Data Structure: Sorting

Yongjun Park
Hanyang University

sorting

- sorting is putting the elements into a list in which the elements are in increasing order

insertion sort

34, 8, 64, 51, 32, 21

0	1	2	3	4	5	
34	8	64	51	32	21	
8	34	64	51	32	21	after p=1
8	34	64	51	32	21	after p=2
8	34	51	64	32	21	after p=3
8	32	34	51	64	21	after p=4
8	21	32	34	51	64	after p=5

insertion sort

- For each pass P = 1 through n - 1, insertion sort ensures that elements in position 0 through P are in sorted order
- In pass P, move the element in position P left until its correct place is found among the first P elements
- $O(n^2)$ comparisons required on average
- any algorithm that sorts by exchanging adjacent elements requires $O(n^2)$ time on average
  - average number of swapping in an array of n distinct numbers is n(n-1)/4 since total number of pairs to be compared is n(n-1)/2

34, 8, 64, 51, 32, 21

0	1	2	3	4	5	
34	8	64	51	32	21	
8	34	64	51	32	21	after p=1
8	34	64	51	32	21	after p=2
8	34	51	64	32	21	after p=3
8	32	34	51	64	21	after p=4
8	21	32	34	51	64	after p=5

insertion sort

```
void insertionSort (ElementType A[ ], int N)
{
        int j,   P;
        for (P = 1;   P < N;   P++)
        {
                Tmp = A [P];
                for (j = P;     j > 0 && A[j-1] > Tmp;    j --)
                        A[j] = A[j-1];
                A[j] = Tmp;
        }
}
```

0	1	2	3	4	5	
34	8	64	51	32	21	
8	34	64	51	32	21	after p=1
8	34	64	51	32	21	after p=2
8	34	51	64	32	21	after p=3
8	32	34	51	64	21	after p=4
8	21	32	34	51	64	after p=5

heap sort

- building binary heap of n elements: O(n)
- DeleteMin operation n times: O(n log n)
- use the last cell in the previous heap to save the noted list (in-place algorithm)

