Insertion sort,
Merge sort,
Quick sort

Insertion sort

1) Initially p = 1

2) Let the first p elements be sorted.

3) Insert the (p+1)th element properly in the list so that now p+1 elements are sorted.

4) increment p and go to step (3)

Insertion Sort

Original	34	8	64	51	32	21	Positions Moved
After $p = 1$	8	34	64	51	32	21	1
After $p = 2$	8	34	64	51	32	21	0
After $p = 3$	8	34	51	64	32	21	1
After $p = 4$	8	32	34	51	64	21	3
After $p = 5$	8	21	32	34	51	64	4

Insertion Sort...

```
for( int p = 1; p < a.size( ); p++ )
{
    Comparable tmp = a[ p ];
    for( j = p; j > 0 && tmp < a[ j - 1 ]; j-- )
        a[ j ] = a[ j - 1 ];
    a[ j ] = tmp;
}
```

- Consists of N - 1 passes
- For pass p = 1 through N - 1, ensures that the elements in positions 0 through p are in sorted order
  - elements in positions 0 through p - 1 are already sorted
  - move the element in position p left until its correct place is found among the first p + 1 elements

Extended Example

To sort the following numbers in increasing order:

34 8 64 51 32 21

P = 1; Look at first element only, no change.

P = 2; tmp = 8;

34 > tmp, so second element is set to 34.

We have reached the front of the list. Thus, 1st position = tmp

After second pass: 8 34 64 51 32 21

(first 2 elements are sorted)

P = 3; tmp = 64;

34 < 64, so stop at 3rd position and set 3rd position = 64

After third pass: 8 34 64 51 32 21

(first 3 elements are sorted)

P = 4; tmp = 51;

51 < 64, so we have 8 34 64 64 32 21,

34 < 51, so stop at 2nd position, set 3rd position = tmp,

After fourth pass: 8 34 51 64 32 21

(first 4 elements are sorted)

P = 5; tmp = 32,

32 < 64, so 8 34 51 64 64 21,

32 < 51, so 8 34 51 51 64 21,

next 32 < 34, so 8 34 34, 51 64 21,

next 32 > 8, so stop at 1st position and set 2nd position = 32,

After fifth pass: 8 32 34 51 64 21

P = 6; tmp = 21, . . .

After sixth pass: 8 21 32 34 51 64

Analysis: worst-case running time

```
for( int p = 1; p < a.size( ); p++ )
{
    Comparable tmp = a[ p ];
    for( j = p; j > 0 && tmp < a[ j - 1 ]; j-- )
        a[ j ] = a[ j - 1 ];
    a[ j ] = tmp;
}
```

- Inner loop is executed p times, for each p=1..N

  $\Rightarrow$ Overall: $1 + 2 + 3 + \ldots + N = O(N^2)$
- Space requirement is O(N)

Analysis

- The bound is tight $\Theta(N^2)$
- That is, there exists some input which actually uses $\Omega(N^2)$ time
- Consider input is a reverse sorted list
  - When A[p] is inserted into the sorted A[0..p-1], we need to compare A[p] with all elements in A[0..p-1] and move each element one position to the right

    $\Rightarrow$ $\Omega(i)$ steps

  - the total number of steps is $\Omega(\Sigma_1^{N-1} i) = \Omega(N(N-1)/2) = \Omega(N^2)$

Analysis: best case

- The input is already sorted in increasing order
  - When inserting A[p] into the sorted A[0..p-1], only need to compare A[p] with A[p-1] and there is no data movement
  - For each iteration of the outer for-loop, the inner for-loop terminates after checking the loop condition once => O(N) time
- If input is *nearly sorted*, insertion sort runs fast

Mergesort

Based on divide-and-conquer strategy

- Divide the list into two smaller lists of about equal sizes
- Sort each smaller list *recursively*
- Merge the two sorted lists to get one sorted list

How do we divide the list? How much time needed?

How do we merge the two sorted lists? How much time needed?

