Quicksort

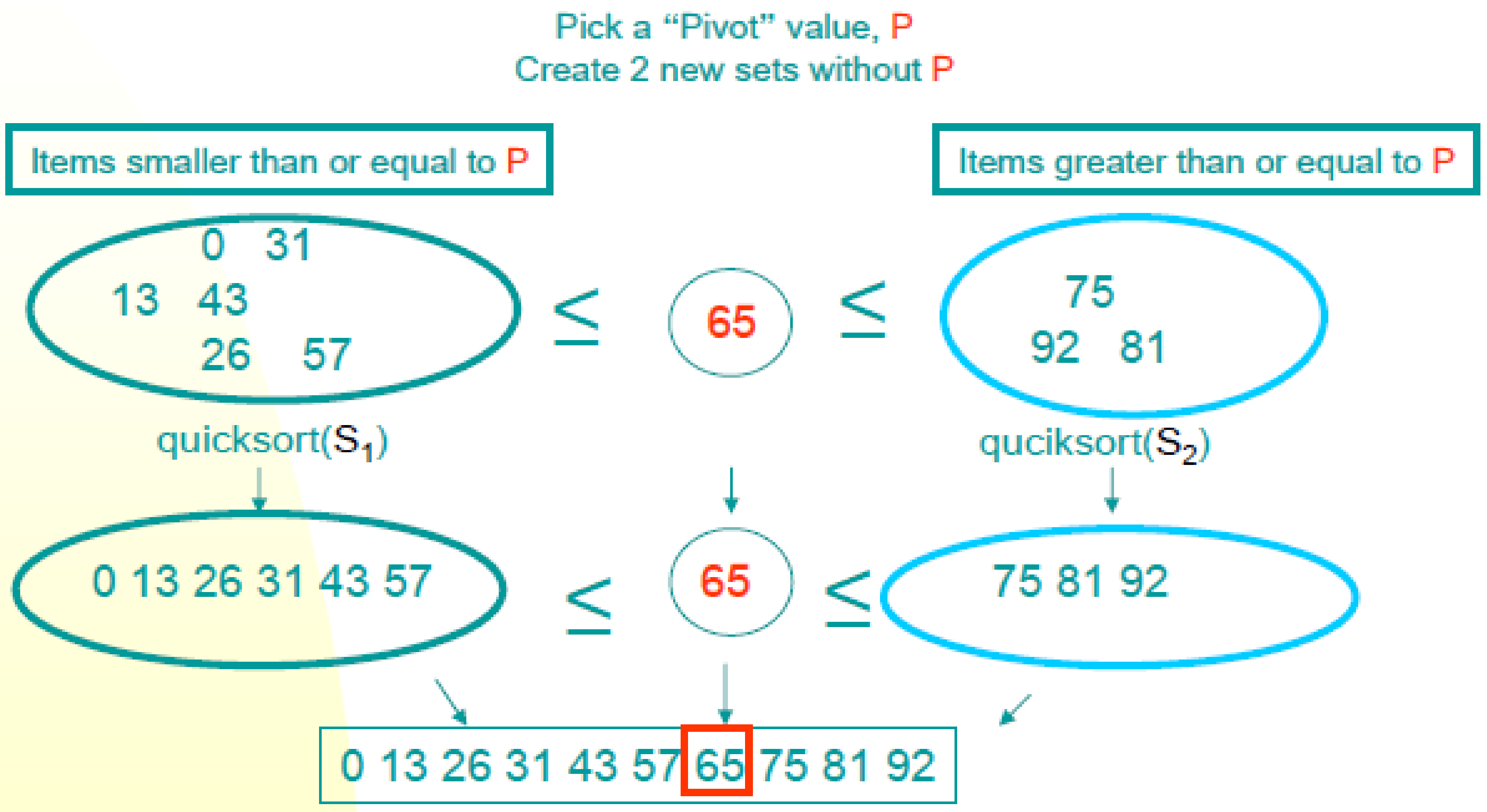
Dr. Bibhudatta Sahoo

Communication & Computing Group

Department of CSE, NIT Rourkela

Email: bdsahu@nitrkl.ac.in, 9937324437, 2462358

Basic idea

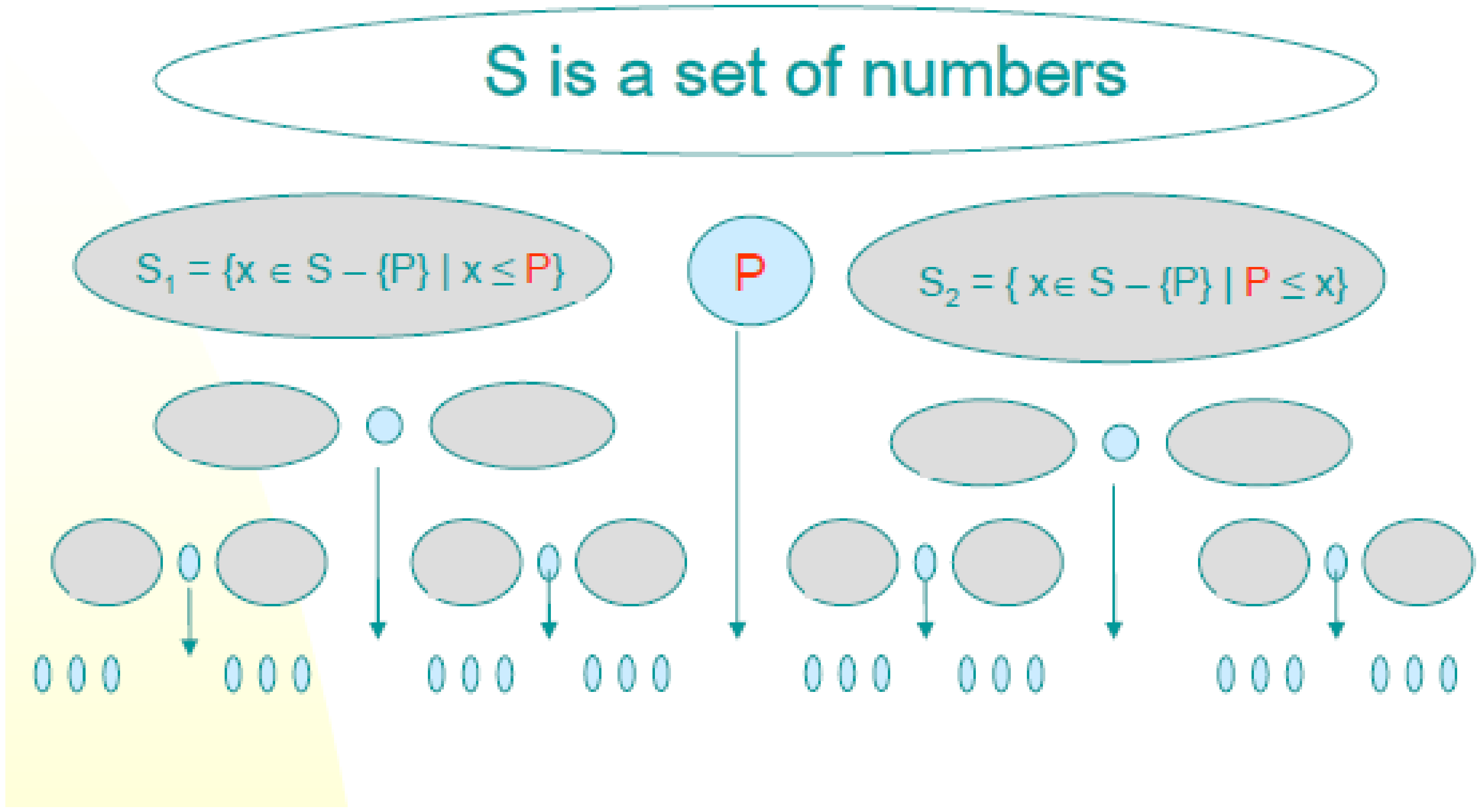

Pick a "Pivot" value, P
Create 2 new sets without P
Items smaller than or equal to P
Items greater than or equal to P
0 31 13 43 26 57
≤
65
≤
75 92 81
quicksort(S_1)
quciksort(S_2)
0 13 26 31 43 57
≤
65
≤
75 81 92
0 13 26 31 43 57 65 75 81 92

Basic idea

- Pick an element, say ***P*** (the pivot)
- Re-arrange the elements into 3 sub-blocks,
  1. those less than or equal to (≤) ***P*** (the **left-**block S_1)
  2. ***P*** (the only element in the **middle**-block)
  3. those greater than or equal to (≥) ***P*** (the **right-**block S_2)
- Repeat the process **recursively** for the **left-** and **right-** sub-blocks. Return {quicksort(S_1), ***P***, quicksort(S_2)}. (That is the results of quicksort(S_1), followed by ***P***, followed by the results of quicksort(S_2))

Basic idea

- As the name implies, it is quick, and it is the algorithm generally preferred for sorting.


S is a set of numbers
$S_1 = \{x \in S - \{P\} \mid x \leq P\}$
P
$S_2 = \{x \in S - \{P\} \mid P \leq x\}$

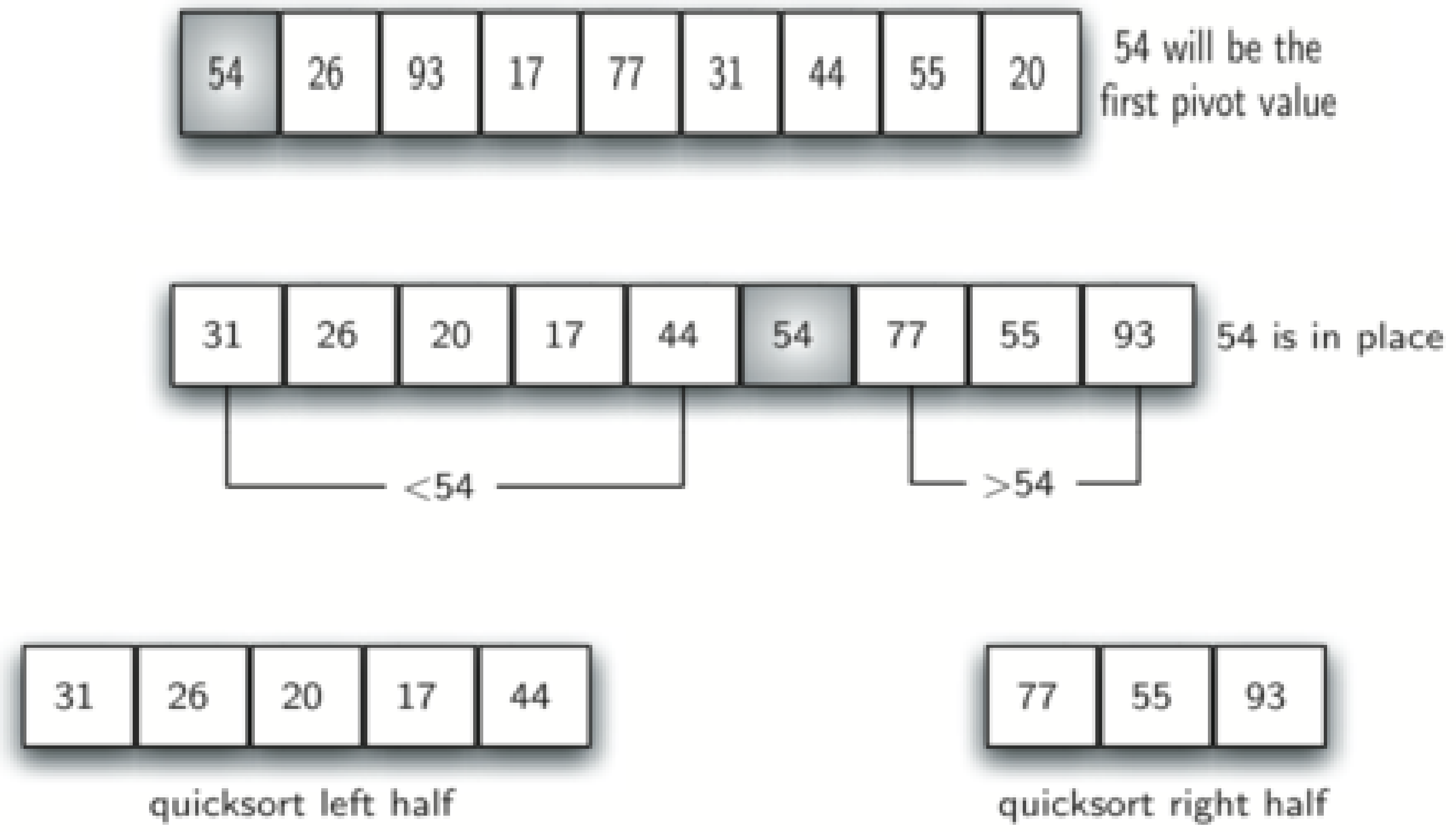
Basic idea

1. Pick one element in the array, which will be the *pivot*.
2. Make one pass through the array, called a *partition* step, rearranging the entries so that:
   - the pivot is in its proper place.
   - entries smaller than the pivot are to the left of the pivot.
   - entries larger than the pivot are to its right.
3. Recursively apply quick sort to the part of the array that is to the left of the pivot, and to the right part of the array.

- <u>Here we don't have the merge step, at the end all the elements are in the proper order</u>.

1st Pass of Quicksort

A quick sort first selects a value, which is called the **pivot value**. Although there are many different ways to choose the pivot value, we will simply use the first item in the list.



Quicksort

- Quicksort is an algorithm based on divide and conquer approach in which the array is split into subarrays and these sub-arrays are recursively called to sort the elements.
- On the basis of **Divide and conquer** approach, quicksort algorithm can be explained as:
- **Divide:** The array is divided into subparts taking pivot as the partitioning point. The elements smaller than the pivot are placed to the left of the pivot and the elements greater than the pivot are placed to the right.
- **Conquer:** The left and the right subparts are again partitioned using the by selecting pivot elements for them. This can be achieved by recursively passing the subparts into the algorithm.
- **Combine:** This step does not play a significant role in quicksort. The array is already sorted at the end of the conquer step.

