DATA STRUCTURE & ALGORITHM

Introduction

Lecture 4

Instructor: Engr. Tayyab Abdul Hannan
Email: tayyab.abdulhannan@gmail.com

Quick-sort *(1)*

- Idea: Choose any value from the array (called the **pivot**). Then **partition** the array into three subarrays such that:
  - the left subarray contains only values less than (or equal to) the pivot;
  - the middle subarray contains only the pivot;
  - the right subarray contains only values greater than (or equal to) the pivot.

  Finally sort the left subarray and the right subarray separately.
- This is another application of the divide-and-conquer strategy.

Quick-sort *(2)*

- **Quick-sort algorithm**:

To sort *a*[*left*…*right*] into ascending order:

1. If *left* < *right*:

   1.1. Partition *a*[*left*…*right*] such that

   a[*left*…*p*–1] are all less than or equal to *a*[*p*], and

   a[*p*+1…*right*] are all greater than or equal to *a*[*p*].

   1.2. Sort *a*[*left*…*p*–1] into ascending order.

   1.3. Sort *a*[*p*+1…*right*] into ascending order.

2. Terminate.

Quick-sort *(3)*

- Animation:

To sort $a[left \ldots right]$ into ascending order:

1. If $left < right$:
   1.1. Partition $a[left \ldots right]$ such that
        $a[left \ldots p–1]$ are all less than or equal to $a[p]$, and
        $a[p+1 \ldots right]$ are all greater than or equal to $a[p]$.
   1.2. Sort $a[left \ldots p–1]$ into ascending order.
   1.3. Sort $a[p+1 \ldots right]$ into ascending order.
2. Terminate.

	left = 0	1	2	3	4	5	6	7	8 = *right*
a	cat	cow	dog	fox	goat	lion	pig	rat	tiger

Quick-sort *(4)*

- In the **best case**, the pivot turns out to be the median value in the array. So the left and right subarrays both have length about $n/2$. Then steps 1.2 and 1.3 take about $comps(n/2)$ comparisons each.

  Therefore:

  $$comps(n) \approx 2\,comps(n/2) + n - 1 \quad \text{if } n > 1$$
  $$comps(n) = 0 \quad \text{if } n \leq 1$$

  Solution:

  $$comps(n) \approx n \log_2 n$$

  Best-case time complexity is $O(n \log n)$.

Quick-sort *(5)*

- In the **worst case**, the pivot turns out to be the smallest value. So the right subarray has length n–1 whilst the left subarray has length 0. Then step 1.3 performs $comps(n–1)$ comparisons, but step 1.2 does nothing at all.

  Therefore:

  $$comps(n) \approx comps(n-1) + n - 1 \quad \text{if } n > 1$$
  $$comps(n) = 0 \quad \text{if } n \leq 1$$

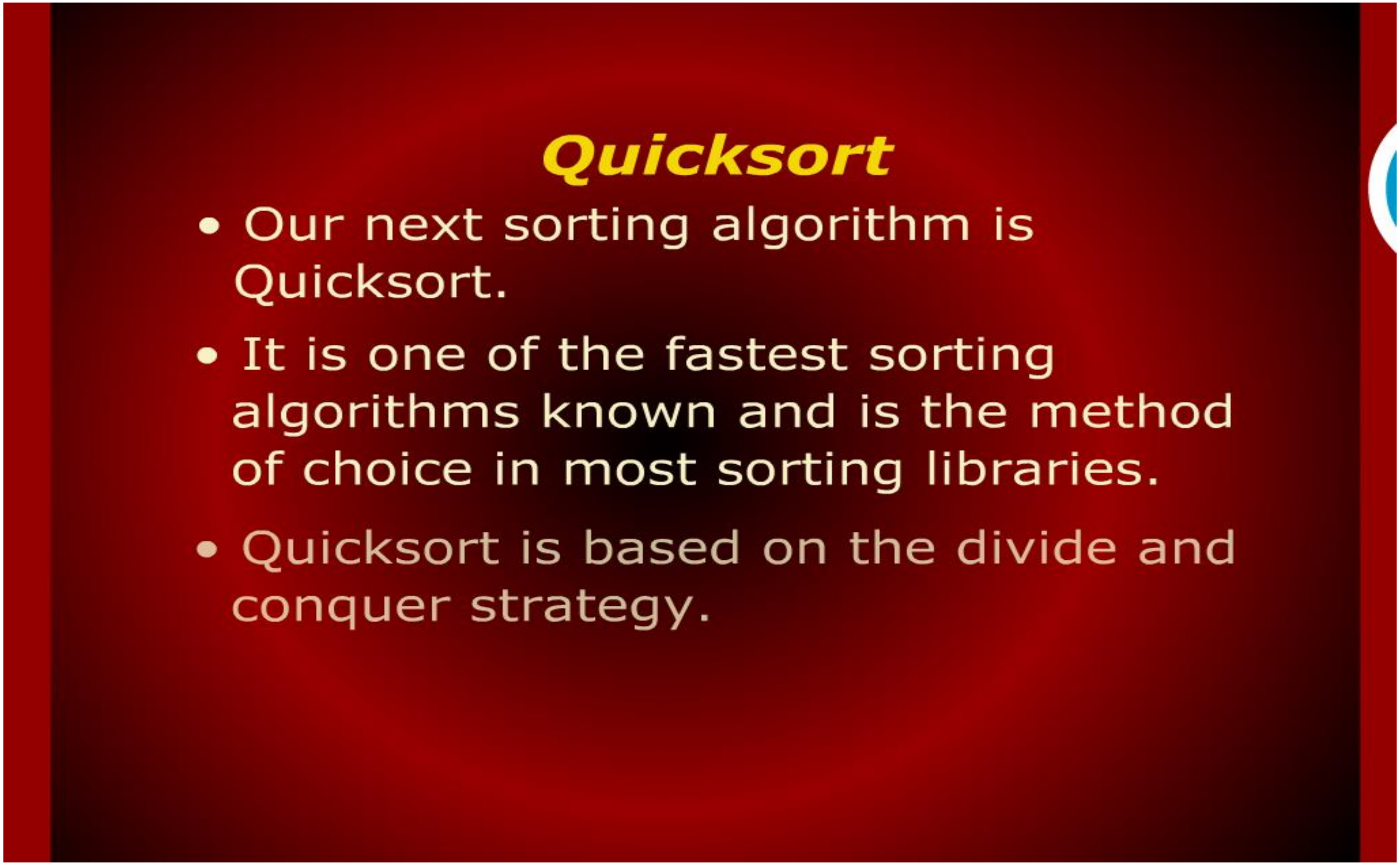
  Solution:

  $$comps(n) \approx (n^2 - n)/2$$

  Worst-case time complexity is $O(n^2)$.

- The worst case arises if the array is already sorted!

QUICK SORT


Quicksort
• Our next sorting algorithm is Quicksort.
• It is one of the fastest sorting algorithms known and is the method of choice in most sorting libraries.
• Quicksort is based on the divide and conquer strategy.

QUICK SORT

Quicksort

```
QUICKSORT( array A, int p, int r)
if (r > p)
    then
      i ← a random index from [p..r]
        swap A[i] with A[p]
      q ← PARTITION(A, p, r)
      QUICKSORT(A, p, q – 1)
      QUICKSORT(A, q + 1, r)
```

QUICK SORT

Partition Algorithm

Recall that the partition algorithm partitions the array A[p..r] into three sub arrays about a pivot element x.

- A[p..q – 1] whose elements are less than or equal to x,
- A[q] = x,
- A[q + 1..r] whose elements are greater than x

QUICK SORT

Choosing the Pivot

- We will choose the first element of the array as the pivot, i.e. x = A[p].
- If a different rule is used for selecting the pivot, we can swap the chosen element with the first element.
- We will choose the pivot randomly.

QUICK SORT

Partition Algorithm

The algorithm works by maintaining the following *invariant condition.*

1. A[p] = x is the pivot value.
2. A[p..q – 1] contains elements that are less than x.
3. A[q + 1..s – 1] contains elements that are greater than or equal to x
4. A[s..r] contains elements whose values are currently unknown.

