Data Structures – Week #9

Sorting

Outline

- Motivation
- Types of Sorting
- Elementary ($O(n^2)$) Sorting Techniques
- Other ($O(n*log(n))$) Sorting Techniques

Sorting

Motivation

- Sorting is a fundamental task in many computer science problems.
- To *sort a set of data* is to put the data points (or records) in some order with regard to some feature of data.
- In a student registration system, an *example to sorting* would be to put the student records in ascending order with respect to students' IDs.

Types of Sorting

- Sorting techniques may be classified based on whether the entirety of data fits in the main memory.
- Sorting techniques for data that entirely fit in the main memory are called the *internal sorting techniques*.
- Those for data that do not are called the *external sorting techniques*.
- We will discuss *internal sorting techniques*.

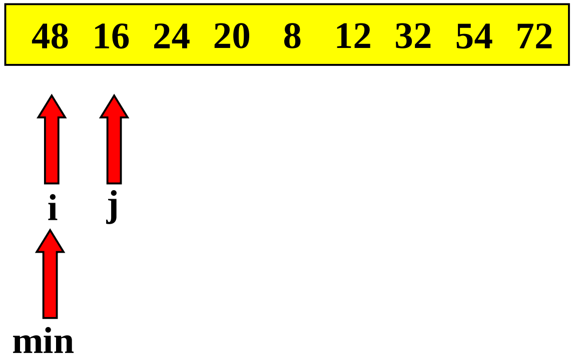
Elementary Sorting ($O(n^2)$)Techniques

- We discuss three (internal) sorting techniques
  - *Selection Sort*
  - *Insertion Sort*
  - *Bubble Sort*

Selection Sort

```
void selectionSort(unsigned int *a, unsigned int n)
//sorts n integers in ascending order (i.e., a[n]<a[n+1])
{
   int i, j, min, t;
   for (i=1; i<n; i++)        {
// ith smallest number gets placed in its correct position.
         min=i;
         for (j=i+1; j<=n; j++)
                  if (a[j] < a[min]) min=j;
         t=a[min]; a[min]=a[i]; a[i]=t;
   }
}
```

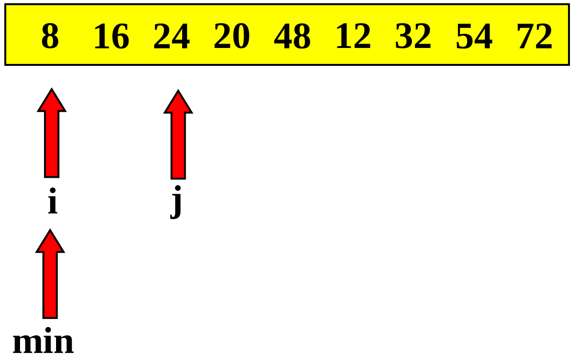
Selection Sort Example



a[j] < a[min]

```
for (i=1; i<n; i++)          {
        min:=i;
        for (j=i+1; j<=n; j++)
                if (a[j] < a[min]) min:=j;
        t:=a[min]; a[min]=a[i]; a[i]=t;
}
```

Selection Sort Example



a[j] < a[min]

```
for (i=1; i<n; i++)           {
            min:=i;
            for (j=i+1; j<=n; j++)
                        if (a[j] < a[min]) min:=j;
            t:=a[min]; a[min]=a[i]; a[i]=t;
}
```

etc.

Algorithm Analysis of Selection Sort

- Barometer statement (or piece of code): the condition of if (*a[j] < a[min]*)
- We find how many times it is executed
- The *outer loop* turns *n-1* times.
- *Inner loop* turns *n-i* times at the *i^{th} turn* of the outer loop.

$$\sum_{i=1}^{n-1}\sum_{j=i+1}^{n}1=\sum_{i=1}^{n-1}(n-i)=\sum_{i=1}^{n}n-\sum_{i=1}^{n}i=$$

$$n^2-\frac{n(n+1)}{2}=\frac{n^2}{2}-\frac{n}{2}\in O(n^2)$$

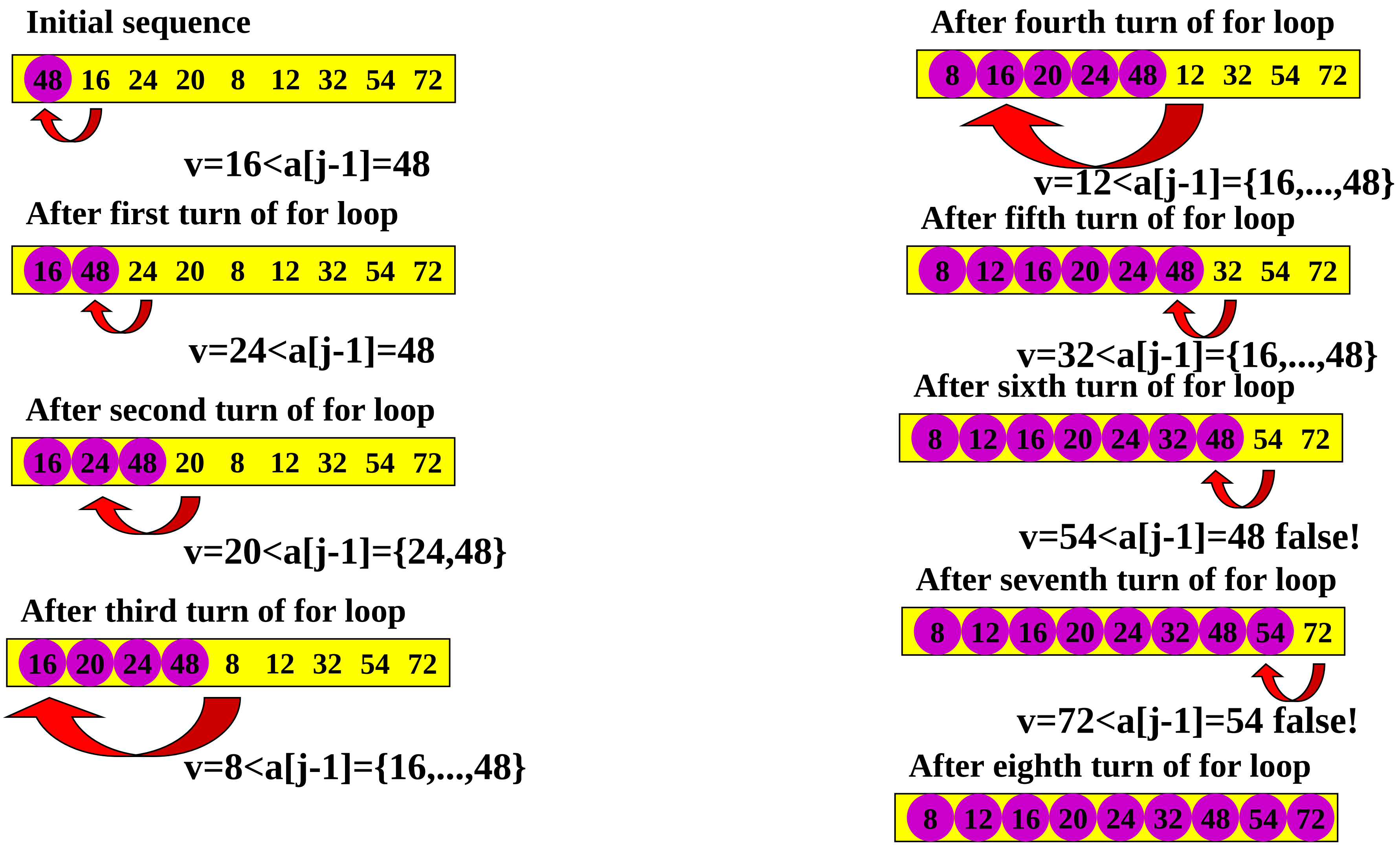
Insertion Sort

- Insertion sort keeps the left portion of the array sorted.
- The insertion sort algorithm places the current (i.e., i^{th}) element at its correct position at the i^{th} turn of the outer loop.

Insertion Sort Algorithm

```
void insertionSort()
{
    int i, j, v;
    int smallest;  // boolean variable
    for (i=2; i<=n; i++) { // a single element array is sorted as is.
            v=a[i]; j=i; smallest=0;
            while (a[j-1] > v && !smallest) {
                        a[j] =a[j-1]; j=j-1;
                        if (j <= 1) smallest=1;
            }
            a[j]=v;
    }
}
```

Insertion Sort Example



Initial sequence

48 16 24 20 8 12 32 54 72

v=16<a[j-1]=48

After first turn of for loop

16 48 24 20 8 12 32 54 72

v=24<a[j-1]=48

After second turn of for loop

16 24 48 20 8 12 32 54 72

v=20<a[j-1]={24,48}

After third turn of for loop

16 20 24 48 8 12 32 54 72

v=8<a[j-1]={16,...,48}

After fourth turn of for loop

8 16 20 24 48 12 32 54 72

v=12<a[j-1]={16,...,48}

After fifth turn of for loop

8 12 16 20 24 48 32 54 72

v=32<a[j-1]={16,...,48}

After sixth turn of for loop

8 12 16 20 24 32 48 54 72

v=54<a[j-1]=48 false!

After seventh turn of for loop

8 12 16 20 24 32 48 54 72

v=72<a[j-1]=54 false!

After eighth turn of for loop

8 12 16 20 24 32 48 54 72

Algorithm Analysis of Insertion Sort

- Here, the inner loop has a stochastic condition. Hence, we either have to work using expected execution times or make a worst case analysis.
- Barometer statement (or piece of code): *j=j-1*
- We find how many times it is executed
- The outer loop again turns *n-1* times.
- In the *worst case*, inner loop turns *i-1 times* at the i^{th} turn of the outer loop.

Algorithm Analysis of Insertion Sort …cont’d

$$\sum_{i=2}^{n}\sum_{j=1}^{i-1}1=\sum_{i=2}^{n}(i-1)=\frac{n(n+1)}{2}-1-(n-1)=\frac{n^2}{2}-\frac{n}{2}\in O(n^2)$$

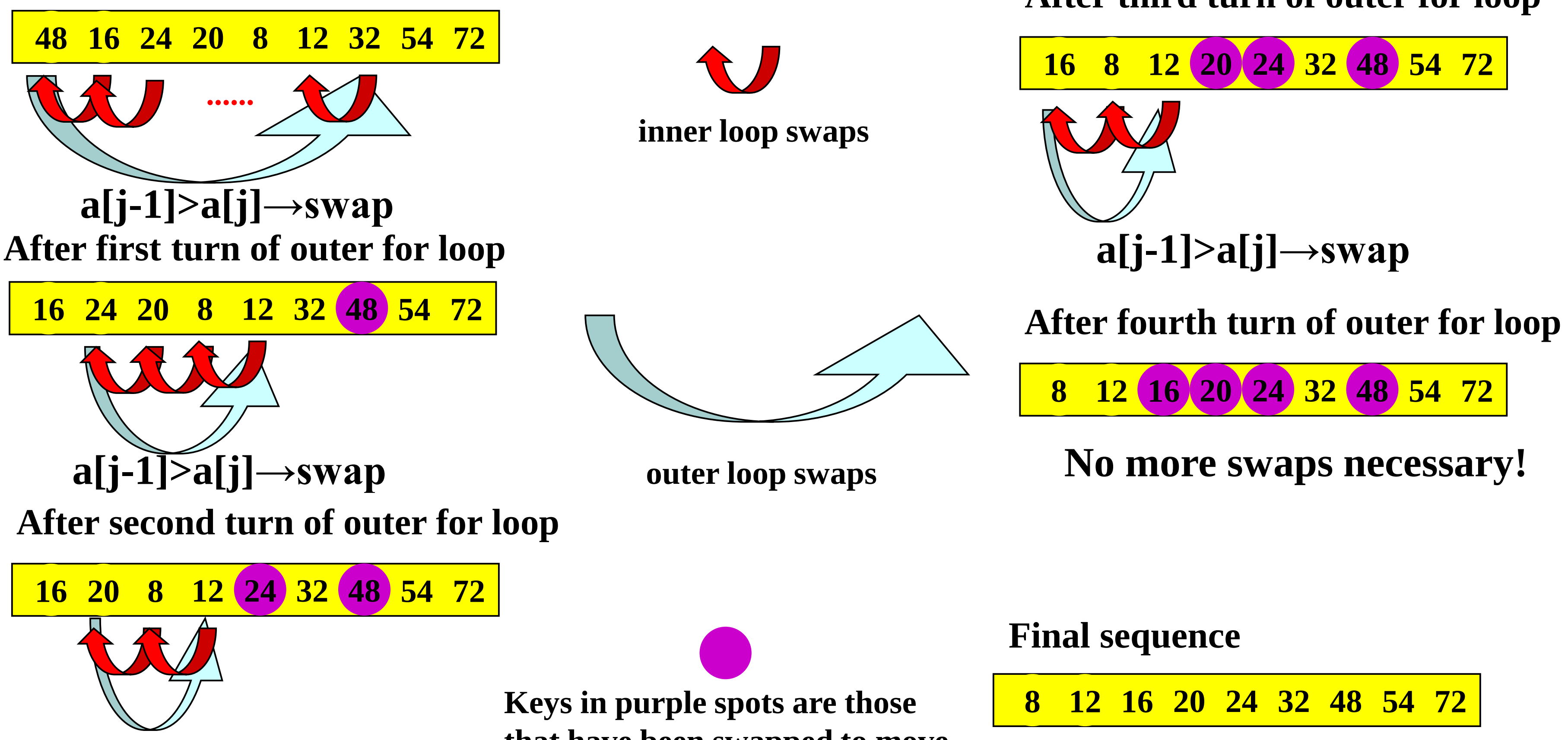
Bubble Sort

- Pass through the array of elements
- Exchange adjacent elements, if necessary
- When no exchanges are required, then array is sorted.
- Make as many passes as the number of elements of the array

Algorithm of Bubble Sort

```
void bubbleSort()
{
   int i, j, t;
   for (i=n; i>=1; i--)
         for (j=2; j<=i; j++) do
                  if (a[j-1] > a[j]) {
                           t:=a[j-1]; a[j-1]:=a[j]; a[j]:=t;
                  }
}
```

Bubble Sort Example



Initial sequence

48 16 24 20 8 12 32 54 72

……

a[j-1]>a[j]→swap

After first turn of outer for loop

16 24 20 8 12 32 48 54 72

a[j-1]>a[j]→swap

After second turn of outer for loop

16 20 8 12 24 32 48 54 72

a[j-1]>a[j]→swap

inner loop swaps

outer loop swaps

● Keys in purple spots are those that have been swapped to move towards their correct position.

After third turn of outer for loop

16 8 12 20 24 32 48 54 72

a[j-1]>a[j]→swap

After fourth turn of outer for loop

8 12 16 20 24 32 48 54 72

No more swaps necessary!

Final sequence

8 12 16 20 24 32 48 54 72

Algorithm Analysis of Bubble Sort

- Barometer statement (or piece of code): the condition of if (*a[j-1] > a[j]*)
- We find how many times it is executed
- The outer loop turns *n* times.
- Inner loop turns *i-1* times at the i^{th} turn of the outer loop.

$$\sum_{i=1}^{n}\sum_{j=2}^{i}1=\sum_{i=1}^{n}(i-1)=\sum_{i=1}^{n}i-\sum_{i=1}^{n}1=$$

$$\frac{n(n+1)}{2}-n=\frac{n^2}{2}-\frac{n}{2}\in O(n^2)$$

Other (O(n*lg(n))) Sorting Techniques

- We discuss three (internal) sorting techniques
  - *Heapsort*
  - *Mergesort*
  - *Quicksort*

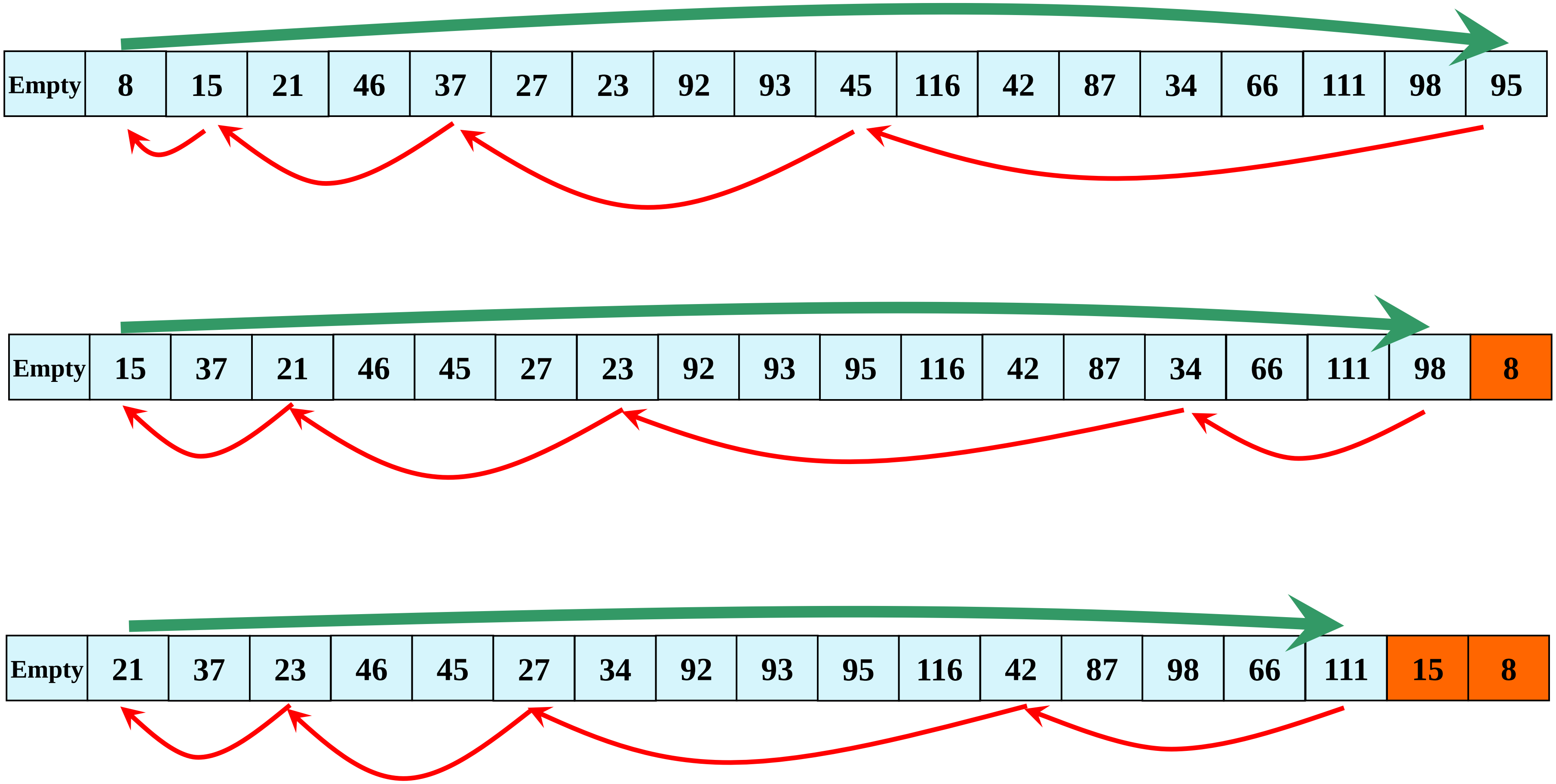
Heapsort

- ***Idea***: *Delete_Min* operation in a minimum heap *removes the key in root*.
- The *hole at the root percolates down* where, in the array we keep the minimum heap, *proper keys are moved left accordingly*.
- The *last cell* in the array becomes *available for storing another key*.
- If we *perform the delete_min operation **n** times* in an *n*-element min heap and *place the $\mathbf{i}^{th}$ key removed* at the available cell *A[n-i]*, we will obtain a sorted sequence of the keys in the heap in *descending* order.