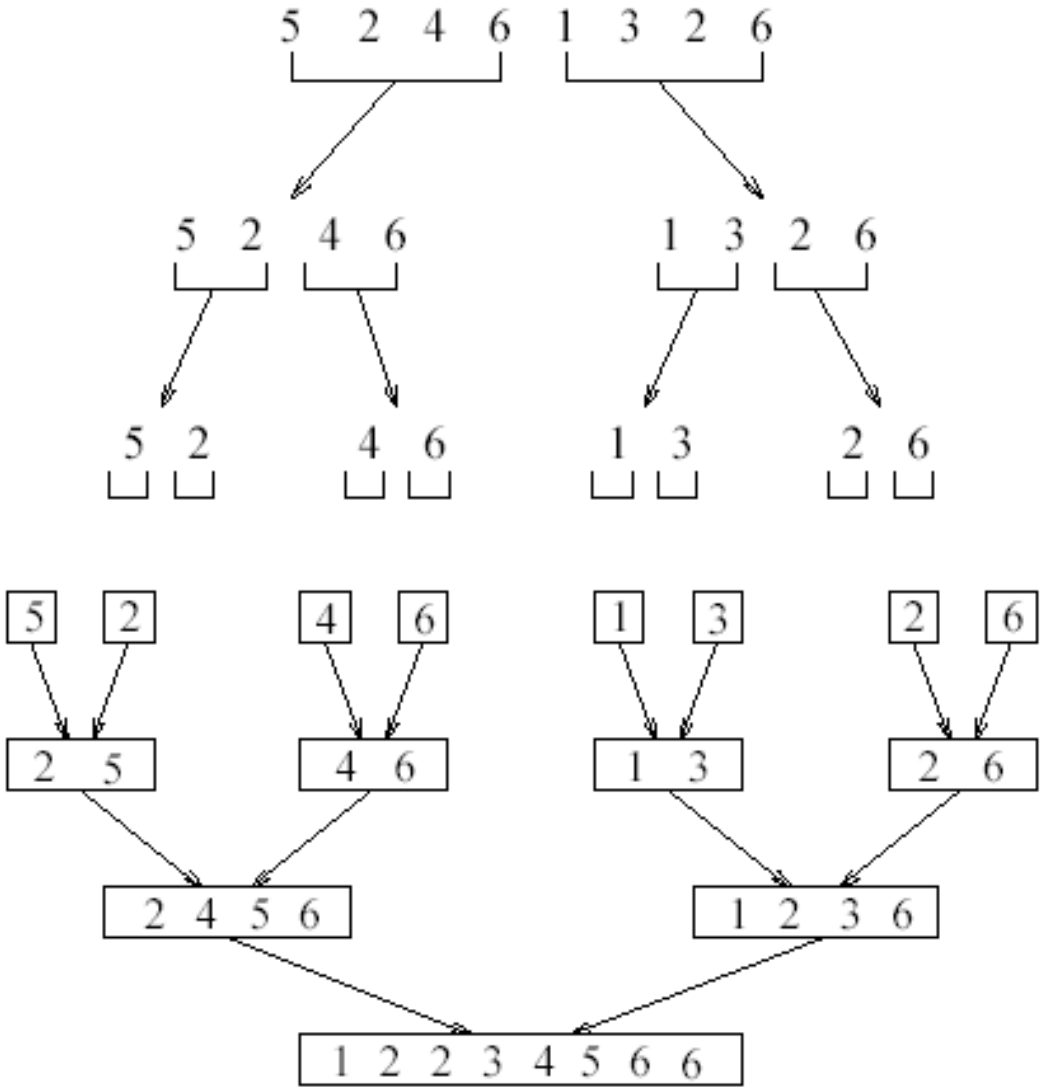
Dividing

- If the input list is a linked list, dividing takes $\Theta(N)$ time
  - We scan the linked list, stop at the $\lfloor N/2 \rfloor$ th entry and cut the link
- If the input list is an array A[0..N-1]: dividing takes $O(1)$ time
  - we can represent a sublist by two integers `left` and `right`: to divide `A[left..Right]`, we compute `center=(left+right)/2` and obtain `A[left..Center]` and `A[center+1..Right]`

Mergesort

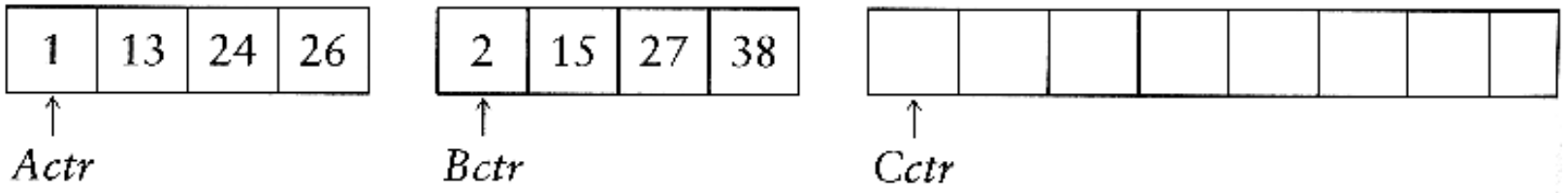
- Divide-and-conquer strategy
  - recursively mergesort the first half and the second half
  - merge the two sorted halves together

```
void mergesort(vector<int> & A, int left, int right)
{
   if (left < right) {
      int center = (left + right)/2;
      mergesort(A,left,center);
      mergesort(A,center+1,right);
      merge(A,left,center+1,right);
   }
}
```


5 2 4 6 1 3 2 6
5 2 4 6 1 3 2 6
5 2 4 6 1 3 2 6
5 2 4 6 1 3 2 6
2 5 4 6 1 3 2 6
2 4 5 6 1 2 3 6
1 2 2 3 4 5 6 6

How to merge?