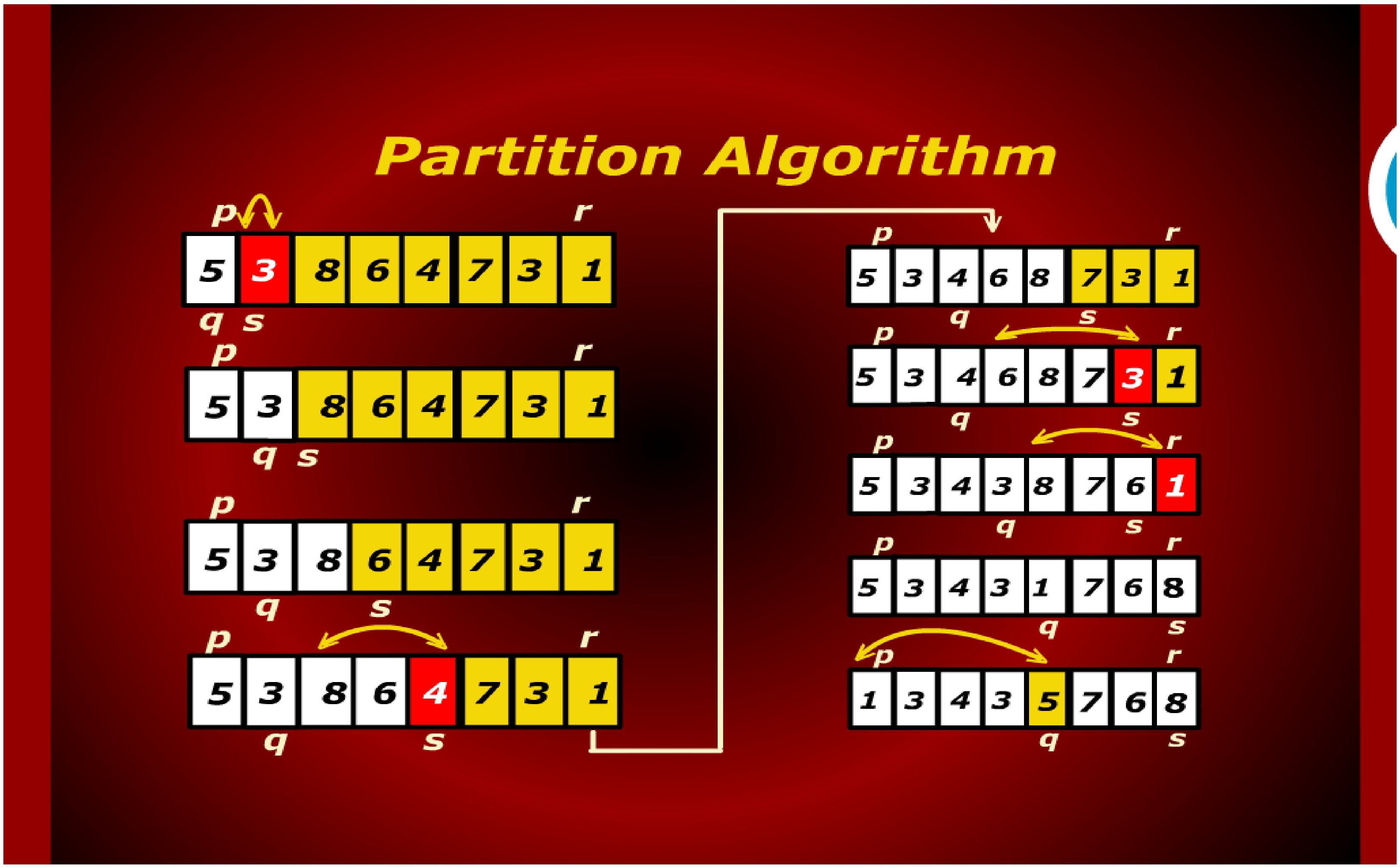
QUICK SORT

Partition Algorithm

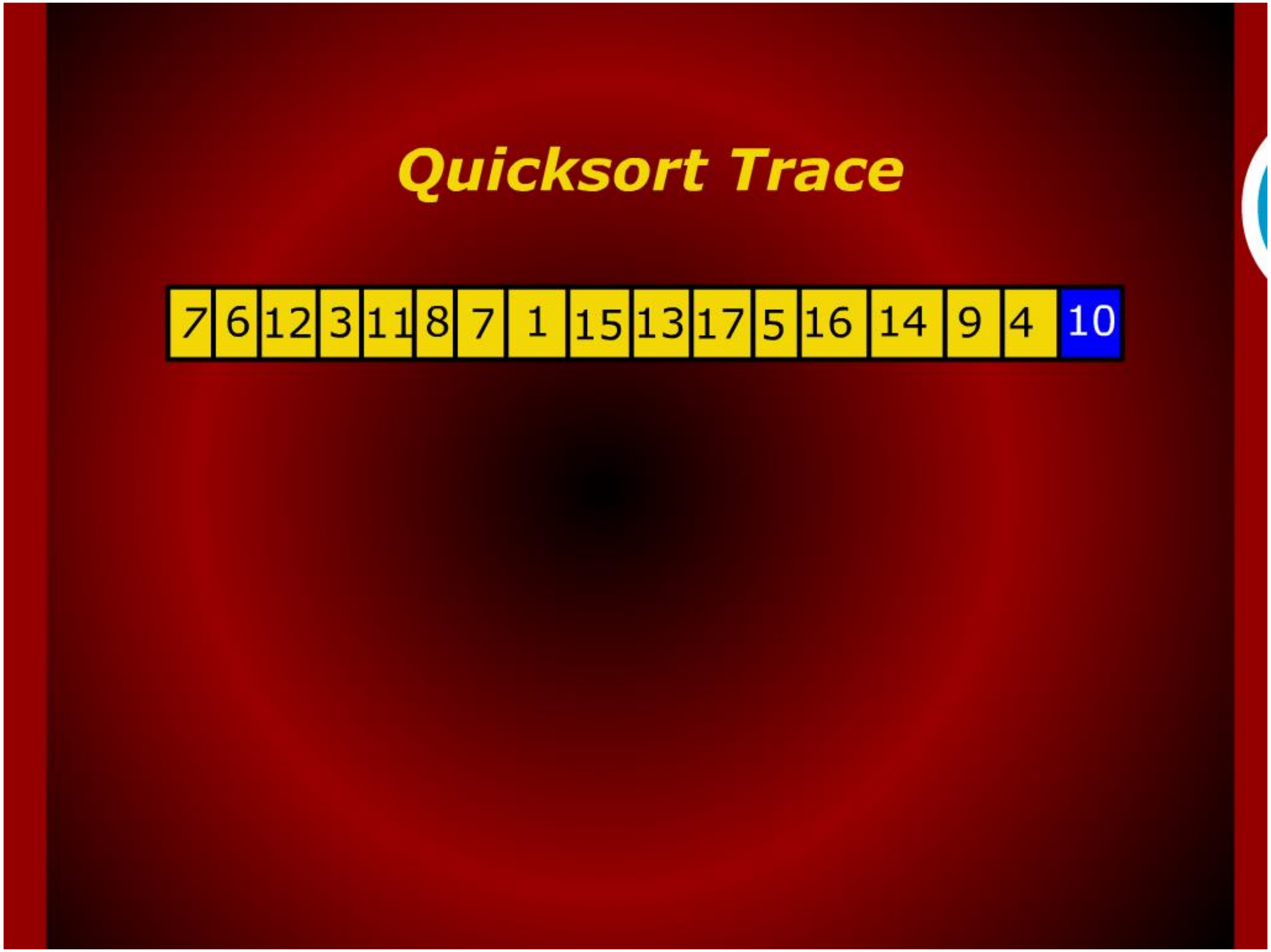
```
PARTITION( array A, int p, int r)
x ← A[p]
q ← p
for s ← p + 1 to r
do if (A[s] < x)
        then q ← q + 1
                swap A[q] with A[s]

 swap A[p] with A[q]
return q
```