Quicksort

Step 1 – Choose the highest(smallest) index value has pivot

Step 2 – Take two variables to point left and right of the list excluding pivot

Step 3 – left points to the low index

Step 4 – right points to the high

Step 5 – while value at left is less than pivot move right

Step 6 – while value at right is greater than pivot move left

Step 7 – if both step 5 and step 6 does not match swap left and right

Step 8 – if left ≥ right, the point where they met is new pivot

Algorithm Quicksort

```
Algorithm QuickSort($p, q$)
// Sorts the elements $a[p], \ldots, a[q]$ which reside in the global
// array $a[1:n]$ into ascending order; $a[n+1]$ is considered to
// be defined and must be $\geq$ all the elements in $a[1:n]$.
{
    if ($p < q$) then  // If there are more than one element
    {
        // divide $P$ into two subproblems.
            $j :=$ Partition($a, p, q+1$);
                // $j$ is the position of the partitioning element.
        // Solve the subproblems.
            QuickSort($p, j-1$);
            QuickSort($j+1, q$);
        // There is no need for combining solutions.
    }
}
```

Algorithm Partition

```
Algorithm Partition($a, m, p$)
// Within $a[m], a[m+1], \ldots, a[p-1]$ the elements are
// rearranged in such a manner that if initially $t = a[m]$,
// then after completion $a[q] = t$ for some $q$ between $m$
// and $p-1$, $a[k] \leq t$ for $m \leq k < q$, and $a[k] \geq t$
// for $q < k < p$. $q$ is returned. Set $a[p] = \infty$.
{
    $v := a[m]$; $i := m$; $j := p$;
    repeat
    {
        repeat
            $i := i + 1$;
        until ($a[i] \geq v$);

        repeat
            $j := j - 1$;
        until ($a[j] \leq v$);

        if ($i < j$) then Interchange($a, i, j$);

    } until ($i \geq j$);

    $a[m] := a[j]$; $a[j] := v$; return $j$;
}
```

```
Algorithm Interchange($a, i, j$)
// Exchange $a[i]$ with $a[j]$.
{
    $p := a[i]$;
    $a[i] := a[j]$; $a[j] := p$;
}
```

Example of partitioning

• **choose pivot:**	**4 3 6 9 2 4 3 1 2 1 8 9 3 5 6**
• **search:**	**4 3 6 9 2 4 3 1 2 1 8 9 3 5 6**
• **swap:**	**4 3 3 9 2 4 3 1 2 1 8 9 6 5 6**
• **search:**	**4 3 3 9 2 4 3 1 2 1 8 9 6 5 6**
• **swap:**	**4 3 3 1 2 4 3 1 2 9 8 9 6 5 6**
• **search:**	**4 3 3 1 2 4 3 1 2 9 8 9 6 5 6**
• **swap:**	**4 3 3 1 2 2 3 1 4 9 8 9 6 5 6**
• **search:**	**4 3 3 1 2 2 3 1 4 9 8 9 6 5 6 (left > right)**
• **swap with pivot:**	**1 3 3 1 2 2 3 4 4 9 8 9 6 5 6**

Clearly, the **Partition** function is a linear (i.e., Θ(n)) one because there is just one scan, iterating over the entire array (of size **n**) and all the other statements are constant time taking processes

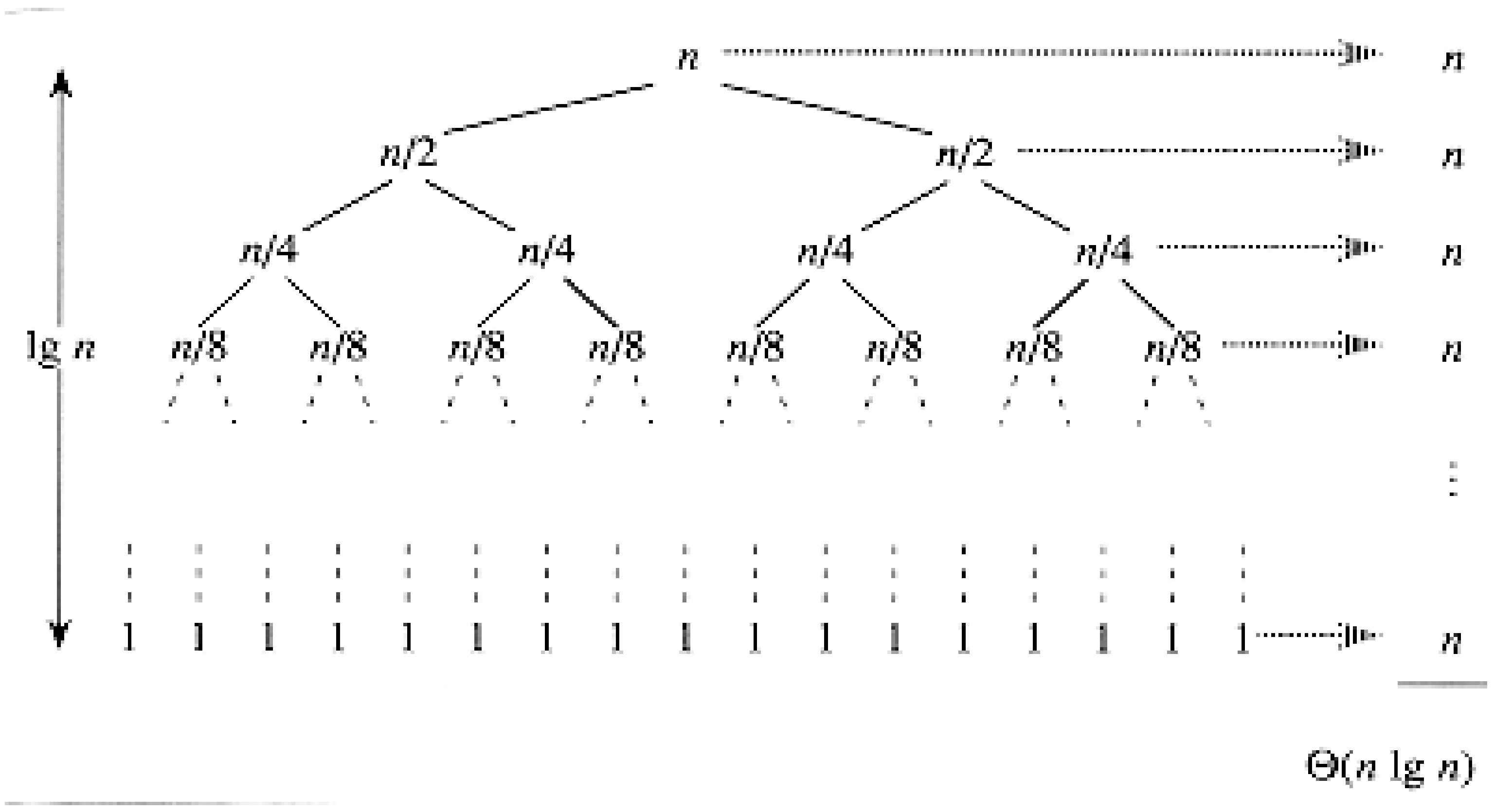
Quicksort Complexity: Time Complexities

- **Worst Case Complexity [Big-O]**:**O(n^2)**
- It occurs when the pivot element picked is either the greatest or the smallest element.
- This condition leads to the case in which the pivot element lies in an extreme end of the sorted array. One sub-array is always empty and another sub-array contains n - 1 elements. Thus, quicksort is called only on this sub-array.
- However, the quick sort algorithm has better performance for scattered pivots.
- **Best Case Complexity [Big-omega]**: **O(n*log n)**
- It occurs when the pivot element is always the middle element or near to the middle element.
- **Average Case Complexity [Big-theta]**: **O(n*log n)**
- It occurs when the above conditions do not occur.

