

Design and Analysis of Algorithms

- **Course Code:** BCSE204L
- **Course Type:** Theory (ETH)
- **Slot:** A1+TA1 & & A2+TA2
- **Class ID:** VL2023240500901
VL2023240500902

A1+TA1

Day	Start	End
Monday	08:00	08:50
Wednesday	09:00	09:50
Friday	10:00	10:50

A2+TA2


Day	Start	End
Monday	14:00	14:50
Wednesday	15:00	15:50
Friday	16:00	16:50

Syllabus- Module 6

Module:6 | **Randomized algorithms**

5 hours

Randomized quick sort - The hiring problem - Finding the global Minimum Cut



Algorithm	Best Case	Average Case	Worst Case
Quicksort (Deterministic)	$O(n \log n)$	$O(n \log n)$	$O(n^2)$
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$

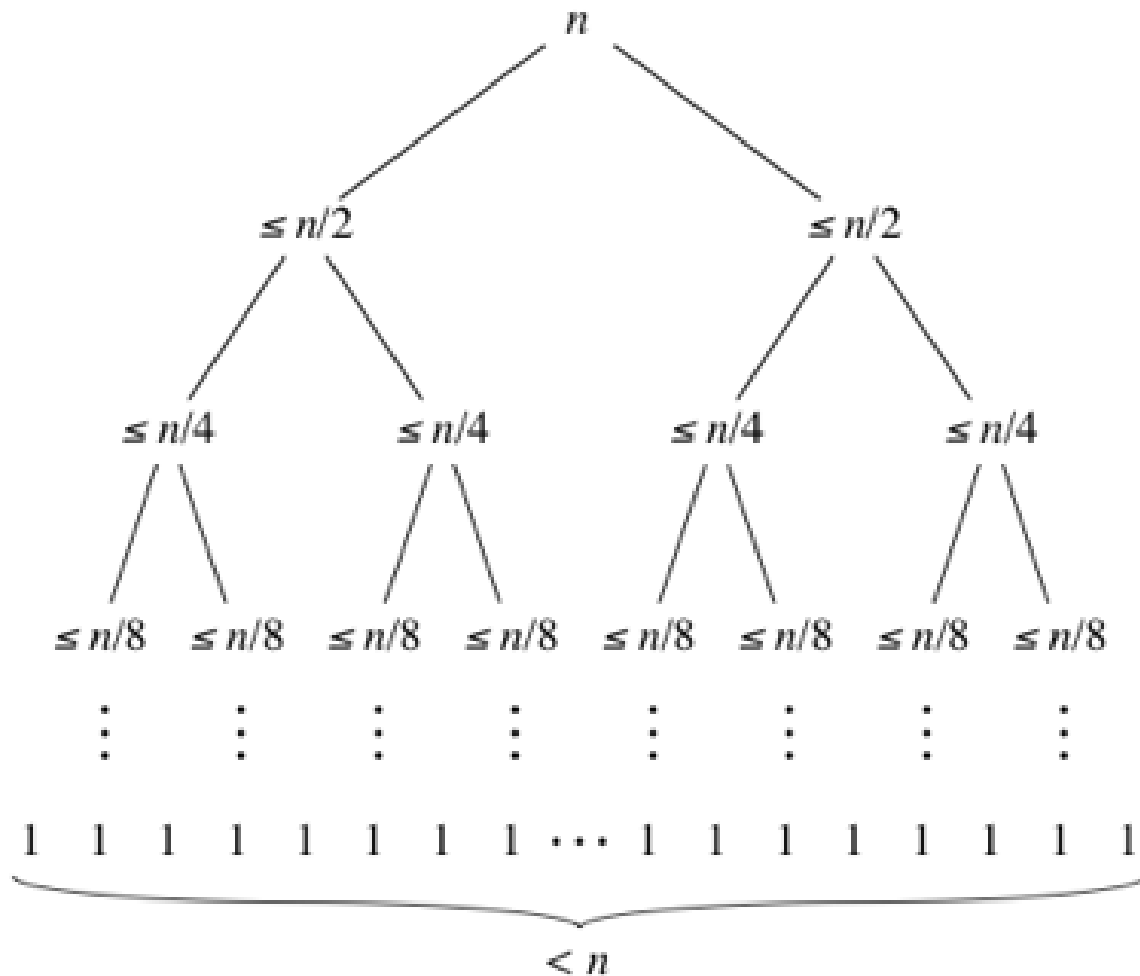
Quicksort - Balanced Partitions (Best Case)

- Occurs when the pivot element splits the subarray into nearly equal halves.
 - This leads to a balanced recursion tree similar to Merge Sort.
-
- The former case occurs if the subarray has an odd number of elements, and after partitioning, the pivot is positioned right in the middle, resulting in two partitions with $(n-1)/2$ elements each.
 - The latter case occurs if the subarray has an even number of elements (n), and after partitioning, one partition has $n/2$ elements while the other has $n/2-1$ elements.
 - In either of these cases, each partition has at most $n/2$ elements. Consequently, the structure of the subproblem sizes resembles that of Merge Sort, with the partitioning times mirroring the merging times.