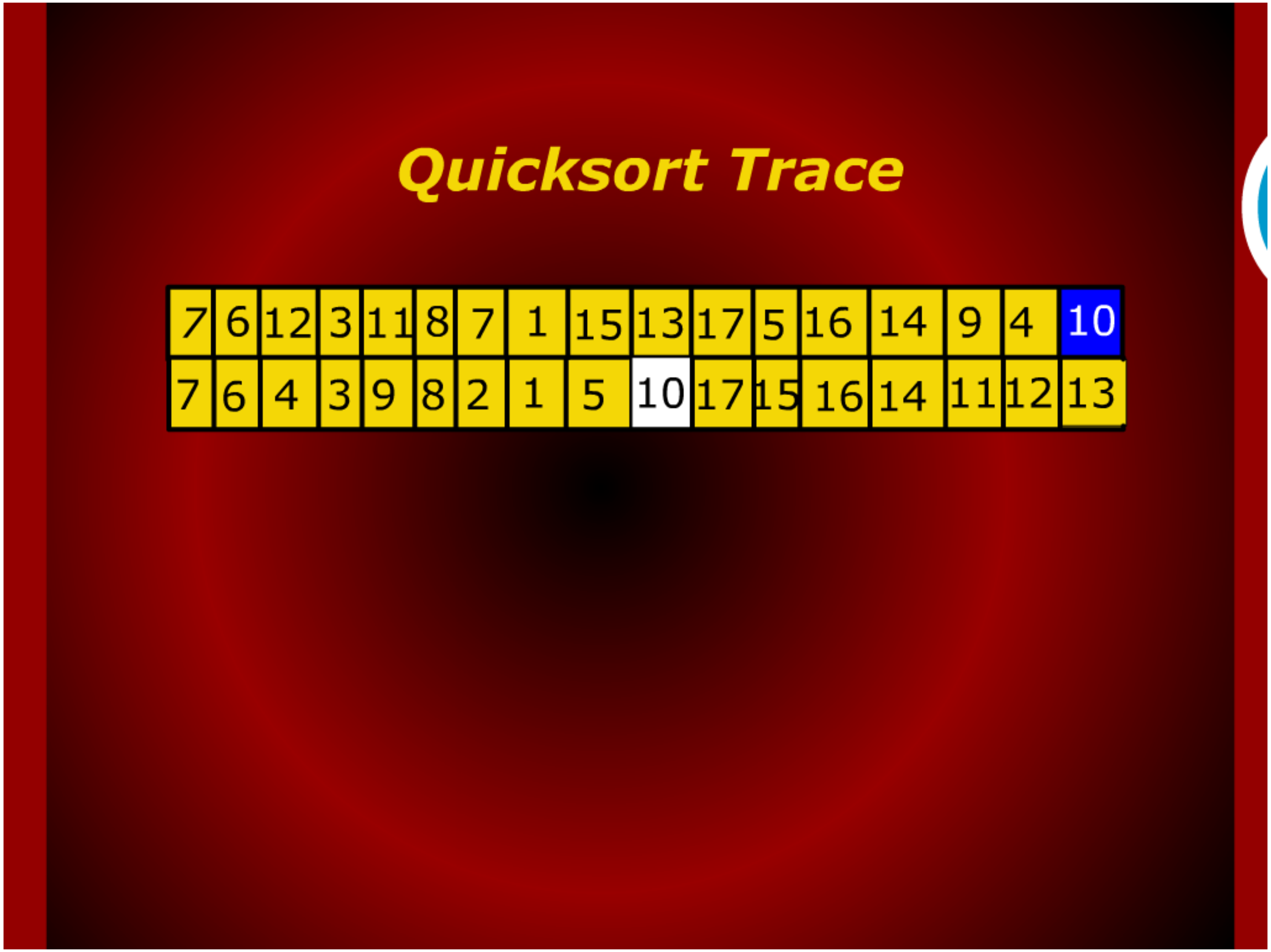
QUICK SORT


Partition Algorithm
p r
5 3 8 6 4 7 3 1
q s
p r
5 3 8 6 4 7 3 1
q s
p r
5 3 8 6 4 7 3 1
q s
p r
5 3 8 6 4 7 3 1
q s
p r
5 3 4 6 8 7 3 1
q s
p r
5 3 4 6 8 7 3 1
q s
p r
5 3 4 3 8 7 6 1
q s
p r
5 3 4 3 1 7 6 8
q s
p r
1 3 4 3 5 7 6 8
q s

