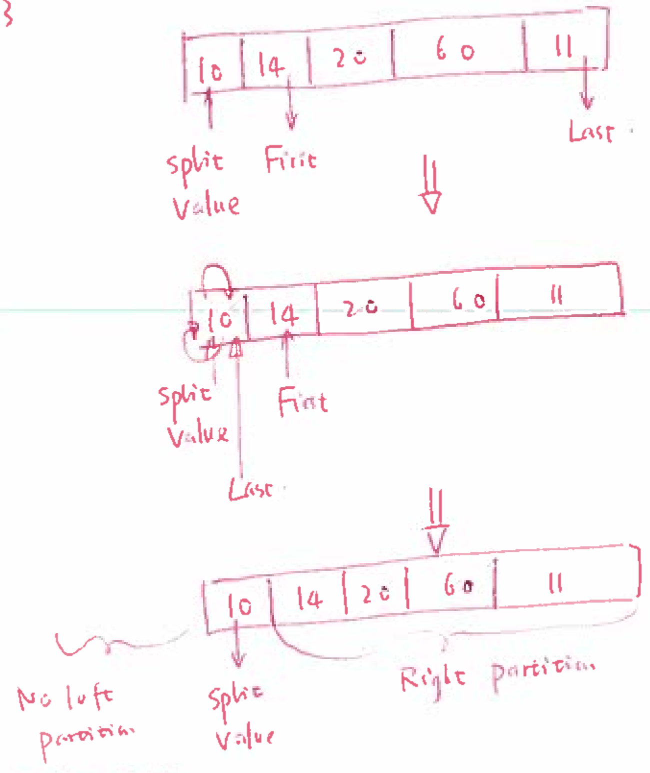
position of — position of

swap the elements of Last and pivot point.

Step2 Quick sort (left)



Step3 Quick Sort (Right)

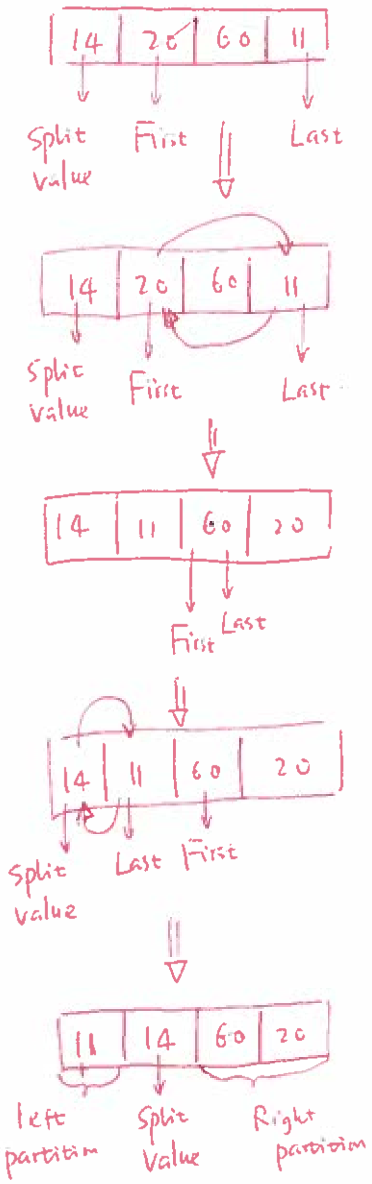


Inefficient?

Q3

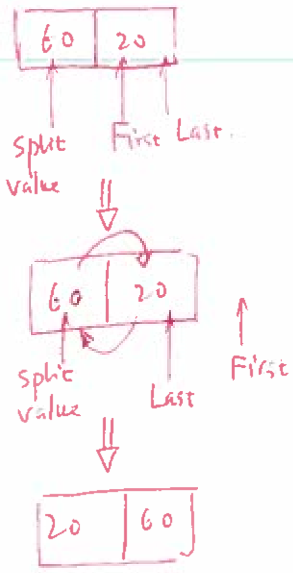
Step 4

(Quick sort (Right)