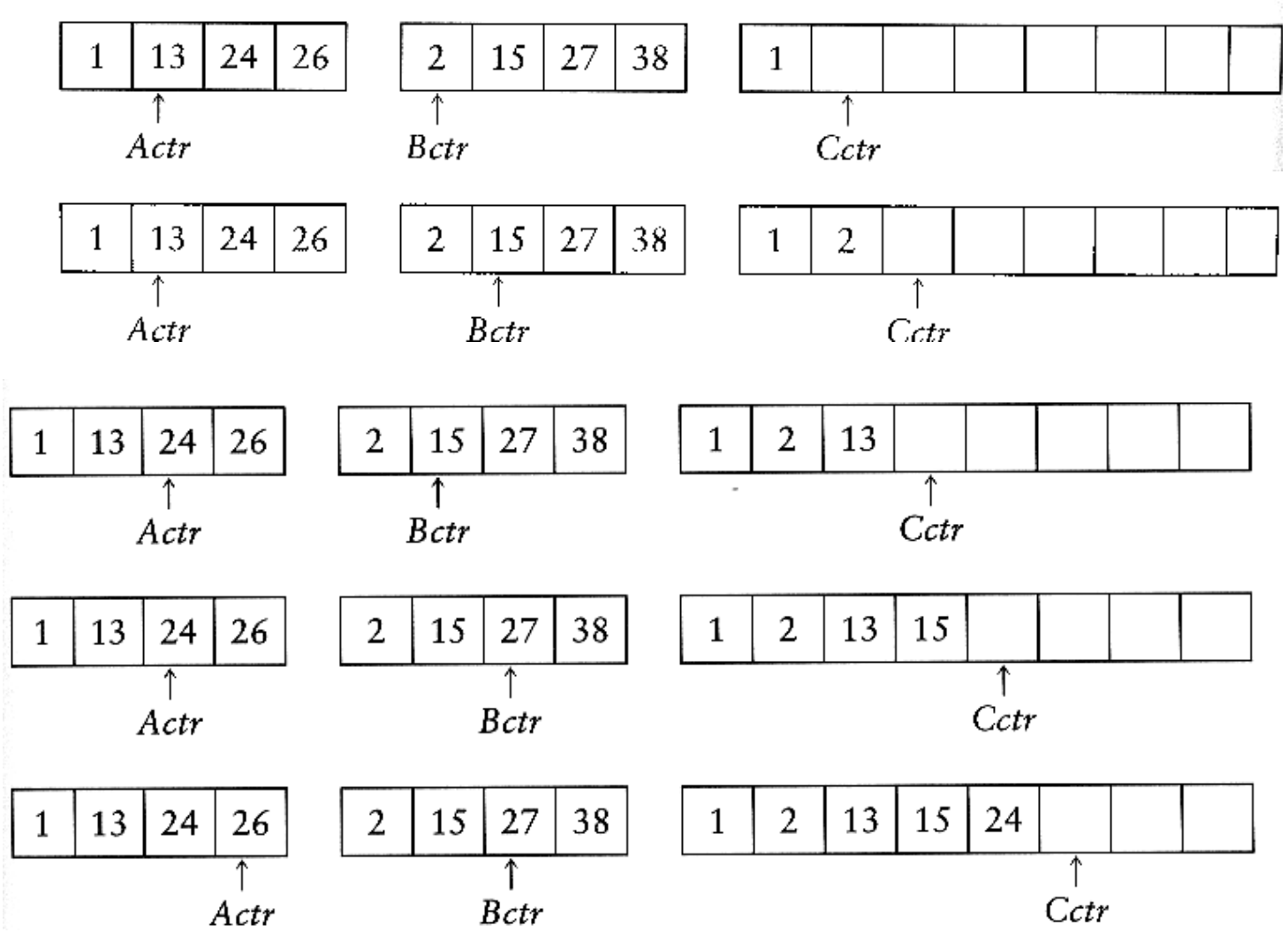
- Input: two sorted array A and B
- Output: an output sorted array C
- Three counters: Actr, Bctr, and Cctr
  - initially set to the beginning of their respective arrays



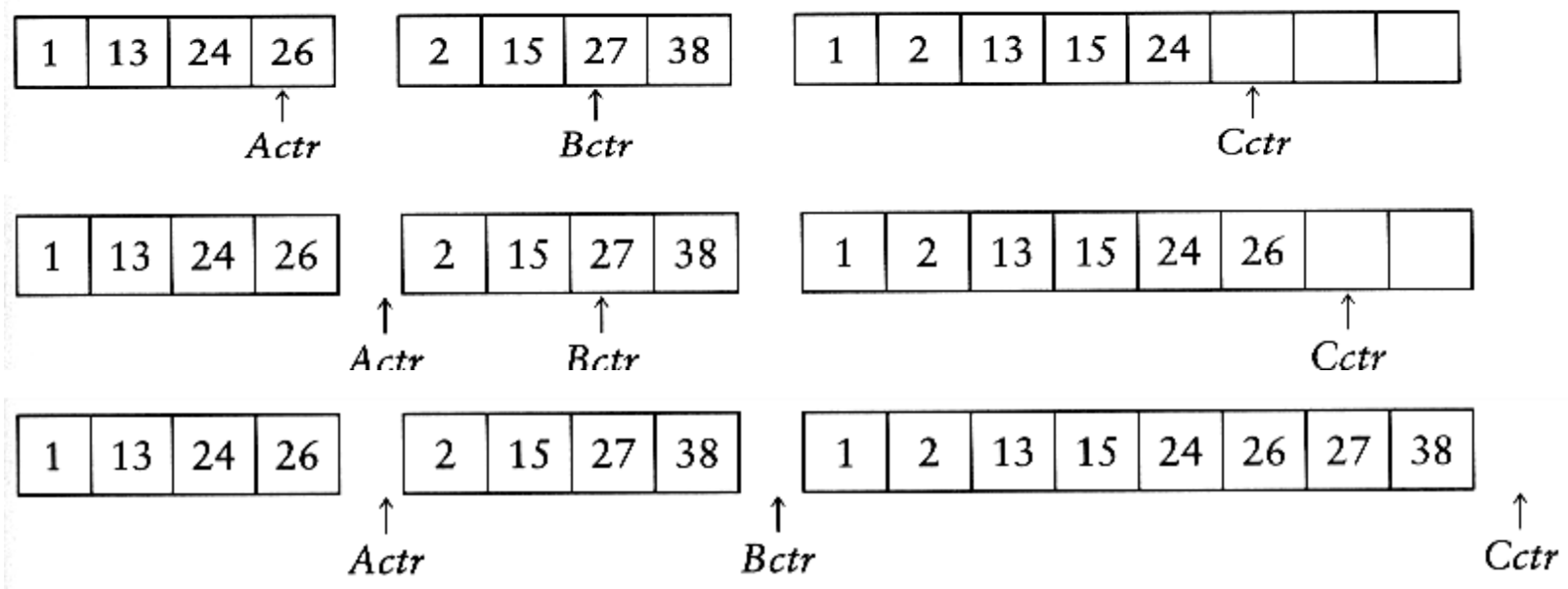
(1) The smaller of A[Actr] and B[Bctr] is copied to the next entry in C, and the appropriate counters are advanced

(2) When either input list is exhausted, the remainder of the other list is copied to C

Example: Merge



Example: Merge...