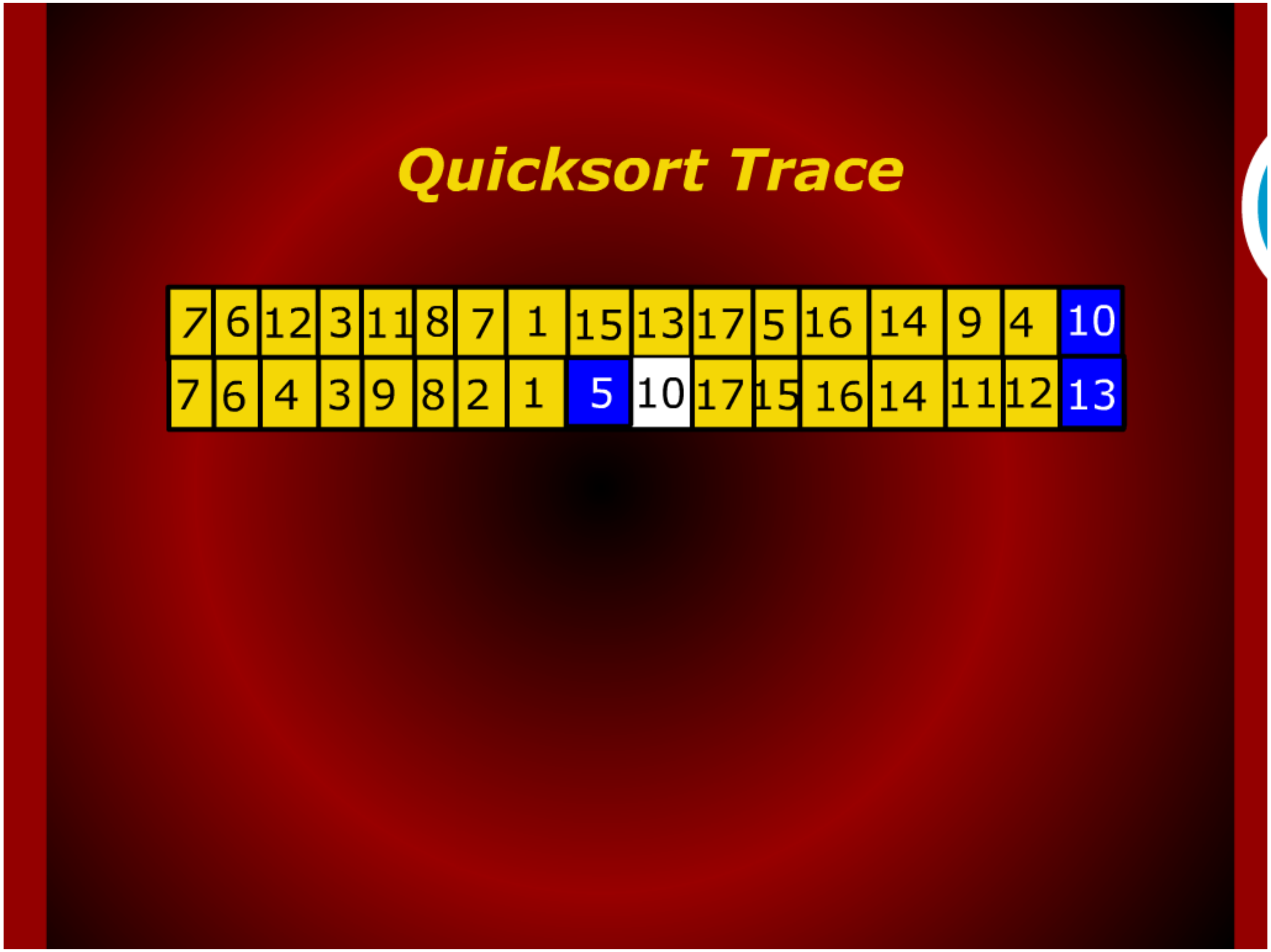
QUICK SORT



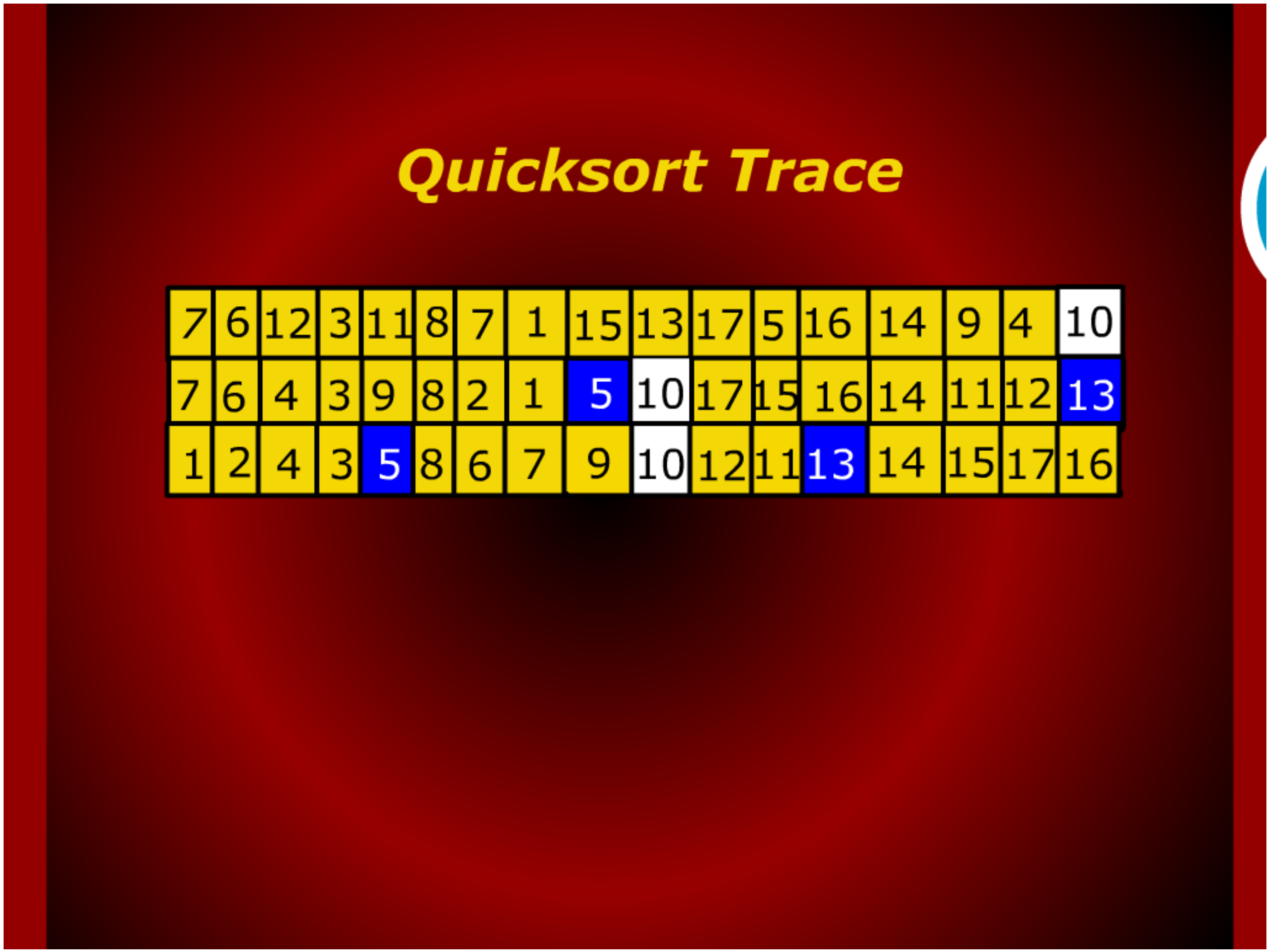
QUICK SORT


Quicksort Trace
7 6 12 3 11 8 7 1 15 13 17 5 16 14 9 4 10
7 6 4 3 9 8 2 1 5 10 17 15 16 14 11 12 13

QUICK SORT


Quicksort Trace
7 6 12 3 11 8 7 1 15 13 17 5 16 14 9 4 10
7 6 4 3 9 8 2 1 5 10 17 15 16 14 11 12 13

QUICK SORT



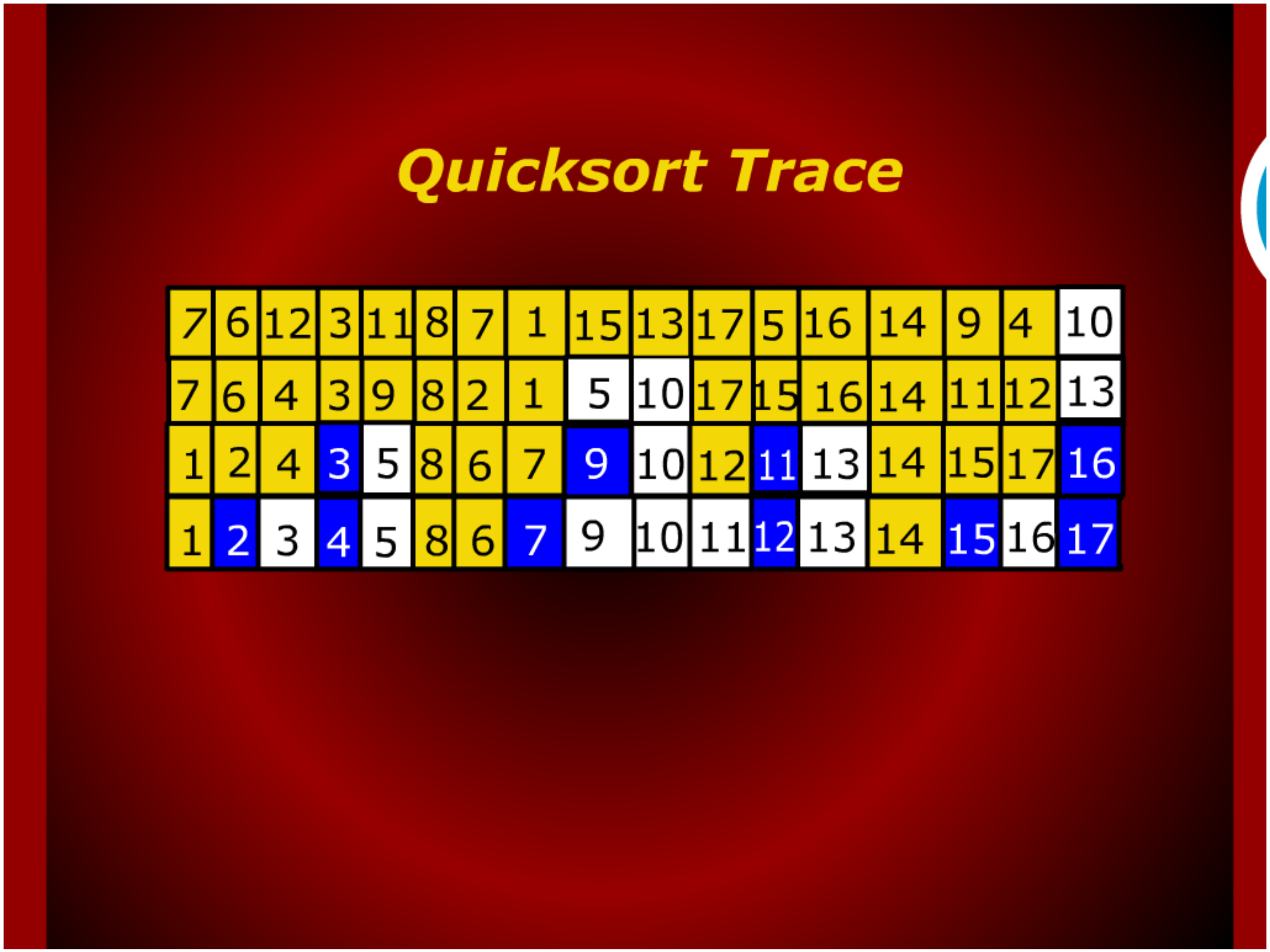
Quicksort Trace
7 6 12 3 11 8 7 1 15 13 17 5 16 14 9 4 10
7 6 4 3 9 8 2 1 5 10 17 15 16 14 11 12 13
1 2 4 3 5 8 6 7 9 10 12 11 13 14 15 17 16

QUICK SORT

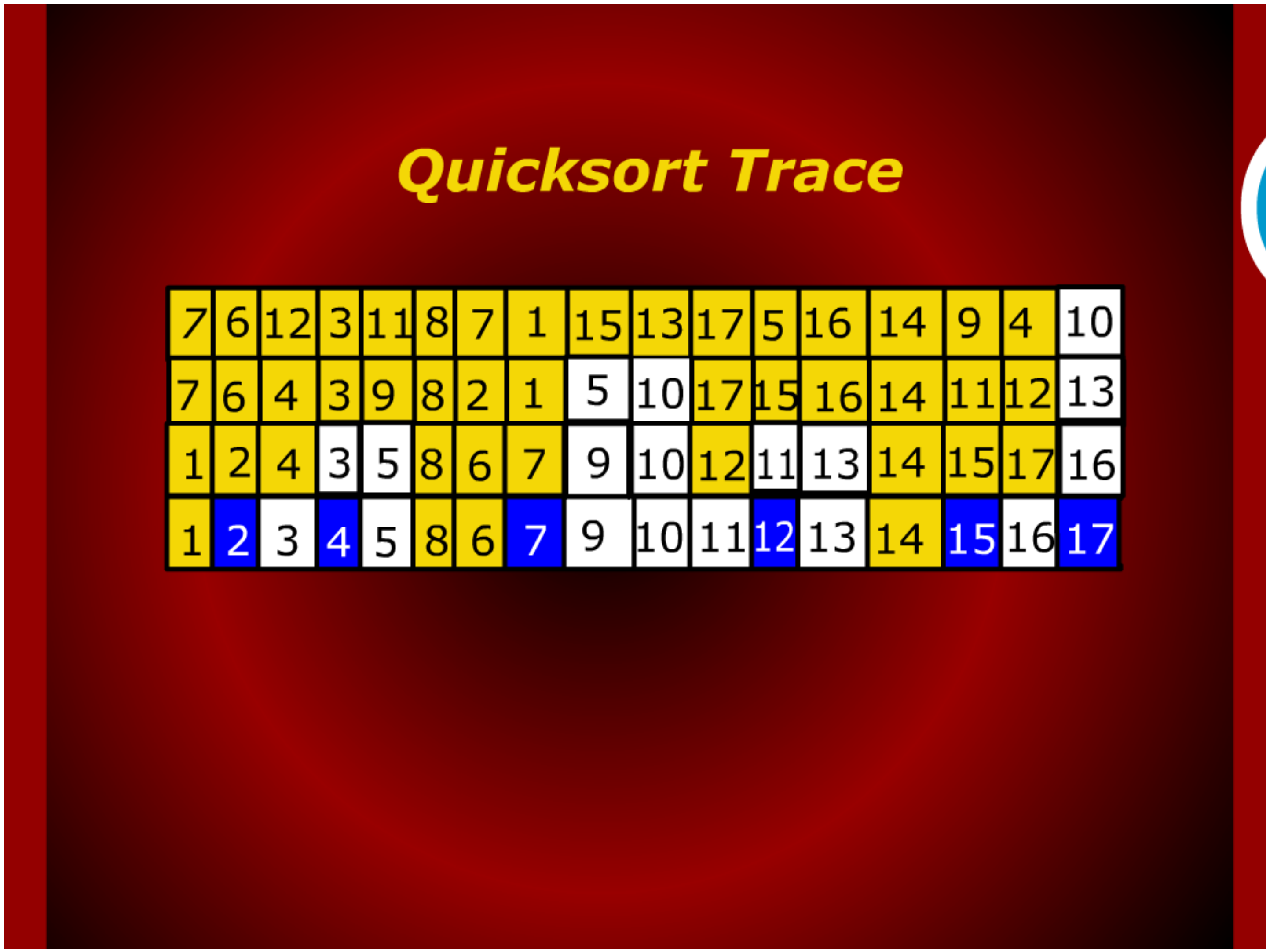
Quicksort Trace

7	6	12	3	11	8	7	1	15	13	17	5	16	14	9	4	10
7	6	4	3	9	8	2	1	5	10	17	15	16	14	11	12	13
1	2	4	3	5	8	6	7	9	10	12	11	13	14	15	17	16