Algorithm of Heapsort

```
int heapsort(Elmnt_Type *A)
{ // sorts keys in minheap A in descending order...
   Elmnt_Type x;
   for (i=1;i<=n;i++)
   {
         x=DeleteMin(A); // O(lgn)
         A[n-i]=x;              // O(1)
   }
}
```

Heapsort Example

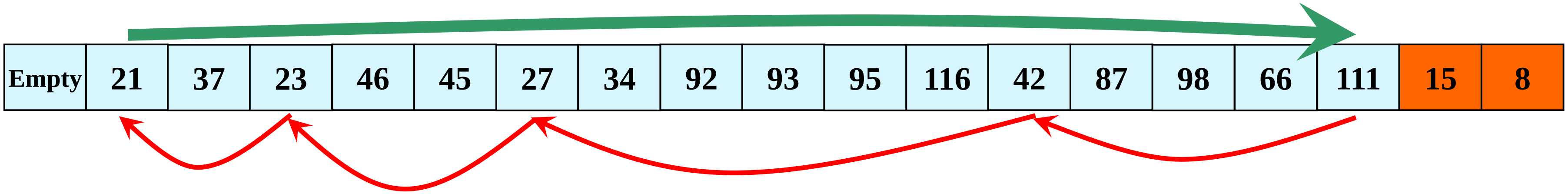


Empty	8	15	21	46	37	27	23	92	93	45	116	42	87	34	66	111	98	95

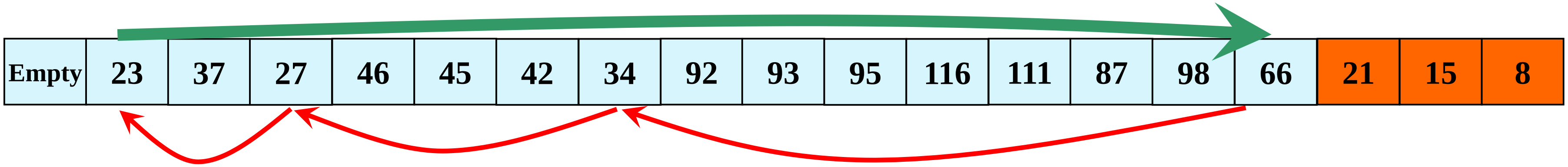
Empty	15	37	21	46	45	27	23	92	93	95	116	42	87	34	66	111	98	8

Empty	21	37	23	46	45	27	34	92	93	95	116	42	87	98	66	111	15	8

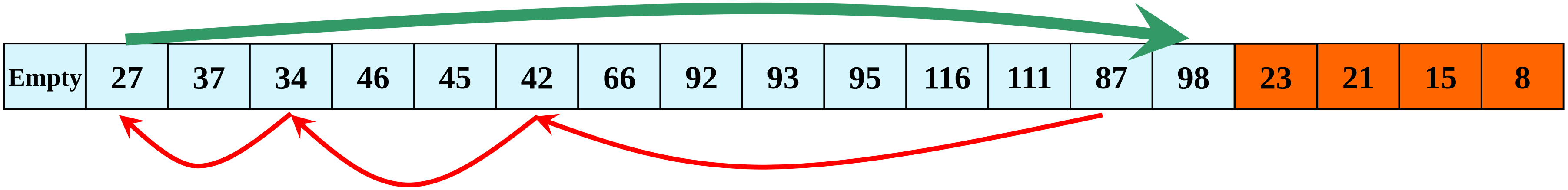
Heapsort Example



Empty	21	37	23	46	45	27	34	92	93	95	116	42	87	98	66	111	15	8

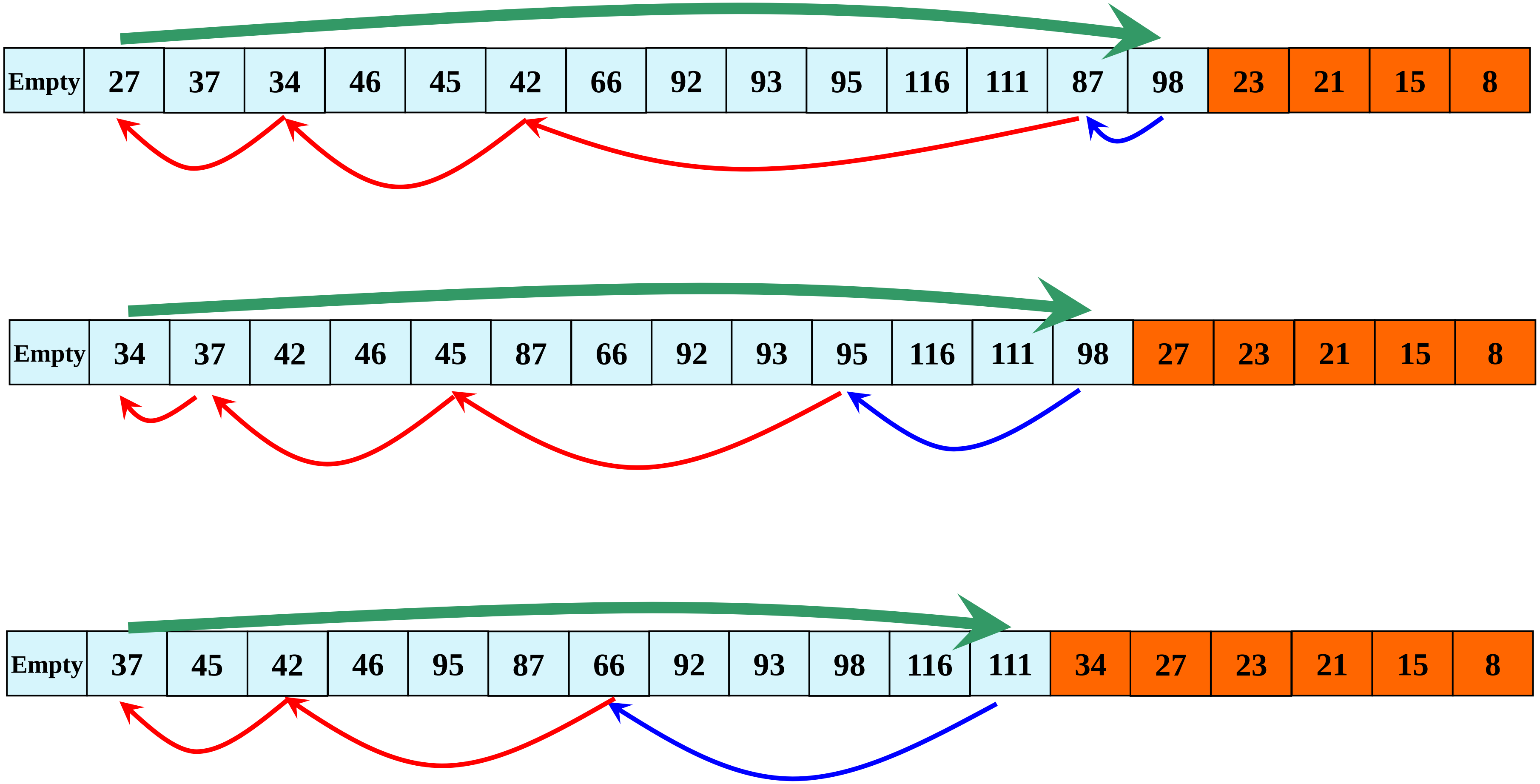


Empty	23	37	27	46	45	42	34	92	93	95	116	111	87	98	66	21	15	8



Empty	27	37	34	46	45	42	66	92	93	95	116	111	87	98	23	21	15	8

Heapsort Example



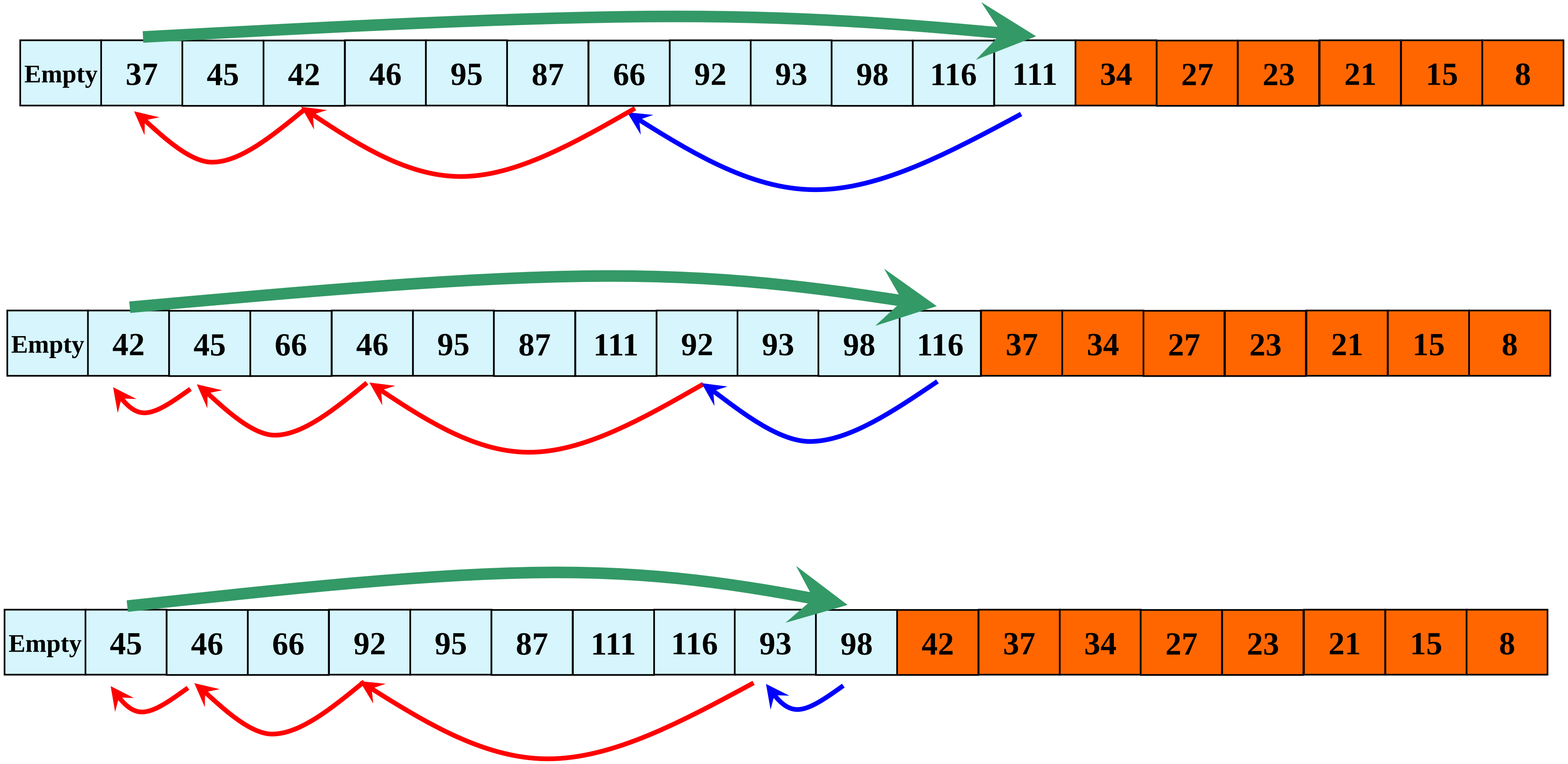
Empty	27	37	34	46	45	42	66	92	93	95	116	111	87	98	23	21	15	8

Empty	34	37	42	46	45	87	66	92	93	95	116	111	98	27	23	21	15	8

Empty	37	45	42	46	95	87	66	92	93	98	116	111	34	27	23	21	15	8

Shifts by ↶ are performed to restore the heap order property of the minimum heap.

Heapsort Example



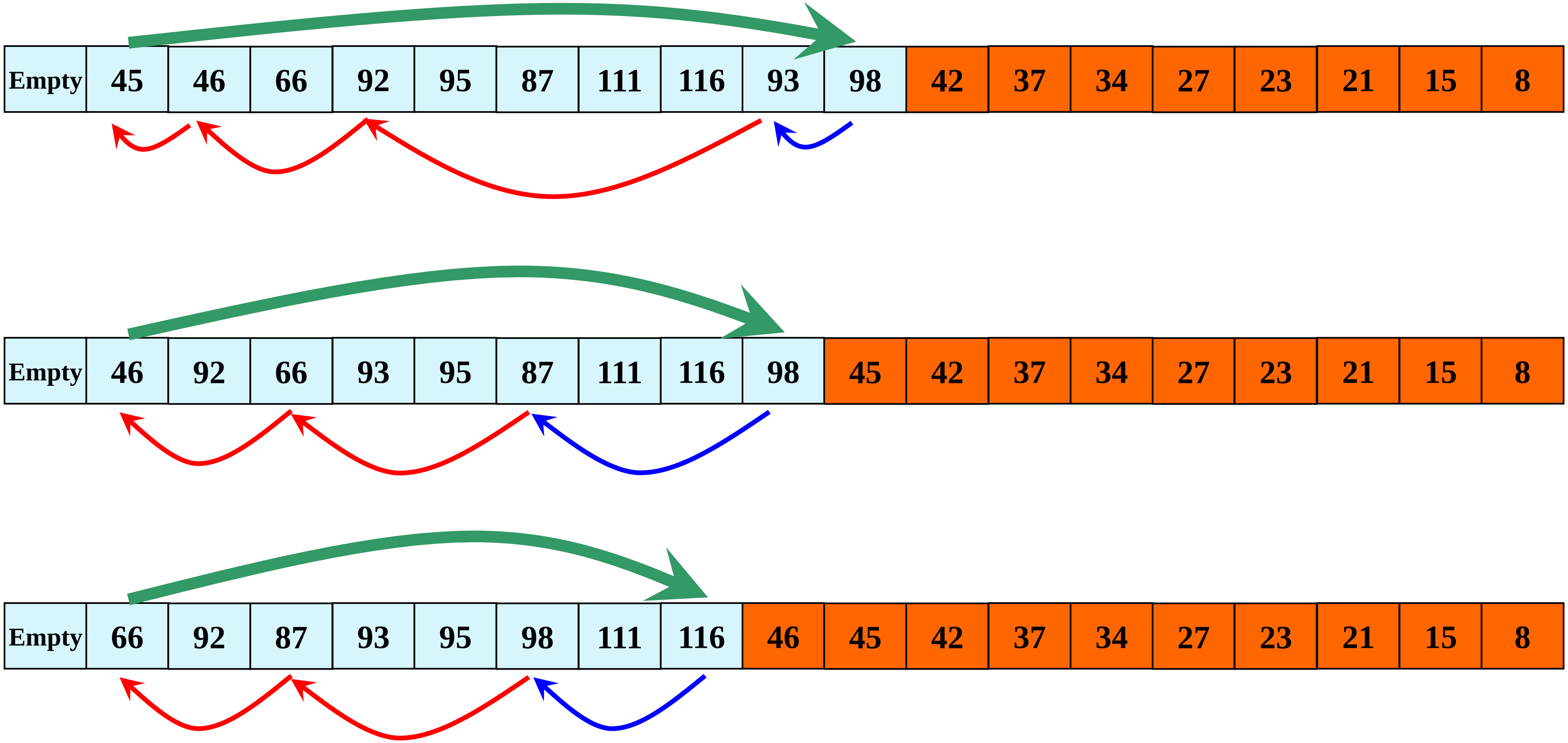
Empty	**37**	**45**	**42**	**46**	**95**	**87**	**66**	**92**	**93**	**98**	**116**	**111**	**34**	**27**	**23**	**21**	**15**	**8**

Empty	**42**	**45**	**66**	**46**	**95**	**87**	**111**	**92**	**93**	**98**	**116**	**37**	**34**	**27**	**23**	**21**	**15**	**8**

Empty	**45**	**46**	**66**	**92**	**95**	**87**	**111**	**116**	**93**	**98**	**42**	**37**	**34**	**27**	**23**	**21**	**15**	**8**

Shifts by ↶ are performed to restore the heap order property of the minimum heap.

Heapsort Example



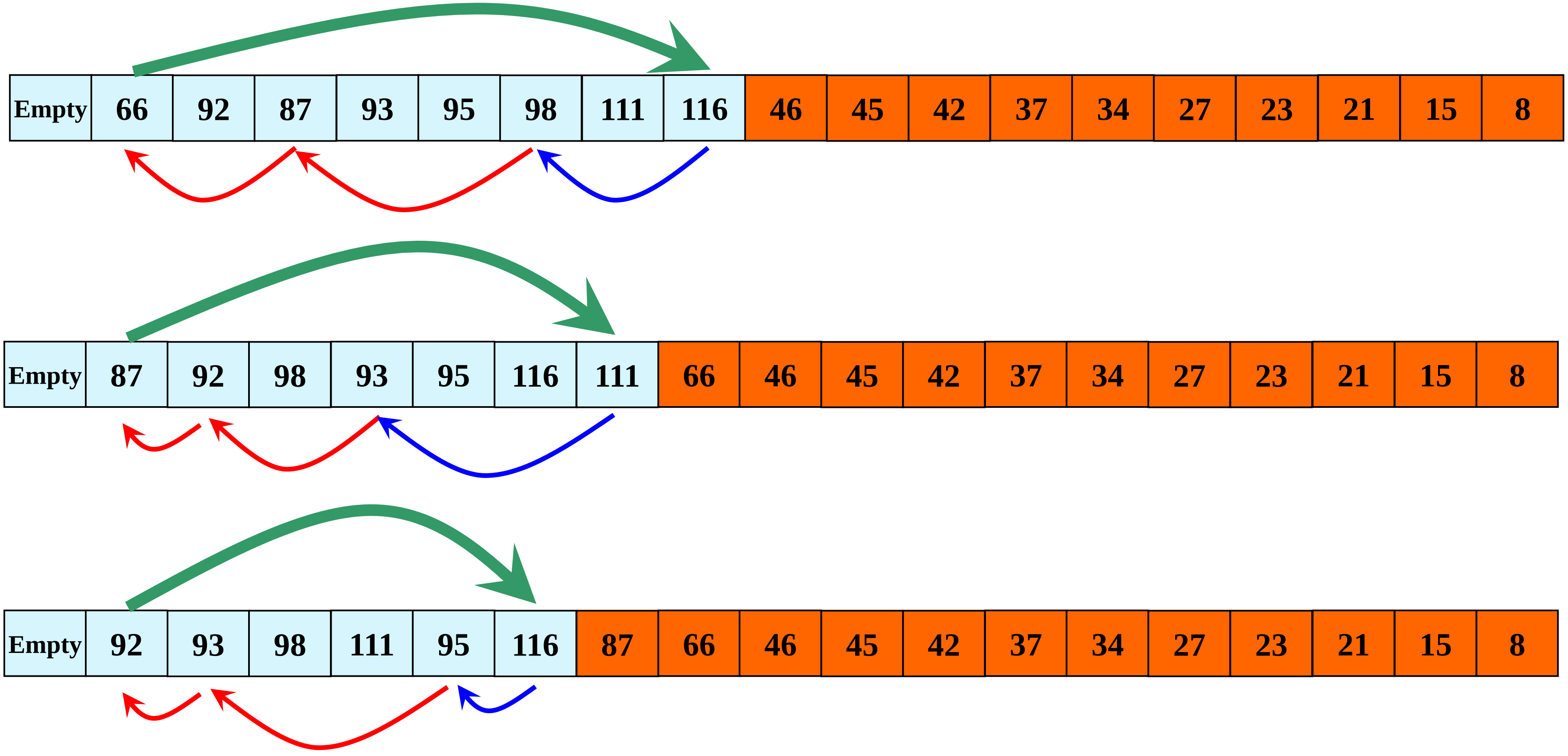
Empty	45	46	66	92	95	87	111	116	93	98	42	37	34	27	23	21	15	8

Empty	46	92	66	93	95	87	111	116	98	45	42	37	34	27	23	21	15	8

Empty	66	92	87	93	95	98	111	116	46	45	42	37	34	27	23	21	15	8

Shifts by ↶ are performed to restore the heap order property of the minimum heap.

Heapsort Example



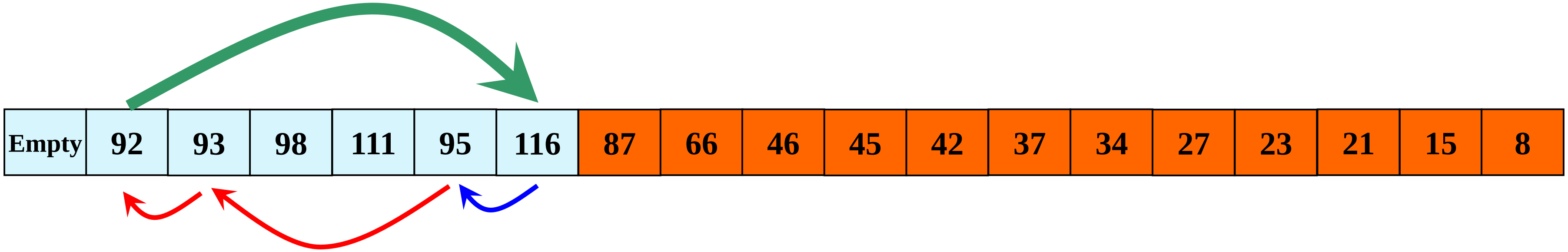
Empty	66	92	87	93	95	98	111	116	46	45	42	37	34	27	23	21	15	8

Empty	87	92	98	93	95	116	111	66	46	45	42	37	34	27	23	21	15	8

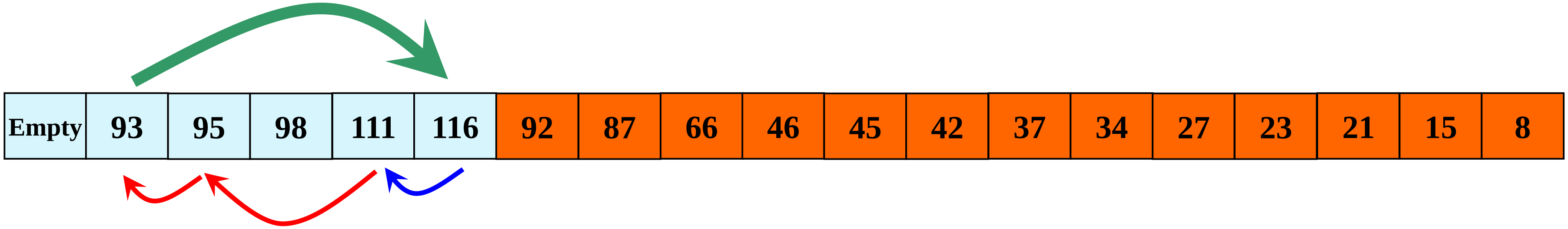
Empty	92	93	98	111	95	116	87	66	46	45	42	37	34	27	23	21	15	8

Shifts by ↶ are performed to restore the heap order property of the minimum heap.