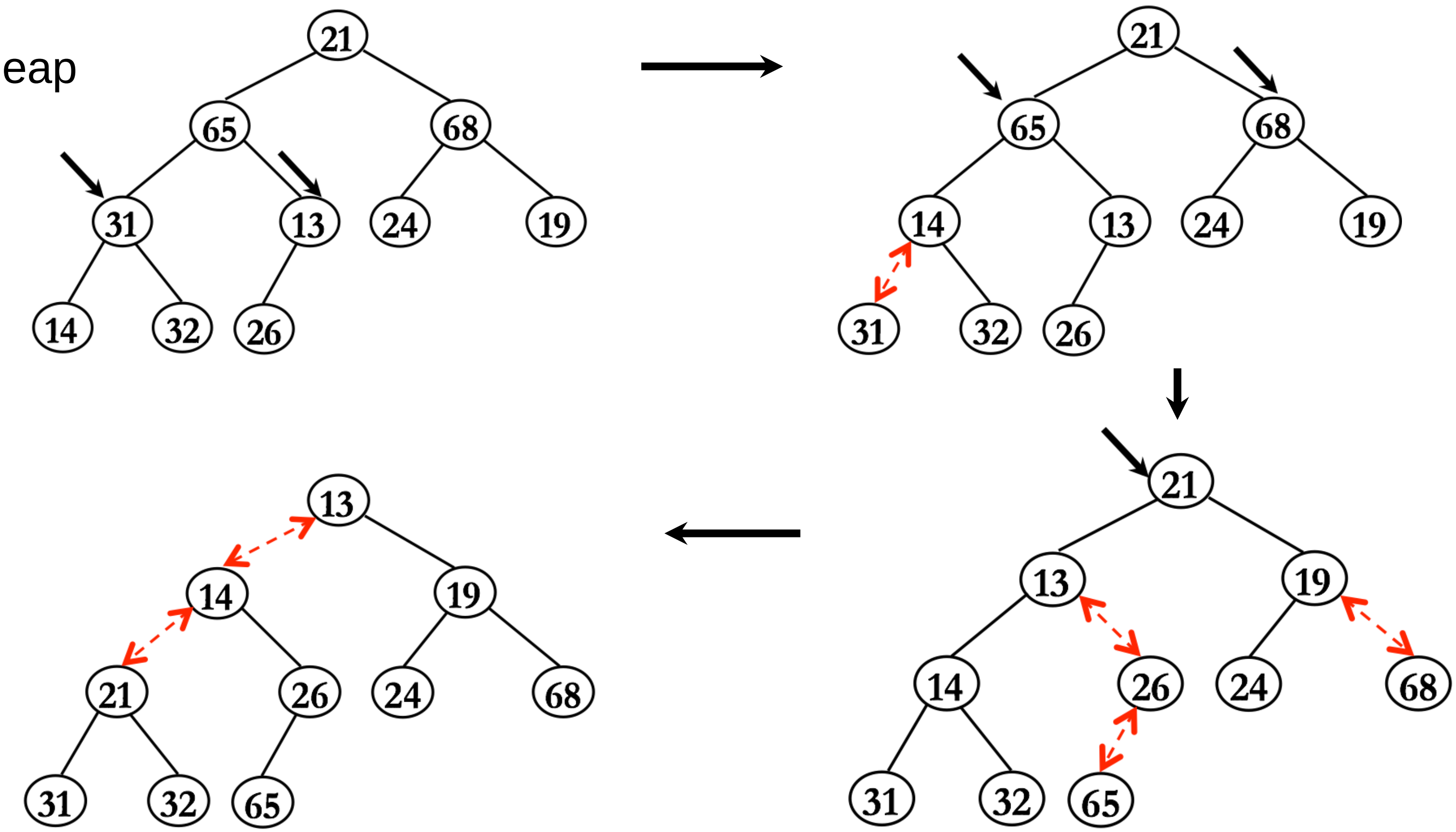
```
void HeapSort (ElementType A[], int N)
{
        int   i;
        for (i = N/2;   i >0;   i --)       /* Build Heap */
                PercDown (A,  i,  N);

        for (i = N;  i > 0;   i --)
        {
                Swap(&A[1],  &A[i] );     /*DeleteMax */
                PercDown(A, 1, i-1);
        }
}
```

BuildHeap

- Build a Heap containing n keys takes $O(n \log n)$ with consecutive insertions
- But it can take $O(n)$ if they are already in array.
- Starting with the lowest non-leaf node, working back towards root, perform percolating-down on each node of the tree.

min heap

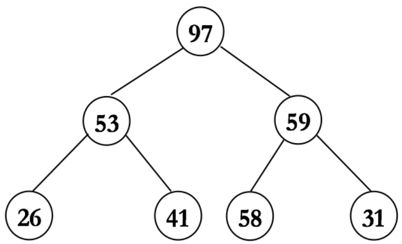


heap sort

Increasing order using Max heap

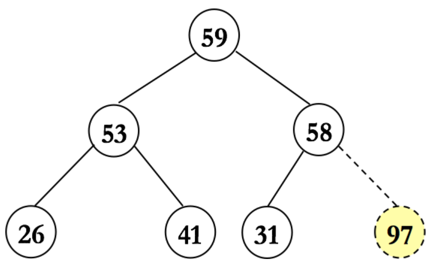
	31	41	59	26	53	58	97		
0	1	2	3	4	5	6	7	8	9

BuildHeap



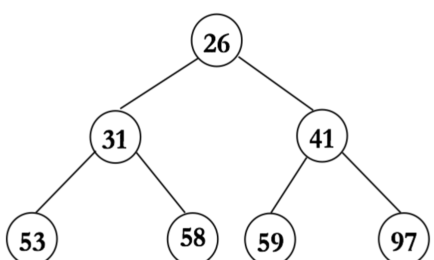
	97	53	59	26	41	58	31		
0	1	2	3	4	5	6	7	8	9

