QuickSort

- The quick sort algorithm designed by Hoare is a simple yet highly efficient algorithm.
- It works as follows:
  - Start with the given array A of n elements.
  - Consider a pivot, say A[n].
  - Now, partition the elements of A into two arrays A_L and A_R such that:
    - the elements in A_L are less than A[n]
    - the elements in A_R are greater than A[n].
  - Sort A_L and A_R, recursively.

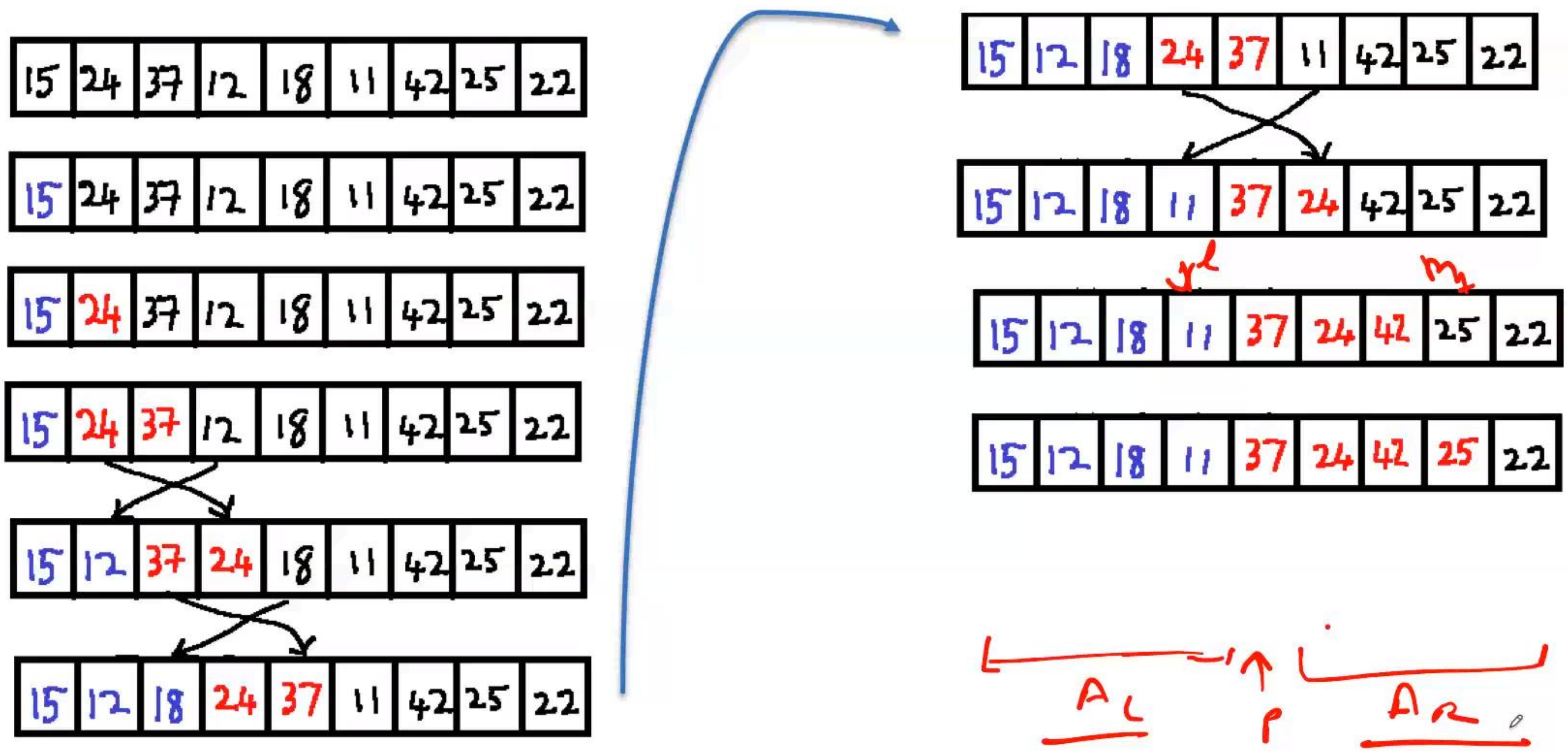
QuickSort

- How to partition ?
  - Suppose we take each element, compare it with A[n] and then move it to A_L or A_R accordingly.
  - Works in O(n) time.
  - Can write the program easily.
  - But, recall that space is also an resource. The above approach requires extra space for the arrays A_L and A_R
  - A better approach exists.

QuickSort

```
Procedure Partition(A,n)
begin
  pivot = A[n];
  less = 0; more = 1;
  for more = 1 to n-1 do
    if A[more] < pivot then
      less++;
      swap(A[more], A[less]);
    end
  end
  swap(A[less+1], A[n]);
end
```

QuickSort


15 24 37 12 18 11 42 25 22
15 24 37 12 18 11 42 25 22
15 24 37 12 18 11 42 25 22
15 24 37 12 18 11 42 25 22
15 12 37 24 18 11 42 25 22
15 12 18 24 37 11 42 25 22
15 12 18 24 37 11 42 25 22
15 12 18 11 37 24 42 25 22
15 12 18 11 37 24 42 25 22
15 12 18 11 37 24 42 25 22
A L
p
A R

Analyzing Quick Sort

- We know that $|A_L| + |A_R| = n-1$.
- But, if the pivot is such that all elements are smaller (or larger) than the pivot, then $|A_L|$ (or $|A_R|$) $= n-1$.
- The recurrence relation in that case is

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

- Suppose the same situation happens over every recursive call. So, the above recurrence relation holds during every recursive call.
- When will this happen ?