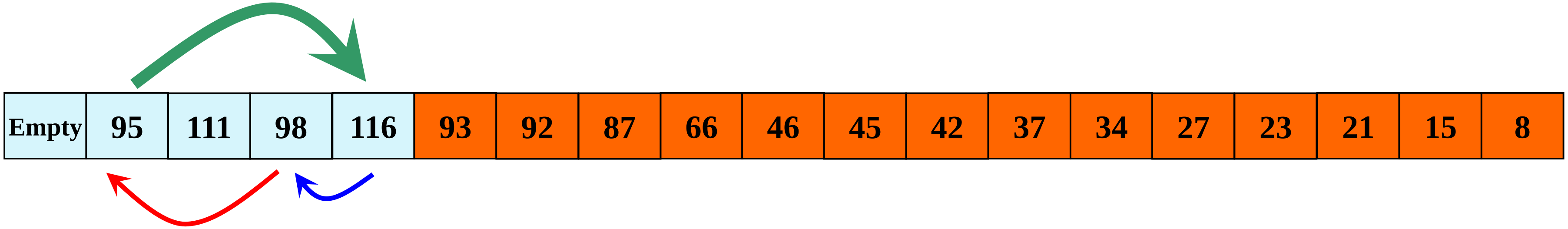
Heapsort Example



Empty	92	93	98	111	95	116	87	66	46	45	42	37	34	27	23	21	15	8



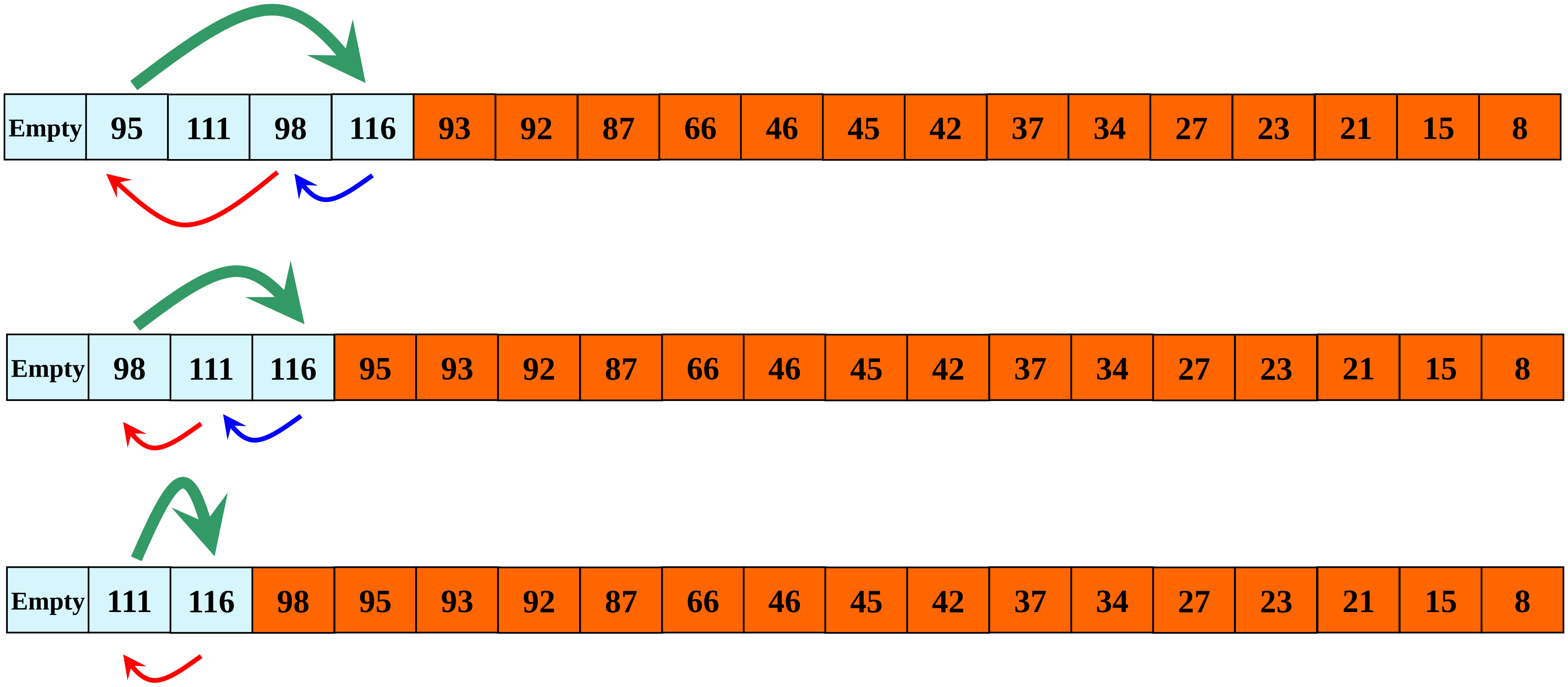
Empty	93	95	98	111	116	92	87	66	46	45	42	37	34	27	23	21	15	8



Empty	95	111	98	116	93	92	87	66	46	45	42	37	34	27	23	21	15	8

Shifts by ↶ are performed to restore the heap order property of the minimum heap.

Heapsort Example



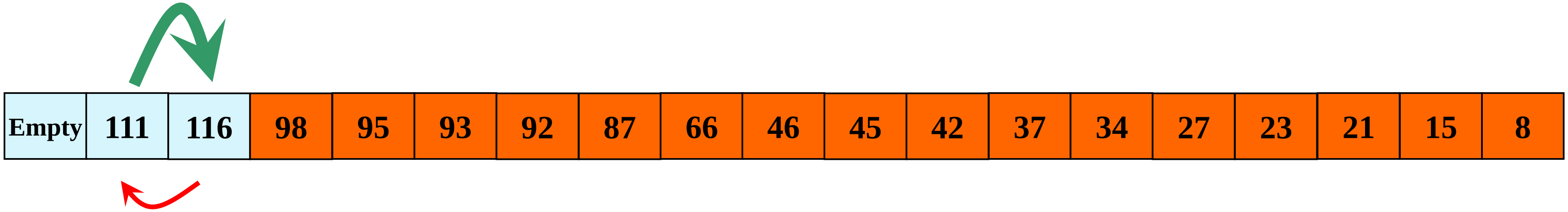
Empty	95	111	98	116	93	92	87	66	46	45	42	37	34	27	23	21	15	8

Empty	98	111	116	95	93	92	87	66	46	45	42	37	34	27	23	21	15	8

Empty	111	116	98	95	93	92	87	66	46	45	42	37	34	27	23	21	15	8

Shifts by ↶ are performed to restore the heap order property of the minimum heap.

Heapsort Example



Empty	**111**	**116**	**98**	**95**	**93**	**92**	**87**	**66**	**46**	**45**	**42**	**37**	**34**	**27**	**23**	**21**	**15**	**8**

Empty	**116**	**111**	**98**	**95**	**93**	**92**	**87**	**66**	**46**	**45**	**42**	**37**	**34**	**27**	**23**	**21**	**15**	**8**

Sequence in the array is sorted in descending order!

Empty	**116**	**111**	**98**	**95**	**93**	**92**	**87**	**66**	**46**	**45**	**42**	**37**	**34**	**27**	**23**	**21**	**15**	**8**

Algorithm Analysis of Heap Sort

- Every turn of the for loop:
  - One delete_min is performed, $O(\lg n)$;
  - Key in the root is placed in last element of the current queue, $O(1)$.
- For loop is executed n times for a heap with n elements
- Hence, running time of heap sort is $O(n \lg n)$.

Merging two sorted lists

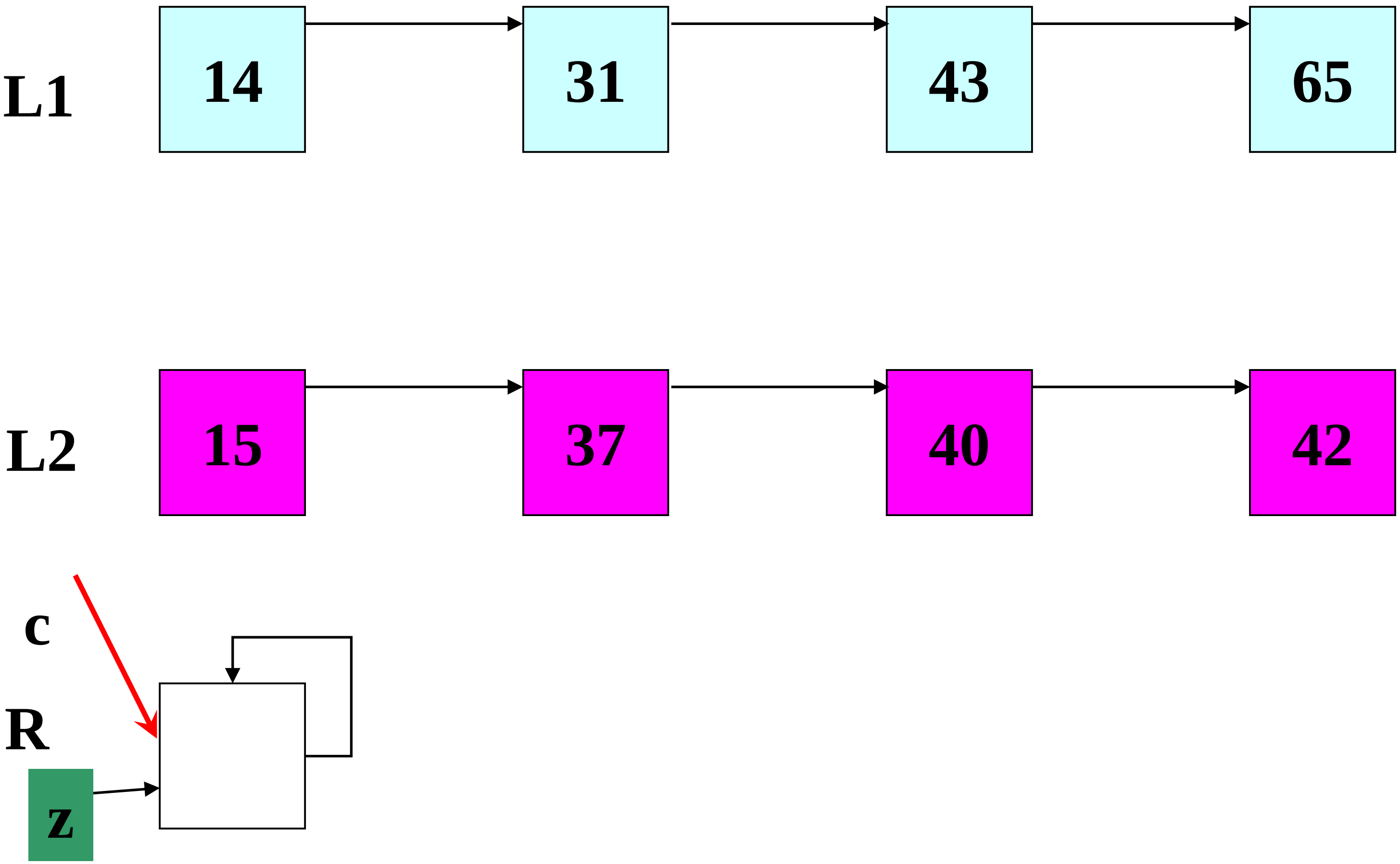
- Consider two sorted lists L1, L2 and an empty list R.
- The following algorithm merges L1 and L2 in R:
  - a and b point to first element of L1 and L2, respectively,
  - c points to the empty resulting list;
  - while both lists have elements left
    - select as the next element of resulting list min(L1[a],L2[b]);
    - advance the pointers or indices of the result list and the input list with the smaller element
  - Append rest of longer list (among L1 and L2) to R

Merging two sorted lists: Algorithm

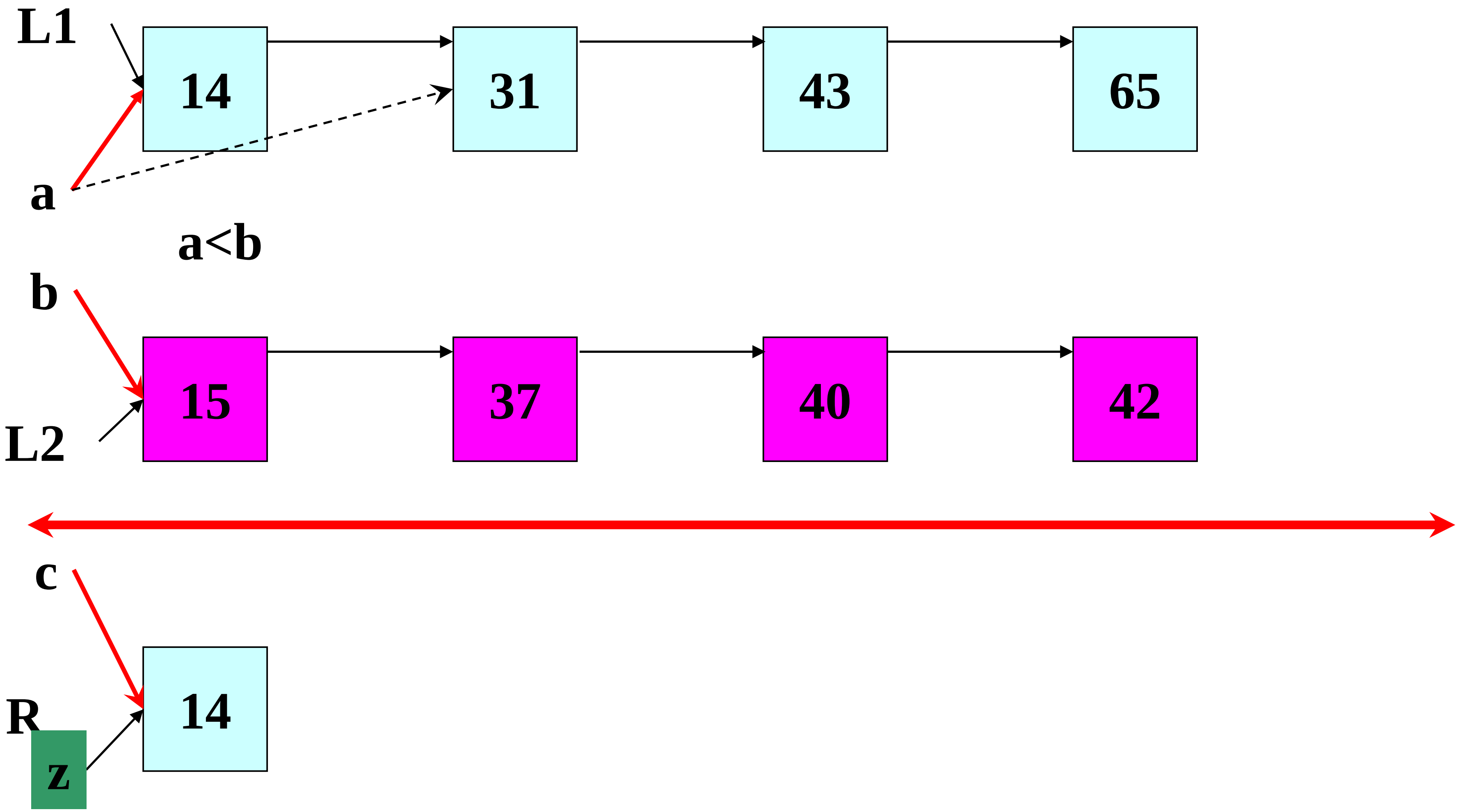
```
struct nodetype {
    int k;
    struct node * next;
}
typedef struct nodetype nodetype;
typedef nodetype * nodeptrtype;
int N,M; nodeptrtype z;// z is a special header
nodeptrtype merge (nodeptrtype a,  nodeptrtype b)
{
    nodeptrtype c;
    c=z;
    do
      if (a->k <=b->k) {c->next=a; c=a; a=a->next;}
      else {c->next=b; c=b; b=b->next;}
    while (c->k < maxint);
    merge=z->next; z->next=z;
}
```

```
main( );
{
   scanf (“%d %d”, &M, &N);
   z=malloc(nodetype);
   z->k=maxint; z->next=z;
   …    /* Lists a and b are
    constructed here */
   merge (a,b);
}
```