DeleteMax



	59	53	58	26	41	31	97		
0	1	2	3	4	5	6	7	8	9

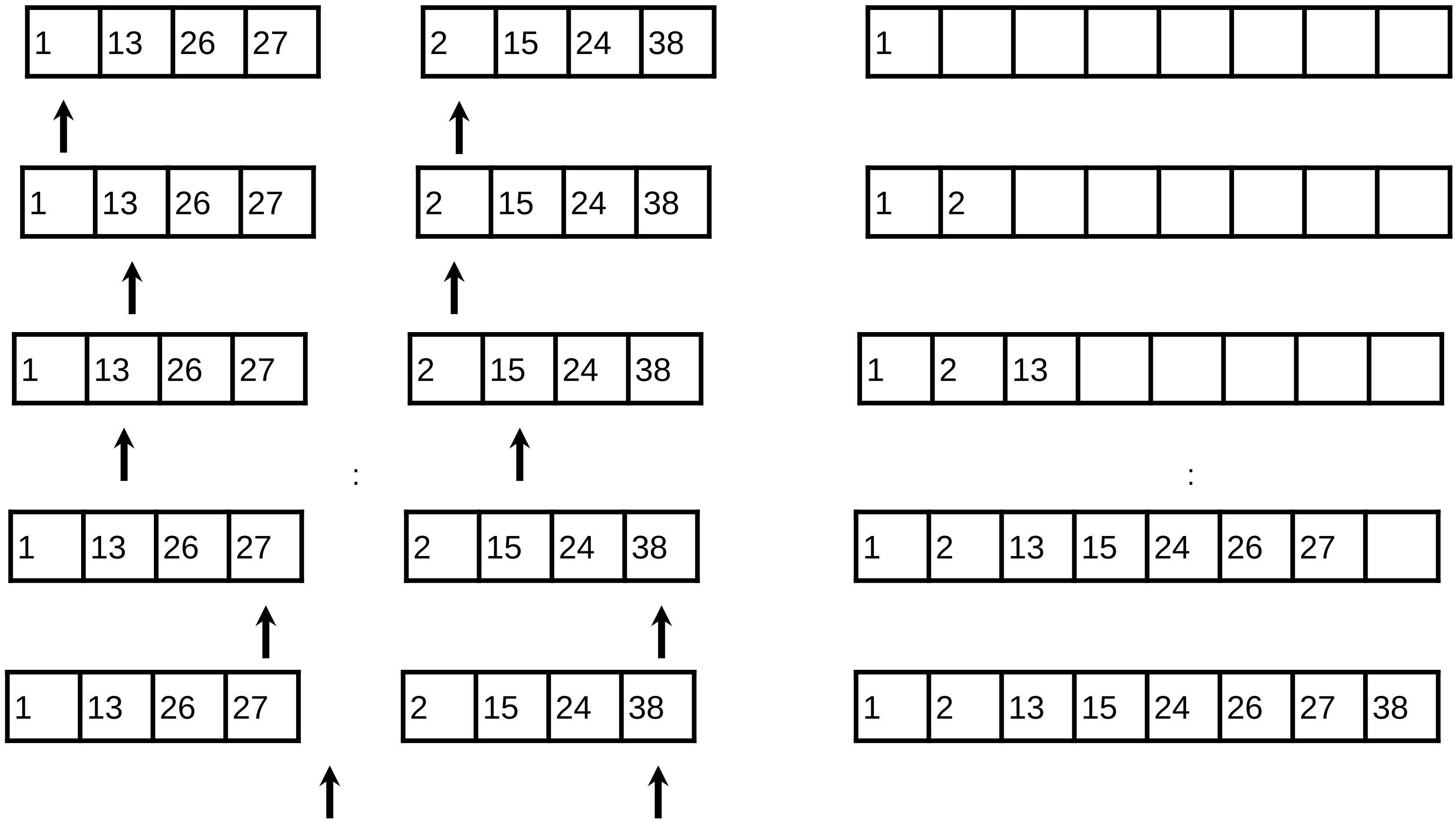
:



	26	31	41	53	58	59	97		
0	1	2	3	4	5	6	7	8	9

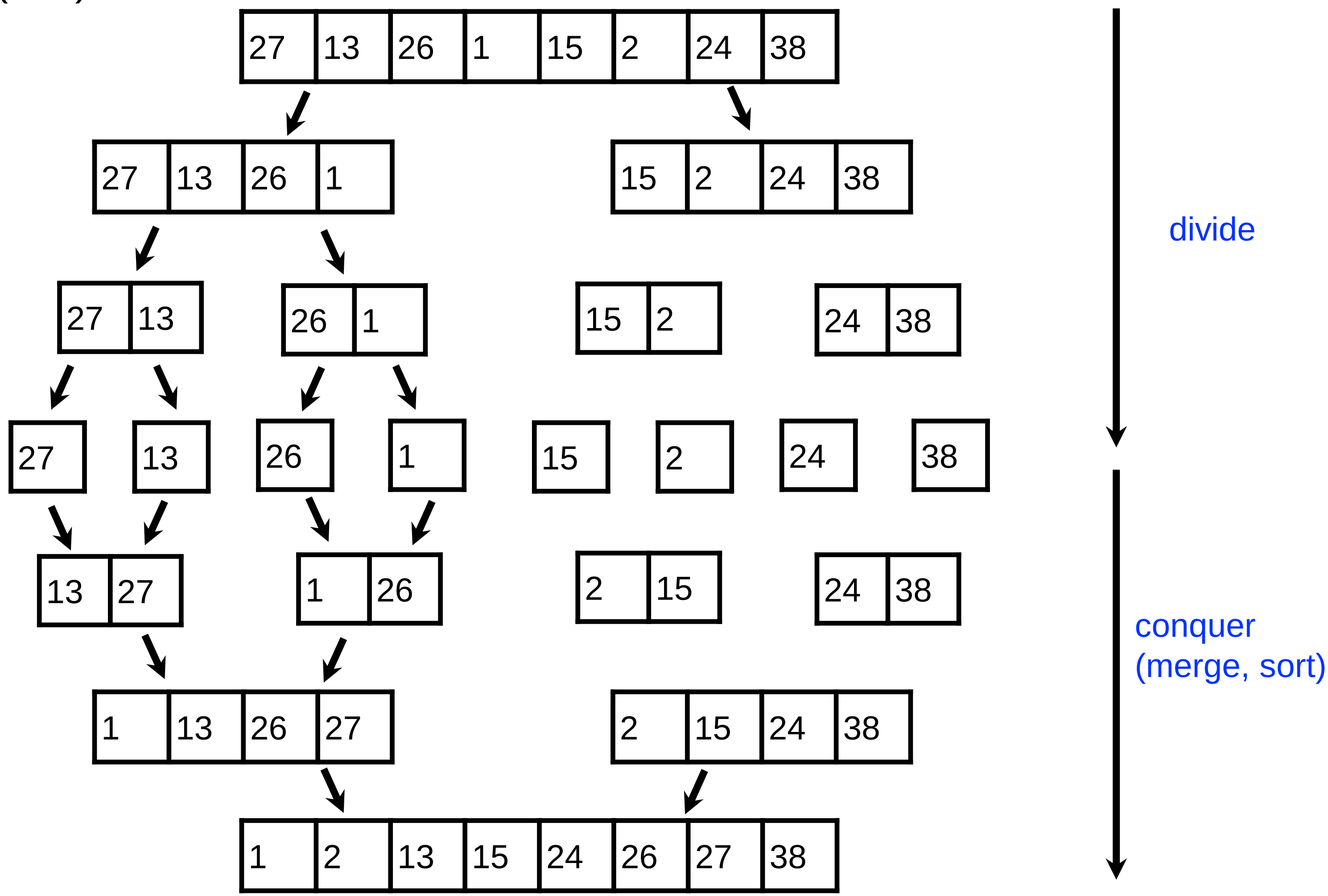
merge sort

- merge two sorted sublists using temporary array



merge sort

- divide a list into two sublists
- conquer (sort) the sorted sublist into the list

