COMP2010
Data Structures and Algorithms

Lecture 5: Bubble sort, Insertion sort, and merge sort

Bubble Sort

- **Sorting takes an unordered collection and makes it an ordered one. Our goal is to obtain an increasing order.**

1	2	3	4	5	6
77	42	35	12	101	5

1	2	3	4	5	6
5	12	35	42	77	101

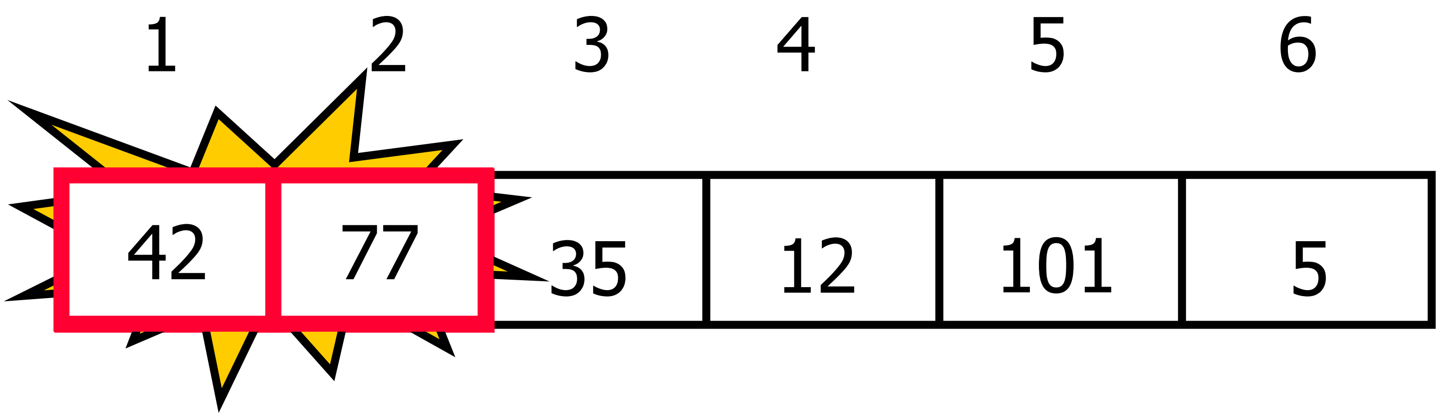
"Bubbling Up" the Largest Element

- **Traverse a collection of elements**
  - Move from the front to the end
  - "Bubble" the largest value to the end using pair-wise comparisons and swapping

1	2	3	4	5	6
77	42	35	12	101	5

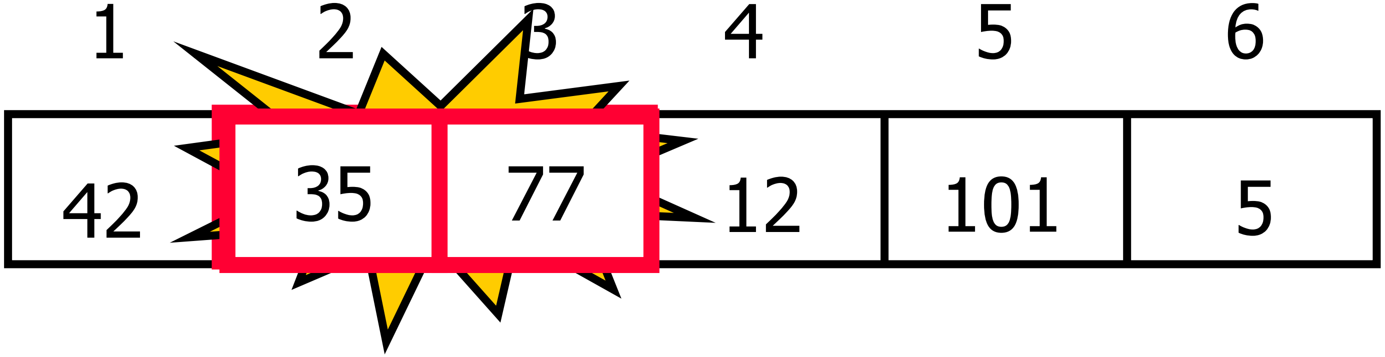
"Bubbling Up" the Largest Element

- **Traverse a collection of elements**
  - Move from the front to the end
  - “Bubble” the largest value to the end using pair-wise comparisons and swapping