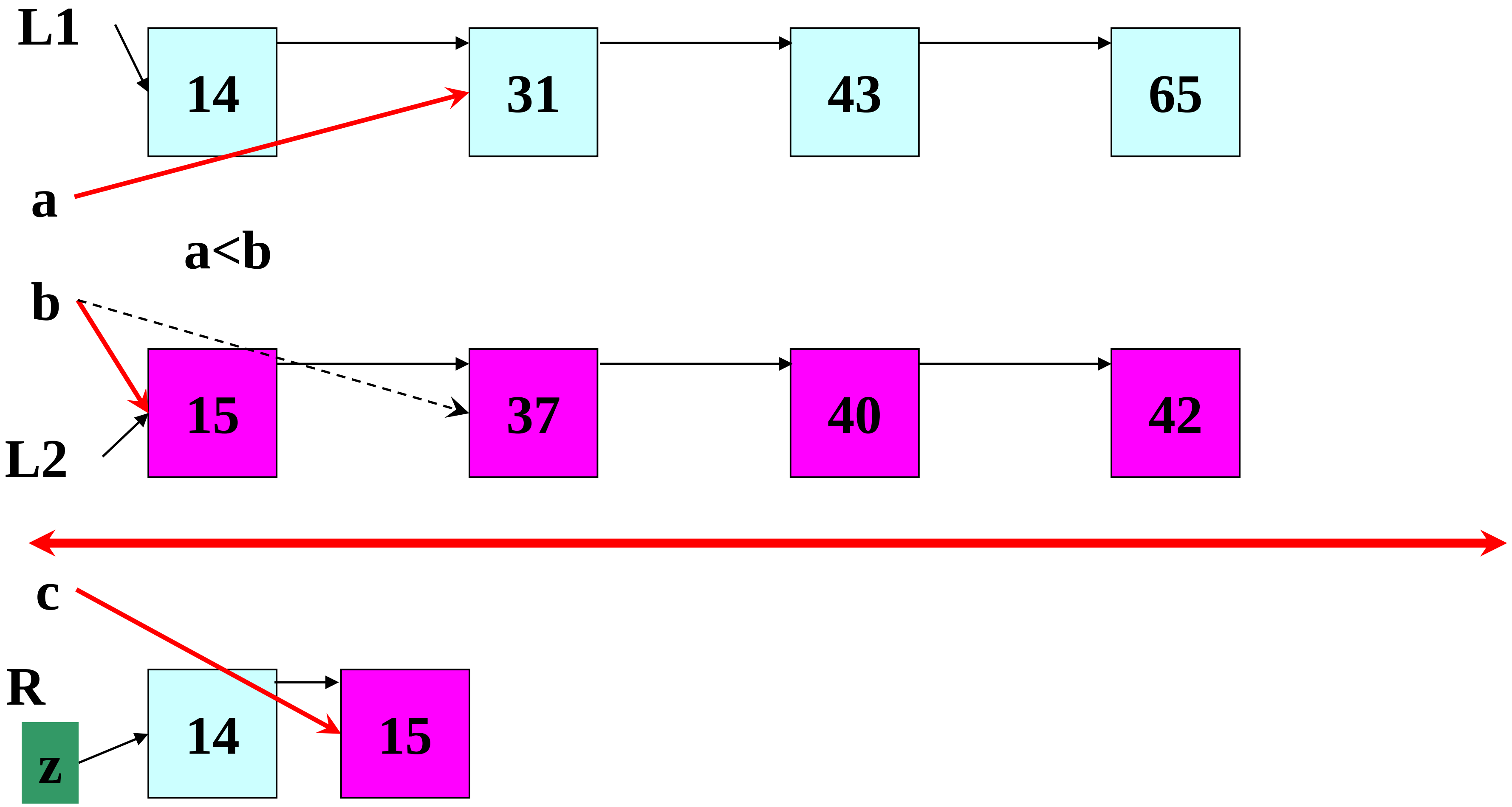
Merging two sorted lists: Example



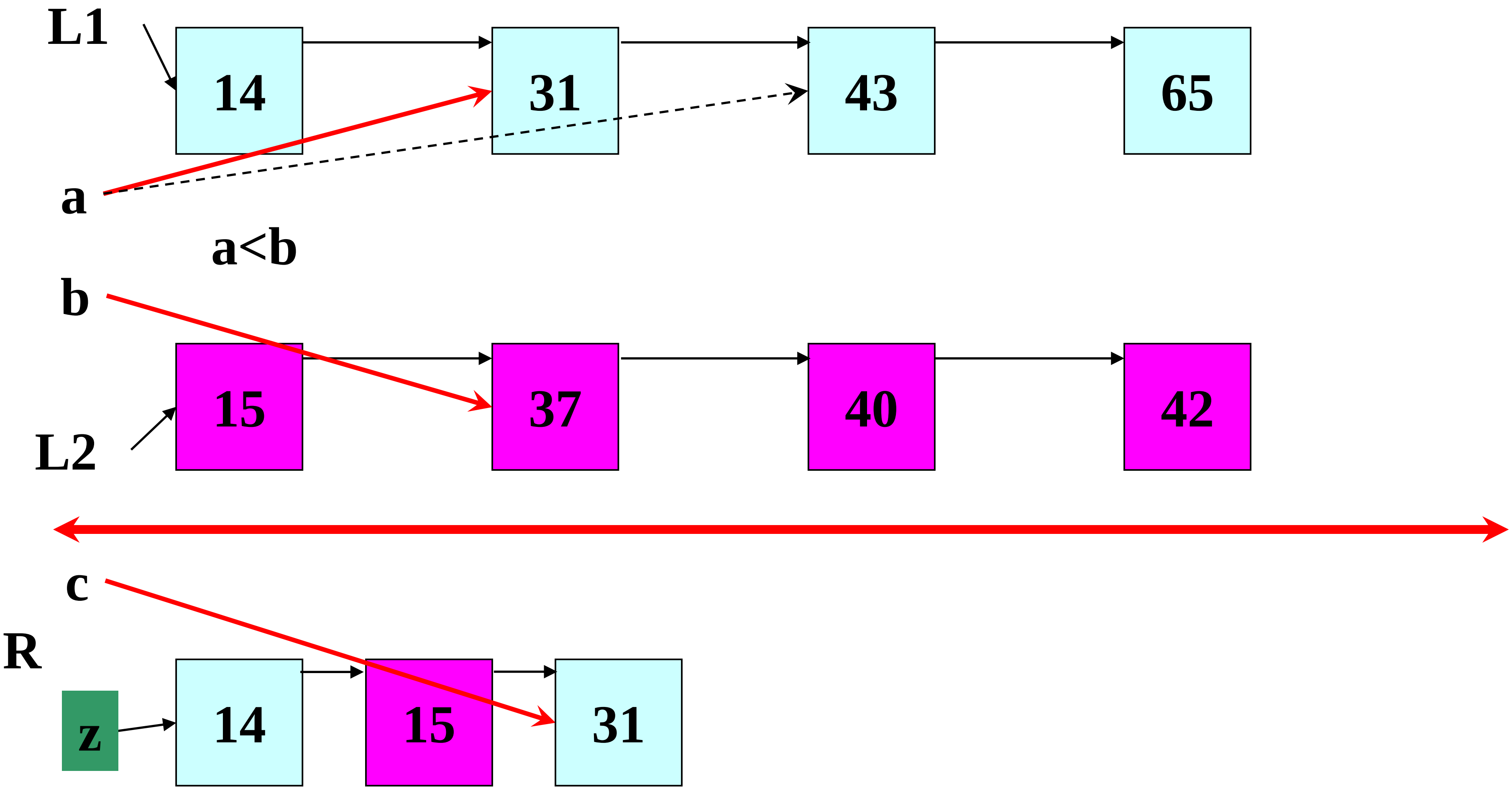
Merging two sorted lists: Example



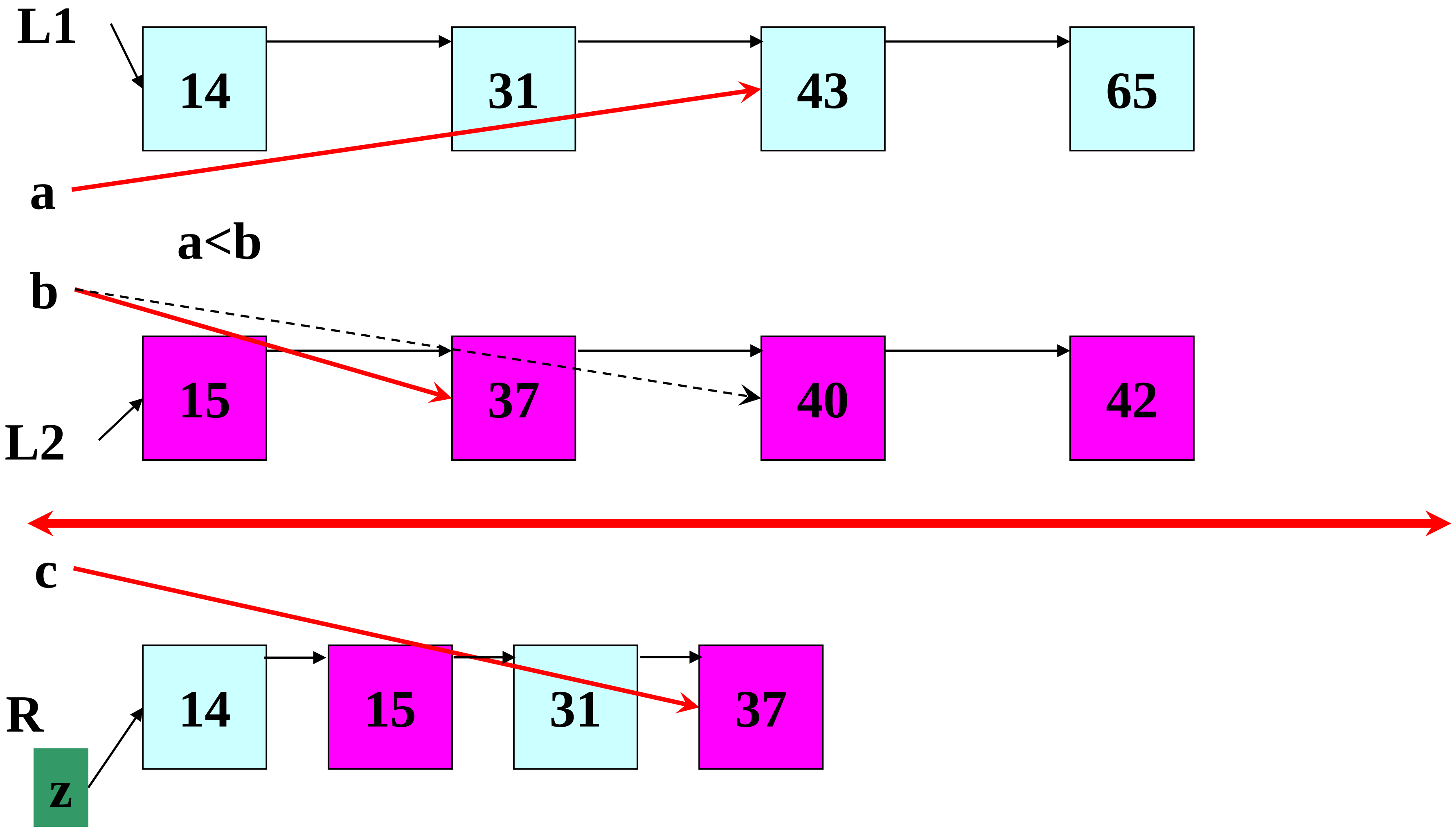
Merging two sorted lists: Example



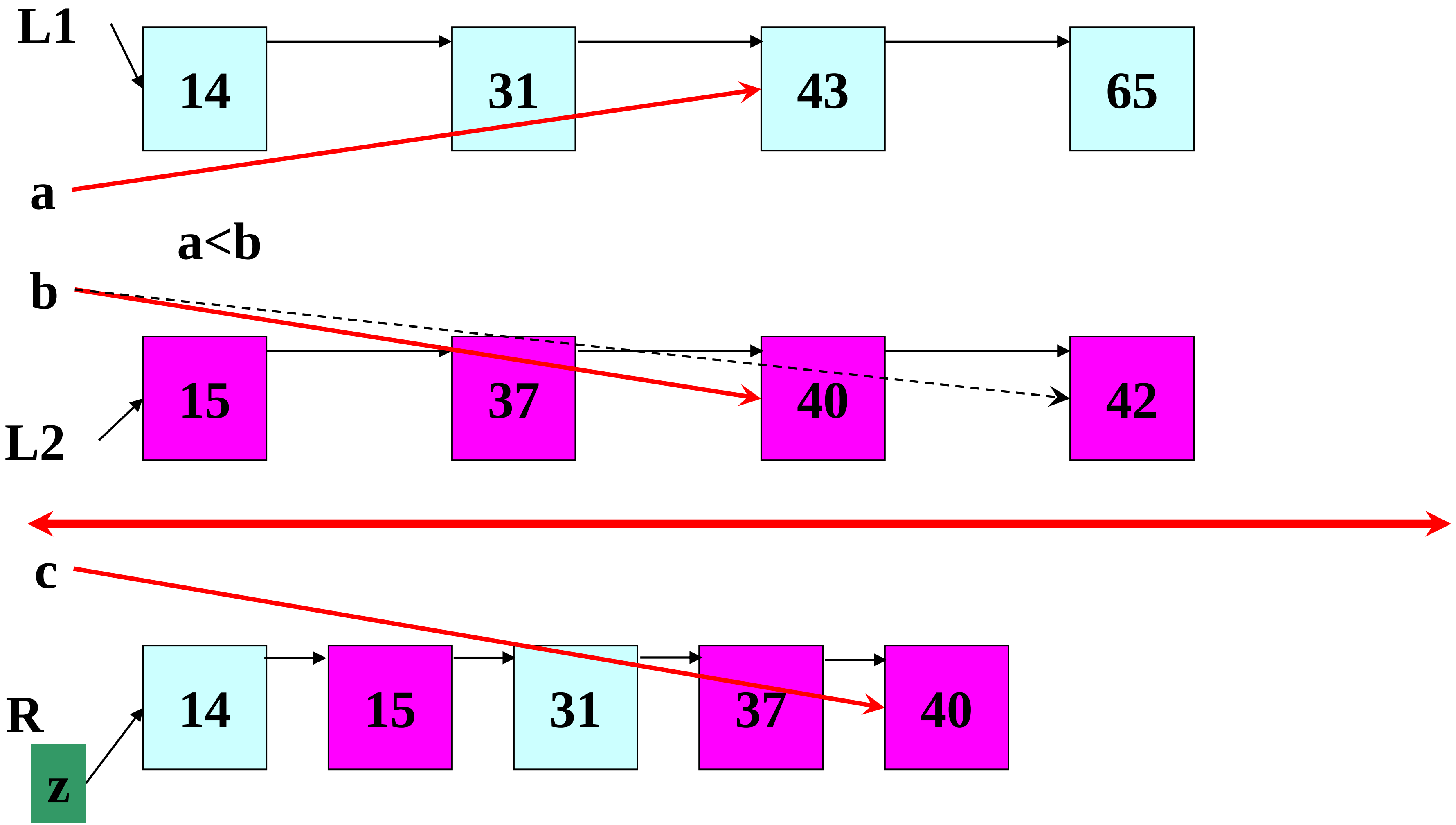
Merging two sorted lists: Example



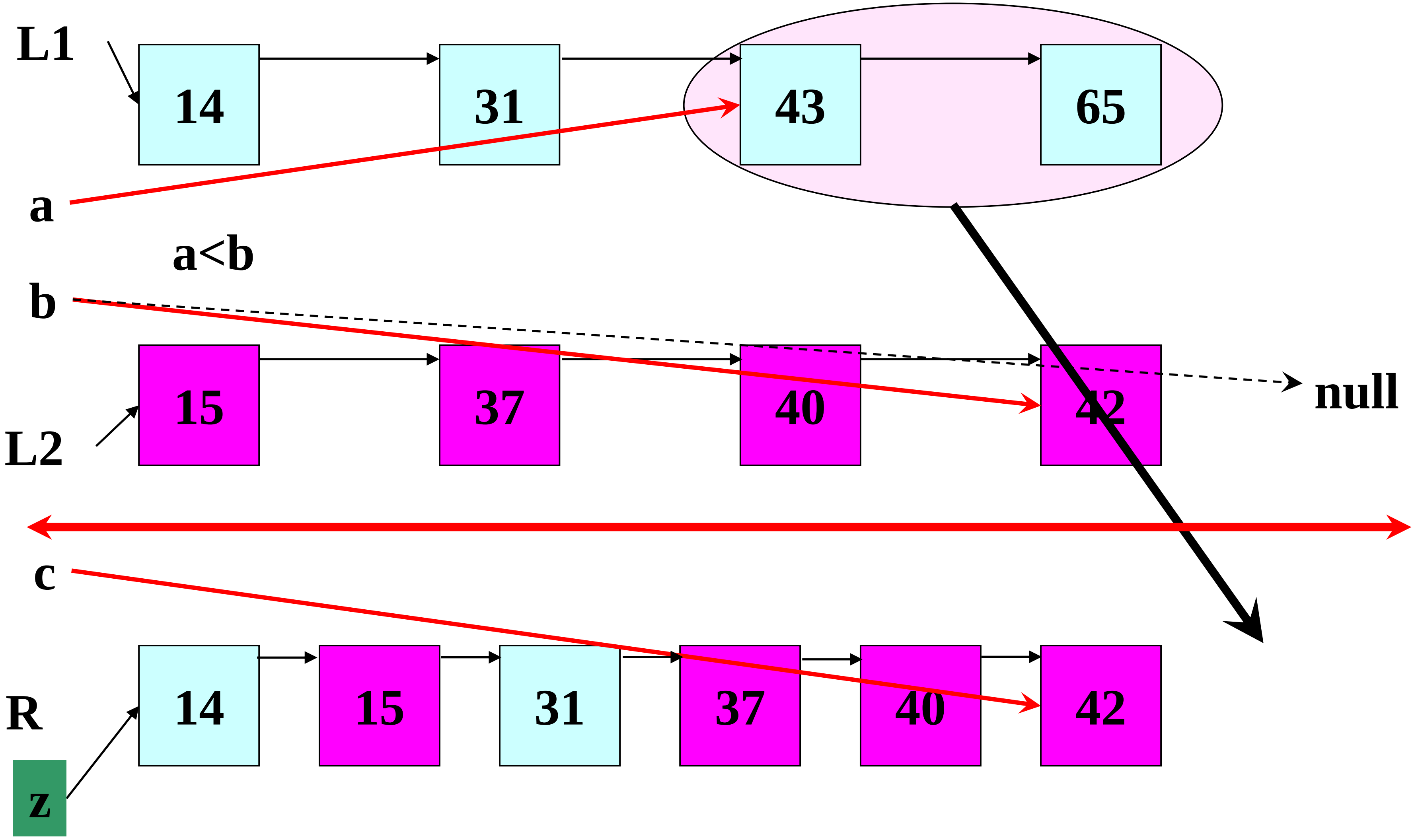
Merging two sorted lists: Example



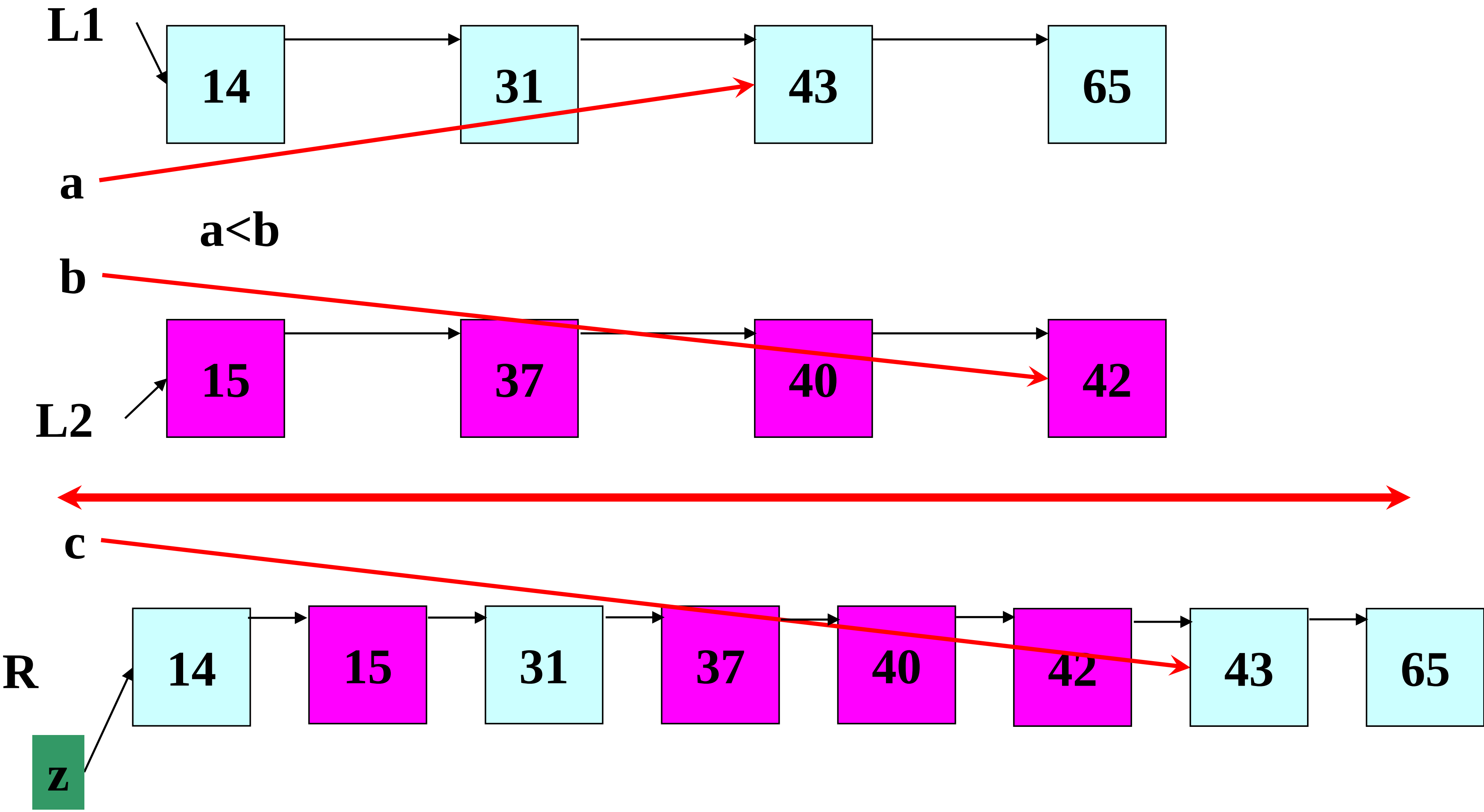
Merging two sorted lists: Example



Merging two sorted lists: Example



Merging two sorted lists: Example



L1

14 31 43 65

a

a<b

b

15 37 40 42

L2

c

R

14 15 31 37 40 42 43 65

z

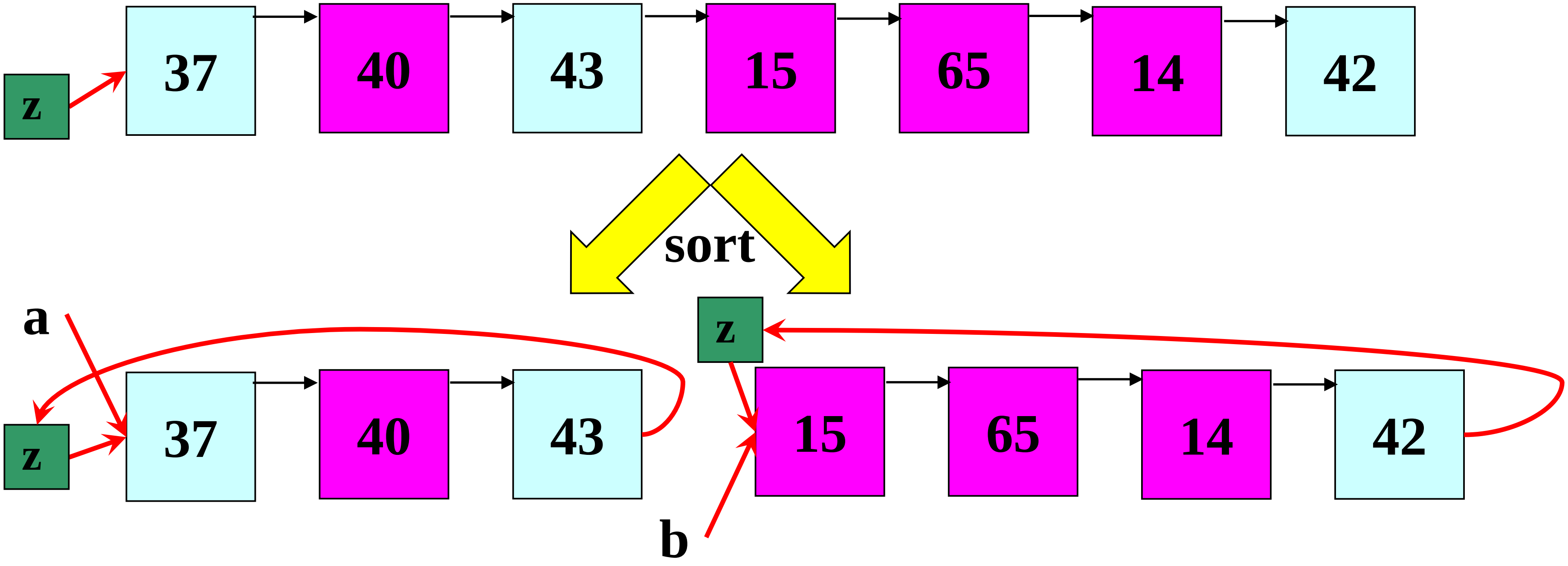
Mergesort: Idea

- This is a divide & conquer algorithm.
- Given the merge algorithm, we may sort a sequence of *n* keys by
  - dividing the sequence into (*divide* section)
    - 2 subsequences each of n/2 elements,
    - 4 subsequences each of n/4 elements,
    - 8 subsequences each of n/8 elements,
    - ...
    - n subsequences each of 1 element,
  - recursively sort (using mergesort) each consecutive pair of sequences, (*conquer* section)
  - merge the two sorted subsequences to obtain the sorted sequence (*combine* section)

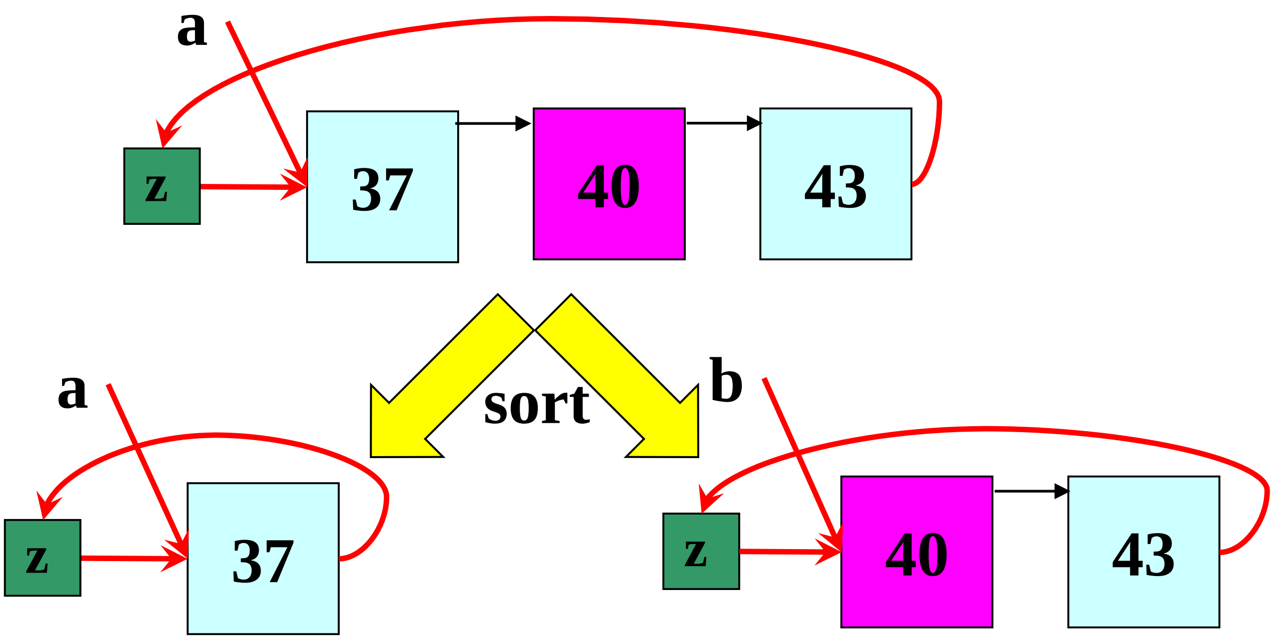
Mergesort: Algorithm

```
nodeptrtype sort (nodeptrtype c; int N)
{
   nodeptrtype a,b; int i;
   if (c->next == z) return c;
   else {
               a=c;
               for (i=1;i<N/2;i++;c=c->next);
               b=c->next; c->next=z;
               return merge(sort(a,N/2), sort(b,N-(N/2))
   }
}
```