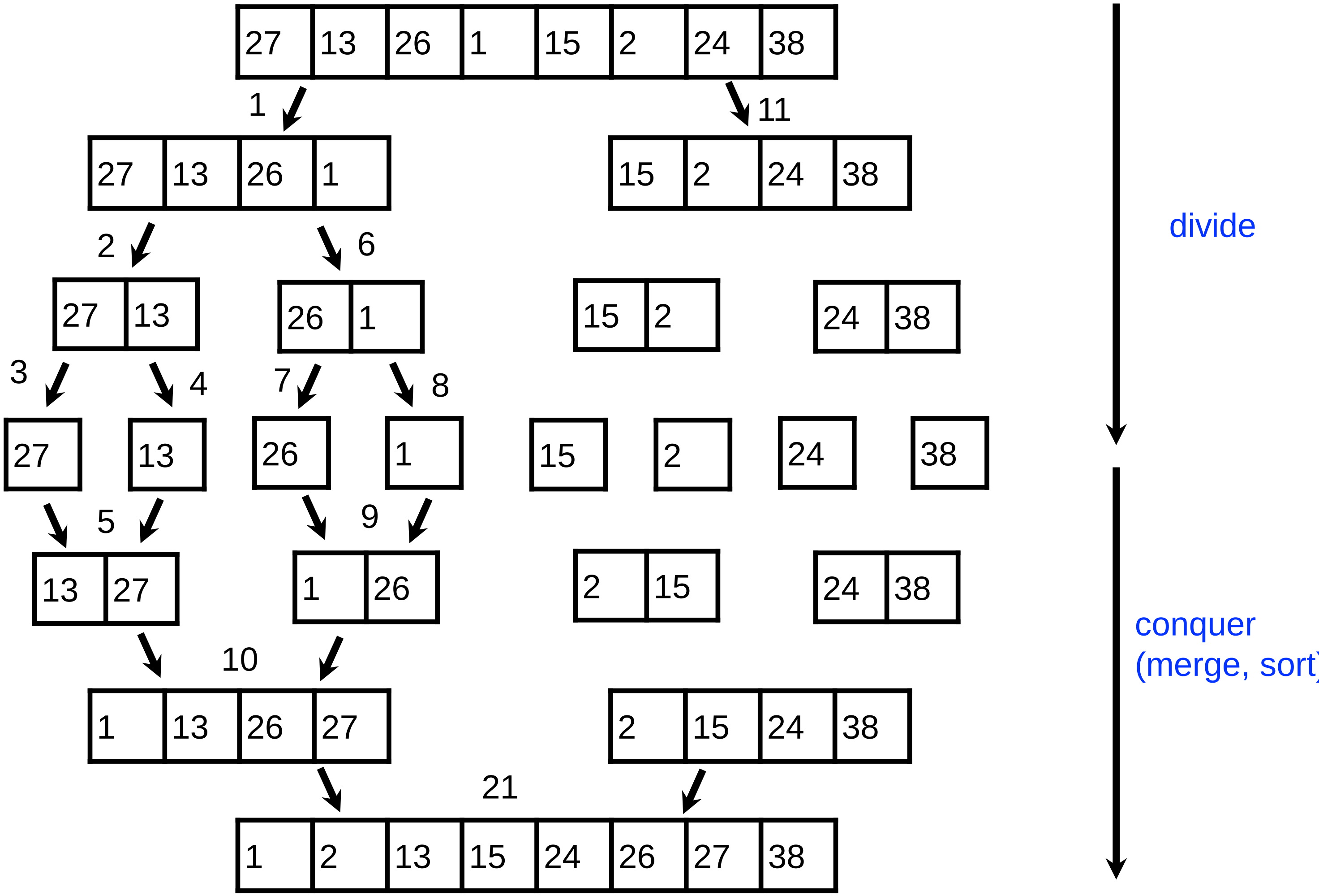


merge sort

```
void  MSort (ElementType A[], ElementType TmpArray[ ], int Left, int Right)
{
        int Center;
        if (Left < Right){
                Center = (Left + Right) / 2;
                  MSort (A, TmpArray, Left, Center);
                  MSort (A, TmpArray, Center+1, Right);
                  Merge (A, TmpArray, Left, Center+1, Right);
        }
}
```