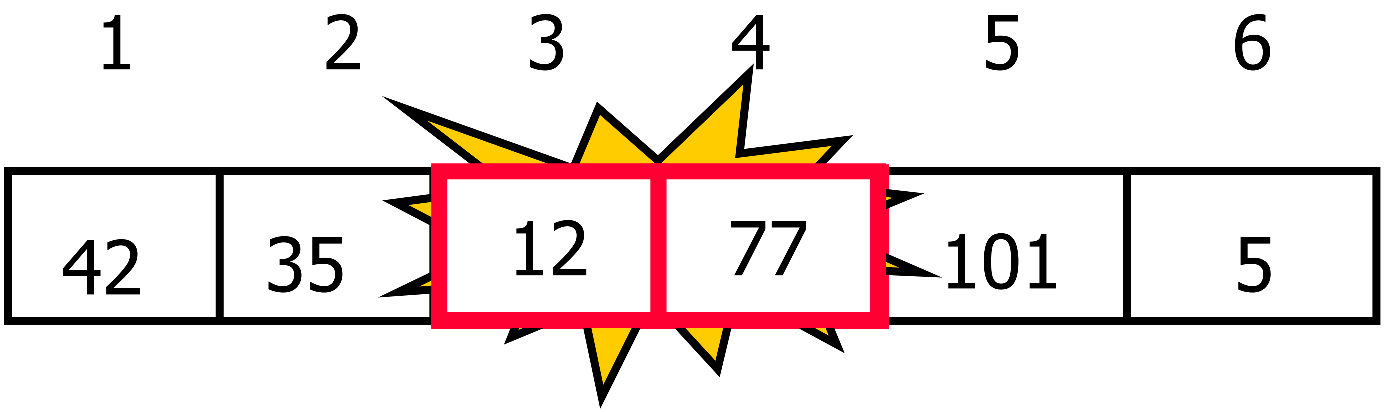
"Bubbling Up" the Largest Element

- **Traverse a collection of elements**
  - Move from the front to the end
  - "Bubble" the largest value to the end using pair-wise comparisons and swapping



"Bubbling Up" the Largest Element

- Traverse a collection of elements
  - Move from the front to the end
  - “Bubble” the largest value to the end using pair-wise comparisons and swapping



"Bubbling Up" the Largest Element

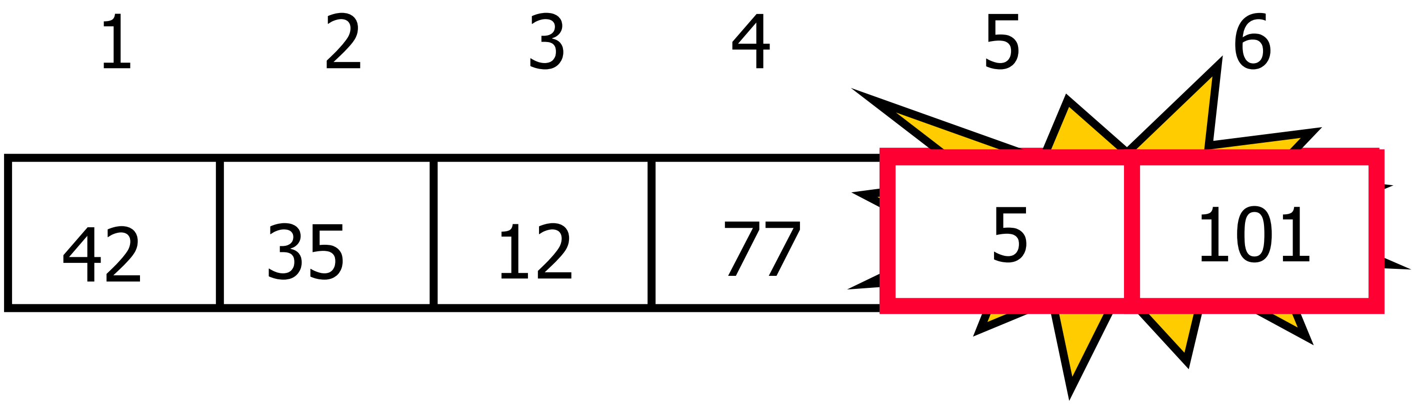
- Traverse a collection of elements
  - Move from the front to the end
  - “Bubble” the largest value to the end using pair-wise comparisons and swapping

1	2	3	4	5	6
42	35	12	77	101	5

No need to swap

"Bubbling Up" the Largest Element

- **Traverse a collection of elements**
  - Move from the front to the end
  - "Bubble" the largest value to the end using pair-wise comparisons and swapping



"Bubbling Up" the Largest Element

- **Traverse a collection of elements**
  - Move from the front to the end
  - “Bubble” the largest value to the end using pair-wise comparisons and swapping

1	2	3	4	5	6
42	35	12	77	5	101

Largest value correctly placed

Items of Interest

- Notice that only the largest value is correctly placed
- All other values are still out of order
- So we need to repeat this process

1	2	3	4	5	6
42	35	12	77	5	101

Largest value correctly placed

Repeat “Bubble Up” How Many Times?