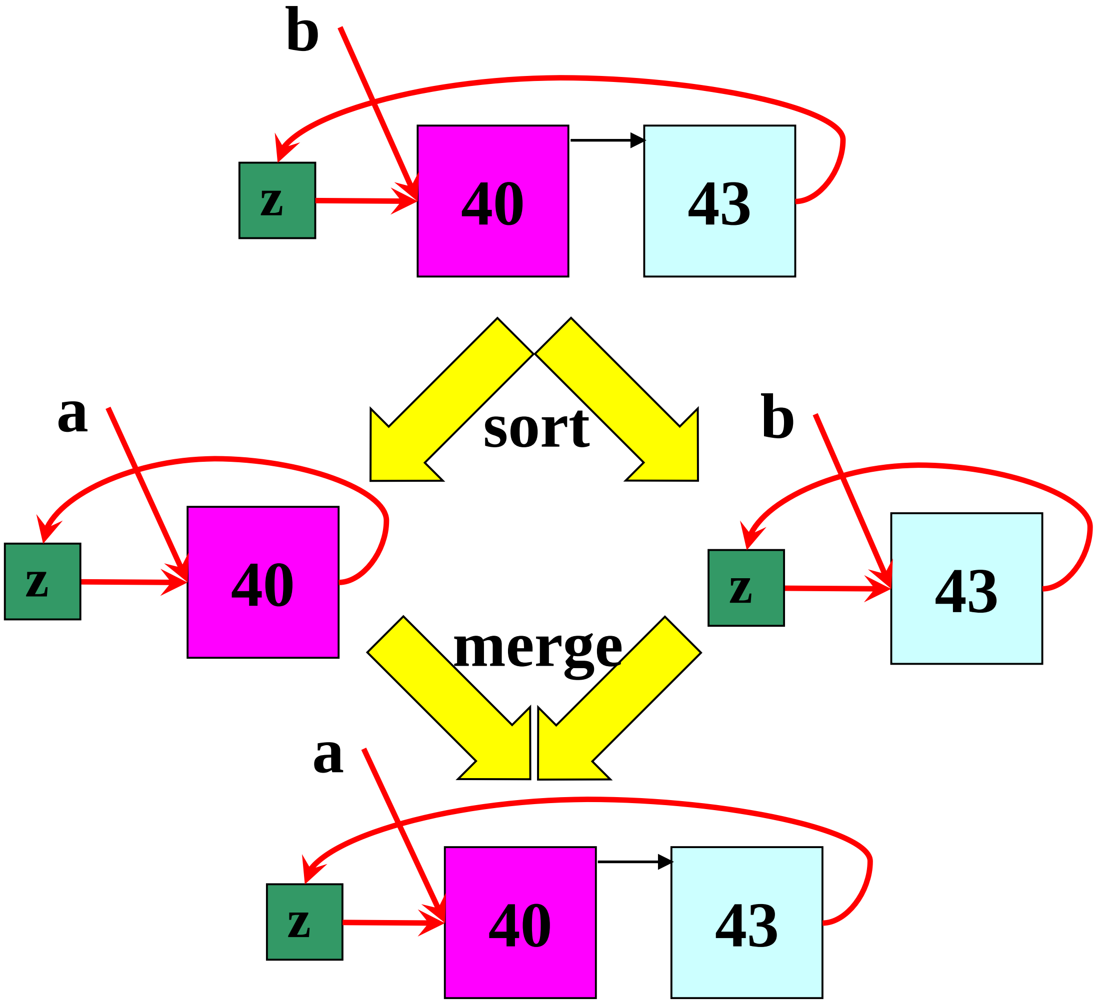
Mergesort: Example



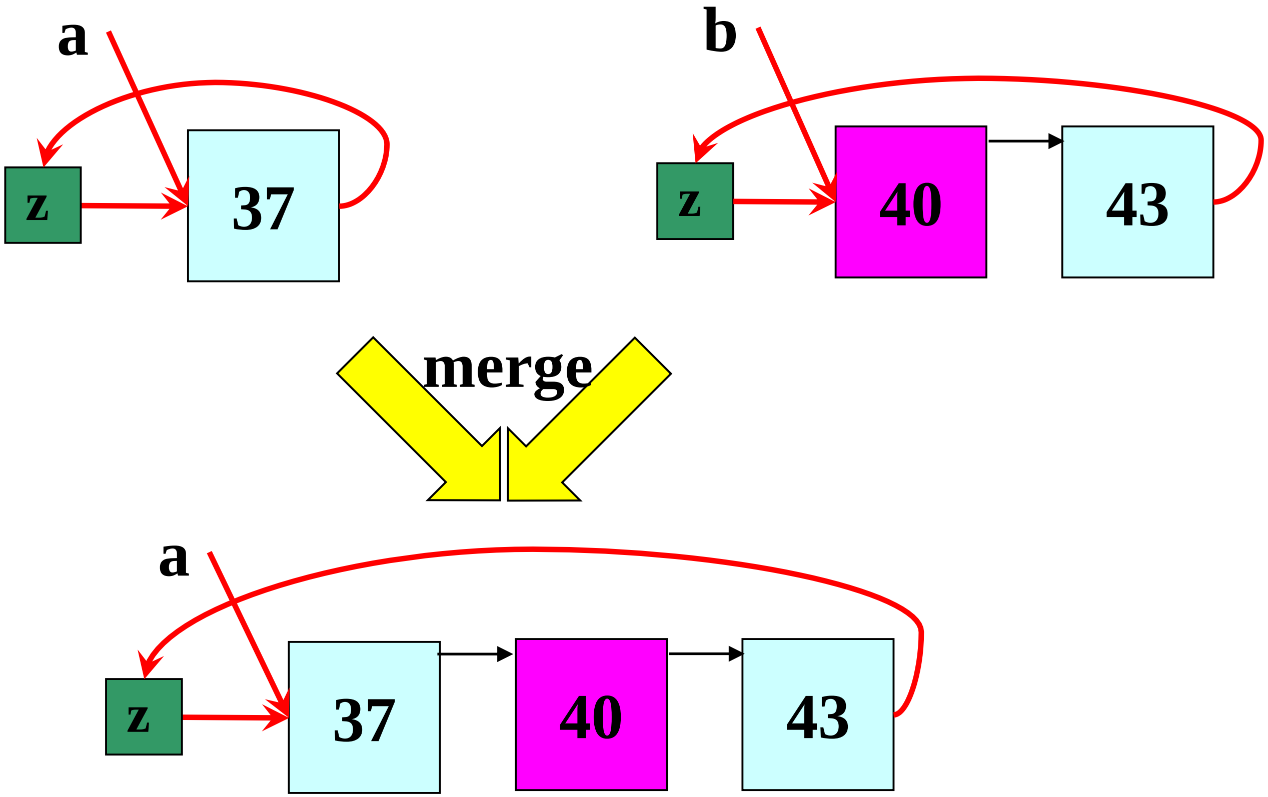
Mergesort: Example



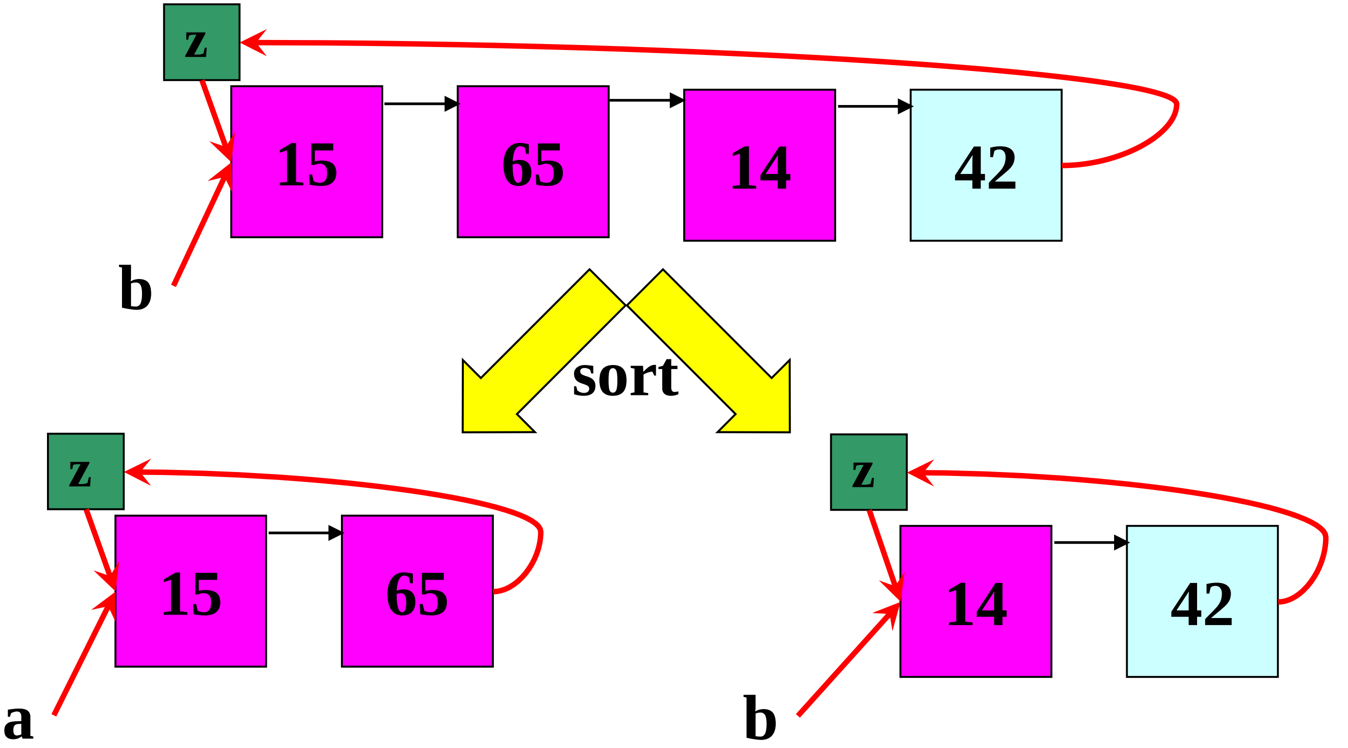
Mergesort: Example



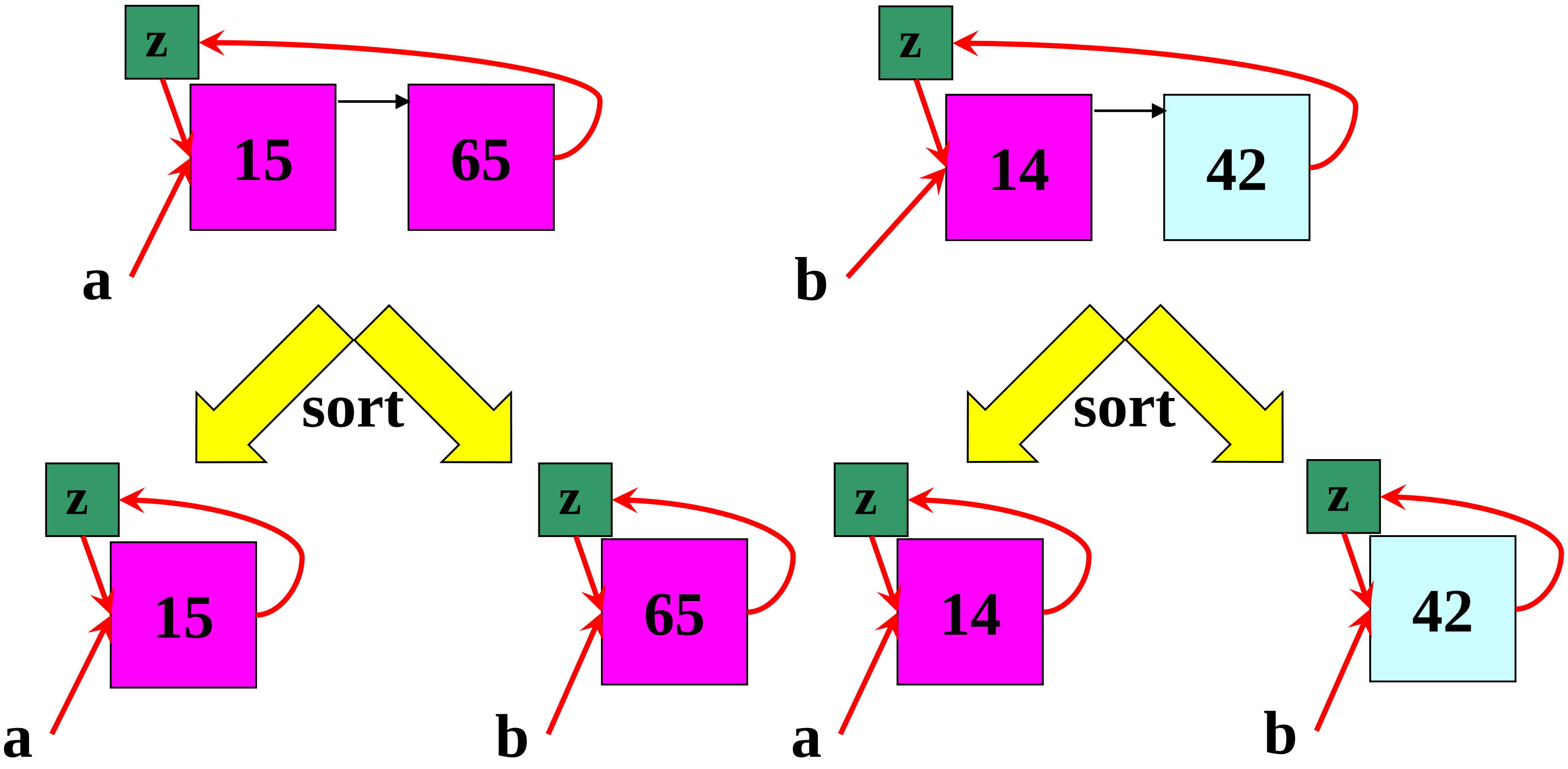
Mergesort: Example



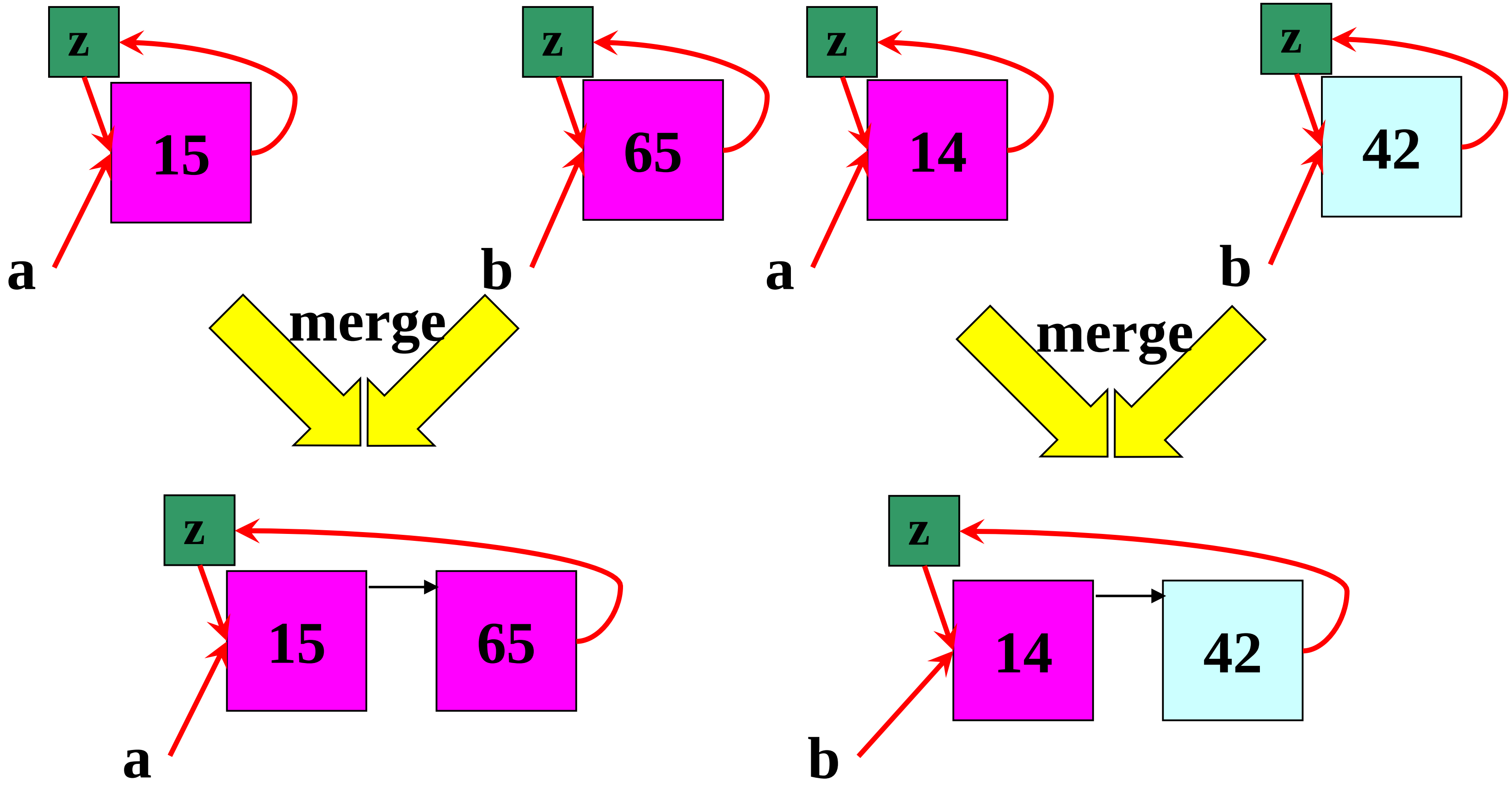
Mergesort: Example



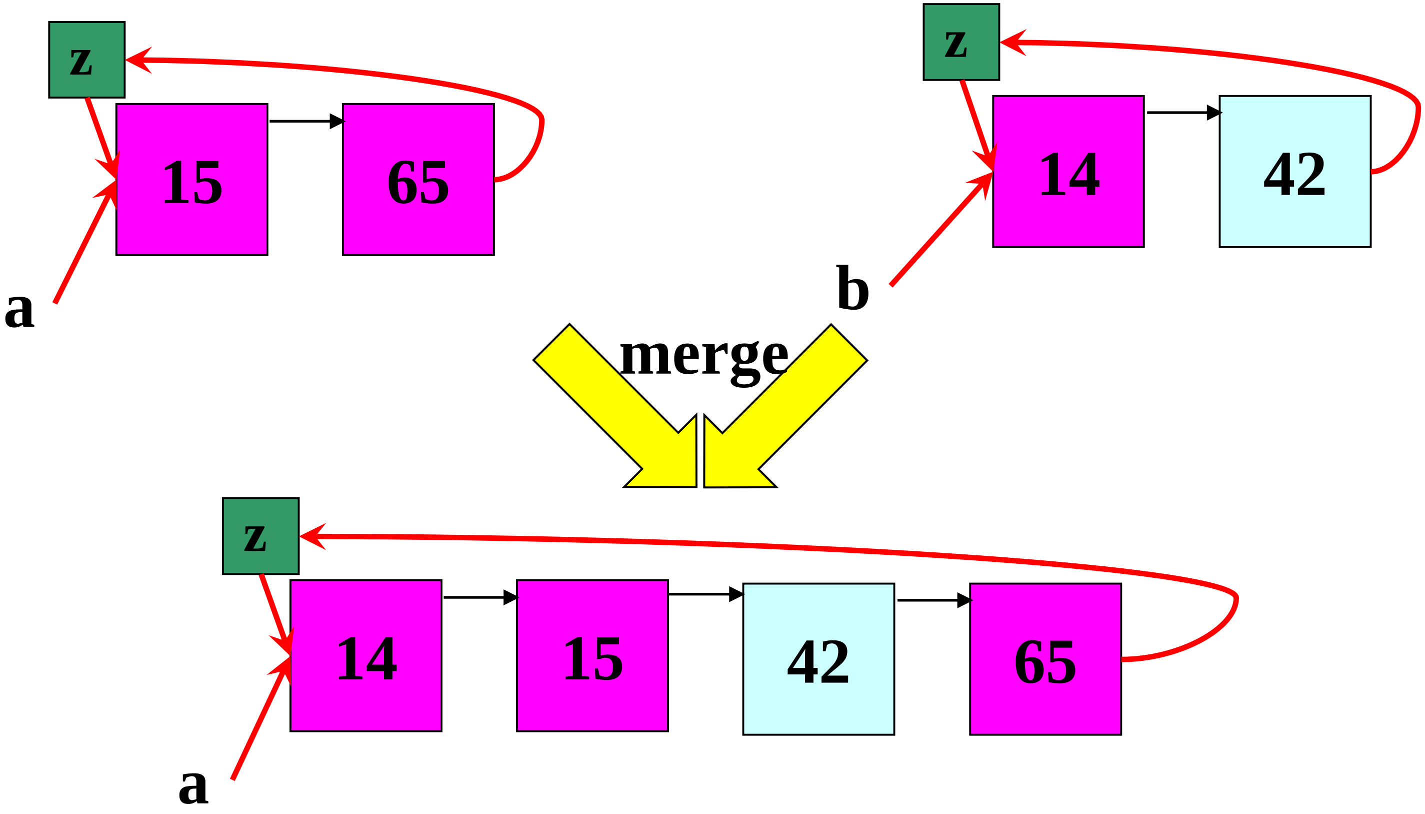
Mergesort: Example



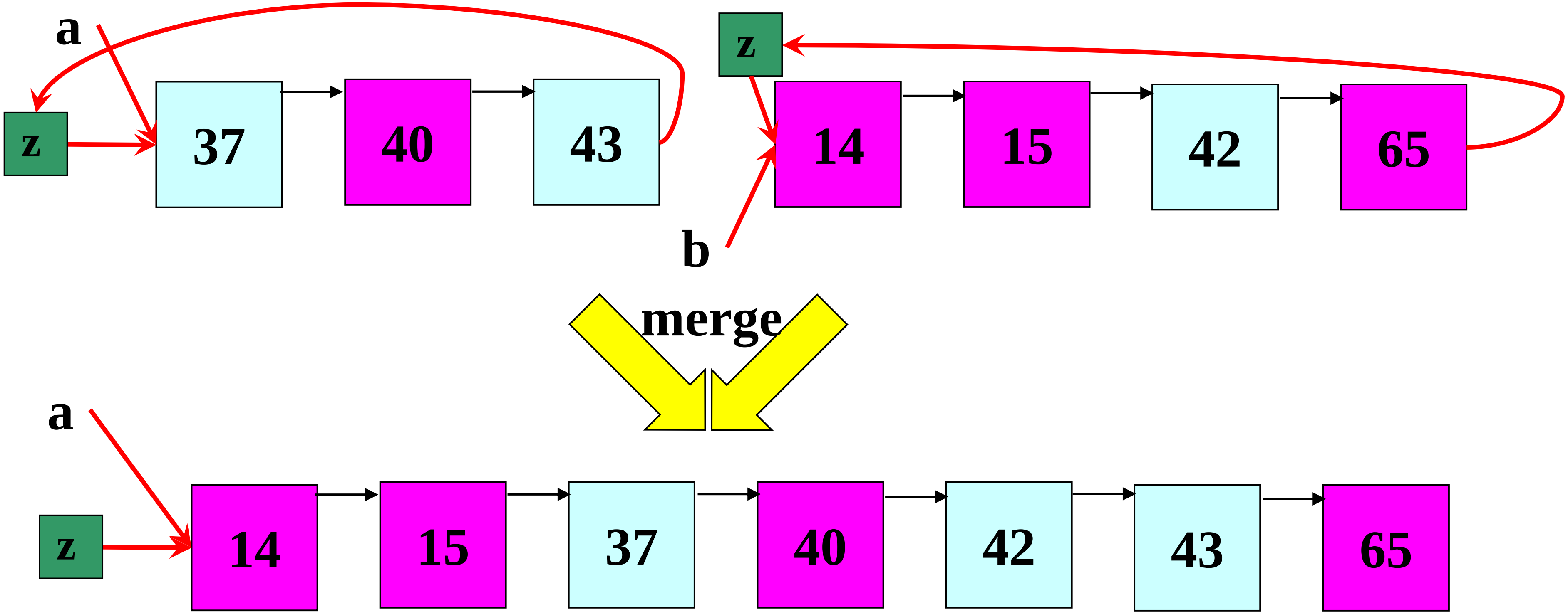
Mergesort: Example



Mergesort: Example



Mergesort: Example



Algorithm Analysis of Mergesort

- The maximum number of comparisons per merging of two sequences for a total of *n* elements is

$$n-1=\Theta(n).$$

- How many partitions of how many elements are merged?
  - Merging once n/2 keys with n/2 keys = n/2+n/2-1
  - Merging twice n/4 keys with n/4 keys = 2(n/4+n/4-1)
  - Merging 4 times n/8 keys with n/8 keys = 4(n/8+n/8-1)
  - ...
  - Merging n/2 times one key with another key = n/2.