merge sort

- divide a list into two sublists
- conquer (sort) the sorted sublist into the list



merge sort

```
void  Merge (ElementType A[], ElementType TmpArray[ ], int Lpos, int Rpos, int RightEnd)
{
        int i, LeftEnd, NumElements, TmpPos;
        LeftEnd = Rpos - 1;
        TmpPos = Lpos;
        NumElements = RightEnd - Lpos + 1;

        while (Lpos <= LeftEnd && Rpos <= RightEnd)
                if (A[Lpos] <= A[Rpos])
                        TmpArray[TmpPos++] = A[Lpos++];
                else
                        TmpArray[TmpPos++] = A[Rpos++];

        while (Lpos <= LeftEnd)
                TmpArray[TmpPos++] = A[Lpos++];
        while (Rpos <= RightEnd)
                TmpArray[TmpPos++] = A[Rpos++];

        for(i=0;   i<NumElements;   i++, RightEnd--)
                A[RightEnd] = TmpArray[RightEnd];
}
```

merge sort: analysis of time complexity

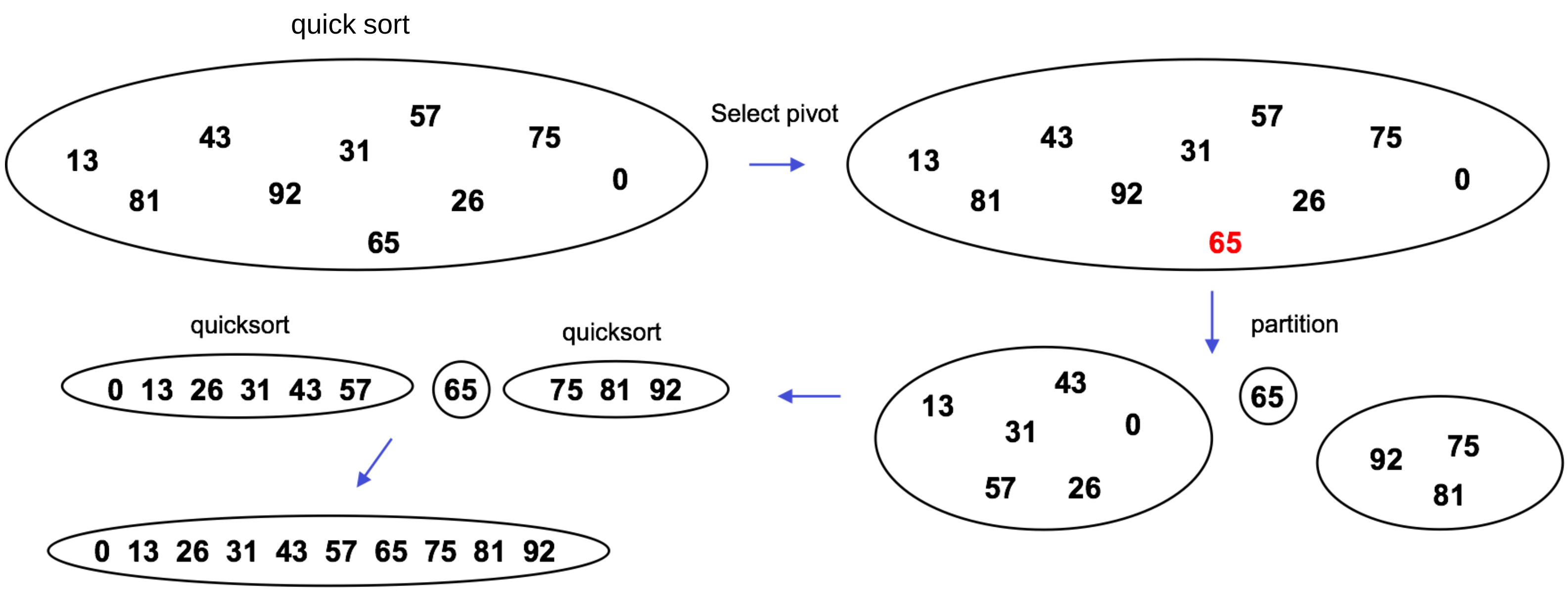
$T(1) = 1$
$T(N) = 2T(N/2) + N$

$T(N)/N = T(N/2) / (N/2) + 1$
$T(N/2) / (N/2) = T(N/4) / (N/4) + 1$
$T(N/4) / (N/4) = T(N/8) / (N/8) + 1$
:
$T(2) / (2) = T(1) / (1) + 1$