- If we have N elements…
- And if each time we bubble an element, we place it in its correct location…
- Then we repeat the “bubble up” process N – 1 times.
- This guarantees we’ll correctly place all N elements.

“Bubbling” All the Elements

42	35	12	77	5	101
35	12	42	5	77	101
12	35	5	42	77	101
12	5	35	42	77	101
5	12	35	42	77	101

N - 1

Reducing the Number of Comparisons

77	42	35	12	101	5
42	35	12	77	5	101
35	12	42	5	77	101
12	35	5	42	77	101
12	5	35	42	77	101

Reducing the Number of Comparisons

- On the i^{th} “bubble up”, we only need to do (N-i) comparisons, where i=1,2,...,N-1.

- **For example:**
  - This is the 4^{th} “bubble up”
  - SIZE is 6
  - Thus we have 2 comparisons to do

1	2	3	4	5	6
12	35	5	42	77	101

Bubble Sort

```
void BubbleSort(int arr[], int n)
{
    for (int i = 0; i < n - 1; i++)
        {
            for (int j = 0; j < n - i - 1; j++)
                {
                    if (arr[j] > arr[j + 1])
                        {
                            int temp = arr[j];
                            arr[j] = arr[j + 1];
                            arr[j + 1] = temp;
                        }
                 }
          }
}
```

Worst-case running time: O(n^2)
Best-case running time: O(n^2)

Pseudo Code for Improved Bubble Sort: efficient implementation

```
Bubble-SORT-OPT(A) // A is a pointer
      FOR i = 0 TO n-2
          flag = false
          FOR j=0 TO n-i-2
              If A[j+1] < A[j]
                  swap A[j] and A[j+1]
                  flag = true
          IF flag is false
              break;
```

- Average Computational time: $O(n^2)$
- Worst case: $O(n^2)$. Best case: $O(n)$.

Insertion sort (similar to playing pokers)

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...

```
#include<iostream>
#include<vector>
using namespace std;
void InsertionSort(vector<int>& a) {
    for (int p  = 1;p < a.size();p++) {
        int tmp = a[p];
        int j;
        for (j=p-1;j >= 0 && tmp < a[j];j--) {
            a[j + 1] = a[j];
        }
        a[j+1] = tmp; // Note that the index is j + 1
    }
}
```

see applet http://www.cis.upenn.edu/~matuszek/cse121-2003/Applets/Chap03/Insertion/InsertSort.html

- Consists of N - 1 passes
- For pass p = 1 through N - 1, ensures that the elements in positions 0 through p are in a 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; tmp = 8;

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

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

After first pass: **8**   **34**   64   51 32   21

(first 2 elements are sorted)

p = 2; tmp = 64;

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

After second pass: 8   34   64   51 32   21

(first 3 elements are sorted)

p = 3; 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 third pass: 8   34   51 64 32   21

(first 4 elements are sorted)

p = 4; 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 fourth pass: 8   32   34   51   64   21

p = 5; tmp = 21, . . .

After fifth pass: 8   21 32 34   51   64

Analysis: worst-case running time

```
#include<iostream>
#include<vector>
using namespace std;
void InsertionSort(vector<int>& a) {
    for (int p  = 1;p < a.size();p++) {
        int tmp = a[p];
        int j;
        for (j=p-1;j >= 0 && tmp < a[j];j--) {
            a[j + 1] = a[j];
        }
        a[j+1] = tmp; // Note that the index is j + 1
    }
}
```

- For example, sort the decreasing order {N,N-1,...,2,1} would lead to the worst-case running time.
- Inner loop is executed p times, for each p=1..N-1

  $\Rightarrow$ Overall: $1 + 2 + 3 + \ldots + N\text{-}1 = O(N^2)$
- 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

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

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

Analysis of Insertion Sort

Best-case Running Time	$O(n)$
Worst-case Running Time	$O(n^2)$
Average Running Time	$O(n^2)$

- The running time of insertion sort largely depends on the input.
- It is considered an $O(n^2)$ algorithm
- Insertion sort is a stable sorting algorithm

Mergesort (归并排序, Jone von Neumann, 1945)

Based on divide-and-conquer (分治) 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
- Merge sort is a stable sorting algorithm.