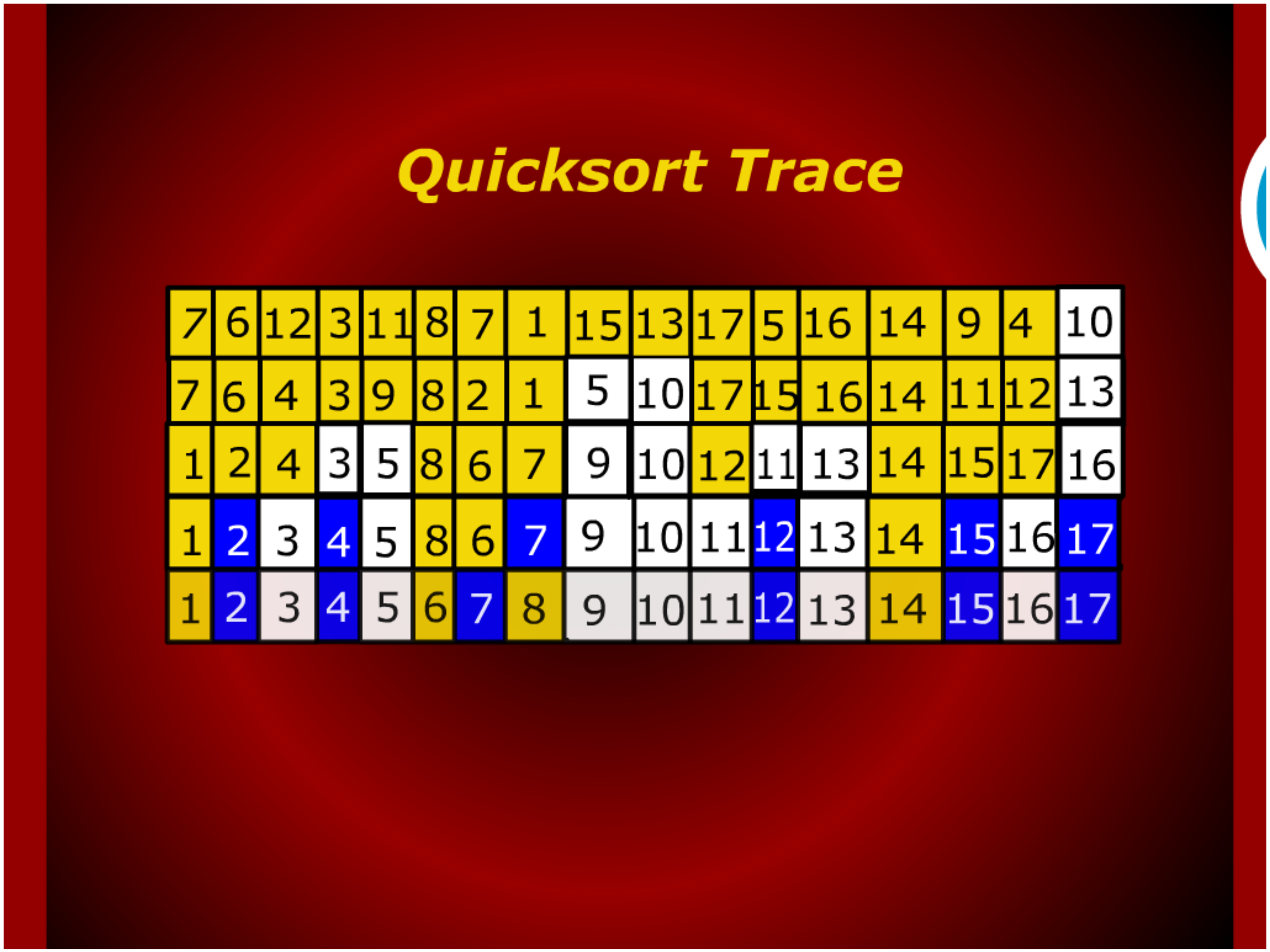
QUICK SORT


Quicksort Trace
7 6 12 3 11 8 7 1 15 13 17 5 16 14 9 4 10
7 6 4 3 9 8 2 1 5 10 17 15 16 14 11 12 13
1 2 4 3 5 8 6 7 9 10 12 11 13 14 15 17 16
1 2 3 4 5 8 6 7 9 10 11 12 13 14 15 16 17

QUICK SORT

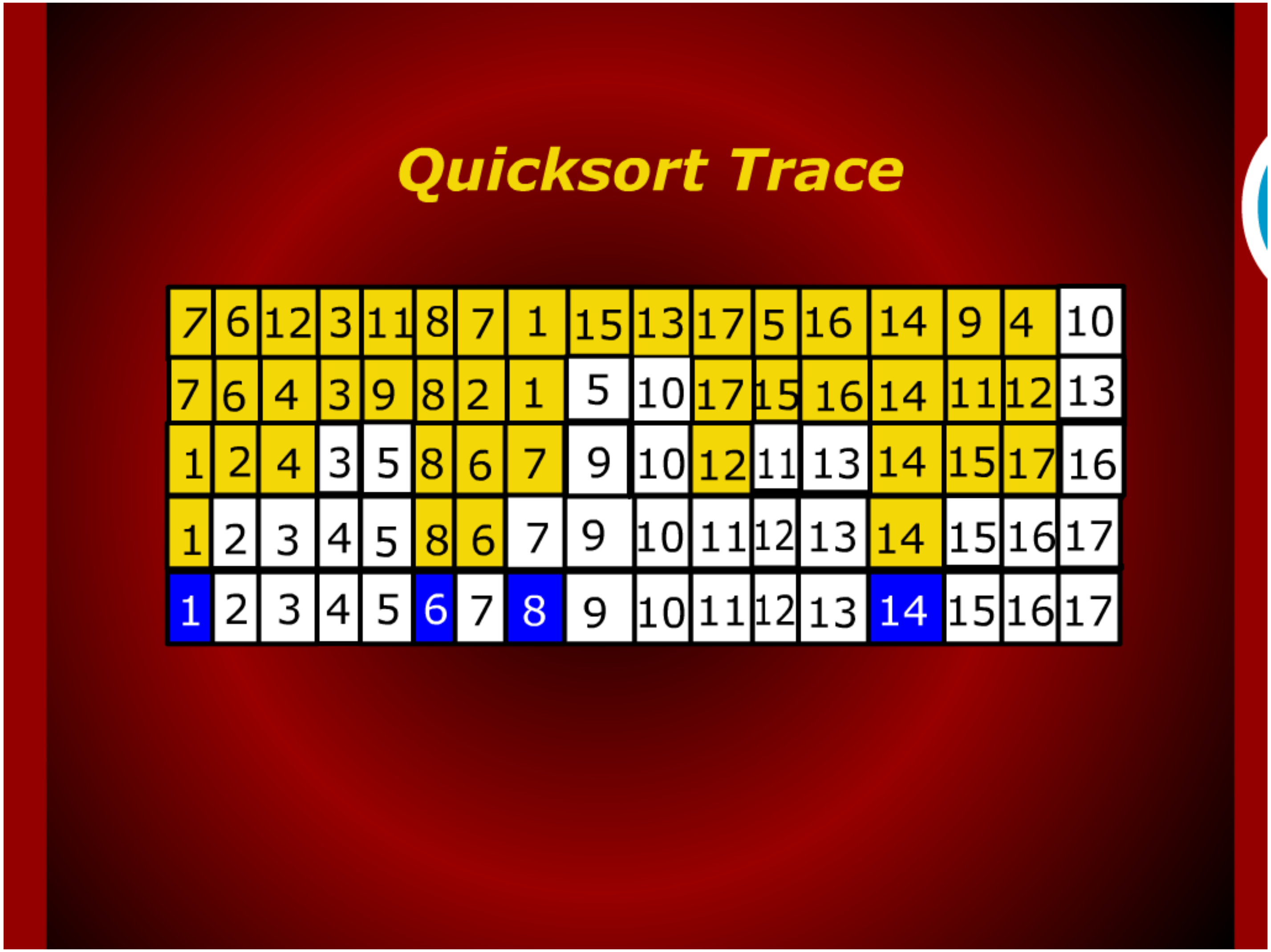

Quicksort Trace
7 6 12 3 11 8 7 1 15 13 17 5 16 14 9 4 10
7 6 4 3 9 8 2 1 5 10 17 15 16 14 11 12 13
1 2 4 3 5 8 6 7 9 10 12 11 13 14 15 17 16
1 2 3 4 5 8 6 7 9 10 11 12 13 14 15 16 17