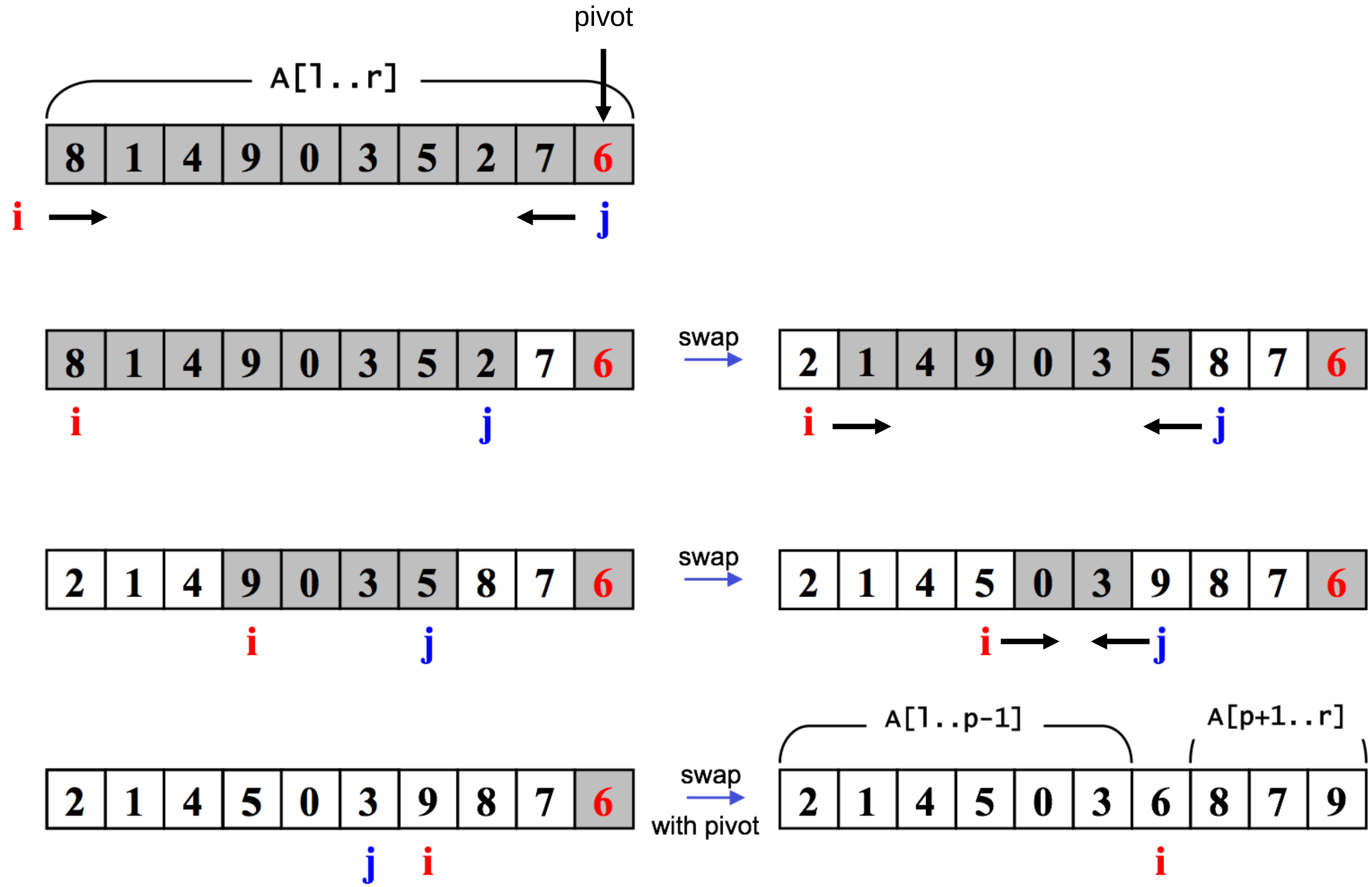
$T(N)/N = T(1)/1 + \log N$
$T(N) = N \log N + N = O(N \log N)$