Analyzing Quick Sort

$n_1 + n_2 = n-1$

- We know that $\cancel{|A_L| + |A_R|} = n-1$.
- But, if the pivot is such that all elements are smaller (or larger) than the pivot, then $|A_L|$ (or $|A_R|$) = n-1.
- The recurrence relation in that case is

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

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

- Suppose the same situation happens over every recursive call. So, the above recurrence relation holds during every recursive call.
- When will this happen ?

Analyzing Quick Sort

- In general, if the sizes of $|A_L|$ and $|A_R|$ are such that they are a constant away from each other, then the recurrence relation is:

  $$T(n) = T(an) + T((1-a)n) + O(n)$$

  where a is a constant < 1.

- In practice, it turns out that most often the partitions are not too skewed.
- So, quick sort runs in O(n log n) time almost always.

27:21 Request control

Divide and Conquer

- Divide the problem P into $k \geq 2$ sub-problems $P_1, P_2, \ldots, P_k$.
- Solve the sub-problems $P_1, P_2, \ldots, P_k$.
- Combine the solutions of the sub-problems to arrive at a solution to P.

29:02

Request control

Divide and Conquer

- A useful paradigm with several applications.
- Examples include merge sort, convex hull, median finding, matrix multiplication, and others.
- Typically, the sub-problems are solved recursively.
  - Recurrence relation

$$T(n) = D(n) + \sum_i T(n_i) + C(n)$$

Divide time

Recursive cost

Combine time

Algorithm Merge

```
Algorithm Merge(L, R)
// L and R are two sorted arrays of size n each.
// The output is written to an array A of size 2n.
int i=1, j=1;
L[n+1] = R[n+1] = MAXINT; // so that index does not
                          // fall over
for k = 1 to 2n do
        if L[i] < R[j] then
                A[k] = L[i]; i++;
        else
                A[k] = R[j]; j++;
end-for
```

Time complexity is O(n).

From Merging to Sorting

- How to use merging to finally sort?
- Using the divide and conquer principle
  - Divide the input array into two halves.
  - Sort each of them.
  - Merge the two sub-arrays. This is indeed procedure Merge.
- The algorithm can now be given as follows.

Analyzing Merge Sort

- Recurrence relation for merge sort as:

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

  - This can be explained by the O(n) time for merge and
  - The two subproblems obtained during the divide step each take T(n/2) time.
  - Now use the general format for divide and conquer based algorithms.
- Solving this recurrence relation is done using say the substitution method giving us $T(n) = O(n \log n)$.
  - Look at previous examples.

Algorithm Merge Sort

```
Algorithm MergeSort(A)
begin
        mid = n/2; //divide step
        L = MergeSort(A[1..mid]);
        R = MergeSort(A[mid+1..n]);
        Merge(L, R); //combine step
end-Algorithm
```

Maxima of n Numbers

- A sequential program resembles the code below.

```
Program Maxima(A)
Begin
        int max = A[1];
        for i = 1 to n do
                if max < A[i]
                        max = A[i]
                End-if
        End-for
End
```

- How do we run this program in parallel?

Parallel Merge Sort

- Need to rethink on a parallel merge algorithm
- Start from the beginning.
  - We have two sorted arrays L and R.
  - Need to merge them into a single sorted array A.
- Define the rank of an element x in a sorted array A as the number of elements of A that are smaller than x.
- To merge L and R, need to know the rank of every element from L and R in the merged array L U R.

Parallel Merge Sort

- Now, consider an element x in L at index k.
- How many elements of L are smaller than x?
  - k-1.
- How many elements of R are smaller than x?
  - No easy answer, but
  - can do binary search for x in R and get the answer.
  - Say k' elements in R are smaller than x.

01:01:04 Request control

Parallel Merge Sort

L = [8 10 12 27]   R = [15 17 24 32]

Element	8	10	12	27	15	17	24	32
Rank in L	0	1	2	3	3	3	3	4
Rank in R	0	0	0	3	0	1	2	3
Rank in L U R	0	1	2	6	3	4	5	7

L U R = [8 10 12 15 17 24 27 32]

01:01:21 Request control

AS

Parallel Merge Sort

```
Algorithm ParallelMergeSort(A)
Begin
        mid = n/2; //divide step
        L = MergeSort(A[1..mid]);
        R = MergeSort(A[mid+1..n]);
        ParallelMerge(L, R); //combine step
 end-Algorithm
```

Parallel Merge Sort

- So, we just have to figure out a way to merge in parallel.

- Recall the merge algorithm as we developed it earlier.
  - Too many dependent tasks.
  - Not feasible in a parallel model.

```
for k = 1 to 2n do
        if L[i] < R[j] then
                A[k] = L[i]; i++;
        else A[k] = R[j]; j++;
end-for
```