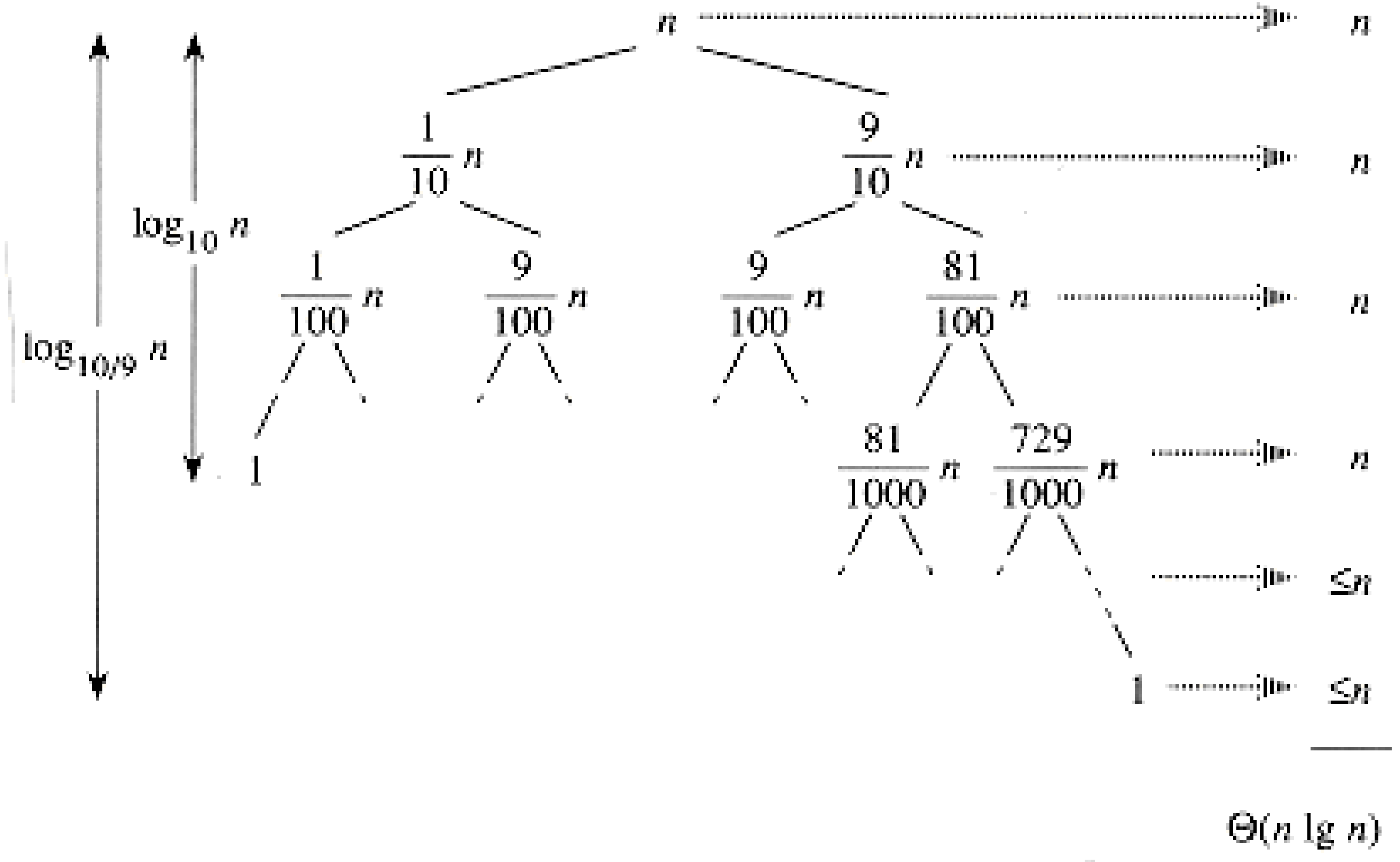
Quicksort Complexity: Space Complexities

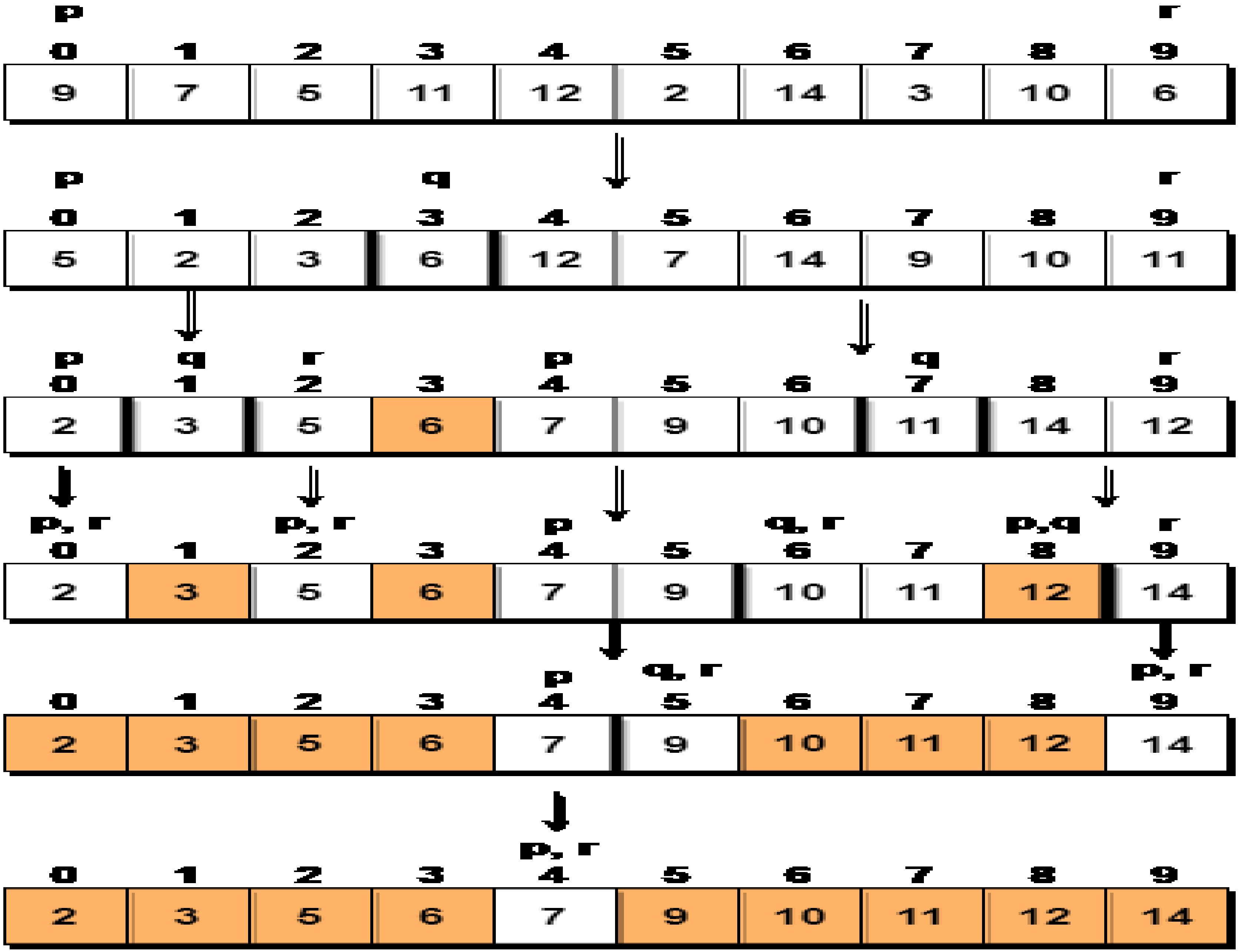
- The in-place version of **quicksort** has a **space complexity** of **O(log n)**.
- The **worst case** complexity of extra space is **O(n)**, when the algorithm encounters its worst case (a sorted list; there will be **n** recursive calls so the call stack will take up **O(n)** extra space).