QUICK SORT

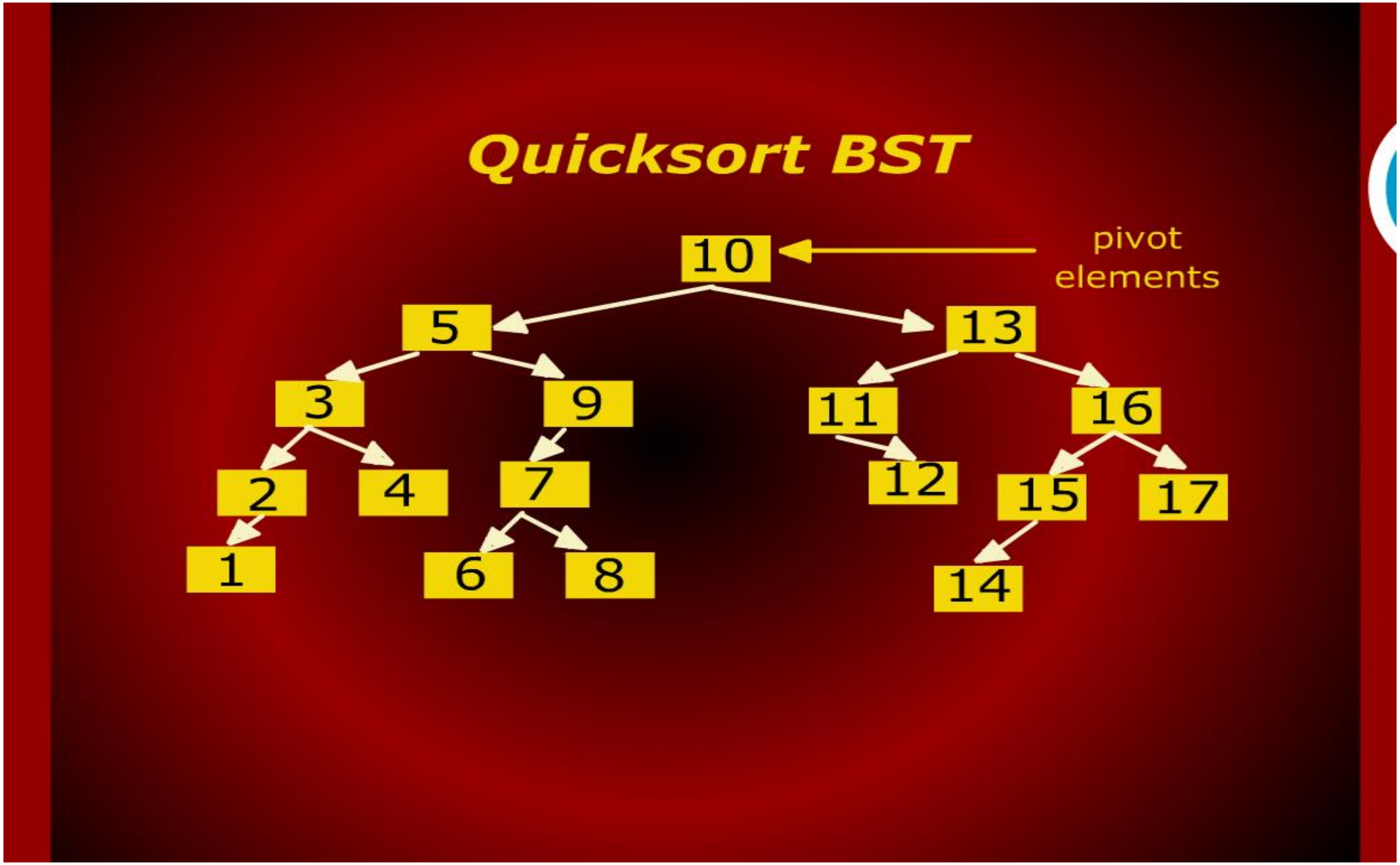


Quicksort Trace
7 6 12 3 11 8 7 1 15 13 17 5 16 14 9 4 10
7 6 4 3 9 8 2 1 5 10 17 15 16 14 11 12 13
1 2 4 3 5 8 6 7 9 10 12 11 13 14 15 17 16
1 2 3 4 5 8 6 7 9 10 11 12 13 14 15 16 17
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17

QUICK SORT

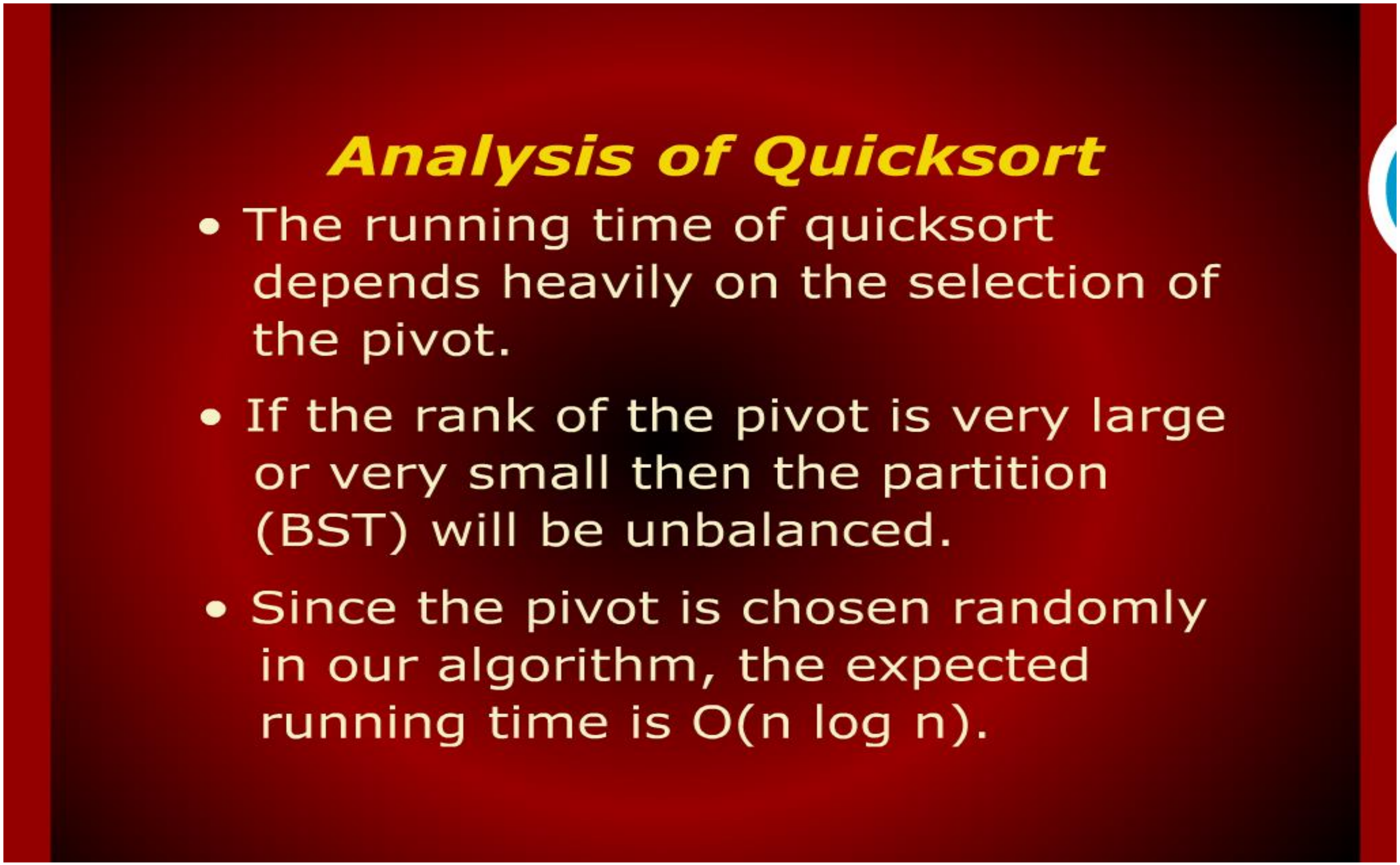

Quicksort Trace
7 6 12 3 11 8 7 1 15 13 17 5 16 14 9 4 10
7 6 4 3 9 8 2 1 5 10 17 15 16 14 11 12 13
1 2 4 3 5 8 6 7 9 10 12 11 13 14 15 17 16
1 2 3 4 5 8 6 7 9 10 11 12 13 14 15 16 17
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17

QUICK SORT



Quicksort BST
10
pivot
elements
5
13
3
9
11
16
2
4
7
12
15
17
1
6
8
14

QUICK SORT


Analysis of Quicksort
• The running time of quicksort depends heavily on the selection of the pivot.
• If the rank of the pivot is very large or very small then the partition (BST) will be unbalanced.
• Since the pivot is chosen randomly in our algorithm, the expected running time is O(n log n).