quick sort

- divide: partition the array A[l..r] into two subarrays A[l..p-1] and A[p+1..r]
  - all elements in A[l..p-1] are less than or equal to a pivot element A[p]
  - all elements in A[p+1..r] are greater than pivot element A[p].
- conquer: sort the two subarrays A[l..p-1] and A[p+1..r] by recursive calls to quicksort.
  -> since the subarrays are sorted in place, no work is needed.



quick sort


pivot
A[l..r]
8 1 4 9 0 3 5 2 7 6
i
j
swap
2 1 4 9 0 3 5 8 7 6
2 1 4 5 0 3 9 8 7 6
swap
with pivot
A[l..p-1]
A[p+1..r]
2 1 4 5 0 3 6 8 7 9

quick sort

```
void Quicksort(A, l, r)
{
        if (l >= r)     return;
        p = Partition(A, l, r);
        Quicksort(A, l, p-1);
        Quicksort(A, p+1, r);
}

int Partition(A, l, r)
{
        pivot = select_pivot(A, l, r);
        i = l – 1;
        j = r;
        for( ; ; ) {
            while( A[--j] > pivot );
            while( A[++i] <= pivot );
            if ( i < j ) swap(&A[i], &A[j]);
            else {
                swap(&A[i], &A[r]);
                return i;
            }
        }
}
```

quick sort: picking the pivot

- use the first element or the last element
  - worst if the input is presorted or in reverse order
- choose the pivot randomly
  - safe, but does not reduce the average running time
  - median-of-three choose the median of the leftmost, rightmost, and center elements

quick sort: picking the pivot

$T(0) = T(1) = 0$

$T(n) = T(i) + T(n - i - 1) + n$

- performance depends on the selection of pivot
  - **worst-case partitioning: divide *n* – 1 and pivot**

    $T(n) = T(n - 1) + n$

    $= \boxed{T(n-2) + n - 1} + n$

    $= :$

    $= T(1) + 2 + 3 + . . . + n$

    $= O(n^2)$
  - **best-case partitioning: divide *n*/2 and *n*/2 elements**

    $T(n) = 2T(n/2) + n$

    $= 4T(n/4) + 2n$

    $= 8T(n/8) + 3n$ $\leftarrow 2(2T(n/4)+n/2) + n$

    $= :$

    $= nT(1) + logn * n$

    $= O(n \log n)$

quick sort: picking the pivot

$T(n) = T(i) + T(n - i - 1) + n$

- **average-case partitioning**

  assume that the size of a partition is equally likely (that is, probability is $1/n$)

  the average value of $T(i)$ or $T(n - i - 1)$ is $\frac{1}{n}\sum_{j=0}^{n-1} T(j)$

$$T(n) = \frac{2}{n}\left[\sum_{j=0}^{n-1} T(j)\right] + n$$

$$nT(n) = 2\left[\sum_{j=0}^{n-1} T(j)\right] + n^2$$

$$(n-1)T(n-1) = 2\left[\sum_{j=0}^{n-2} T(j)\right] + (n-1)^2$$

$$nT(n) - (n-1)T(n-1) = 2T(n-1) + 2n - 1$$

$$nT(n) = (n+1)T(n-1) + 2n$$

$$\frac{T(n)}{n+1} = \frac{T(n-1)}{n} + \frac{2}{n+1}$$

$$\frac{T(n)}{n+1} = \frac{T(1)}{2} + 2\sum_{i=3}^{n+1} \frac{1}{i}$$

$$\frac{T(n)}{n+1} = O(\log n), \quad T(n) = O(n \log n)$$