A recursion tree for QUICKSORT in which PARTITION always balances the two sides of the partition equally [Best case]



A recursion tree for QUICKSORT in which PARTITION always produces a 9-to-1 split [worst case]





p r
0 1 2 3 4 5 6 7 8 9
9 7 5 11 12 2 14 3 10 6
p q r
0 1 2 3 4 5 6 7 8 9
5 2 3 6 12 7 14 9 10 11
p q r p q r
0 1 2 3 4 5 6 7 8 9
2 3 5 6 7 9 10 11 14 12
p, r p, r p q, r p,q r
0 1 2 3 4 5 6 7 8 9
2 3 5 6 7 9 10 11 12 14
p q, r p, r
0 1 2 3 4 5 6 7 8 9
2 3 5 6 7 9 10 11 12 14
p, r
0 1 2 3 4 5 6 7 8 9
2 3 5 6 7 9 10 11 12 14

Analysis of quicksort

- Quicksort is recursive; therefore, its analysis requires solving a recurrence formula.
- We will do the analysis for a quicksort, assuming a random pivot (no median of three partitioning) and no cutoff for small arrays.
- Let $\boldsymbol{T(0) = T(1) = 1}$, based on no. of comparisons with n=0 or 1.
- The running time of quicksort is equal to the running time of the two recursive calls plus the linear time spent in the partition (the pivot selection takes only constant time).
- This gives the basic quicksort relation

$$T(N) = T(i) + T(N - i - 1) + cN$$

where $i = |S_1|$ is the number of elements in S_1.

- Based on pivot selection, Time complexity of Quicksort leads to worst case, best case and average case analysis

Worst-Case Analysis O(N^2)

- The pivot is the **smallest element**, all the time. Then $i = 0$, and if we ignore $T(0) = 1$, which is insignificant, the recurrence is

$$T(1) = 1 \qquad \text{for } N = 1$$

$$T(N) = T(N-1) + cN, \qquad \text{for } N > 1 \qquad (1)$$

$$T(N) = \cancel{T(N-1)} + cN,\ N > 1 \qquad (2)$$

With **telescoping** , using Equation (2) repeatedly. Thus

$$\cancel{T(N-1)} = \cancel{T(N-2)} + c(N-1) \qquad (3)$$

$$\cancel{T(N-2)} = T(N-3) + c(N-2) \qquad (4)$$

…

$$T(2) = T(1) + c(2) \qquad (5)$$

Adding up all these equations yields

$$T(N) \quad = \quad T(1) + c\sum_{i=2}^{N} i \quad \text{as} \quad \sum_{i=1}^{N} i = \frac{N(N+1)}{2} \approx \frac{N^2}{2}$$

$\Rightarrow$ **T(N) = O(N^2)** for N > 1

Best case Analysis: O(N log N)

- If the pivot element is selected in each pass , divides the array exactly equals to half, that leads to the best case.
- Assumption: We assume that the **two subarrays are exactly half size** of the original , and although this gives a slight over estimate. This is acceptable because we are looking for the answer in a Big-Oh notation (upper bound only)

$T(1) = 1$ for $N = 1$

$T(N) = 2T(N/2) + cN,$ for $N > 1$ **(6)**

by dividing equation (6) by N ...

$\Rightarrow T(N)/N = 2T(N/2)/(N/2) + c,$ for $N > 1$ (7)

Let $N = 2^k, \Rightarrow \log N = \log 2^k \Rightarrow \log N = k \log 2 \Rightarrow \log N = k$

$\Rightarrow k = \log N$

Using telescpic sum method to equation (7)

Best case Analysis: O(N log N)

- $\frac{T(N)}{N} = \frac{T(\frac{N}{2})}{(\frac{N}{2})} + c \qquad \ldots(8)$
- $\frac{T(\frac{N}{2})}{(\frac{N}{2})} = \frac{T(\frac{N}{4})}{(\frac{N}{4})} + c \qquad \ldots(9)$
- $\frac{T(\frac{N}{4})}{(\frac{N}{4})} = \frac{T(\frac{N}{8})}{(\frac{N}{8})} + c \qquad \ldots(10)$

  .
- .

  .
- $\frac{T(2)}{2} = \frac{T(1)}{(1)} + c \qquad \ldots(11)$
- Adding above equations
- $\frac{T(N)}{N} = \mathbf{T(1) + kc} \qquad \ldots(12)$

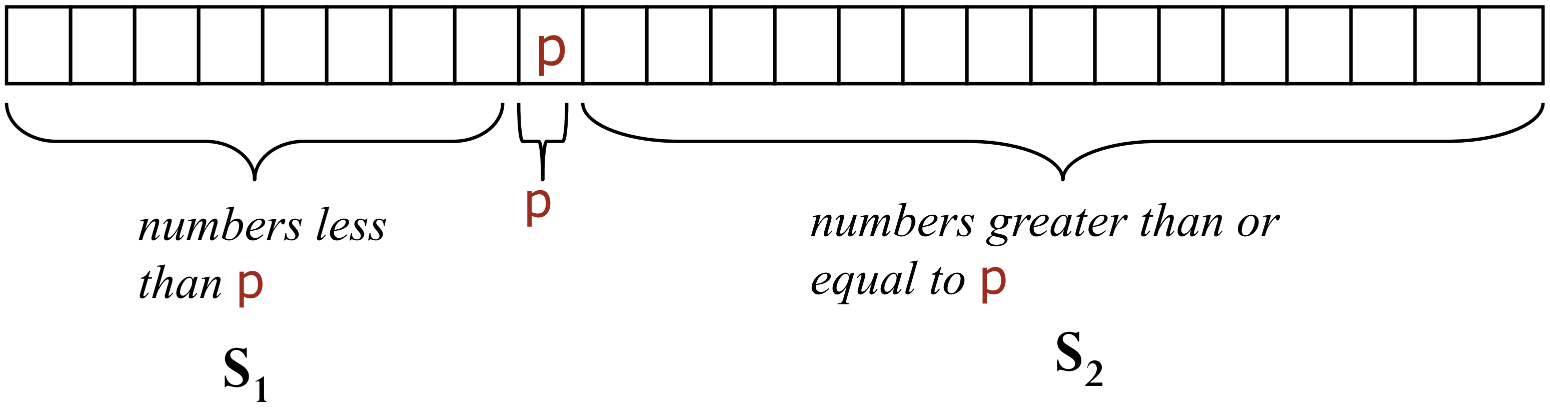
k equations as N = 2^k

Best case Analysis: O(N log N)