- Running time analysis:
  - Clearly, `merge` takes O(m1 + m2) where m1 and m2 are the sizes of the two sublists.
- Space requirement:
  - merging two sorted lists requires linear extra memory
  - additional work to copy to the temporary array and back

Algorithm *merge*(A, p, q, r)
Input: Subarrays $A[p..l]$ and $A[q..r]$ s.t. $p \leq l = q - 1 < r$.
Output: $A[p..r]$ is sorted.
$(*\ T$ is a temporary array. $*)$

```
1.  k = p; i = 0; l = q − 1;
2.  while p ≤ l and q ≤ r
3.      do if A[p] ≤ A[q]
4.             then T[i] = A[p]; i = i + 1; p = p + 1;
5.             else  T[i] = A[q]; i = i + 1; q = q + 1;
6.  while p ≤ l
7.      do T[i] = A[p]; i = i + 1; p = p + 1;
8.  while q ≤ r
9.      do T[i] = A[q]; i = i + 1; q = q + 1;
10. for i = k to r
11.     do A[i] = T[i − k];
```

Analysis of mergesort

Let T(N) denote the worst-case running time of mergesort to sort N numbers.

Assume that N is a power of 2.

- Divide step: O(1) time
- Conquer step: 2 T(N/2) time
- Combine step: O(N) time

Recurrence equation:

$T(1) = 1$

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

Analysis: solving recurrence

$$T(N) = 2T(\frac{N}{2}) + N$$

$$= 2(2T(\frac{N}{4}) + \frac{N}{2}) + N$$

$$= 4T(\frac{N}{4}) + 2N$$

$$= 4(2T(\frac{N}{8}) + \frac{N}{4}) + 2N$$

$$= 8T(\frac{N}{8}) + 3N = \cdots$$

$$= 2^k T(\frac{N}{2^k}) + kN$$

Since $N=2^k$, we have $k=\log_2 n$

$$T(N) = 2^k T(\frac{N}{2^k}) + kN$$

$$= N + N\log N$$

$$= O(N\log N)$$

Comparing $n \log_{10} n$ and n^2

n	$n \log_{10} n$	n^2	Ratio
100	0.2K	10K	50
1000	3K	1M	333.33
2000	6.6K	4M	606
3000	10.4K	9M	863
4000	14.4K	16M	1110
5000	18.5K	25M	1352
6000	22.7K	36M	1588
7000	26.9K	49M	1820
8000	31.2K	64M	2050

An experiment

- Using Unix `time` utility

n	Isort (secs)	Msort (secs)	Ratio
100	0.01	0.01	1
1000	0.18	0.01	18
2000	0.76	0.04	19
3000	1.67	0.05	33.4
4000	2.90	0.07	41
5000	4.66	0.09	52
6000	6.75	0.10	67.5
7000	9.39	0.14	67
8000	11.93	0.14	85

Quicksort

Introduction

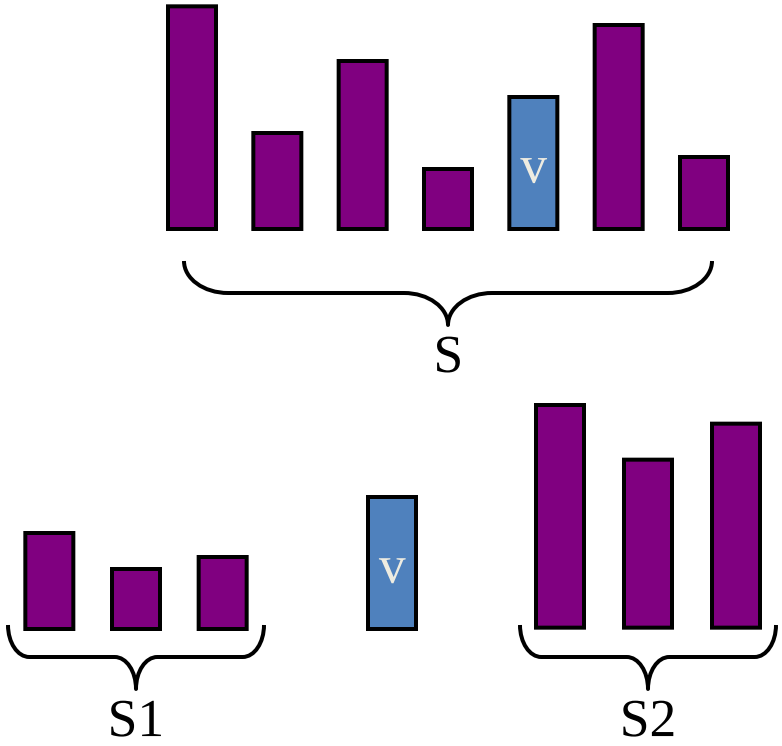
- Fastest known sorting algorithm in practice
- Average case: O(N log N)
- Worst case: $O(N^2)$
  - But, the worst case seldom happens.
- Another divide-and-conquer recursive algorithm like mergesort