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

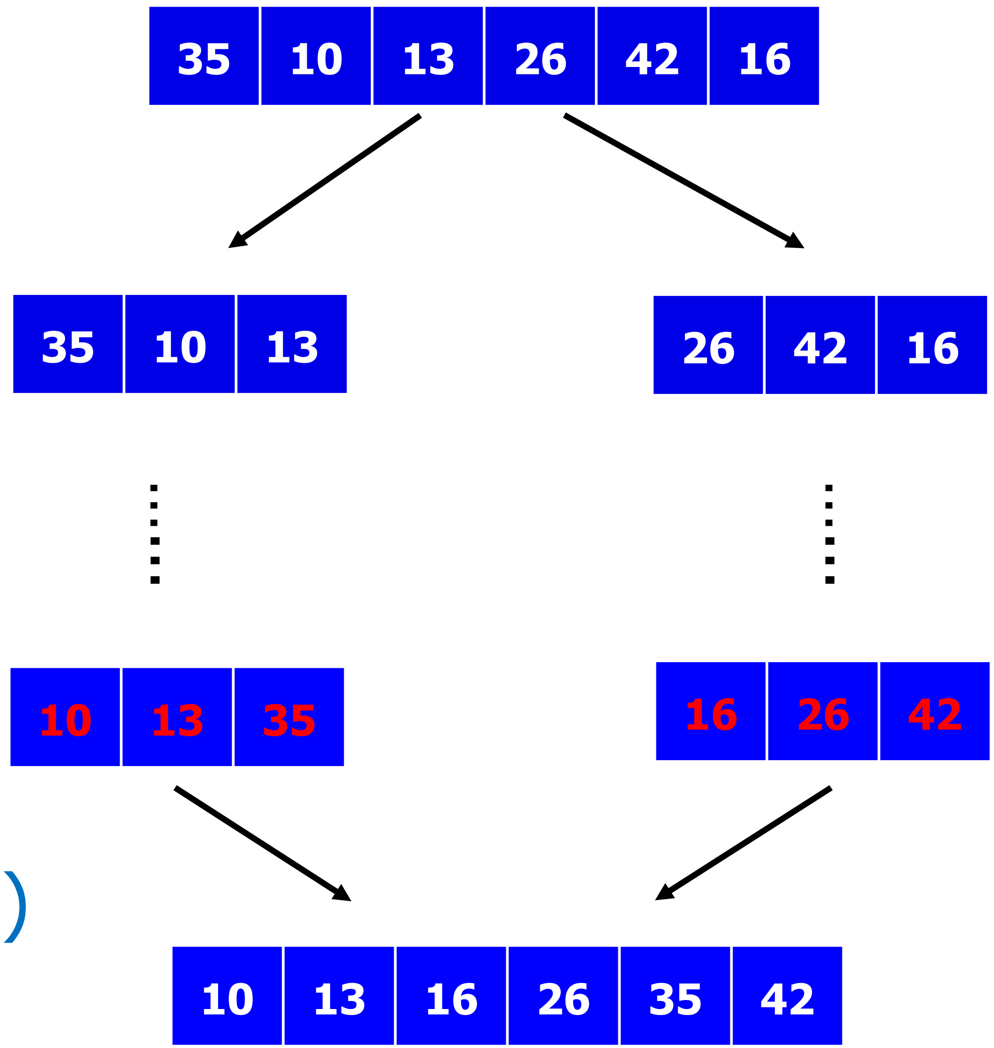
How do we merge the two sorted lists? How much time is needed?

Merge Sort

Divide (Split)

Conquer

Combine (Merge)



Dividing

- If the input list is a linked list, dividing takes $\Theta(N)$ time
  - We scan the linked list, stop at the $\lfloor \mathbf{N/2} \rfloor$-th entry and cut the list ($\lfloor \mathbf{N/2} \rfloor$ denotes the nearest integer to N/2 in the direction of negative infinity, e.g., $\lfloor 1/2 \rfloor = 0$, $\lfloor 3/2 \rfloor = 1$)
- 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=`$\lfloor$`(left+right)/2`$\rfloor$ and obtain `A[left...Center]` and `A[center+1...Right]`

