Algorithms and Data Structures
Lecture notes: Merge sort and quicksort

Lecturer: Michel Toulouse

Hanoi University of Science & Technology
michel.toulouse@soict.hust.edu.vn

20 avril 2023

Introduction

This section introduces two divide-and-conquer sorting algorithms, mergesort and quicksort

Iterative sorting algorithms

These two algorithms run in $\Theta(n^2)$ using worst case analysis

```
Selection sort(A[1..n])
for i = 1 to n − 1 do
    minj = i ; minx = A[i] ;
    for j = i + 1 to n do
        if A[j] < minx then
            minj = j ; minx = A[j] ;
    A[minj] = A[i] ; A[i] = minx ;
```

```
InsertionSort(A[1..n])
for (i = 2; i <= n; i++)
    v=A[i] ; j=i-1 ;
    while(j > 0 & A[j] > v)
            A[j+1]=A[j] ; j− ;
    A[j+1] = v ;
```

Merge Sort

Merge sort is a divide-&-conquer sorting algorithm. It works as follow :

- **Divide** the array to sort in half.
- **Recursively** sort each half.
- **Merge** the two sorted halves.

Pseudo-code of merge sort is as follows :

Mergesort($A[p..r]$)
  if $p < r$
    $q = \lfloor \frac{p+r}{2} \rfloor$
    Mergesort($A[p..q]$)
    Mergesort($A[q+1..r]$)
    Merge(A, p, q, r)

Merge function

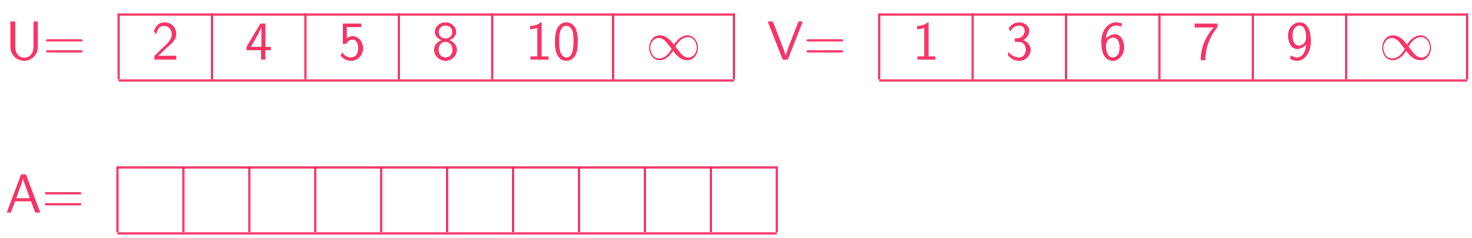
The merge function is a non-recursive component of merge sort which combines 2 sub-arrays already sorted