Subproblem
size

Total partitioning time
for all subproblems of
this size

Best case:
Using big- Θ notation $\Theta(n \log n)$



$$cn$$

$$\leq 2 \cdot cn/2 = cn$$

$$\leq 4 \cdot cn/4 = cn$$

$$\leq 8 \cdot cn/8 = cn$$

\vdots

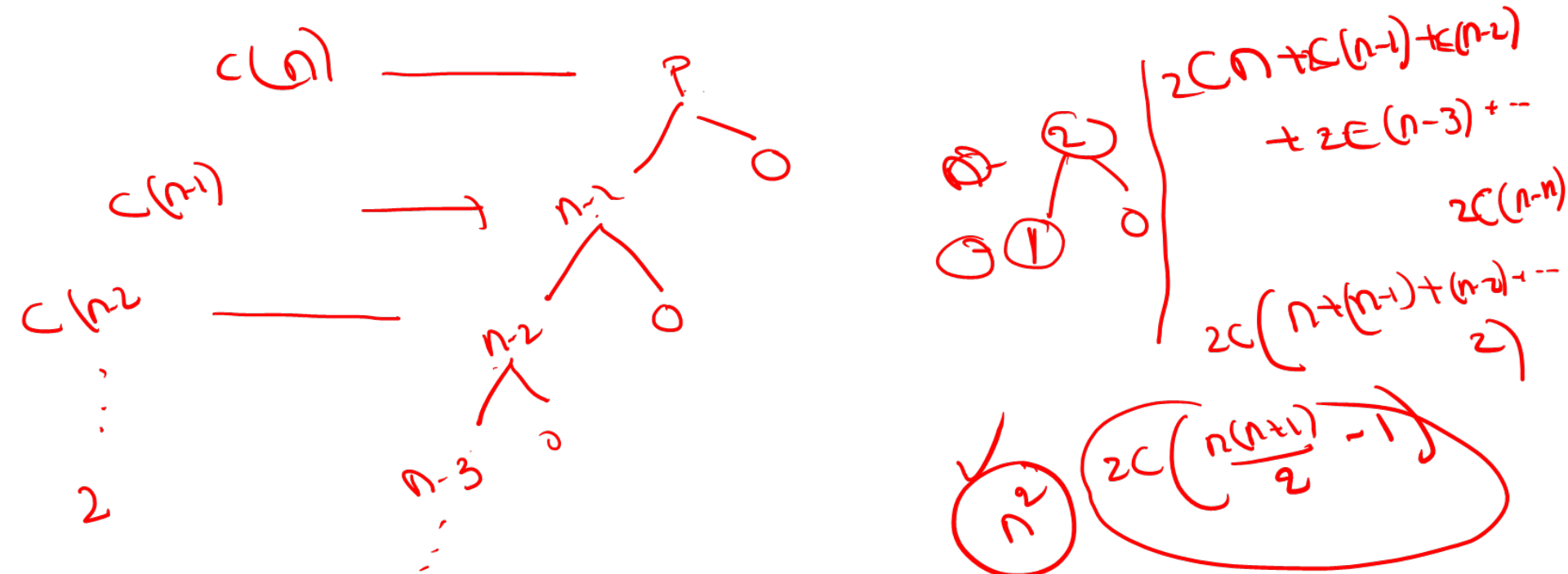
$$< n \cdot c = cn$$

Quicksort - Worst Case

- Occurs when the chosen pivot is always the smallest or largest element.
- This creates highly unbalanced partitions, with one subarray having no elements and the other having $n-1$ elements.
- Recursive calls become inefficient, leading to $O(n^2)$ time complexity.