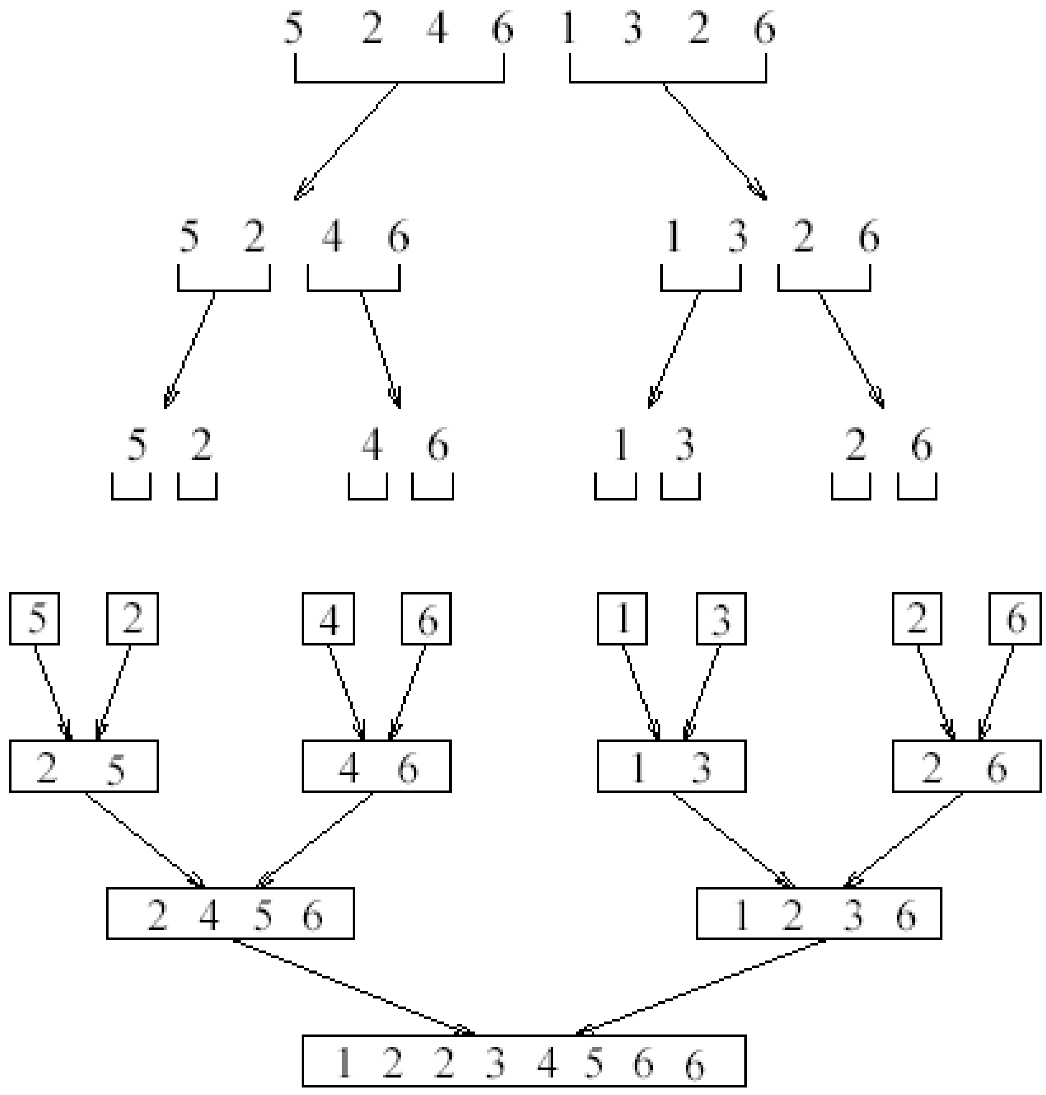
Mergesort

- Divide-and-conquer strategy
  - Recursively mergesort the first half and the second half
  - Merge the two sorted halves together
  - The base case: the subarray contains only one element

```
void MergeSort(vector<int> & A, int left, int right){
if(left>=right)return;// The base case
int center = (left+right)/2;// Divide
MergeSort(A,left,center);// Sort subarray A[left...center] into a sorted array
MergeSort(A,center+1,right);// Sort subarray A[(center+1)...right] into a sorted array
Merge(A,left,middle+1,right);// Merge the two subarrays into a large sorted array
}
```