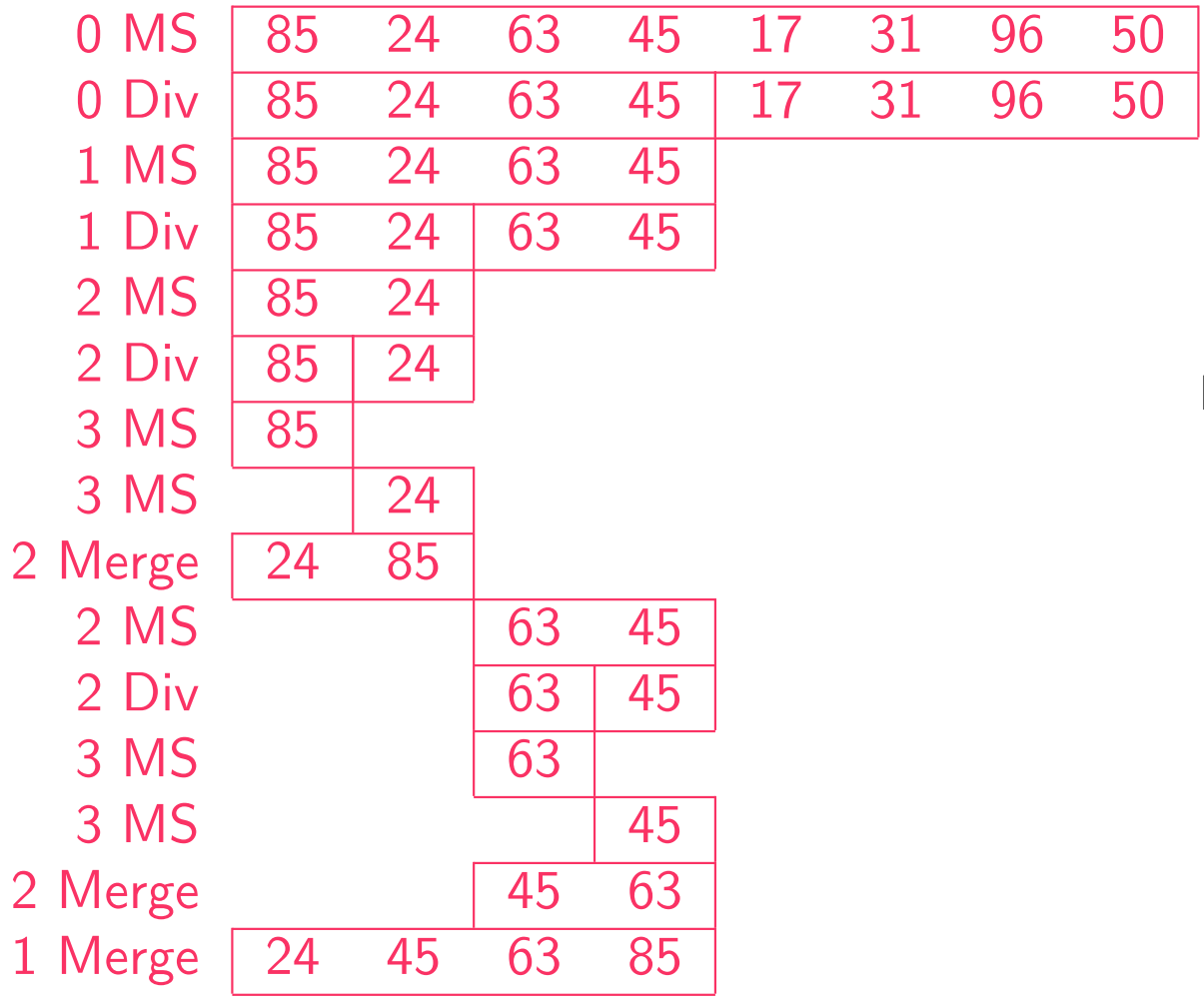
Takes in input 2 sorted sub-arrays U and V respectively of size m and n and return a sorted array A of size $m + n$

$Merge(U[1..m+1], V[1..n+1], A[1..m+n])$;
  $i = j = 1; U[m+1] = V[n+1] = \infty$;
  for $(k = 1; k \leq m + n; k++)$
    if $U[i] < V[j]$ **then**
      $A[k] = U[i]$; $i = i + 1$;
    else
      $A[k] = V[j]$; $j = j + 1$;