Quicksort

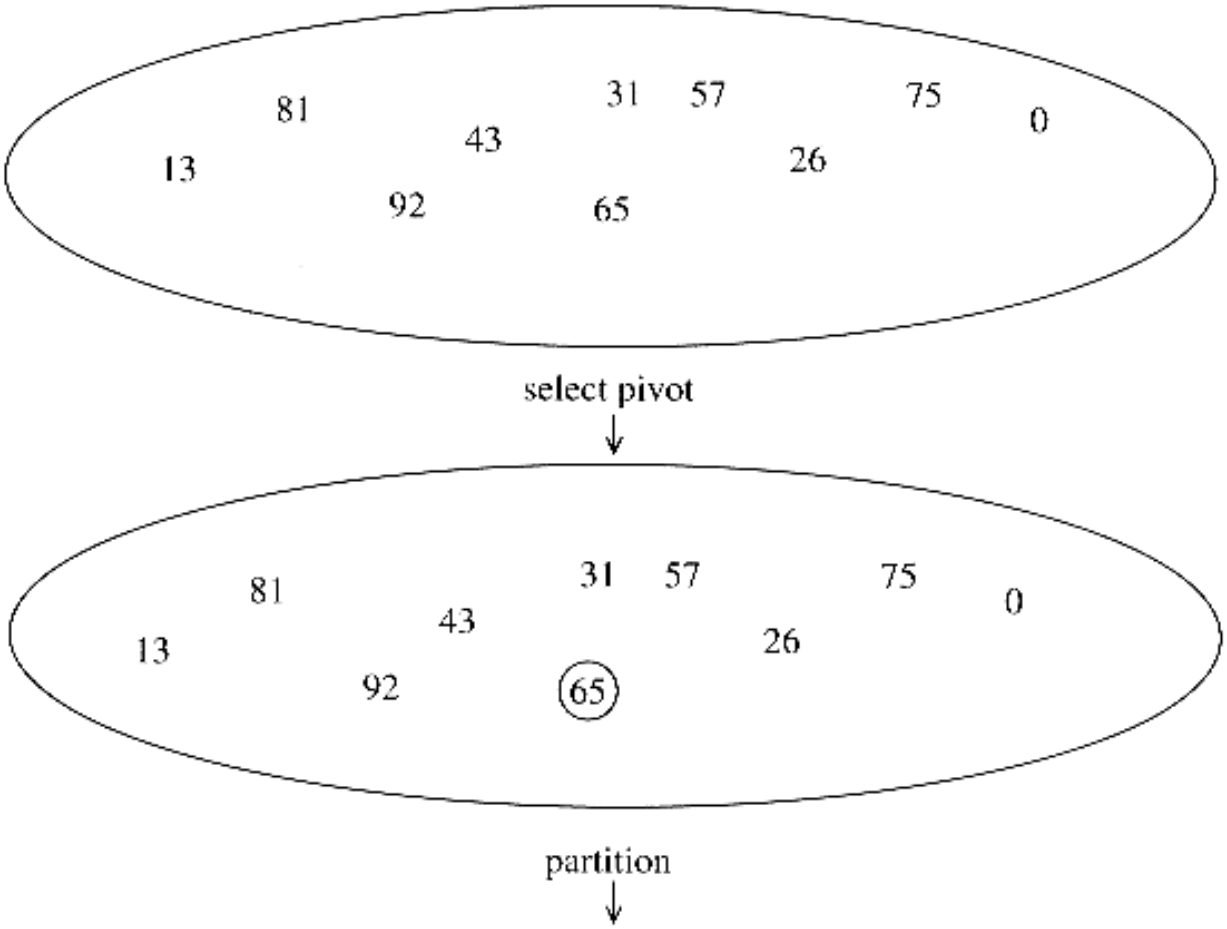
- Divide step:
  - Pick any element (***pivot***) v in S
  - Partition S – {v} into two disjoint groups

    S1 = {x ∈ S – {v} | x ≤ v}

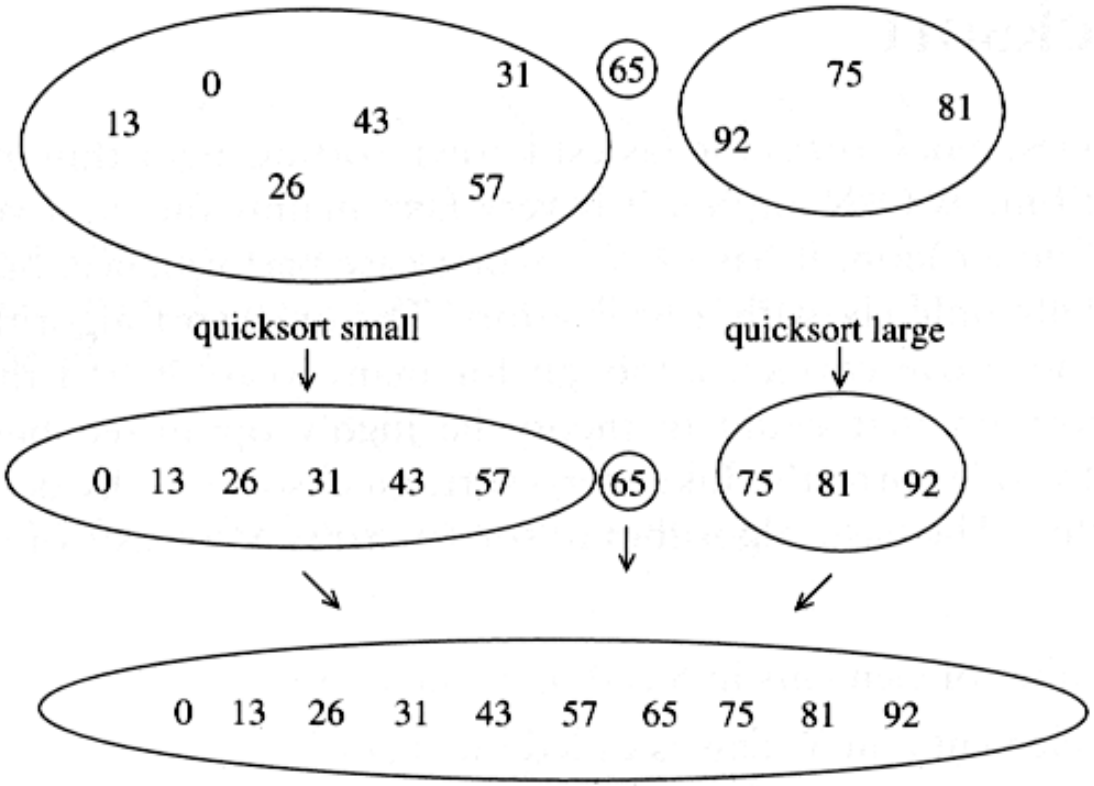
    S2 = {x ∈ S – {v} | x ≥ v}
- Conquer step: recursively sort S1 and S2
- Combine step: combine the sorted S1, followed by v, followed by the sorted S2



Example: Quicksort



Example: Quicksort...