The simplified one

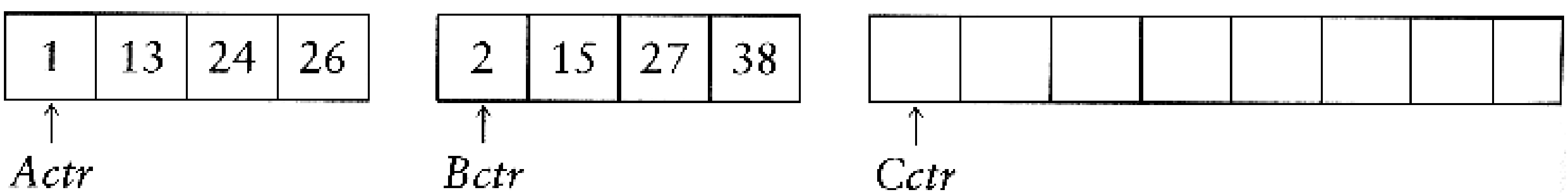
```
void mergesort(vector<int> & A, int left, int right)
{
    if (left < right) {
        int center = (left + right)/2;
        mergesort(A,left,center);
        mergesort(A,center+1,right);
        merge(A,left,center+1,right);
    }
}
```



see applet http://www.cosc.canterbury.ac.nz/people/mukundan/dsal/MSort.html

How to merge?

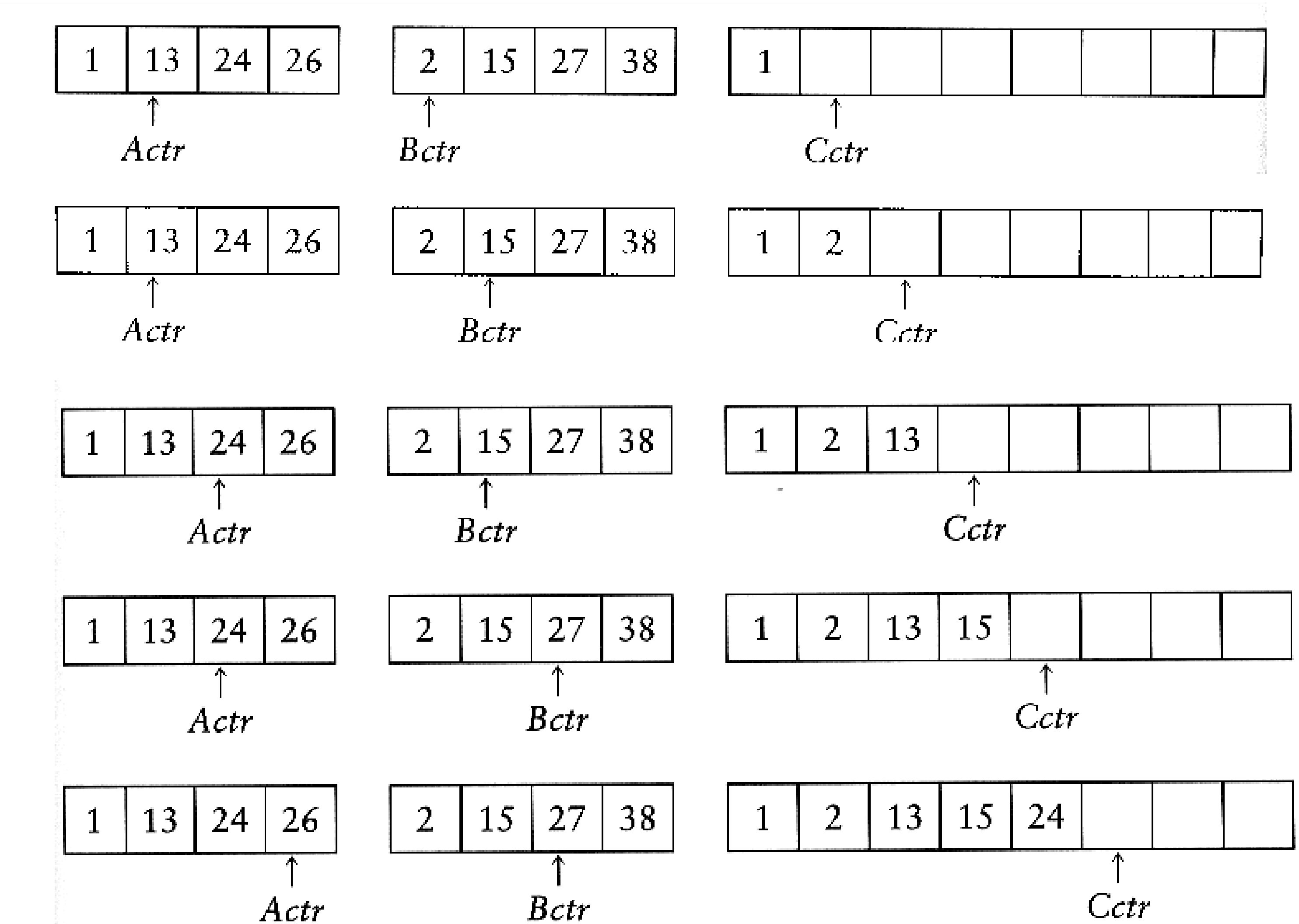
- Input: two sorted arrays A and B
- Output: a sorted array C
- Three counters: Actr, Bctr, and Cctr
  - initially set to the beginning of their respective arrays