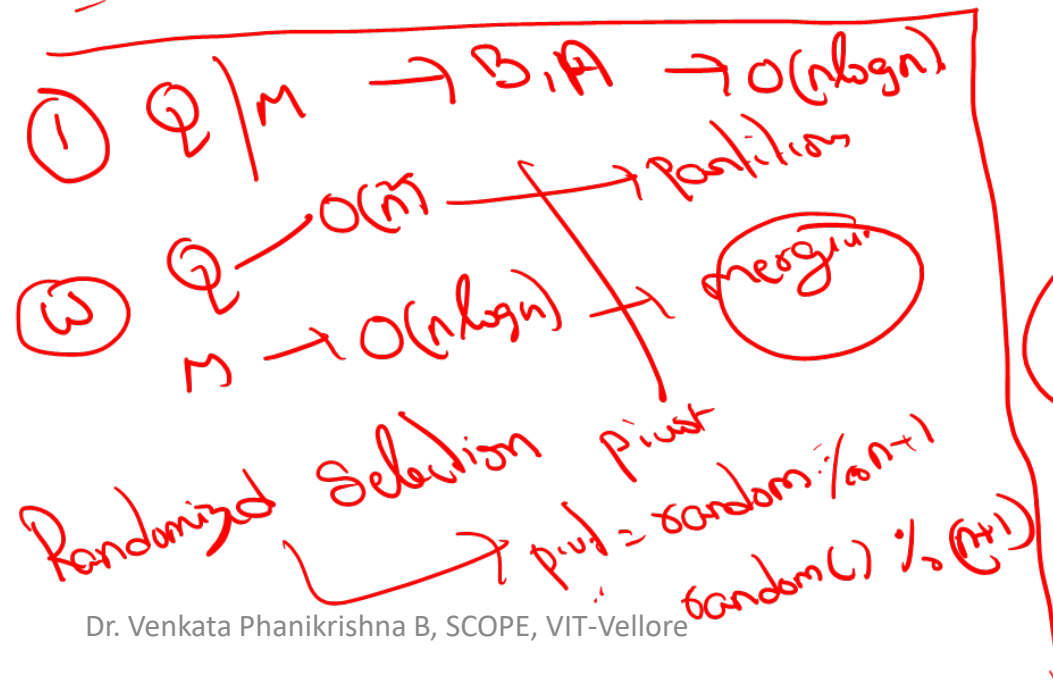
Quicksort's worst-case

- The pivot chosen by the partition function is always either the smallest or the largest element in the n -element subarray.
- Then one of the partitions will contain no elements and the other partition will contain $n-1$ elements—all but the pivot.
- So the recursive calls will be on subarrays of sizes 0 and $n-1$.



Why merge sort is better in such case than Quick sort.

- As in merge sort, the **time for a given recursive call** on an **n-element subarray** is $\Theta(n)$.
- In merge sort, that was the **time for merging**, but in quicksort it's the time **for partitioning**.
- Merge Sort's merging step takes $\Theta(n)$ time regardless of the input.
- In Quicksort's worst case, the partitioning step becomes the bottleneck due to unbalanced subarrays.



Solution: Randomized Quicksort

- Randomized Quicksort aims to address the **worst-case scenario of Quicksort** by introducing **randomness** in the selection of the pivot element.
- Instead of always selecting the first (min) or last element (max) as the pivot, Randomized Quicksort chooses the **pivot randomly** from the array.
- By doing so, the probability of encountering the worst-case scenario is significantly reduced, as the randomly chosen pivot is less likely to consistently lead to unbalanced partitions.
- On average, the randomly chosen pivot will result in more evenly balanced partitions, leading to a much faster sorting process.
- As a result, Randomized Quicksort typically achieves an average-case time complexity of $O(n \log n)$, similar to standard Quicksort, but with a significantly lower probability of degrading to $O(n^2)$ time complexity.

Quicksort

```
def partition(arr, low, high):
    comparisons = 0
    swaps = 0
    pivot = arr[high]
    i = low - 1
    for j in range(low, high):
        comparisons += 1
        if arr[j] < pivot:
            i += 1
            arr[i], arr[j] = arr[j], arr[i]
            swaps += 1
    arr[i + 1], arr[high] = arr[high], arr[i + 1]
    swaps += 1
    return i + 1, comparisons, swaps
```

```
def quick_sort(arr, low, high):
    comparisons = 0
    swaps = 0
    if low < high:
        pi, comp1, swap1 = partition(arr, low, high)
        comparisons += comp1
        swaps += swap1
        comp2, swap2 = quick_sort(arr, low, pi - 1)
        comparisons += comp2
        swaps += swap2
        comp3, swap3 = quick_sort(arr, pi + 1, high)
        comparisons += comp3
        swaps += swap3
    return comparisons, swaps
```