- $\frac{T(N)}{N} = \mathrm{T}(1) + \mathrm{kc}$ ...(12)
- $\frac{T(N)}{N} = \mathrm{T}(1) + \mathrm{c} \log \mathrm{N}$ ...(13)
- $\frac{T(N)}{N} = 1 + \mathrm{c} \log \mathrm{N}$ ...(14) as $\mathrm{T}(1) = 1$

$\Rightarrow$ $T(N) = N + cN \log N$, Hence $\boldsymbol{T(N) = \mathrm{O}(N \log N)}$

Average case Analysis Quicksort :O(N log N)



- Let us assume that in a given set of N element, selecting any element as pivot is equally likely i.e. **1/N**
- A key step in the Quicksort algorithm is partitioning the array. In a pass , the partition divided the array into three parts: **S_1**, p, **S_2**.
- The |**S_1**| is 0 or 1 or 2 or …(N-1), each of the sizes for **S_1** is equally likely with a probability **1/N.**

$$T(1) = 1 \qquad \textbf{for } N = 1$$

$$T(N) = T(i) + T(N-i-1) + cN, \quad \textbf{for } N > 1 \qquad (15)$$

Average case Analysis Quicksort :O(N log N)

$T(1) = 1$ for N =1

$T(N) = T(\mathrm{i}) + \mathrm{T(N\text{-}i\text{-}1)} + cN,$ for $N > 1$ (15)

- The average value of $T(i) = {}^{1}/_{N} \sum_{j=0}^{N-1} T(j)$

All possible sub problem

T(0)	T(N-1)
T(1)	T(N-2)
T(2)	T(N-3)
.	.
.	.
.	.
T(N-1)	T(0)

Total sub problems = $2 \sum_{j=0}^{N-1} T(j)$

Average case Analysis Quicksort :O(N log N)

Using **Total sub problems** = $2\sum_{j=0}^{N-1} T(j)$ in equation 15 $\Rightarrow$

$T(1) = 1$ **for N =1**

$T(N) = \frac{2}{N}\sum_{j=0}^{N-1} T(j) + cN,$ **for** $N > 1$ **(16)**

Multiplying the equation (16) with N

$\Rightarrow NT(N) = 2\sum_{j=0}^{N-1} T(j) + cN^2$ **(17)**

Inorder to remove the summation sign from equation (17), we can telescope with one more equation:

$\Rightarrow (N-1)T(N-1) = 2\sum_{j=0}^{N-2} T(j) + c(N-1)^2$ **(18)**

- Subtracting (18) from (17)

$\Rightarrow NT(N) - [(N-1)T(N-1)] = [2\sum_{j=0}^{N-1} T(j) + cN^2] - [2\sum_{j=0}^{N-2} T(j) + c(N-1)^2]$

$\Rightarrow NT(N) - (N-1)T(N-1) = 2T(N-1) + cN^2 - cN^2 - c + 2cN$

Average case Analysis Quicksort :O(N log N)

- $NT(N) - (N-1)T(N-1) = 2T(N-1) + cN^2 - cN^2 - c + 2cN$

$\Rightarrow NT(N) - (N-1)T(N-1) = 2T(N-1) + 2cN - c$

By dropping the insignificant term –c

$\Rightarrow NT(N) - (N-1)T(N-1) = 2T(N-1) + 2cN$

$\Rightarrow NT(N) = 2T(N-1) + (N-1)T(N-1) + 2cN$

$\Rightarrow NT(N) = (N+1)T(N-1) + 2cN$ (19)

Dividing the above equation by N(N-1)

$$\Rightarrow \frac{NT(N)}{N(N+1} = \frac{(N+1)T(N-1)}{N(N+1)} + \frac{2cN}{N(N+1)}$$

$$\Rightarrow \frac{T(N)}{(N+1} = \frac{T(N-1)}{N} + \frac{2c}{(N+1)} \qquad (20)$$

Using the method of telescopic sum:

Average case Analysis Quicksort :O(N log N)

$$\Rightarrow \frac{T(N)}{(N+1)} = \frac{T(N-1)}{N} + \frac{2c}{(N+1)} \qquad (20)$$

$$\Rightarrow \frac{T(N-1)}{N} = \frac{T(N-2)}{(N-1)} + \frac{2c}{N} \qquad (21)$$

$$\Rightarrow \frac{T(N-2)}{(N-1)} = \frac{T(N-3)}{(N-2)} + \frac{2c}{N-1} \qquad (22)$$

.

.

.

$$\Rightarrow \frac{T(2)}{3} = \frac{T(1)}{2} + \frac{2c}{3} \qquad (23)$$

Adding all equations from (20) ….

$$\Rightarrow \frac{T(N)}{(N+1)} = \frac{T(1)}{2} + 2c \sum_{i=3}^{N+1} 1/i \qquad (21)$$

Average case Analysis Quicksort :O(N log N)

$\Rightarrow \frac{T(N)}{(N+1} = \frac{T(1)}{2} + 2c \sum_{i=3}^{N+1} 1/i$ (21)

Assume that the sum $\mathbf{2c \sum_{i=3}^{N+1} 1/i}$ is about **$\log_e$ (N+1)** +ν - 3/2, where ν ≈ 0.577 and known as Euler's constant.

$\Rightarrow \frac{T(N)}{(N+1)} = O(\log N)$

$\Rightarrow T(N) = (N+1)\, O(\log N)$

$\Rightarrow \boldsymbol{T(N) = O(N \log N)}$

Motivation for Iterative version of Quicksort

- In worst case Quicksort requires a stack space **O(n-1)** to manage the recursion.
- The amount of stack space needed can be reduced to **O(log n)** by using iterative version of quicksort in which the smaller of the two subarray a[p:j-1] and a[j+1:q] is always sorted first.
- The second recursion call can be replaced by some assignment statements and a jump to the beginning of the algorithm.
- The algorithm **Quicksort2** uses stack space of size **2 log n**.
- **Add** and **Delete** are the PUSH and POP operation equivalent in a typical STACK

We can now verify that the maximum stack space needed is $O(\log n)$. Let $S(n)$ be the maximum stack space needed. Then it follows that

$$S(n) \leq \begin{cases} 2 + S(\lfloor (n-1)/2 \rfloor) & n > 1 \\ 0 & n \leq 1 \end{cases}$$

which is less than $2 \log n$.