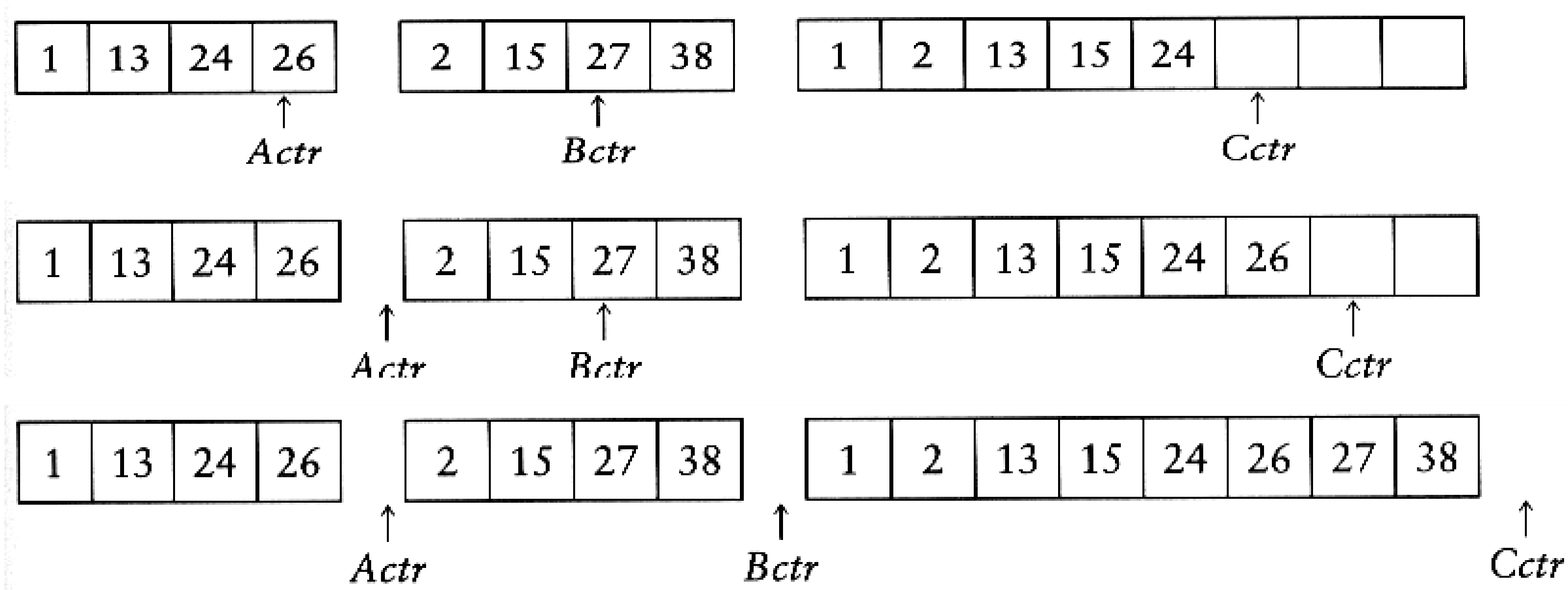
(1) The smaller of A[Actr] and B[Bctr] is copied to the next entry in C, and the appropriate counters are advanced

(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. $p \le l = q - 1 < r$.

Output: $A[p..r]$ is sorted.