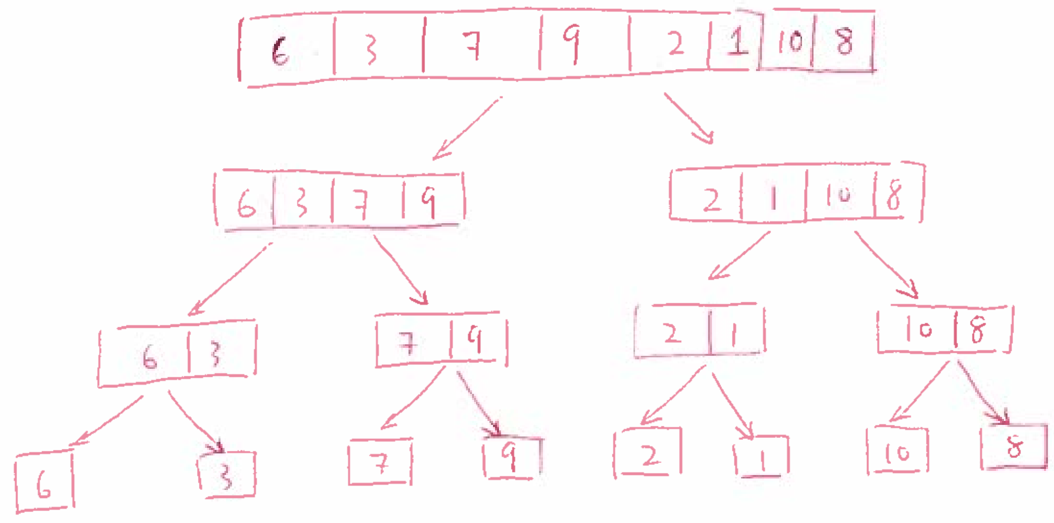
Step 5

Quick sort (Right)



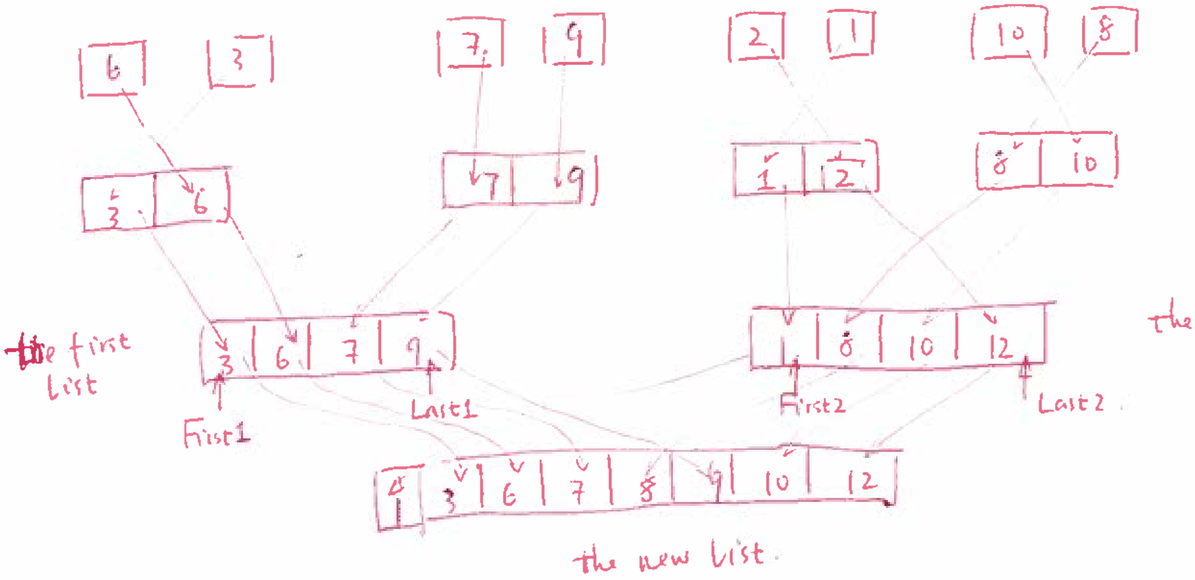
Merge Sort

M1.



Step 1
Splitting Process
top-down

Step 2
Merge Process
bottom-up.



Merge Rule:

Rule 1 If First1 < First2, move First1 to the new list and advance First1 to the right by one position.

Rule 2 If First1 > First2, move First2 to the new list and advance First2 to right by one position.

Rule 3 If Last1 has been moved to the new list, the remainding part of second list can be entirely copied to the new list.

Rule 4 If Last2 has been moved to the new List, the remaining part of the first list can be entirely copied to the new list.