QUICK SORT

Analysis of Quicksort

- If the rank of the pivot is very large or very small then the partition (BST) will be unbalanced.
- Since the pivot is chosen randomly in our algorithm, the expected running time is $O(n \log n)$.
- The worst case time, however, is $O(n^2)$. Luckily, this happens rarely.

QUICK SORT ALGORITHM (Method2)

- QUICKSORT(A,lb,ub)
- If (lb<ub)
- {
- loc=partition(A,lb,ub)
- QUICKSORT(A,lb,loc-1)
- QUICKSORT(A,loc+1,ub)
- }

QUICK SORT ALGORITHM (Method2)

- PARTITION(A,lb,ub){
- Pivot=A[lb]; start=lb; end=ub;
- While(start<end)
- {
- While(A[start]<=pivot) start++;
- While(A[end]>pivot) end--;
- If(start<end) swap(A[start],A[end]);
- }
- Swap(A[lb],A[end])
- Return end';
- }

SHELL SORT

Introduction

Shell sort is one of the oldest sorting algorithm devised by Donald Shell in 1959.

Shell sort is also called **the diminishing increment sort.**

The shell sort was introduced to improve the efficiency of simple insertion sort.

In insertion sort, we move elements only one position ahead. When an element has to be moved far ahead, many movements are involved.

The shell sort improved the insertion sort by comparing elements far apart, then elements less far apart and finally comparing adjacent elements.

The shell sort algorithm avoids large shifts as in the case of insertion sort, if the smaller values is to the far right and has to be moved to the far left.

The shell sort improved insertion sort by breaking the original list into a number of smaller sublist, each of which is sorted using an insertion sort.

The unique way these sublists are chosen is through the key to the shell sort.

Instead of breaking the list into sublists of contiguous items, the shell sort uses an incremental i, sometimes called the gap or interval to create a sublist by choosing all items that i items apart.

The idea of shell Sort is to allow exchange of far items

key terms

Gap or interval: the spacing between elements or the distance between items being sorted. As we progress, the gap decreases and that is why Shell Sort is also called Diminishing Gap Sort.

Swap: exchanging one thing for another. (interchange, exchange, switch). When an element is swapped, the move the position of the swapped item.

Sort: the arrangement of data in a prescribed sequence.

Integer (INT): a whole number; a number that is not a fraction.

Sub-list: a list gotten from a larger list. For instance, in sets a sub-set is gotten from the universal set.

Shuttle sort: A sorting method based on swaps of pairs of numbers. Swapping may be done or no swapping may be required. The sort works from left to right, reconsidering earlier pairs when a swap is made.

How Shell Sort Works steps

1. Count number of elements in a given array;
2. Divide the list into $\frac{n}{2}$, $\frac{n}{4}$, … sub-lists. (*ignore the remainder if n is odd*)
3. Sort each sub-list using shuttle sort;
4. Merge the sub-lists;

Repeat step 2 and 3 until the number of sub-list is 1.

There are many ways of choosing the next gap.

In any item, objects are compared with less than (<), equal (=) or greater then (>) in cases where strings may be applied.

1. Count the number of elements in a given array. E.g an array containing

35, 33,42,10,14,19,27,44

This array has how many elements?

Ans: 8

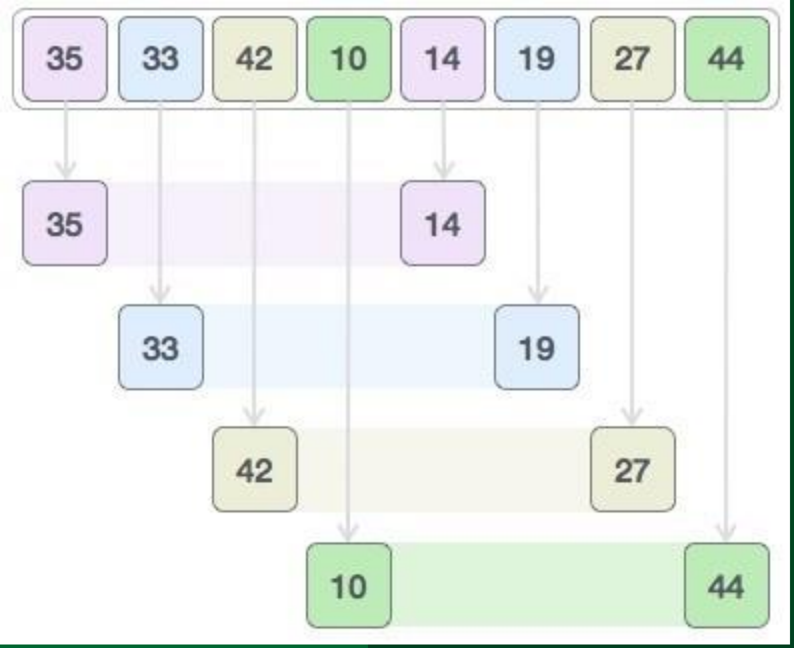
- Divide the list into $\text{INT}\left(\frac{N}{2}\right)$ sublists for the first gap then $\text{INT}\left(\frac{n}{4}\right)$ for the second gap or the value of the first gap divide by 2. (*ignore the remainder if n is odd*).
- 35, 33,42,10,14,19,27,44 we have 8 elements. Then $\text{INT}\left(\frac{8}{2}\right) = 4$. This means that the gap or interval is 4.
-

We make a virtual sub-list of all values located at the interval of 4 position.

The values for the 4 sub-list for the first gap.

{35, 14}, {33, 19}, {42, 27} and {10, 44}

How comes



{35, 14}if 35>14 then we swap in there position otherwise is left.

Next {33, 19} if 33>19 then swap to their position.

Next {42, 27} if 42>27 then swap.

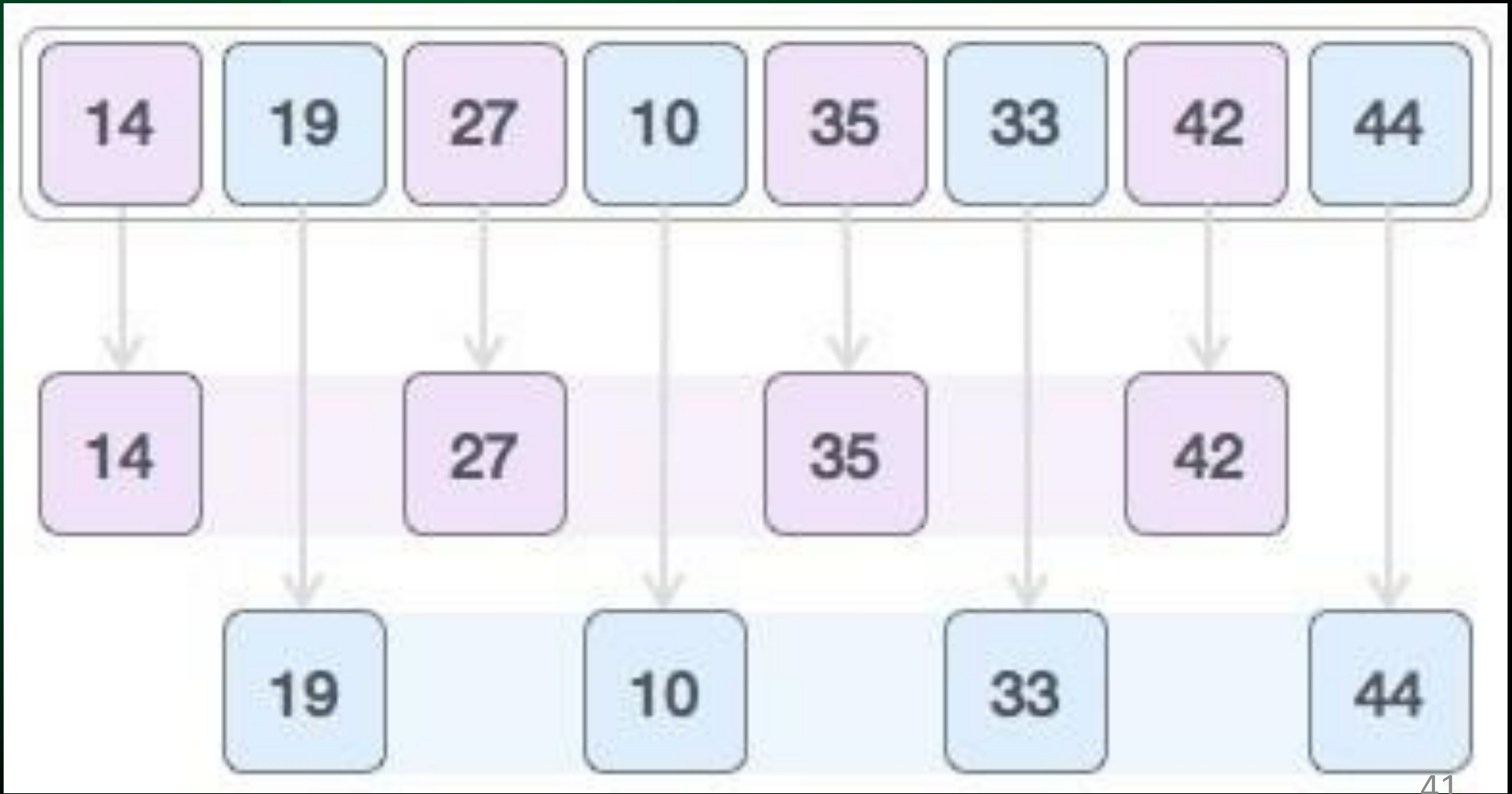
Next {10, 44} if 10> 44 swap otherwise remain in the same position.