(* T is a temporary array. *)

$k = p$; $i = 0$; $l = q - 1$;
while $p \le l$ and $q \le r$
    do if $A[p] \le A[q]$
            then $T[i] = A[p]$; $i = i + 1$; $p = p + 1$;
            else $T[i] = A[q]$; $i = i + 1$; $q = q + 1$;
while $p \le l$
    do $T[i] = A[p]$; $i = i + 1$; $p = p + 1$;
while $q \le r$
    do $T[i] = A[q]$; $i = i + 1$; $q = q + 1$;
for $i = k$ to r
    do $A[i] = T[i - k]$;

```
void Merge(vector<int>& A, int left, int q, int right) {
    // Merge two sorted subarrays A[left...(q-1)] and A[q...right]
    int i = left, center = q - 1, j = q;// Then the subarrays are A[left...center] and A[(center+1) ... right]
    int n = right - left + 1;// The number of elements in array A[left...right]
    vector<int> C(n); // Creat an extra array of size n
    int k =0;
    while (i <= center && j<=right) {
        if (A[i] <= A[j]) {
            C[k] = A[i];
            i++; k++;
        }
        else {
            C[k] = A[j];
            j++; k++;
        }
    }
    while (i <= center) {
        C[k] = A[i];
        i++; k++;
    }
    while (j <= right) {
        C[k] = A[j];
        j++; k++;
    }
    for (i = left;i <= right;i++)
        A[i] = C[i-left];
}
```

Analysis of mergesort

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

Assume that N is a power of 2, that is $N = 2^k$.

- 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

- Code from textbook (using template)
- 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

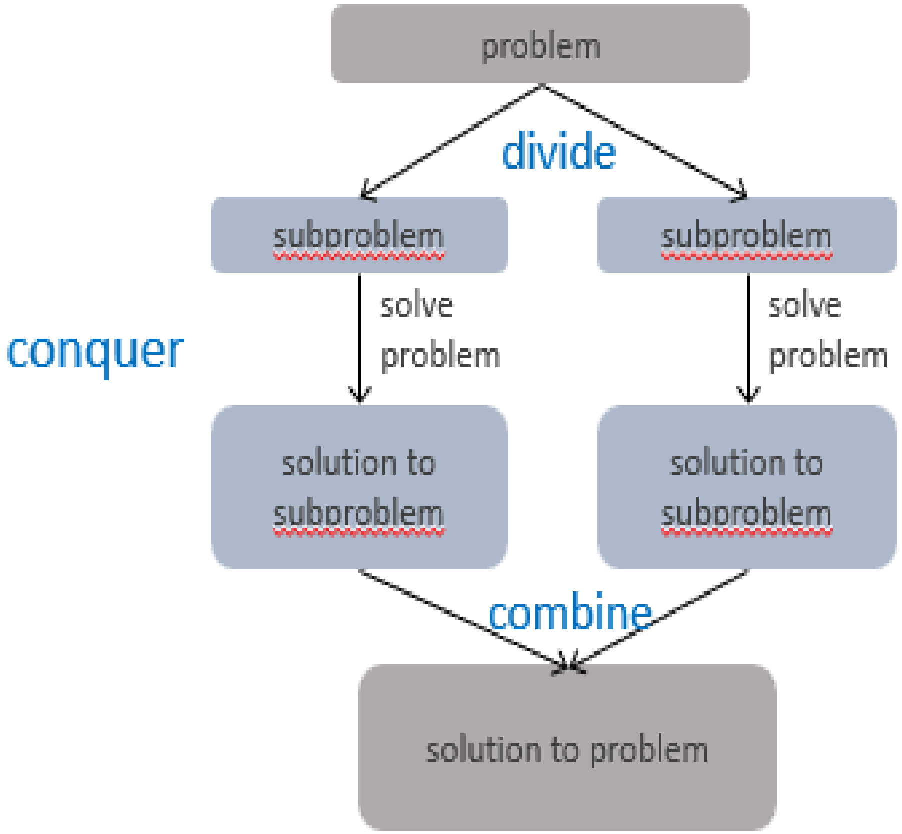
Divide and Conquer (DC)

If the problem is large, break it into sub-problems that are smaller in size but are similar in structure to the original problem, **recursively** solve the sub-problems, and finally combine the sub-solutions into a final solution that solves the original problem.

Three Phases of DC

- Divide: top $\longrightarrow$ bottom
  - Divide a problem into sub-problems
- Conquer: bottom level
  - Solve the sub-problems recursively
  - If the sub-problems are small enough, solve them as base cases
- Combine: bottom $\longrightarrow$ top
  - Combine the solutions to the sub-problems into that of the original problem
  - Usually the key!

Divide-Conquer-Combine



Divide-Conquer-Combine