Randomized Quicksort

```
def randomized_partition(arr, low, high):
    comparisons = 0
    swaps = 0
    pivot_index = random.randint(low, high)
    arr[pivot_index], arr[high] = arr[high], arr[pivot_index]
    return partition(arr, low, high)
```

```
def randomized_quick_sort(arr, low, high):
    comparisons = 0
    swaps = 0
    if low < high:
        pi, comp1, swap1 = randomized_partition(arr, low, high)
        comparisons += comp1
        swaps += swap1
        comp2, swap2 = randomized_quick_sort(arr, low, pi - 1)
        comparisons += comp2
        swaps += swap2
        comp3, swap3 = randomized_quick_sort(arr, pi + 1, high)
        comparisons += comp3
        swaps += swap3
    return comparisons, swaps
```

```
def quicksort(A, low, high):  
    if low < high:  
        parti = lomutoPartition(A, low, high)  
        quicksort(A, low, parti-1)  
        quicksort(A, parti+1, high)  
  
def lomutoPartition(A, low, high):  
    pivot = A[high]  
    i = low-1  
    for j in range(low, high):  
        if pivot >= A[j]:  
            i += 1  
            A[i], A[j] = A[j], A[i]  
    A[i+1], A[high] = A[high], A[i+1]  
    return i+1  
  
if __name__ == "__main__":  
    A = [4, 2, 7, 3, 1, 9, 6, 0, 8]  
    low, high = 0, len(A)-1  
    print("BEFORE SORTING:", A)  
    quicksort(A, low, high)  
    print("AFTER SORTING:", A)
```

Hoare Partition Scheme (Optinal)

	Hoare Partition	Lomuto Partition
1	Makes use of two pointers for partitioning	Also makes use of two pointers for partitioning
2	The first element of the array is usually chosen as the pivot element, although there is no restriction	The last element of the array is usually chosen as the pivot element although it can be random as well
3	It is a linear algorithm	It is also a linear algorithm
4	It is more comparatively more efficient and faster because of fewer swap operations on average	It is more comparatively less efficient and slower because of more swap operations on average
5	It causes Quicksort to downgrade to $O(n^2)$ if the array is already almost or completely sorted	It also causes Quicksort to downgrade to $O(n^2)$ if the array is already almost or completely sorted
6	It is slightly complex to understand and implement	It is comparatively easier to understand and implement

Lomuto Partition Scheme

```
partition(arr[], lo, hi)
    pivot = arr[hi]
    i = lo    // place for swapping
    for j := lo to hi - 1 do
        if arr[j] <= pivot then
            swap arr[i] with arr[j]
            i = i + 1
    swap arr[i] with arr[hi]
    return i
```

```
partition_r(arr[], lo, hi)
    r = Random Number from lo to hi
    Swap arr[r] and arr[hi]
    return partition(arr, lo, hi)
```

```
quicksort(arr[], lo, hi)
    if lo < hi
        p = partition_r(arr, lo, hi)
        quicksort(arr, lo , p-1)
        quicksort(arr, p+1, hi)
```

Lomuto Partition Scheme

```
partition(arr[], lo, hi)
    pivot = arr[hi]
    i = lo    // place for swapping
    for j := lo to hi - 1 do
        if arr[j] <= pivot then
            swap arr[i] with arr[j]
            i = i + 1
    swap arr[i] with arr[hi]
    return i
```

```
partition_r(arr[], lo, hi)
    r = Random Number from lo to hi
    Swap arr[r] and arr[hi]
    return partition(arr, lo, hi)
```

```
quicksort(arr[], lo, hi)
    if lo < hi
        p = partition_r(arr, lo, hi)
        quicksort(arr, lo, p-1)
        quicksort(arr, p+1, hi)
```

Hoare Partitioning

```
partition(arr[], lo, hi)
    pivot = arr[lo]
    i = lo - 1 // Initialize left index
    j = hi + 1 // Initialize right index

    while(True)
        // Find a value in left side greater than pivot
        do
            i = i + 1
            while arr[i] < pivot
        // Find a value in right side smaller than pivot
        do
            j = j - 1
            while arr[j] > pivot

        if i >= j then
            return j
        else
            swap arr[i] with arr[j]
    end while
```

```
partition_r(arr[], lo, hi)
    r = Random number from lo to hi
    Swap arr[r] and arr[lo]
    return partition(arr, lo, hi)
```

```
quicksort(arr[], lo, hi)
    if lo < hi
        p = partition_r(arr, lo, hi)
        quicksort(arr, lo, p)
        quicksort(arr, p+1, hi)
```

Any

Question



PresenterMedia

Thank You!

**FOR YOUR
ATTENTION**