Algorithm Analysis of Mergesort

$$t(n) = \sum_{i=0}^{\lfloor \lg n \rfloor - 1} n - 2^i = \sum_{i=0}^{\lfloor \lg n \rfloor - 1} n - \underbrace{\sum_{i=0}^{\lfloor \lg n \rfloor - 1} 2^i}_{\frac{1-2^{\lfloor \lg n \rfloor}}{1-2} = 2^{\lfloor \lg n \rfloor} - 1}$$

$$t(n) = \underbrace{n \lfloor \lg n \rfloor}_{O(n \lg n)} - \underbrace{(\underbrace{2^{\lfloor \lg n \rfloor}}_{\leq n} - 1)}_{O(n)}$$

$$\Rightarrow t(n) \in \Theta\left(n \lg n\right)$$

Hence, running time of mergesort is $\Theta(n\ lgn)$.

Quicksort: Idea

- Another popular divide & conquer sort algorithm.
- The idea here is that, at every call of a specific *partitioning* algorithm, a selected key (pivot) is placed at its correct position, and all keys less are moved to left while those greater than that key are carried over to the right of the key.
- The original partitioning algorithm used in quicksort is devised by C.A.R. Hoare.

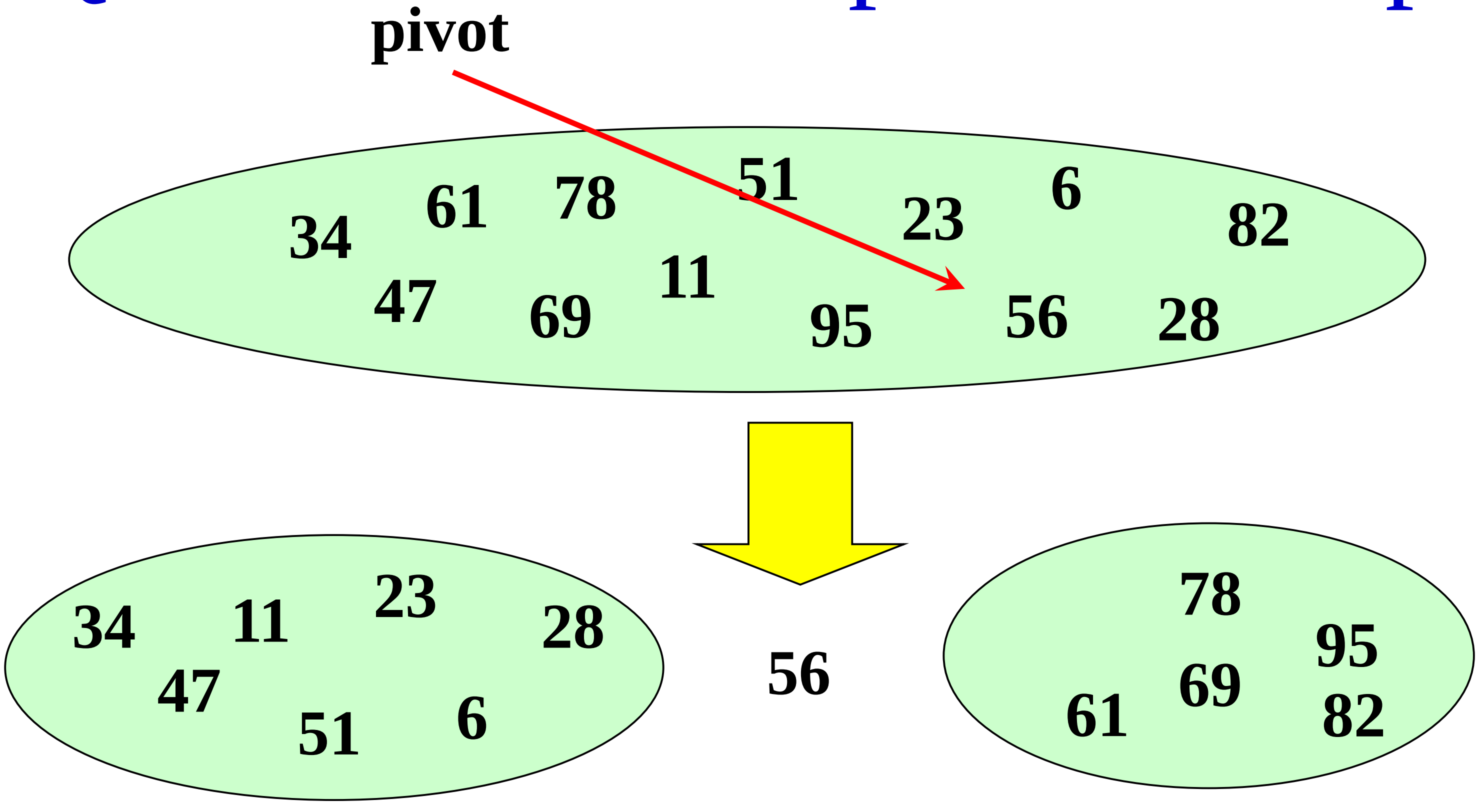
Quicksort: Idea

- *After partitioning*, the original sequence is divided into *three subsequences*:
  1. Numbers to the *left of pivot* (still needs sorting)
  2. *Pivot* (correctly placed)
  3. Numbers to the *right of pivot* (still needs sorting)
- Recursively calling *quicksort* function for the left and right subsequences, we sort the entire array.

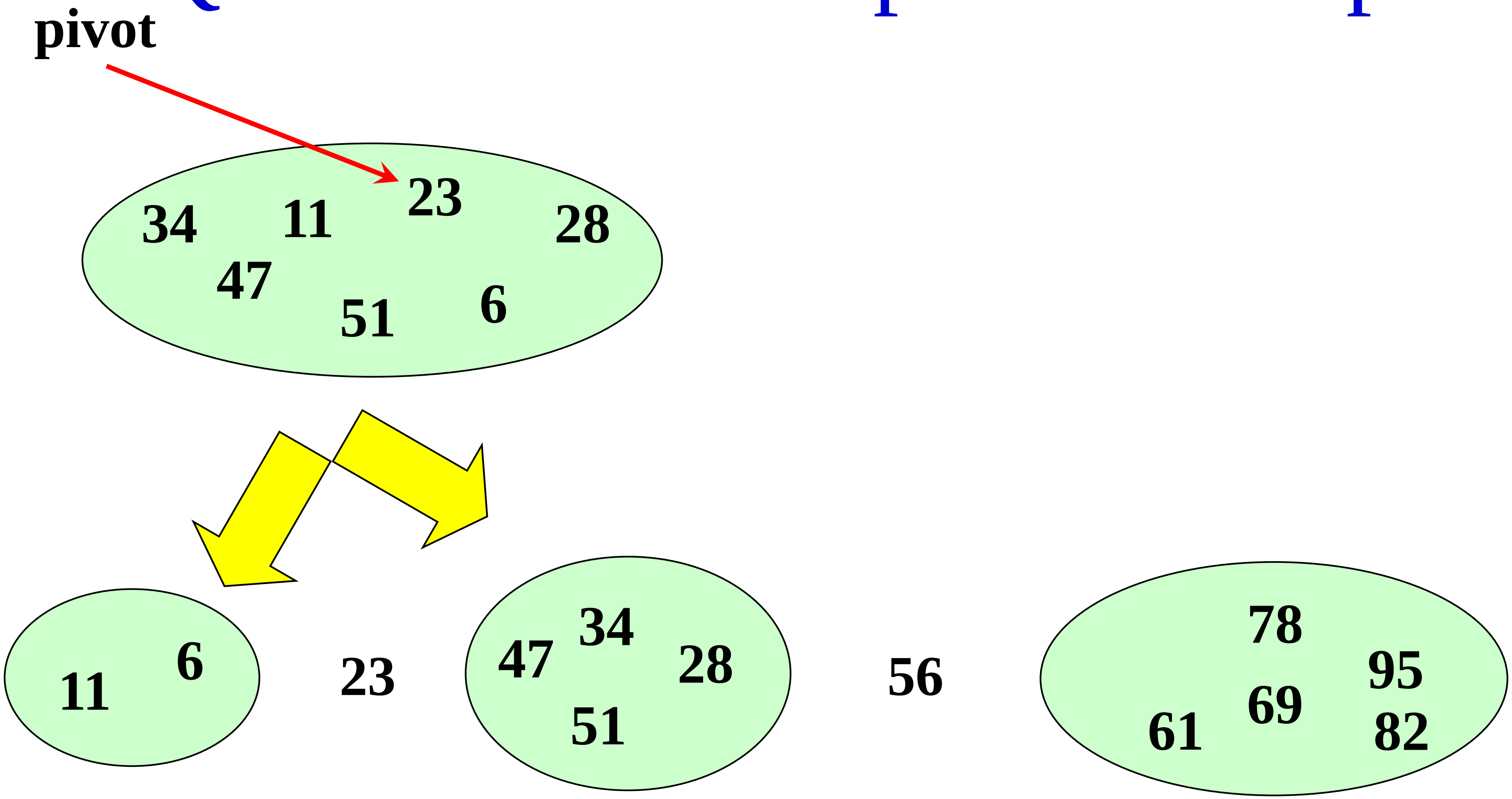
Quicksort: Algorithm

```
Quicksort(A,l,r)
{ //A is the array holding the key sequence
   // l,r are the lowest and highest index values of
   // the key sequence in respective order.
   // m is the index value of the pivot.
   if (l<r)
         m=partition(A,l,r);
         Quicksort(A,l,m-1);
         Quicksort(A,m+1,r);
}
```

Quicksort: Conceptual Example



Quicksort: Conceptual Example



pivot

34 11 23 28 47 51 6

11 6

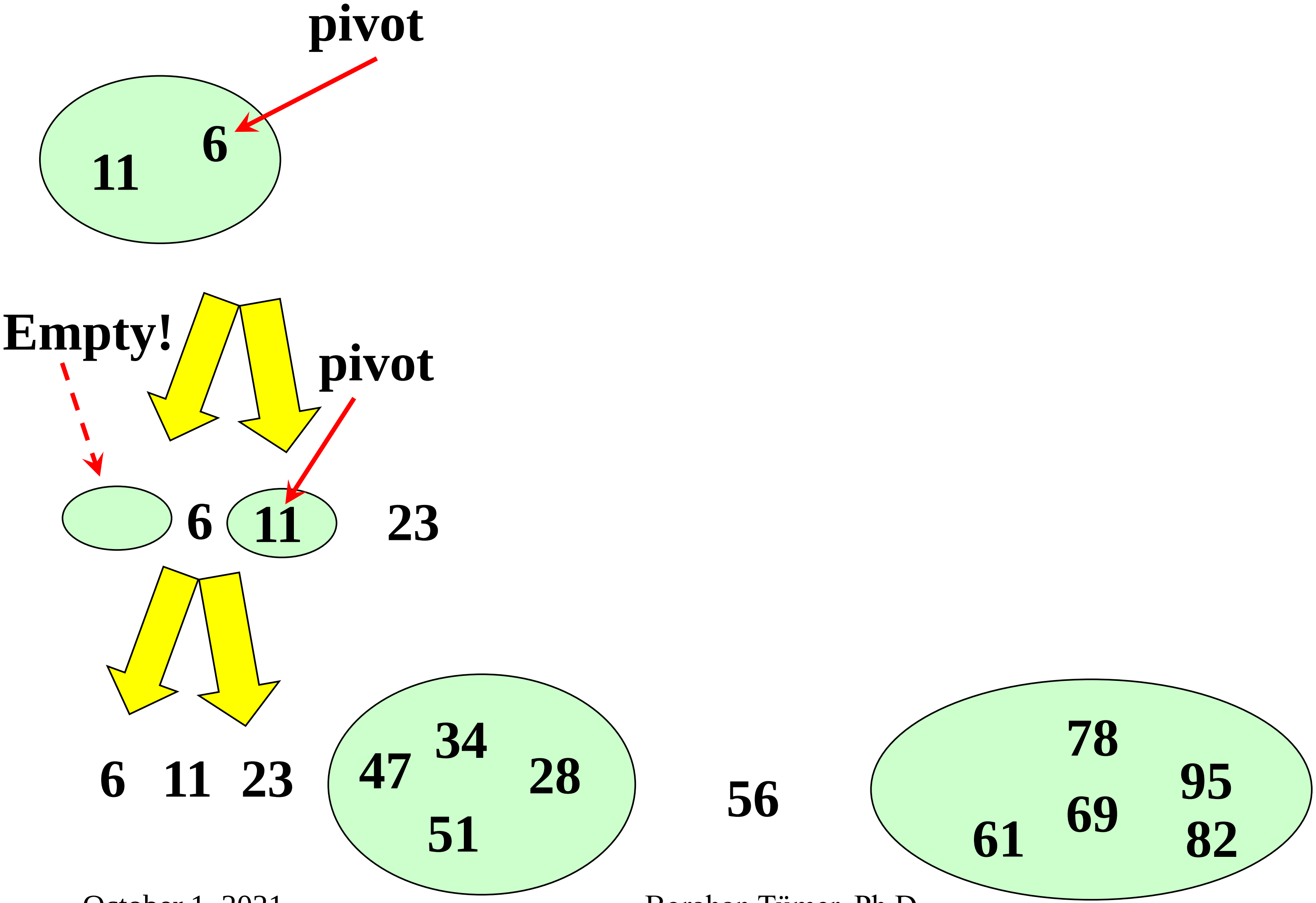
23

47 34 28 51

56

78 95 69 61 82

Quicksort: Conceptual Example



pivot

11 6

Empty!