Pseudocode

Input: an array A[p, r]

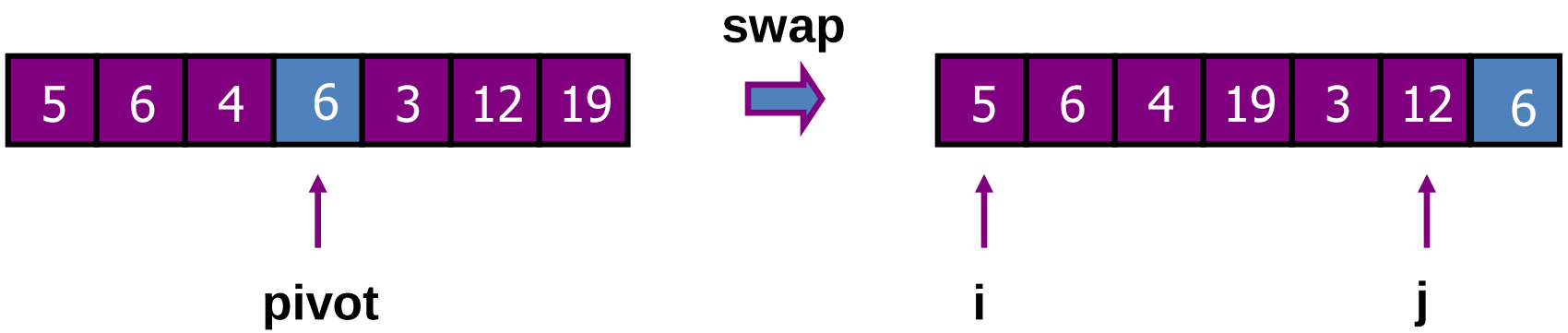
```
Quicksort (A, p, r) {
    if (p < r) {
        q = Partition (A, p, r)   //q is the position of the pivot element
        Quicksort (A, p, q-1)
        Quicksort (A, q+1, r)
    }
}
```

Partitioning

- Partitioning
  - Key step of quicksort algorithm
  - Goal: given the picked pivot, partition the remaining elements into two smaller sets
  - Many ways to implement
  - Even the slightest deviations may cause surprisingly bad results.
- We will learn an easy and efficient partitioning strategy here.
- How to pick a pivot will be discussed later

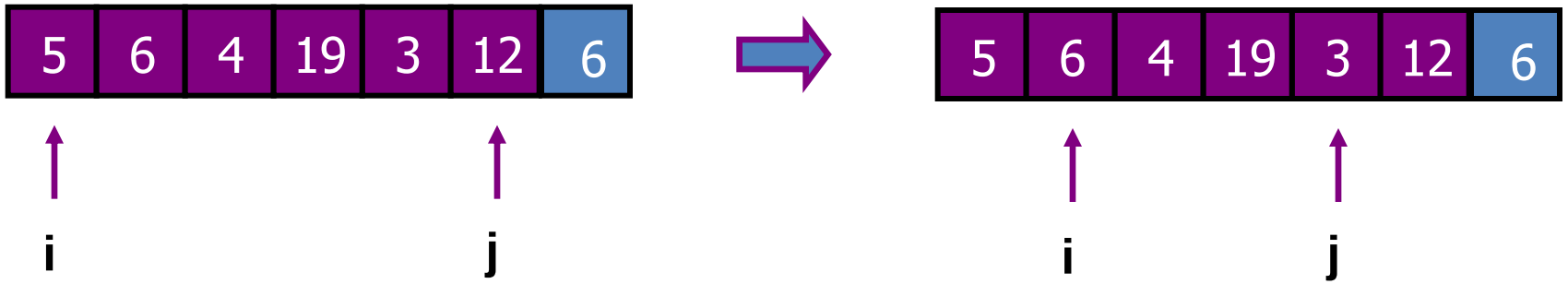
Partitioning Strategy

- Want to partition an array A[left .. right]
- First, get the pivot element out of the way by swapping it with the last element. (Swap pivot and A[right])
- Let i start at the first element and j start at the next-to-last element (i = left, j = right – 1)



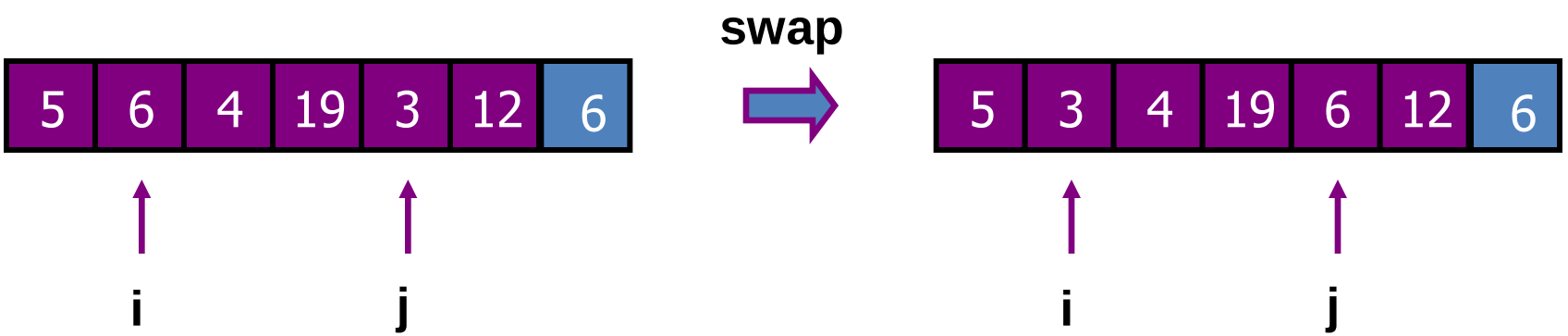
Partitioning Strategy

- Want to have
  - A[p] <= pivot, for p < i
  - A[p] >= pivot, for p > j
- When i < j
  - Move i right, skipping over elements smaller than the pivot
  - Move j left, skipping over elements greater than the pivot
  - When both i and j have stopped
    - A[i] >= pivot
    - A[j] <= pivot