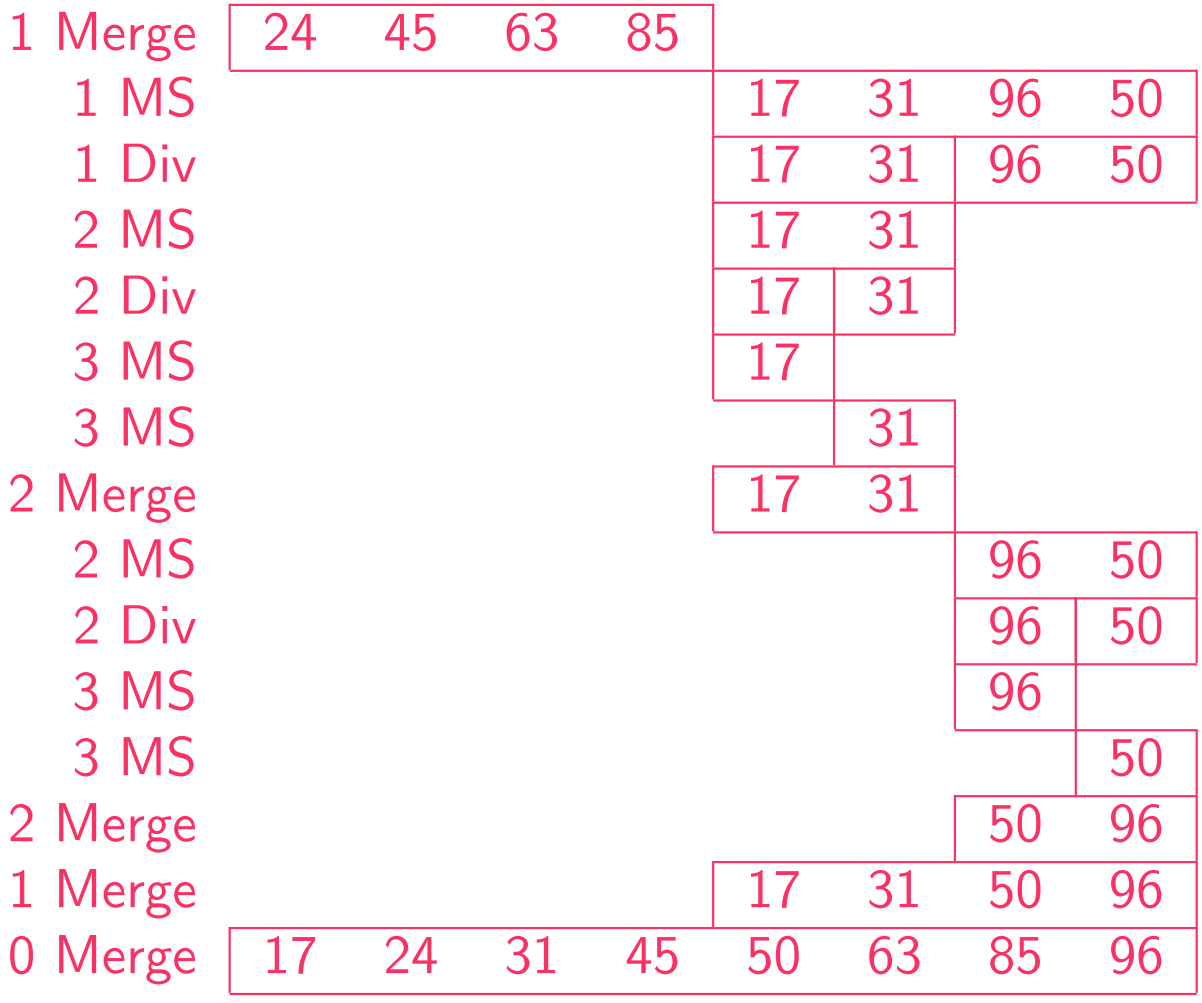
Merge algorithm

```
Merge(U[1..m+1], V[1..n+1], A[1..m+n]);
  i = j = 1; U[m+1] = V[n+1] = ∞;
  for (k = 1; k ≤ m+n; k++)
      if U[i] < V[j] then
          A[k] = U[i]; i = i+1;
      else
          A[k] = V[j]; j = j+1;
```





Mergesort($A[p..r]$)
  if $p < r$
    $q = \lfloor \frac{p+r}{2} \rfloor$
    Mergesort($A[p..q]$)
    Mergesort($A[q+1..r]$)
    Merge(A, p, q, r)


1 Merge
24 45 63 85
1 MS
17 31 96 50
1 Div
17 31 96 50
2 MS
17 31
2 Div
17 31
3 MS
17
3 MS
31
2 Merge
17 31
2 MS
96 50
2 Div
96 50
3 MS
96
3 MS
50
2 Merge
50 96
1 Merge
17 31 50 96
0 Merge
17 24 31 45 50 63 85 96

Divide-and-Conquer

Running time of merge sort

The running time of recursive algorithms cannot be analyzed in the same way as iterative algorithms

There are no loops, so summations to count the number of basic operations in loops do not apply for recursive algorithms

Rather recurrence relations will be used to express the number of times basic operations are executed in recursive algorithms

Mergesort($A[p..r]$)
  if $p < r$
    $q = \lfloor \frac{p+r}{2} \rfloor$
    Mergesort($A[p..q]$)
    Mergesort($A[q+1..r]$)
    Merge(A, p, q, r)

Recurrence relations

The equivalent to polynomial expressions such as $n^2 + 6n + 4$ for iterative algorithms is recurrence relations
A recurrence relation is an equation in which at least one term appears on both sides of the equation such as

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

in which the term $T()$ appears on both side of the equation.