pivot

6 11 23

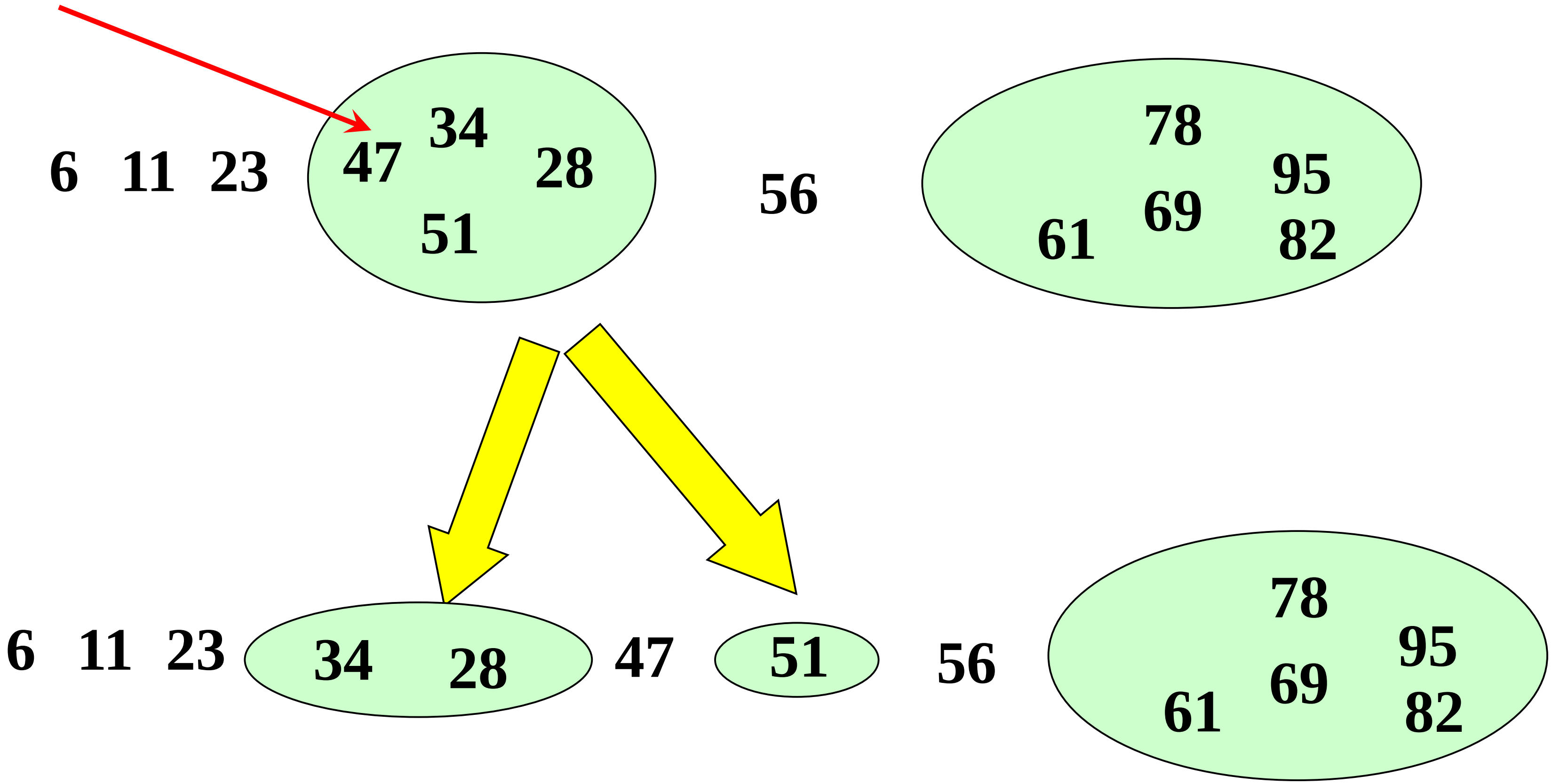
6 11 23

47 34 28 51

56

78 95 69 61 82

Quicksort: Conceptual Example

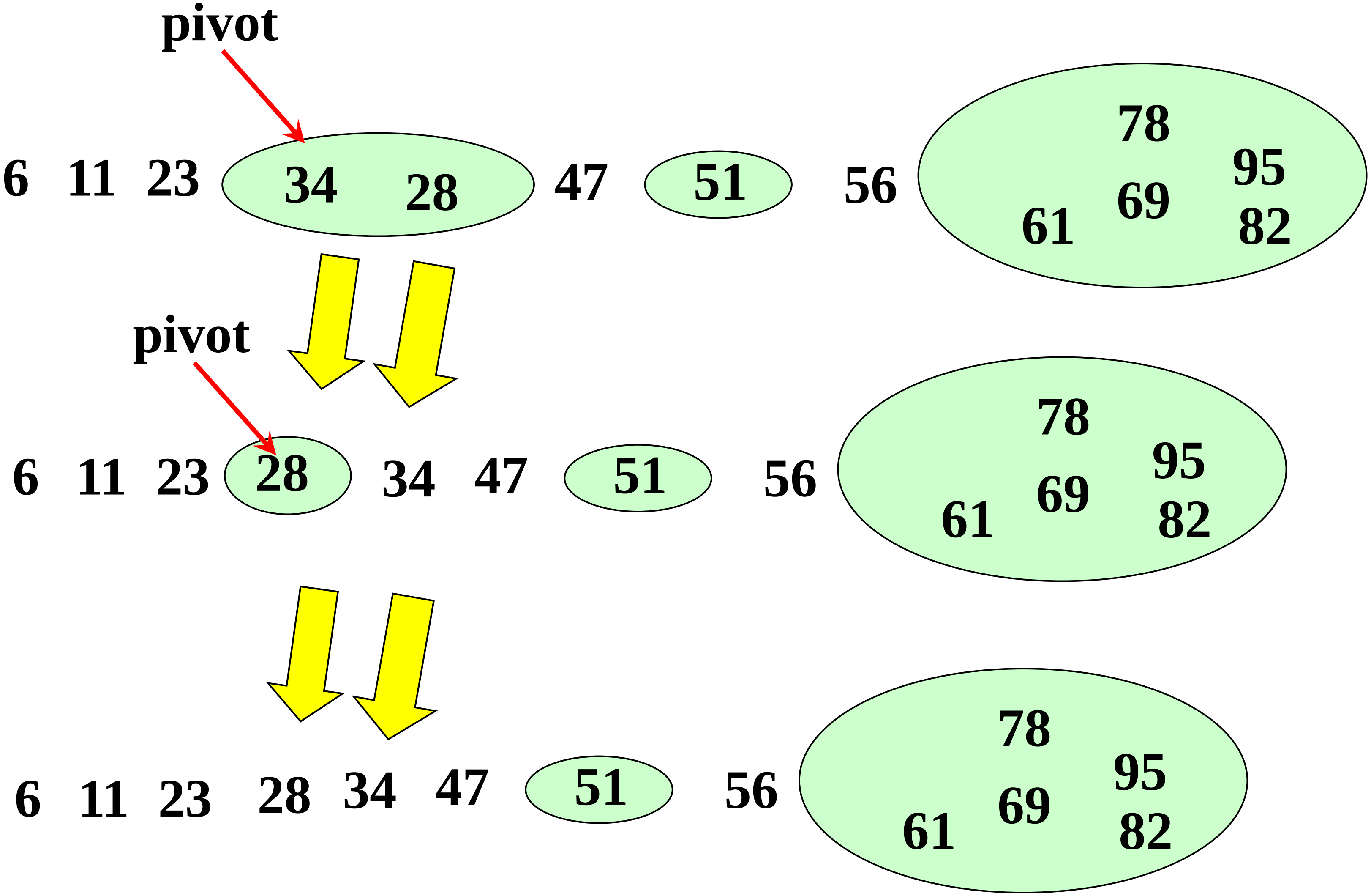


pivot

6 11 23 47 34 28 51 56 78 95 69 61 82

6 11 23 34 28 47 51 56 78 95 69 61 82

Quicksort: Conceptual Example



pivot

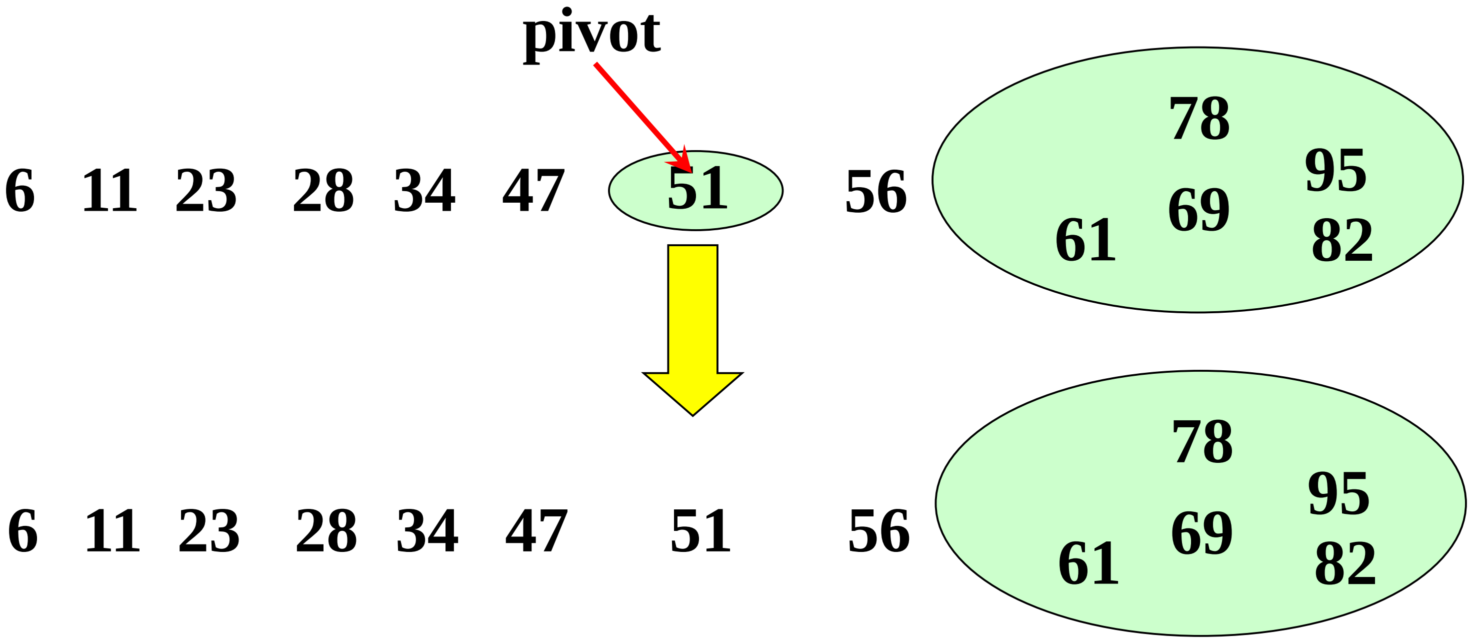
6 11 23 34 28 47 51 56 78 95 61 69 82

pivot

6 11 23 28 34 47 51 56 78 95 61 69 82

6 11 23 28 34 47 51 56 78 95 61 69 82

Quicksort: Conceptual Example



pivot

6 11 23 28 34 47 51 56

78 95 69 61 82

6 11 23 28 34 47 51 56

78 95 69 61 82

The right subsequence of the original sequence is sorted similarly.

Partitioning*: Idea

- The idea here is to group keys so that
  - *all keys less than pivot are at left of pivot*, and
  - *all greater keys are at the right of pivot*,
- The pivot is the last key of the sequence.
- At any time, the rest of the sequence consists of three subsequences:
  1. the left subsequence holds keys less than the pivot;
  2. the middle subsequence holds keys greater than the pivot;
  3. the right subsequence holds keys not yet sorted;
- Initially, the left and middle subsequences are empty. Any time a key in the right subsequence is processed (i.e., compared with pivot), it is purged from the right and placed to the left or middle sequence.

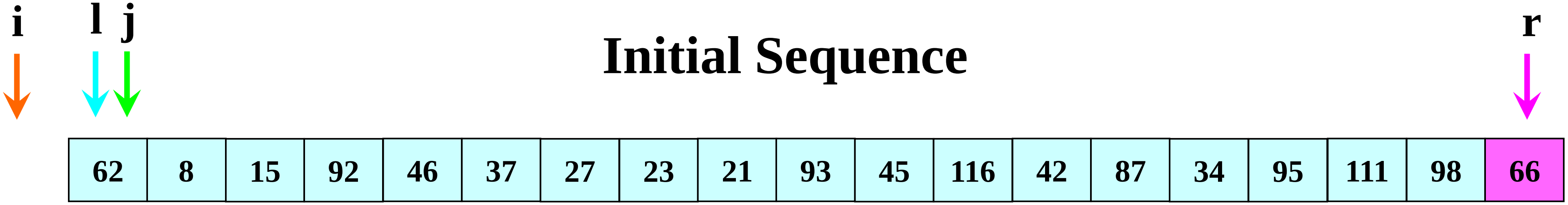
***main reference for this algorithm is [1]**

Partitioning: Algorithm

```
Partition(A,l,r)
{
    x=A[r];  //   x is the pivot!
    i=l-1;
    for (j=l;j<r; j++) {
        if (A[j]<=x) {i++; swap(A[i],A[j])}
    }
    swap(A[i+1],A[r]);
    return i+1;
}
```

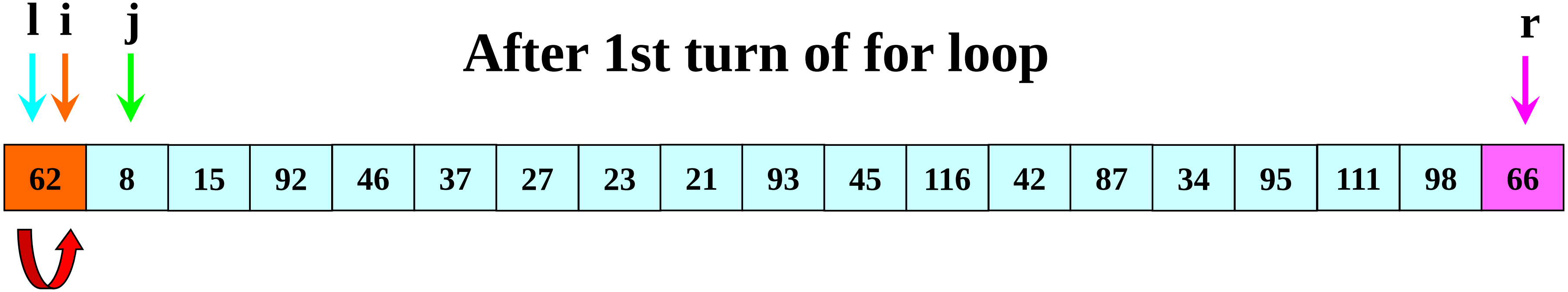
***main reference for this algorithm is [1]**

Partitioning: Example



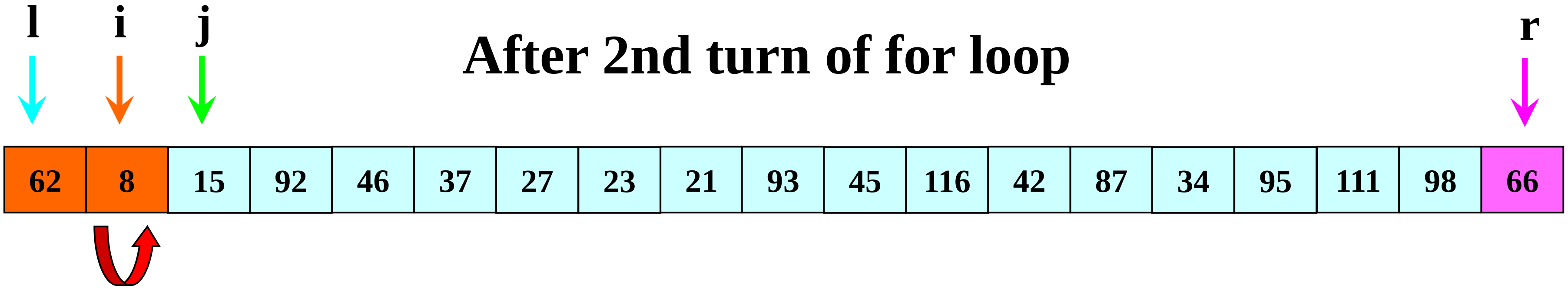
Initial Sequence

62	8	15	92	46	37	27	23	21	93	45	116	42	87	34	95	111	98	66



After 1st turn of for loop

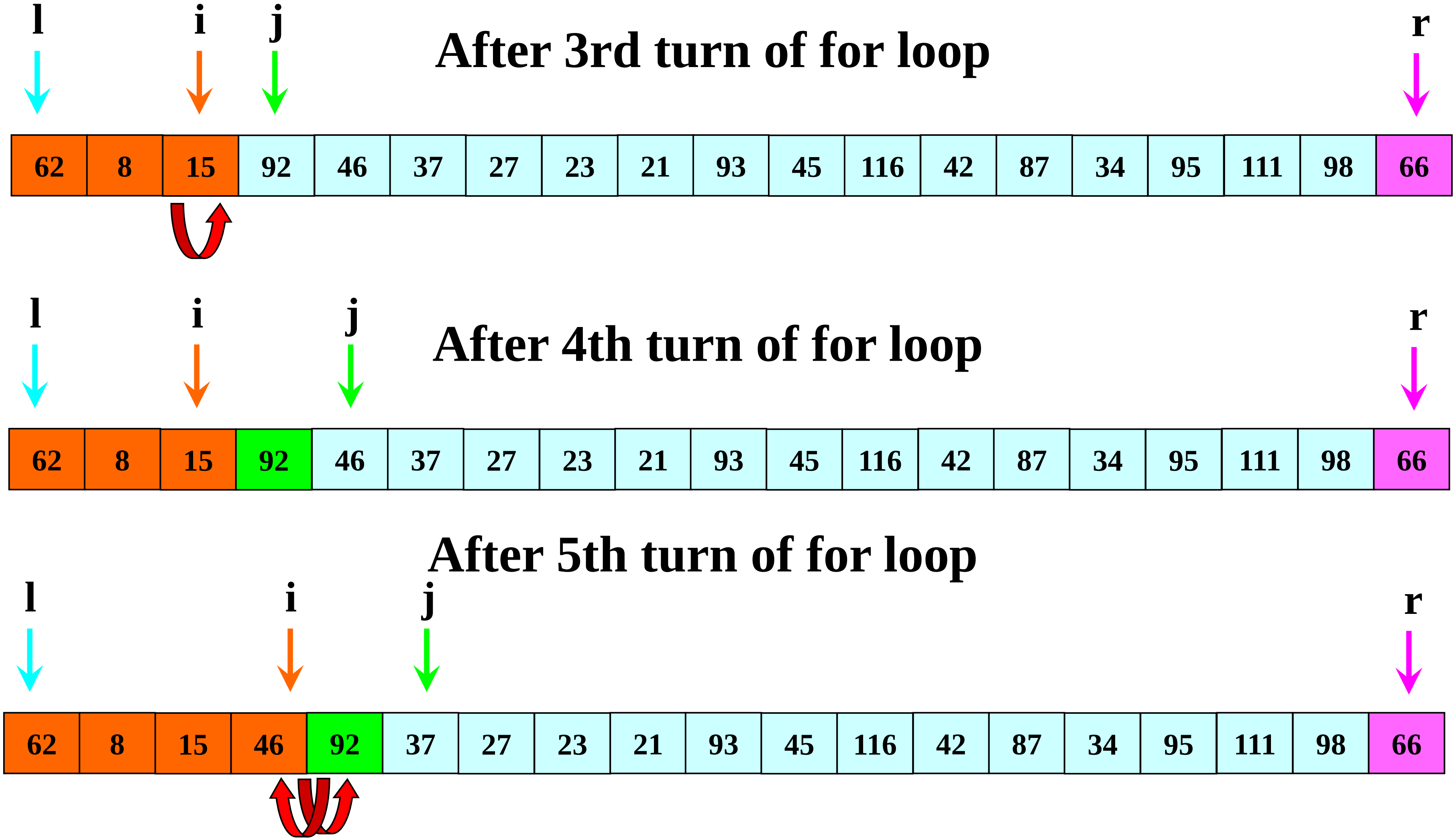
62	8	15	92	46	37	27	23	21	93	45	116	42	87	34	95	111	98	66



After 2nd turn of for loop

62	8	15	92	46	37	27	23	21	93	45	116	42	87	34	95	111	98	66

Partitioning: Example



After 3rd turn of for loop

62	8	15	92	46	37	27	23	21	93	45	116	42	87	34	95	111	98	66

After 4th turn of for loop

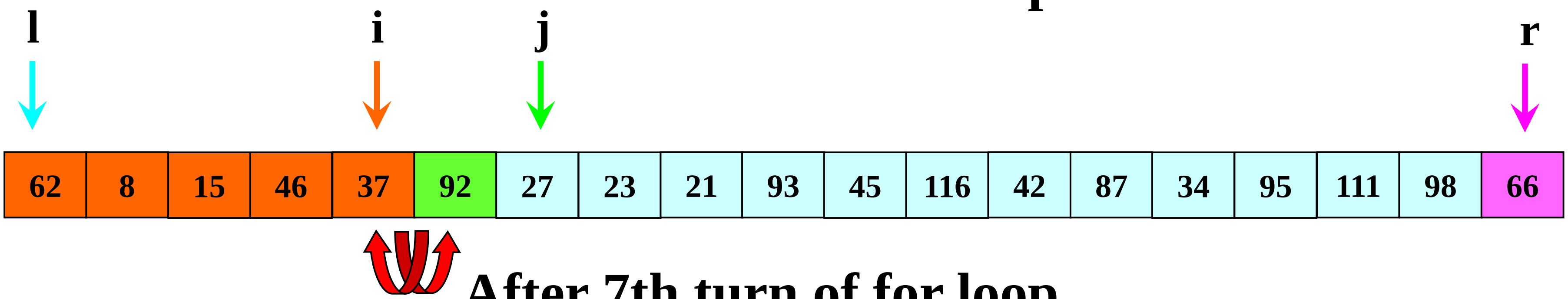
62	8	15	92	46	37	27	23	21	93	45	116	42	87	34	95	111	98	66

After 5th turn of for loop

62	8	15	46	92	37	27	23	21	93	45	116	42	87	34	95	111	98	66

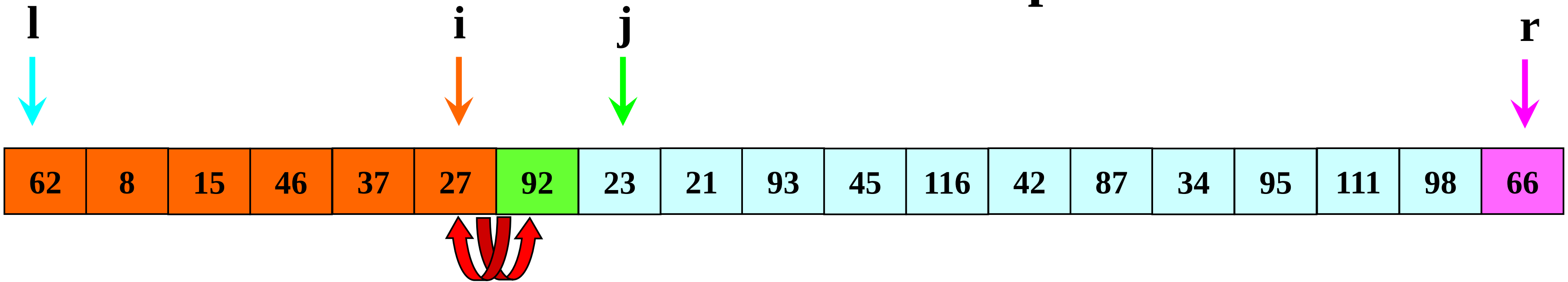
Partitioning: Example

After 6th turn of for loop



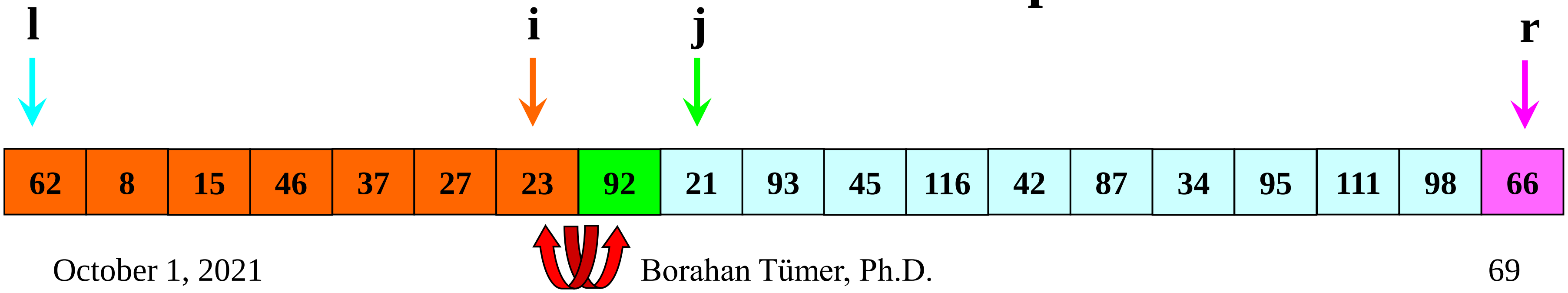
l				i		j												r
62	8	15	46	37	92	27	23	21	93	45	116	42	87	34	95	111	98	66

After 7th turn of for loop



l					i		j											r
62	8	15	46	37	27	92	23	21	93	45	116	42	87	34	95	111	98	66

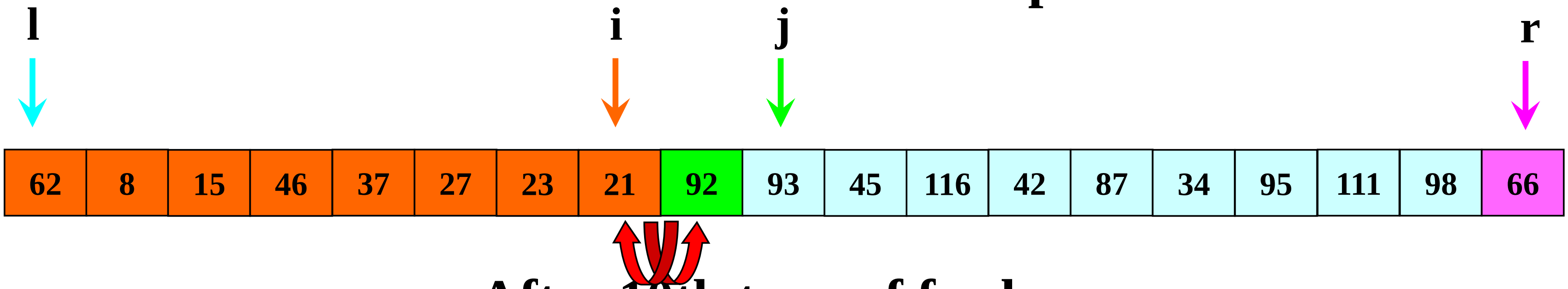
After 8th turn of for loop



l						i		j										r
62	8	15	46	37	27	23	92	21	93	45	116	42	87	34	95	111	98	66

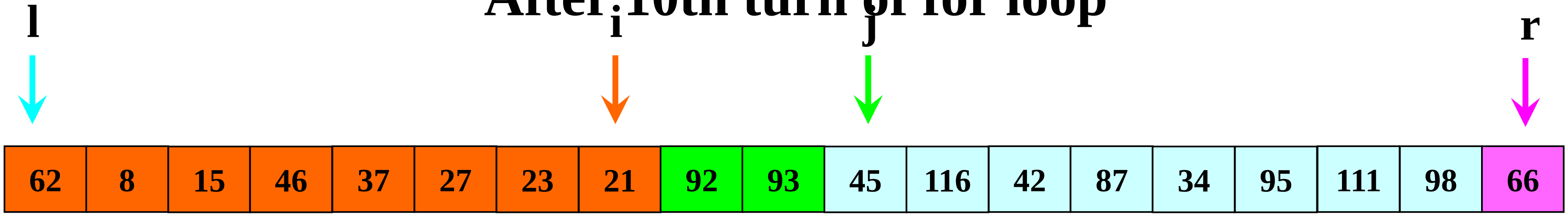
Partitioning: Example

After 9th turn of for loop



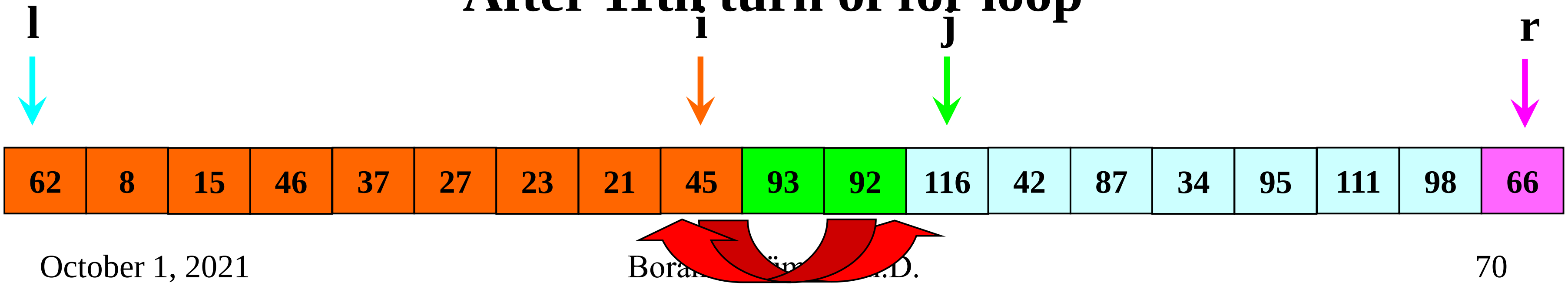
62	8	15	46	37	27	23	21	92	93	45	116	42	87	34	95	111	98	66