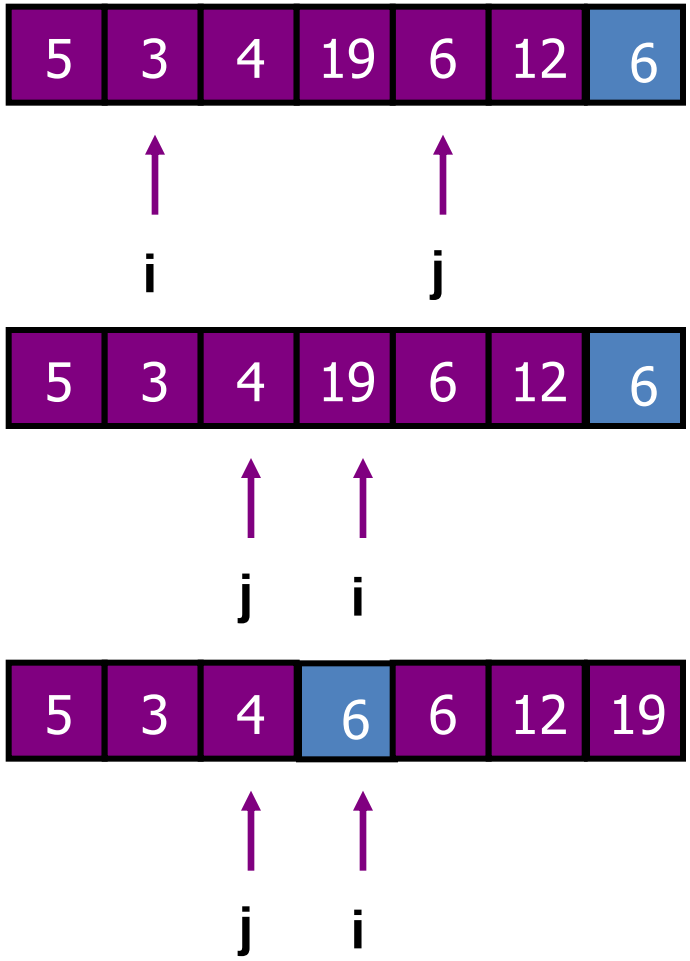
Partitioning Strategy

- When i and j have stopped and i is to the left of j
  - Swap A[i] and A[j]
    - The large element is pushed to the right and the small element is pushed to the left
  - After swapping
    - A[i] <= pivot
    - A[j] >= pivot
  - Repeat the process until i and j cross



Partitioning Strategy

- When i and j have crossed
  - Swap A[i] and pivot
- Result:
  - A[p] <= pivot, for p < i
  - A[p] >= pivot, for p > i



Small arrays

- For very small arrays, quicksort does not perform as well as insertion sort
  - how small depends on many factors, such as the time spent making a recursive call, the compiler, etc
- Do not use quicksort recursively for small arrays
  - Instead, use a sorting algorithm that is efficient for small arrays, such as insertion sort

Picking the Pivot

- Use the first element as pivot
  - if the input is random, ok
  - if the input is presorted (or in reverse order)
    - all the elements go into S2 (or S1)
    - this happens consistently throughout the recursive calls
    - Results in $O(n^2)$ behavior (Analyze this case later)
- Choose the pivot randomly
  - generally safe
  - random number generation can be expensive

Picking the Pivot

- Use the median of the array
  - Partitioning always cuts the array into roughly half
  - An optimal quicksort (O(N log N))
  - However, hard to find the exact median
    - e.g., sort an array to pick the value in the middle