Iterative version of Quicksort

```
Algorithm QuickSort2($p, q$)
// Sorts the elements in $a[p : q]$.
{
    // $stack$ is a stack of size $2\log(n)$.
    repeat
    {
        while ($p < q$) do
        {
            $j$ := Partition($a, p, q + 1$);
            if (($j - p$) < ($q - j$)) then
            {
                Add($j + 1$); // Add $j + 1$ to $stack$.
                Add($q$); $q := j - 1$; // Add $q$ to $stack$
            }
            else
            {
                Add($p$); // Add $p$ to $stack$.
                Add($j - 1$); $p := j + 1$; // Add $j - 1$ to $stack$
            }
        } // Sort the smaller subfile.
        if $stack$ is empty then return;
        Delete($q$); Delete($p$); // Delete $q$ and $p$ from $stack$.
    } until (false);
}
```

Iterative version of Quicksort

Iterative Quicksort Algorithm

- ***Create a stack which has the size of the array***

1. PUSH Initial values of start and end in the stack S, parent array(full array) start and end indexes
2. Till the stack is empty
3. POP start and end indexes in the stack
4. call the partition function and store the return value it in pivot index
5. Now, PUSH the left subarray indexes that is less to the pivot into the stack S, start, pivot_index -1
6. push the right subarray indexes that is greater than pivot into the stack S, pivot+1, end

How to choose the pivot in Quicksort

- A **quicksort algorithm** should always aim to choose the middle-most element as its **pivot**.
- Some **algorithms** will literally select the center-most item as the **pivot**, while others will select the first or the last element.
- A safe course is merely to choose the **pivot randomly**. This strategy is generally perfectly safe, unless the random number generator has a flaw, since it is very unlikely that a random pivot would consistently provide a poor partition. On the other hand, random number generation is generally an expensive commodity and does not reduce the average running time of the rest of the algorithm at all.
- **Median-of-Three Partitioning***:* The median of a group of *N* numbers is the *N*/2th largest number. The best choice of pivot would be the median of the array. Unfortunately, this is hard to calculate and would slow down quicksort considerably.
- A good estimate can be obtained by **picking three elements randomly** and using the median of these three as pivot.

Quicksort : Notes

- Quicksort is the fastest general-purpose sort.
- Quick Sort is a tail-recursive, in-place algorithm that makes it suitable for use in case of arrays of a large number of elements.
- In most practical situations, quicksort is the method of choice. If stability is important and space is available, merge sort might be best.
- Commercial applications use Quick sort - generally it runs fast, no additional memory, this compensates for the rare occasions when it runs with $O(N^2)$
- **Never use in applications which require guaranteed response time:** Life-critical (medical monitoring, life support in aircraft and space craft) Mission-critical (monitoring and control in industrial and research plants handling dangerous materials, control for aircraft, defense, etc) unless you assume the worst-case response time.

Quicksort : Notes

- In some performance-critical applications, the focus may be on just sorting numbers, so it is reasonable to avoid the costs of using references and sort primitive types instead.
- If you need to sort a data structure like linked list for example merge sort preferred and not quicksort.
- When there is a limitation on memory then randomised quicksort can be used. merge sort is not an in-place sorting algorithm and requires some extra space.
- Choice of sorting algorithm can also depend on the data. if the data is almost sorted then quicksort can be used. (See **Dutch national flag problem** this an O(N) complexity sorting algorithm).
- Choice of sorting algorithm may also depend on the architecture of the machine. The time it take to write into the memory and read from the memory depends on the processor and memory design.

Comparison

Comparison with heap sort:

- both algorithms have O(N log N) complexity
- quick sort runs faster, (does not support a heap tree)
- the speed of quick sort is not guaranteed

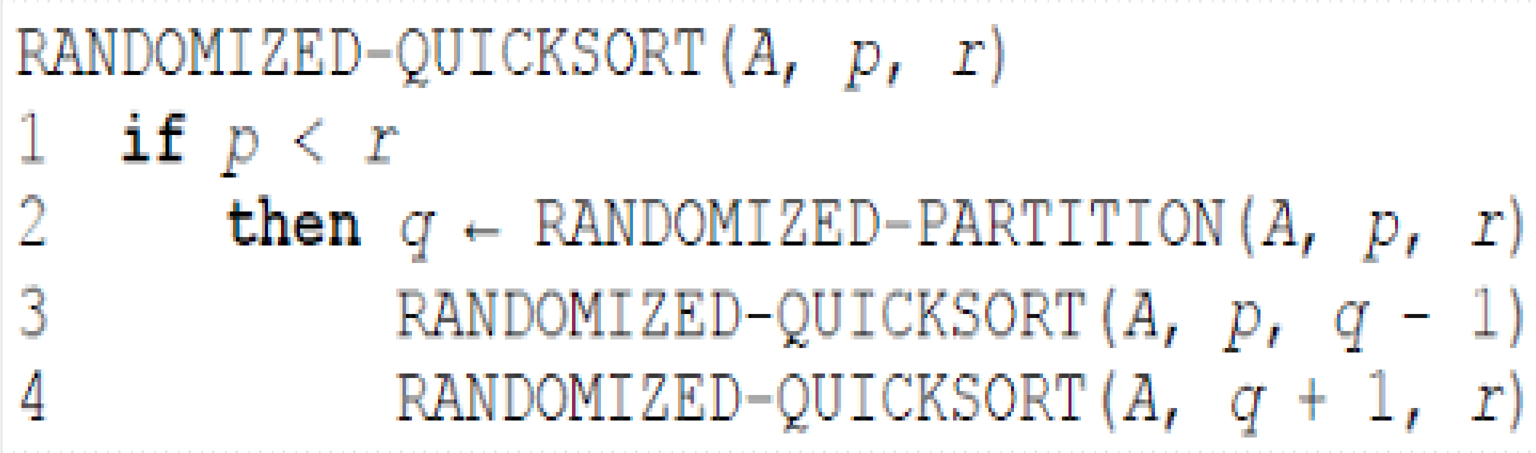
Comparison with merge sort:

- merge sort guarantees O(N log N) time, however it requires additional memory with size N.
- quick sort does not require additional memory, however the speed is not guaranteed
- usually merge sort is not used for main memory sorting, only for external memory sorting.

Summary of Sorting Algorithms

Algorithm	Time	Notes
selection-sort	$O(n^2)$	◆ in-place ◆ slow (good for small inputs)
insertion-sort	$O(n^2)$	◆ in-place ◆ slow (good for small inputs)
quick-sort	$O(n \log n)$ expected	◆ in-place, randomized ◆ fastest (good for large inputs)
heap-sort	$O(n \log n)$	◆ in-place ◆ fast (good for large inputs)
merge-sort	$O(n \log n)$	◆ sequential data access ◆ fast (good for huge inputs)