We compare values in each sub-list and swap them (if necessary) in the original array.

The new array after pass 1:

Pass 2: $INT\left(\frac{4}{2}\right) = 2$ OR $INT\left(\frac{8}{4}\right) = 2$. Now the interval will be 2 and this gap will generate two sub-list

{ 14, 27, 35, 42}, {19, 10, 33, 44}



Algorithm

```
SHELLSORT(A,n)
FOR(gap=n/2;gap>=1;gap/2)
{
For(j=gap;j<n;j++)
{
For(i=j-gap:i>=0;i-gap)
{
If(A[i+gap]>A[i])          break;
Else
swap(A[i+gap],A[i])
    }}}
```

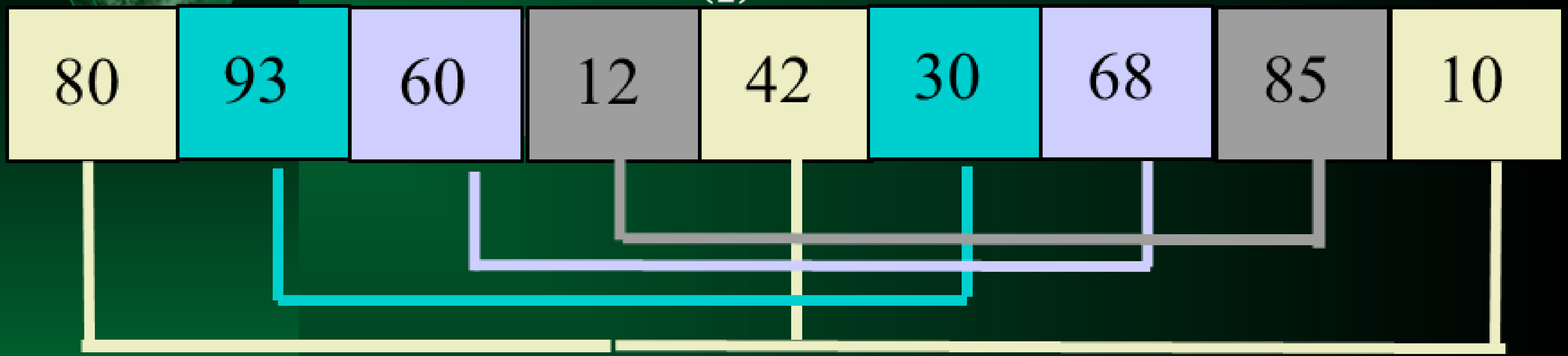
Example

Consider the following unsorted array A with elements

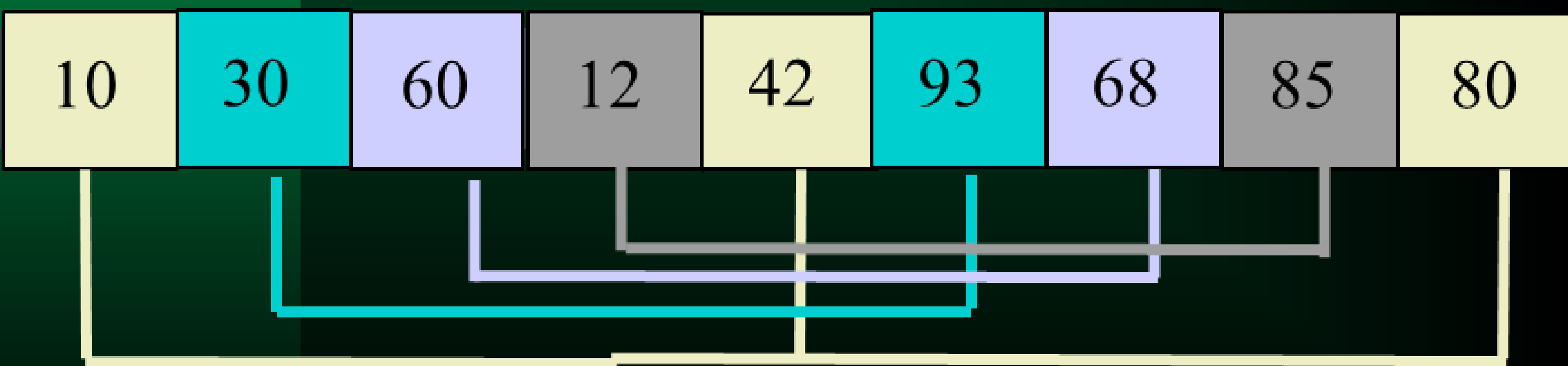
80, 93, 60, 12, 42, 30, 68, 85, 10

Sort using the Shell sort.

Initial Segmenting gap = $\text{INT}\left(\frac{9}{2}\right) = 4$



No of comparison: 6
No of swaps: 4



Re segmenting Gap

$$\mathrm{INT}\left(\frac{4}{2}\right) = 2$$

10	30	60	12	42	93	68	85	80

10	12	42	30	60	85	68	93	80

Re-segmenting Gap = INT $\left(\frac{2}{2}\right)$ = 1

10	12	42	30	60	85	68	93	80

10	12	30	42	60	68	80	85	93

Best case

The best case is when the array is already sorted in the right order.

The number of comparisons is almost zero and swapping.

Average case

Less comparison and swamping

Worst case

More comparison and swapping

Advantages

1. Shell sort is easy to implement but not easier than insertion sort.
2. Shell sort can be sorted for a gap greater than one and thus less exchange than insertion sort hence, fast.
3. Efficient for medium size lists.
4. It is easy to understand.
5. It is easy to implement.

Disadvantages

1. It is a complex algorithm and its not nearly as efficient as the merge, heap and quick sorts.
2.

Algorithm

The difference between insertion sort from the shell sort are

- There is one additional loop (the outer loop)- each run processes one increment.
- The middle and the most inner loops are same as in the insertion sort with gap = 1.

Let A be a linear array of n elements, A [1], A [2], A [3], ... A[n] and Incr be an array of sequence of span to be incremented in each pass. X is the number of elements in the array Incr. Span is to store the span of the array from the array Incr.

1. Input n numbers of an array A
2. Initialise i = 0 and repeat through step 6 if (i < x)
3. Span = Incr[i]
4. Initialise j = span and repeat through step 6 if (j < n)
   (a) Temp = A[j]
5. Initialise k = j-span and repeat through step 5 if (k > = 0) and (temp < A [k])
   (a) A [k + span] = A [k]
6. A [k + span] = temp
7. Exit

Shell Sort Algorithm

1. Set **gap** to n/2
2. while **gap** > 0
3. for each element from **gap** to end, by **gap**
4. Insert element in its **gap**-separated sub-array
5. if **gap** is 2, set it to 1
6. otherwise set it to **gap** / 2.2

Shell Sort Algorithm: Inner Loop

set **nextPos** to position of element to insert

2. set **nextVal** to value of that element
3. while **nextPos** > **gap** and

   element at **nextPos-gap** is > **nextVal**

3.4 Shift element at **nextPos-gap** to **nextPos**

3.5 Decrement **nextPos** by **gap**

3.6 Insert **nextVal** at **nextPos**

```
# Sort an array a[0...n-1].
gaps = [701, 301, 132, 57, 23, 10, 4, 1]

# Start with the largest gap and work down to a gap of 1
For each (gap in gaps)
{
# Do a gapped insertion sort for this gap size.
# The first gap elements a[0..gap-1] are already in gapped order
# keep adding one more element until the entire array is gap sorted
for (i = gap; i < n; i += 1)
{
# add a[i] to the elements that have been gap sorted
# save a[i] in temp and make a hole at position i
temp = a[i]
# shift earlier gap-sorted elements up until the correct location for a[i] is found
for (j = i; j >= gap and a[j - gap] > temp; j -= gap)
{
a[j] = a[j - gap]
}
# put temp (the original a[i]) in its correct location
a[j] = temp
}
}
```

```
int j, p, gap;
comparable tmp;
for (gap = N/2; gap > 0; gap = gap/2)

for ( p = gap; p < N ; p++)
{
tmp = a[p];
for (j = p; j >= gap && tmp < a[j- gap]; j = j - gap)

a[j] = a[j-gap];

a[j] = tmp;
}

}
```