Pivot: median of three

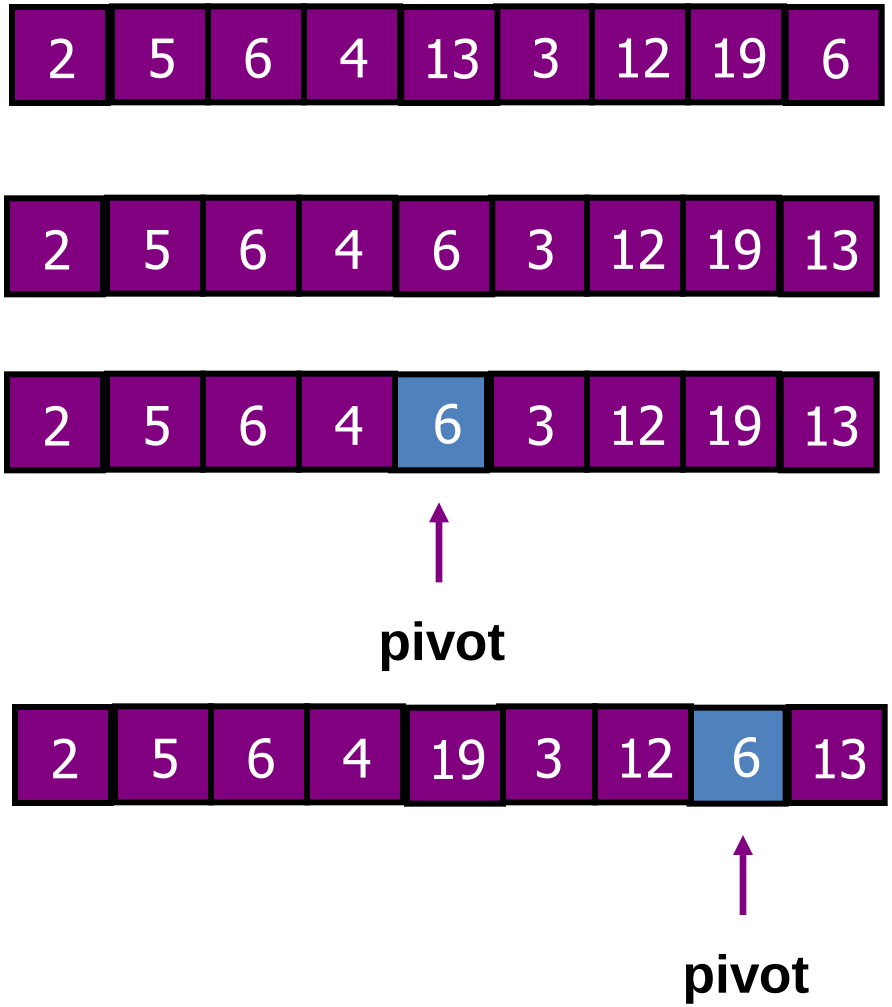
- We will use median of three
  - Compare just three elements: the leftmost, rightmost and center
  - Swap these elements if necessary so that
    - A[left] = Smallest
    - A[right] = Largest
    - A[center] = Median of three
  - Pick A[center] as the pivot
  - Swap A[center] and A[right – 1] so that pivot is at second last position (why?)

median3

```
int center = ( left + right ) / 2;
if( a[ center ] < a[ left ] )
    swap( a[ left ], a[ center ] );
if( a[ right ] < a[ left ] )
    swap( a[ left ], a[ right ] );
if( a[ right ] < a[ center ] )
    swap( a[ center ], a[ right ] );

    // Place pivot at position right - 1
swap( a[ center ], a[ right - 1 ] );
```

Pivot: median of three



Note we only need to partition A[left + 1, …, right – 2]. Why?

Main Quicksort Routine

```
if( left + 10 <= right )
{
    Comparable pivot = median3( a, left, right );

        // Begin partitioning
    int i = left, j = right - 1;
    for( ; ; )
    {
        while( a[ ++i ] < pivot ) { }
        while( pivot < a[ --j ] ) { }
        if( i < j )
            swap( a[ i ], a[ j ] );
        else
            break;
    }

    swap( a[ i ], a[ right - 1 ] );  // Restore pivot

    quicksort( a, left, i - 1 );     // Sort small elements
    quicksort( a, i + 1, right );    // Sort large elements
}
else  // Do an insertion sort on the subarray
    insertionSort( a, left, right );
```

Choose pivot

Partitioning

Recursion

For small arrays

Partitioning Part

- Works only if pivot is picked as median-of-three.
  - A[left] <= pivot and A[right] >= pivot
  - Thus, only need to partition A[left + 1, …, right – 2]
- j will not run past the end
  - because a[left] <= pivot
- i will not run past the end
  - because a[right-1] = pivot

```
int i = left, j = right - 1;
for( ; ; )
{
    while( a[ ++i ] < pivot ) { }
    while( pivot < a[ --j ] ) { }
    if( i < j )
        swap( a[ i ], a[ j ] );
    else
        break;
}
```

Quicksort Faster than Mergesort

- Both quicksort and mergesort take O(N log N) in the average case.
- Why is quicksort faster than mergesort?
  - The inner loop consists of an increment/decrement (by 1, which is fast), a test and a jump.
  - There is no extra juggling as in mergesort.

```
int i = left, j = right - 1;
for( ; ; )
{
    while( a[ ++i ] < pivot ) { }
    while( pivot < a[ --j ] ) { }
    if( i < j )
        swap( a[ i ], a[ j ] );
    else
        break;
}
```

inner loop

Analysis

- Assumptions:
  - A random pivot (no median-of-three partitioning
  - No cutoff for small arrays
- Running time
  - pivot selection: constant time O(1)
  - partitioning: linear time O(N)
  - running time of the two recursive calls
- $T(N)=T(i)+T(N-i-1)+cN$ where c is a constant
  - i: number of elements in S1

Worst-Case Analysis

- What will be the worst case?
  - The pivot is the smallest element, all the time
  - Partition is always unbalanced

$$
\begin{aligned}
T(N) &= T(N-1) + cN \\
T(N-1) &= T(N-2) + c(N-1) \\
T(N-2) &= T(N-3) + c(N-2) \\
&\vdots \\
T(2) &= T(1) + c(2) \\
T(N) &= T(1) + c\sum_{i=2}^{N} i = O(N^2)
\end{aligned}
$$

Best-case Analysis

- What will be the best case?
  - Partition is perfectly balanced.
  - Pivot is always in the middle (median of the array)

$$
\begin{aligned}
T(N) &= 2T(N/2) + cN \\
\frac{T(N)}{N} &= \frac{T(N/2)}{N/2} + c \\
\frac{T(N/2)}{N/2} &= \frac{T(N/4)}{N/4} + c \\
\frac{T(N/4)}{N/4} &= \frac{T(N/8)}{N/8} + c \\
&\vdots \\
\frac{T(2)}{2} &= \frac{T(1)}{1} + c \\
\frac{T(N)}{N} &= \frac{T(1)}{1} + c \log N \\
T(N) &= cN \log N + N = O(N \log N)
\end{aligned}
$$

Average-Case Analysis

- Assume
  - Each of the sizes for S1 is equally likely
- This assumption is valid for our pivoting (median-of-three) and partitioning strategy
- On average, the running time is O(N log N)