After 10th turn of for loop



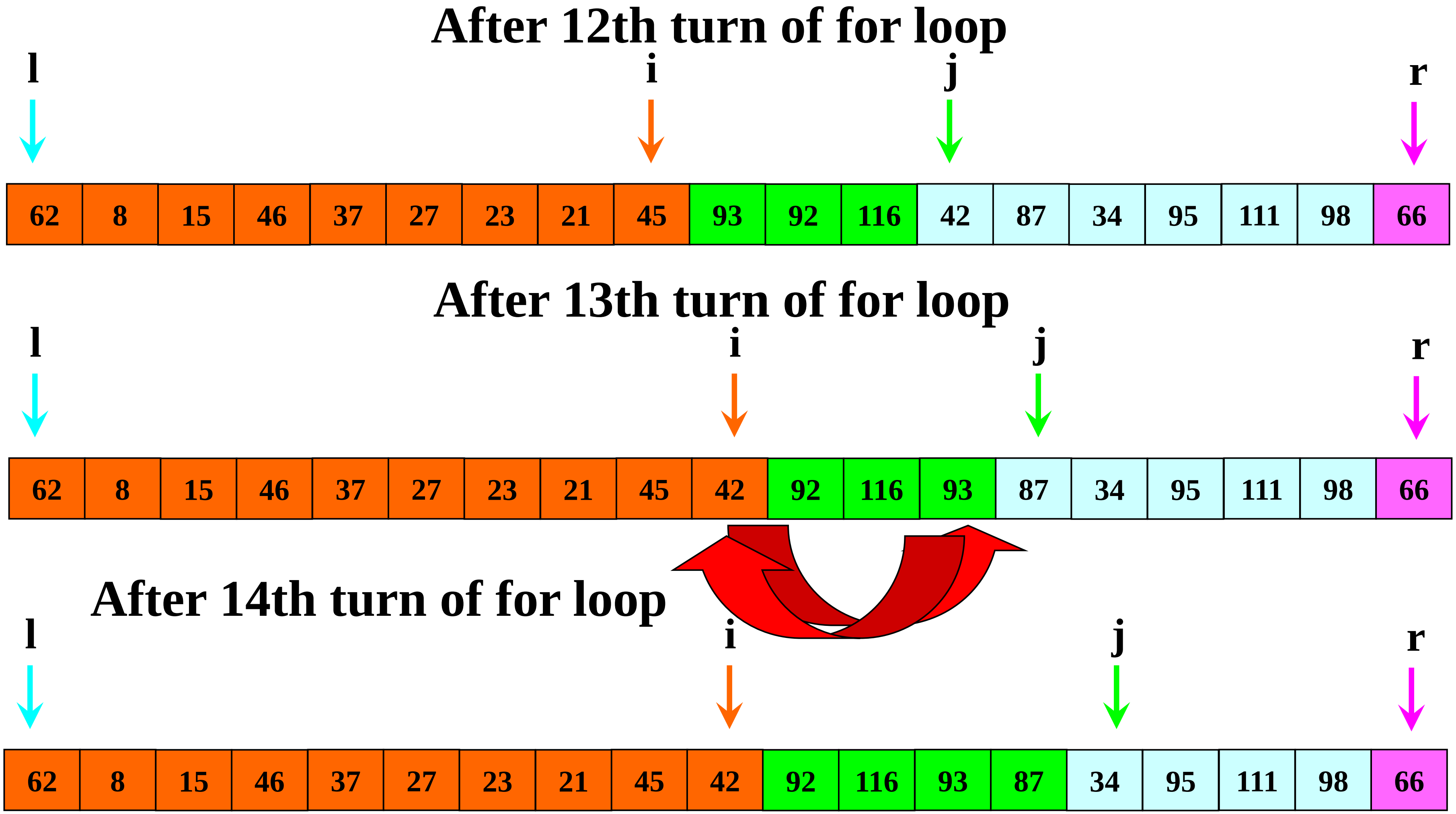
62	8	15	46	37	27	23	21	92	93	45	116	42	87	34	95	111	98	66

After 11th turn of for loop



62	8	15	46	37	27	23	21	45	93	92	116	42	87	34	95	111	98	66

Partitioning: Example



After 12th turn of for loop

l								i				j						r
62	8	15	46	37	27	23	21	45	93	92	116	42	87	34	95	111	98	66

After 13th turn of for loop

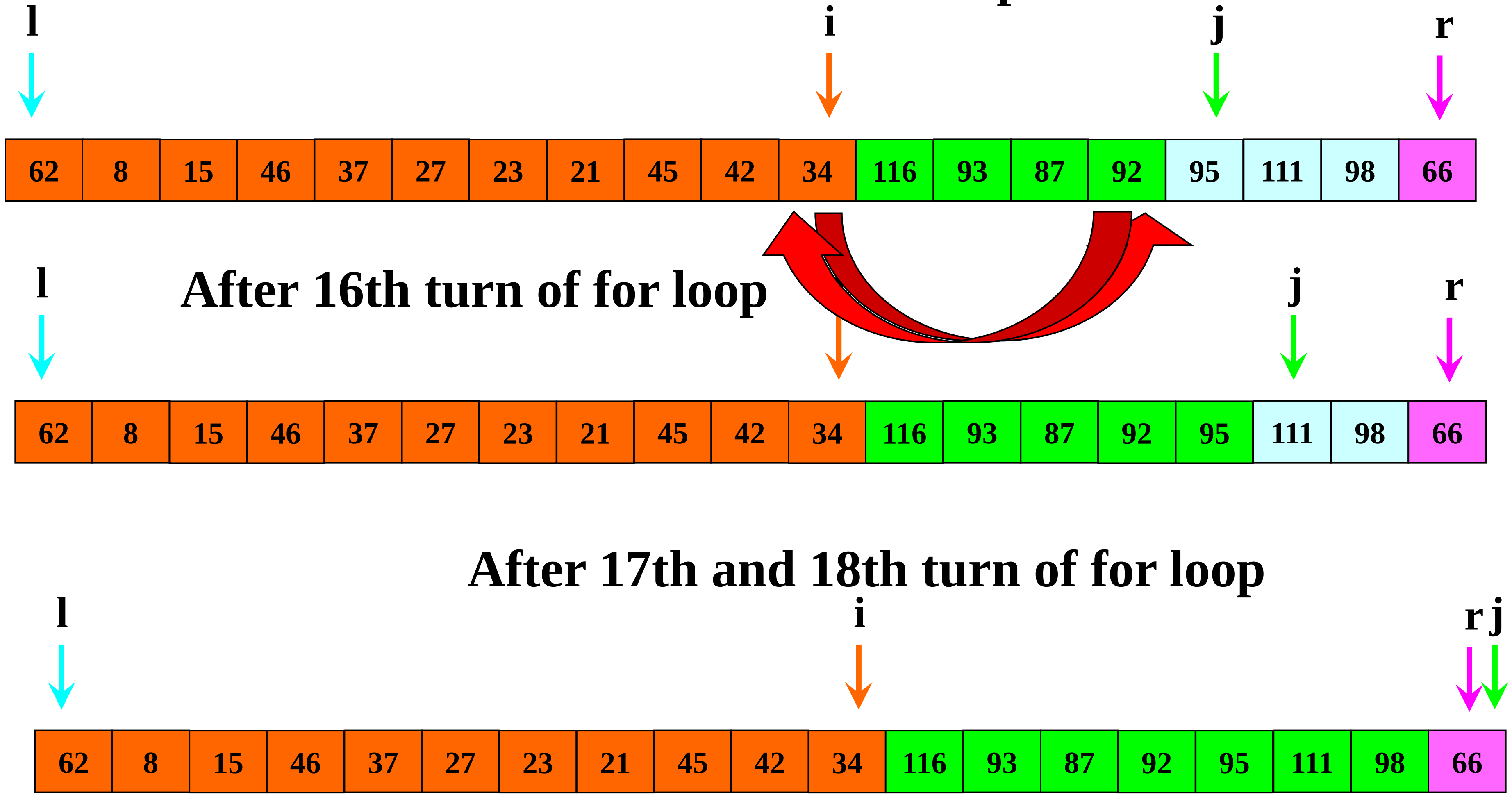
l									i				j					r
62	8	15	46	37	27	23	21	45	42	92	116	93	87	34	95	111	98	66

After 14th turn of for loop

l									i					j				r
62	8	15	46	37	27	23	21	45	42	92	116	93	87	34	95	111	98	66

Partitioning: Example

After 15th turn of for loop



62	8	15	46	37	27	23	21	45	42	34	116	93	87	92	95	111	98	66

After 16th turn of for loop

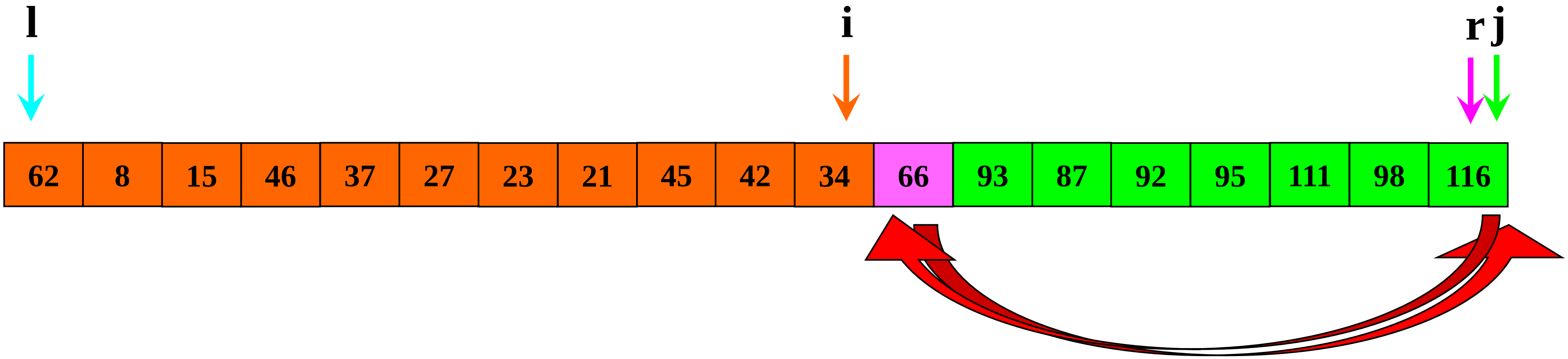
62	8	15	46	37	27	23	21	45	42	34	116	93	87	92	95	111	98	66

After 17th and 18th turn of for loop

62	8	15	46	37	27	23	21	45	42	34	116	93	87	92	95	111	98	66

Partitioning: Example

After execution of return statement



Hoare’s Partitioning: Idea

- The idea is to simply check the sequence starting at both ends to see that
  - no keys less than pivot should remain to right of pivot, and
  - no greater keys to left,
- If any two such keys exist, they are swapped to have them at their correct sides.

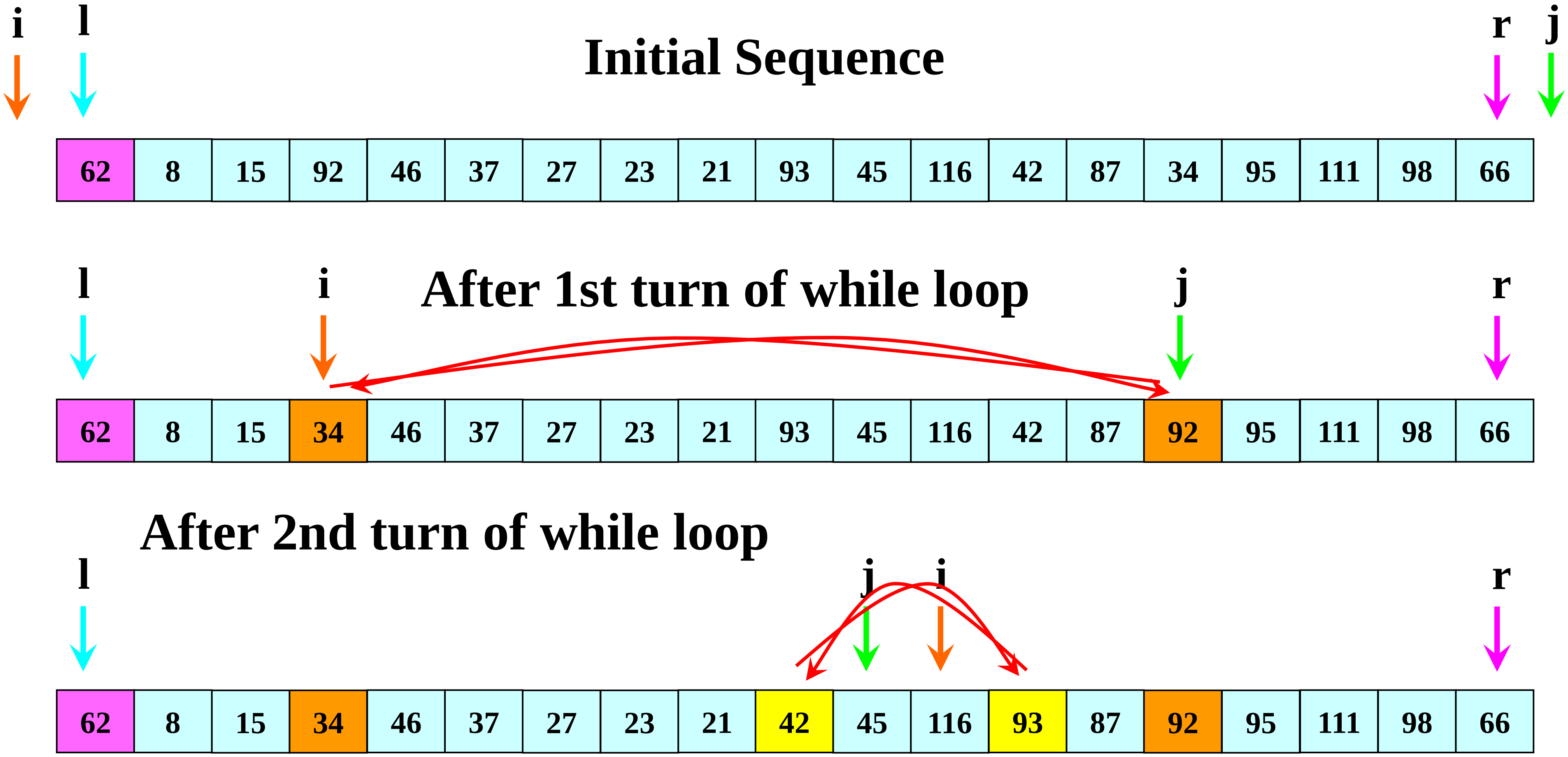
Hoare’s Partitioning: Idea

- To accomplish this, two pointers (e.g., *i* and *j*) start at both ends of the sequence to scan through moving towards each other.
- Whenever a pointer locates *a key in an incorrect position* (i.e., a greater key to left of pivot or a key less than the pivot at its right), it *stops moving*. When both pointers stop, the keys they point to are swapped and they restart moving.
- Scanning terminates when pointers pass crossing each other, and the key is placed at the position pointed to by *j*.

Hoare’s Partitioning: Algorithm

```
Partition_Hoare(A,l,r)
{
    x=A[l];
    i=l-1; j=r+1;
    while true
        do j-=1 while A[j]>x;
        do i+=1 while A[i]<x;
        if (i<j) swap A[i]↔A[j];
        else {
            swap(A[j],A[l])
            return j;
        }
}
```

Hoare’s Partitioning: Example



Initial Sequence

62	8	15	92	46	37	27	23	21	93	45	116	42	87	34	95	111	98	66

After 1st turn of while loop

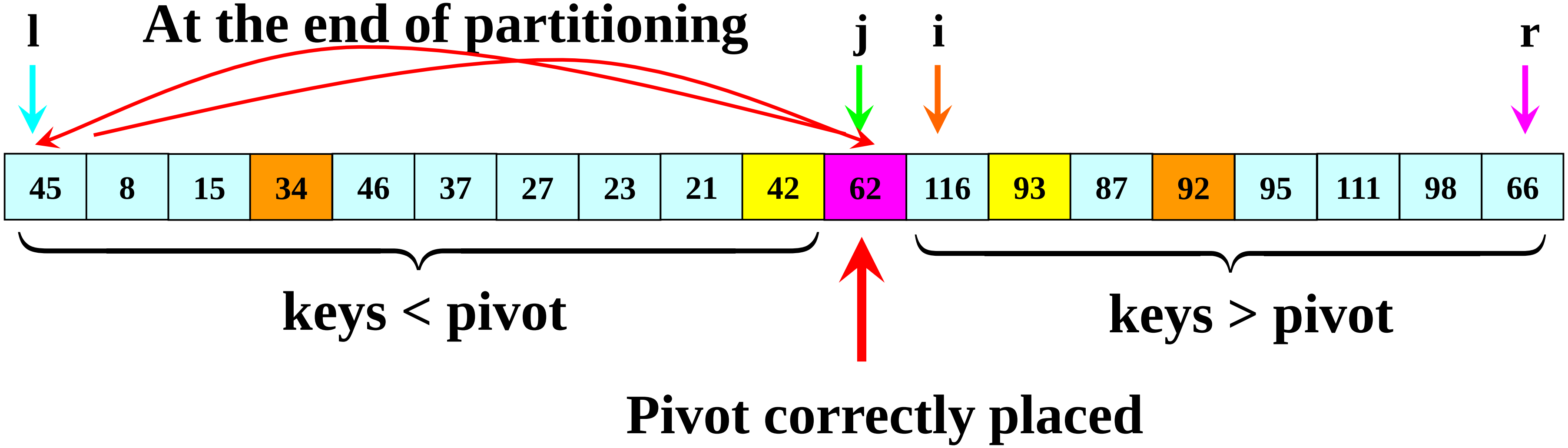
62	8	15	34	46	37	27	23	21	93	45	116	42	87	92	95	111	98	66

After 2nd turn of while loop

62	8	15	34	46	37	27	23	21	42	45	116	93	87	92	95	111	98	66

i>j, not exchanged

Hoare’s Partitioning: Example



At the end of partitioning

l j i r

45	8	15	34	46	37	27	23	21	42	62	116	93	87	92	95	111	98	66

keys < pivot

keys > pivot

Pivot correctly placed

Algorithm Analysis for Quicksort

Average case:

Each pivot at position m has a left and a right subsequence with $m-1$ and $n-m$ keys, respectively, to sort. Hence, the running time $f(n)$ of quicksort for n elements can be expressed as follows

$$f(n)=f(m-1)+f(n-m)+\Theta(n).$$

The term "$\Theta(n)$" is the time that partitioning takes to place the key selected as pivot at its correct position m.

Algorithm Analysis for Quicksort

m and *n-m* can be any number between between *1* and *n*.

To come up with a general solution, we can average *f(m-1)* and *f(n-m)*:

$$E\left[f(n)\right] = E\left[\frac{1}{n}\sum_{m=1}^{n} f(m-1) + f(n-m) + \Theta(n)\right]$$

$$E\left[f(n)\right] = E\left[\frac{2}{n}\sum_{m=2}^{n-1} f(m)\right] + \Theta(n)$$

$$E\left[f(n)\right] = \frac{2}{n} E\left[\sum_{m=2}^{n-1} f(m)\right] + \Theta(n)$$

$$f(n) = O(n + n\lg(n)) = O(n\lg(n))$$

Algorithm Analysis for Quicksort

Best Case: *The pivot is placed always in the middle.*

$$f(n)=2f(n/2)+\Theta(n);$$

$$f(n)=2f(n/2)+\Theta(n); \quad n=2^k$$

$$f(k)=2f(k-1)+c2^k;$$

$$CE:(x-2)^2=0$$

$$f(k)=c_1 2^k+c_2 2^k k; n=2^k \Leftrightarrow k=\lg n$$

$$f(n)=c_1 n+c_2 n\lg n$$

$$f(n)=O(n+n\lg(n))=O(n\lg(n))$$

Algorithm Analysis for Quicksort

Worst Case: The sequence is sorted in the opposite order..

f(n)=f(n-1)+Θ(n);

$$f(n) = f(n-1) + \Theta(n);$$

$$f(n) = f(n-1) + cn;$$

$$CE: (x-1)^3 = 0$$

$$f(n) = c_1 + c_2 n + c_2 n^2;$$

$$f(n) = O(1 + n + n^2) = O(n^2)$$

Selecting the pivot...

- Pivot selection method may essentially affect quicksort performance.
- *Selecting the first* (or the last) would work *alright if the sequence is purely randomly built*. If not, the worst case is not totally unlikely to occur.
- *Selecting the pivot randomly works well*. However, the random number generator should generate numbers sufficiently randomly. Furthermore, random number generation is an *expensive* process.
- ***Median-of-3 partitioning*** (median of a sequence is the $\lceil n/2 \rceil$ highest among n keys in a sequence) is to select as the pivot the second largest among the leftmost, rightmost and middle keys in the sequence.

A $O(n)$-*Expected-Time* Selection Algorithm

Quicksort can be modified to select the k^{th} largest key in the sequence.

1. Select a pivot
2. Have it placed at its correct position *m*
3. If *(k<m)* recursively call quicksort for the left subsequence only
4. Else recursively call quicksort for the right subsequence only.

Conceptual Example: $O(n)$-Expected-Time Selection

Pivot

Problem: Find *third* smallest key (i.e., key at cell 2)!

Initial Sequence

62	8	15	92	46	37	27	23	21	93	45	116	42	87	34	95	111	98	66

45	8	15	34	46	37	27	23	21	42	62

62 placed at array cell 10… Since 2 < 10, we consider only numbers less than 62

21	8	15	34	42	37	27	23	45

45 placed at array cell 8… Since 2 < 8, we consider only numbers less than 45

15	8	21	34	42	37	27	23

21 placed at array cell 2… Since 2=2 , we stop. Third smallest key is *21*.

Reference...

[1] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, "*Introduction to Algorithms*," 2nd edition, MIT Press, 2003

[2] M.A. Weiss, "*Data Structures and Algorithm Analysis in C*," 2nd edition, Addison-Wesley, 1997