The above recurrence must be read as "the running $T(n)$ on an instance of size n is equal to twice the running time of an instance of size $n-1$ (i.e. $2T(n-1)) + n$"

The recurrence for merge sort

The recurrence relation for merge sort is

$$\begin{aligned} T(n) &= T(\tfrac{n}{2}) + T(\tfrac{n}{2}) + \Theta(n) \\ &= 2T(n/2) + \Theta(n) \end{aligned}$$

which must be read as "the running time $T(n)$ of merge sort is equal to :

- the running time of solving two input instances of input size $\frac{n}{2}$ plus
- the running time of merging two sub-arrays of size n, $\Theta(n)$"

Mergesort($A[p..r]$)
  if $p < r$
    $q = \lfloor \frac{p+r}{2} \rfloor$
    Mergesort($A[p..q]$)
    Mergesort($A[q+1..r]$)
    Merge(A, p, q, r)

Quicksort

Another divide-and-conquer sorting algorithm

An array A[p..r] of integers is partitioned into two subarrays A[p..q] and A[q+1..r]

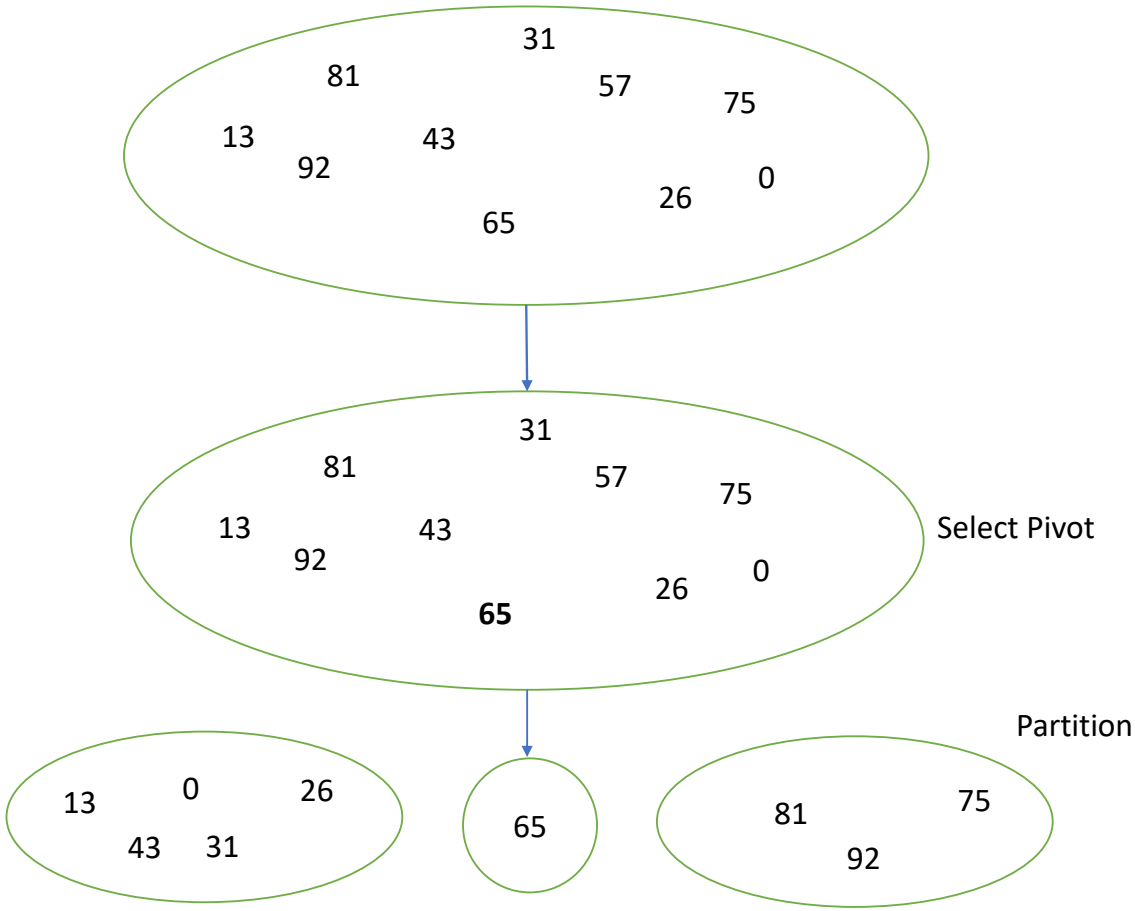
- such that the elements in A[p..q] are less than all elements in A[q+1..r]

The subarrays are recursively sorted by calls to quicksort

Quicksort Code

```
Quicksort(A, p, r)
  if (p < r) /* the array has at least 2 elements */
    q = Partition(A, p, r); /* q is the index in A of the pivot element */
    Quicksort(A, p, q-1);
    Quicksort(A, q+1, r);
```

Quicksort idea



After partition, the pivot is in its right position in a sorted array. Quicksort is recursively applied to the next two subarrays.

Partition

```
Quicksort(A, p, r)
  if (p < r)
    q = Partition(A, p, r);
    Quicksort(A, p, q-1);
    Quicksort(A, q+1, r);
```

- All the action takes place in the partition() function which rearranges the subarray
- End result : Two subarrays, all values in first subarray $<$ all values in second
- Returns the index of the "pivot" element separating the two subarrays

Partition code

Partition() takes in input the array A and the section between indexes p and r to partition

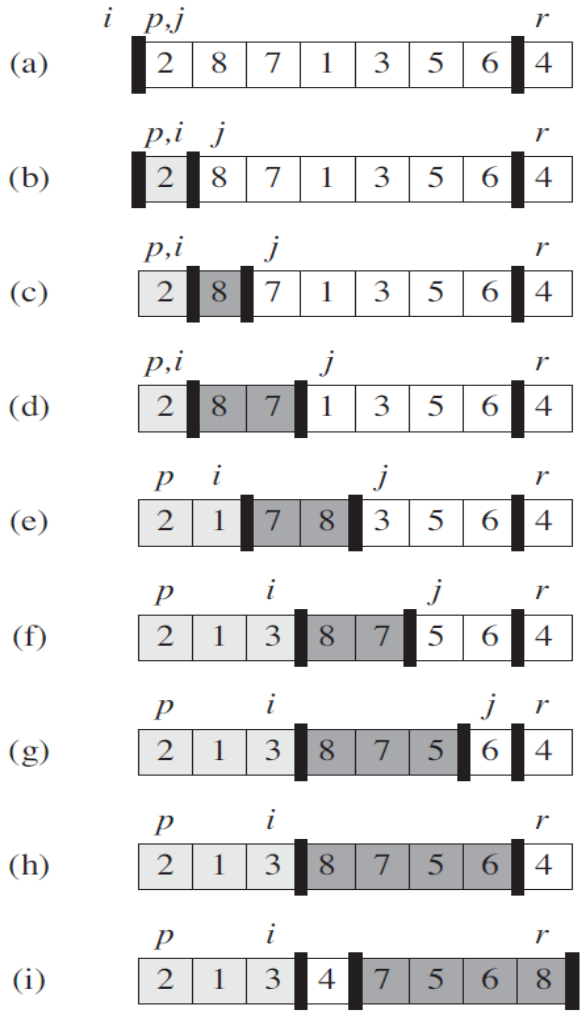
x is the pivot element, in this code the last element in the array is selected as pivot

Partition() returns the index in A of the pivot element once partition is completed

The position of the pivot element will never be part of a sub-array to partition, it is at its right position in the sorted array

```
Partition(A, p, r)
    x = A[r];
    i = p - 1;
    for (j = p; j < r; j++)
       if (A[j] ≤ x)
          i = i +1;
          swap(A[i],A[j]);
    swap(A[i+1],A[r]);
    return i+1;
```

Example with running partition()




```
Partition(A, p, r)
    x = A[r] ;
    i = p - 1 ;
    for (j = p ; j < r ; j++)
        if (A[j] ≤ x)
            i = i +1 ;
            swap(A[i],A[j]) ;
    swap(A[i+1],A[r]) ;
    return i+1 ;
```

Performance of Quicksort

Quicksort calls itself on 2 sub-arrays with likely different sizes. The asymptotic running time of Quicksort depends on how these two sub-arrays are balanced

$$T(n) = T(L) + T(R) + O(n)$$

where L and R are the number of elements in each sub-array.

- If the subarrays are balanced, then quicksort can run as fast as merge sort.
- If they are unbalanced, then quicksort can run as slowly as insertion sort.

Worst case analysis of Quicksort

Occurs when the subarrays are completely unbalanced for each recursive call. Has 0 element in one subarray and $n-1$ elements in the other subarray. The recurrence is

$$T(n) = T(n-1) + T(0) + O(n) \quad (1)$$
$$= T(n-1) + O(n) \quad (2)$$

This recurrence cannot be solved using the Master theorem, but can be solved using the substitution method or the recursion tree method (will see them later)

Intuitively, since only element of the array is sorted properly at each recursive call (the pivot element), the input size reduces only by 1, thus $n-1$ recursive calls are made.

Each recursive call entails a call to partition() which runs in $O(n)$, thus the worst case running time of Quicksort is $O(n^2)$

```
Quicksort(A, p, r)
  if (p < r)
    q = Partition(A, p, r);
    Quicksort(A, p, q-1);
    Quicksort(A, q+1, r);
```