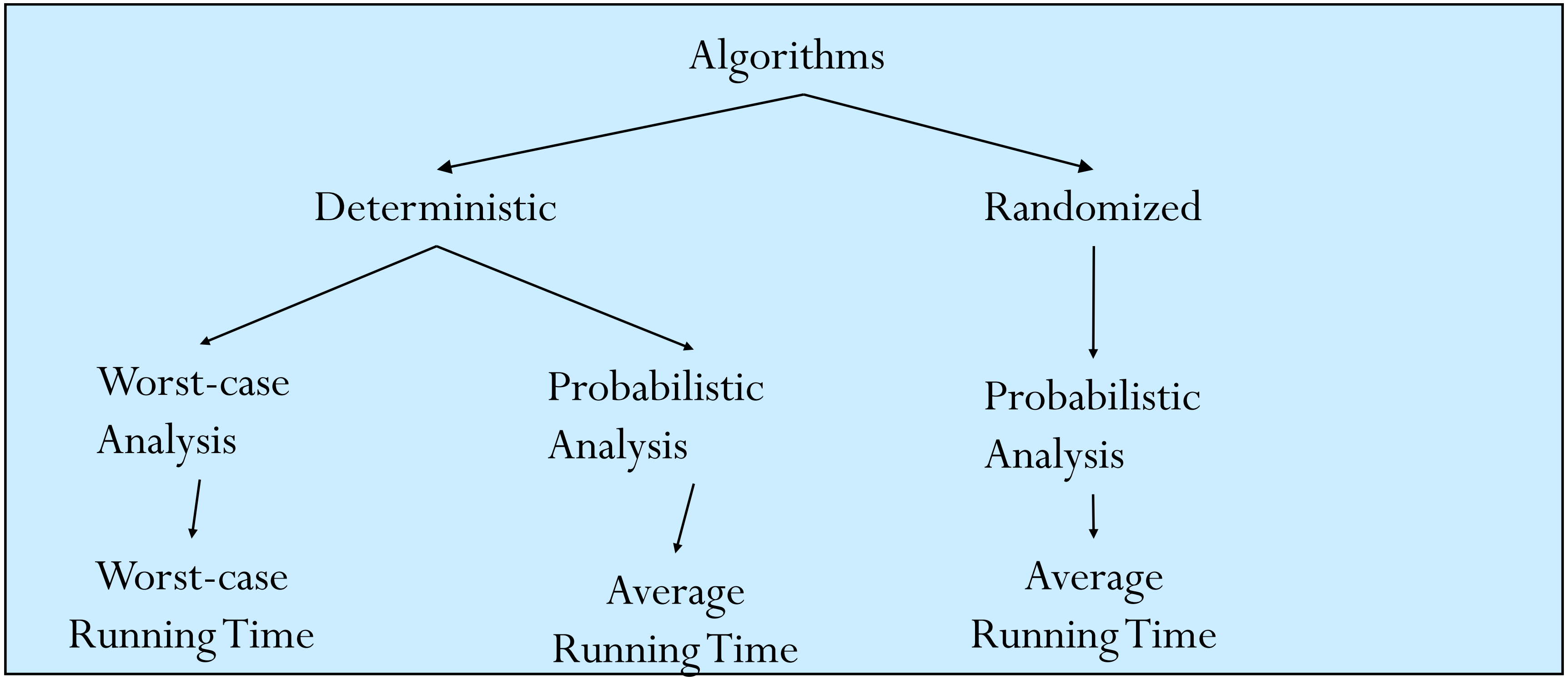
Randomized Quick Sort



```
RANDOMIZED-QUICKSORT($A$, $p$, $r$)
if $p < r$
    then $q \leftarrow$ RANDOMIZED-PARTITION($A$, $p$, $r$)
         RANDOMIZED-QUICKSORT($A$, $p$, $q - 1$)
         RANDOMIZED-QUICKSORT($A$, $q + 1$, $r$)
```

Deterministic vs. Randomized Algorithms

- **Deterministic Algorithm : Identical behavior for different runs for a given input.**
- **Randomized Algorithm : Behavior is generally different for different runs for a given input.**



Classification of Randomized Algorithm

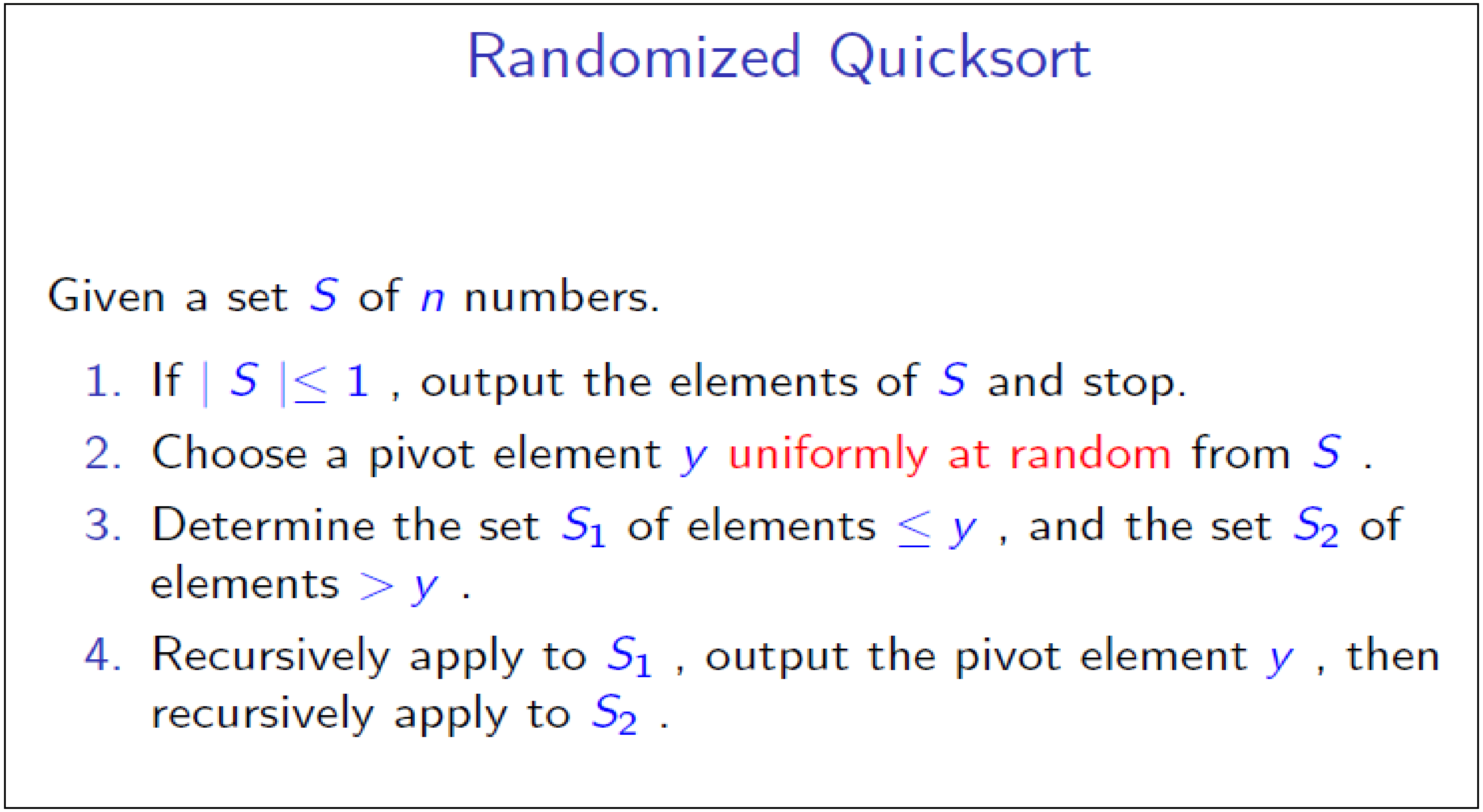
1. **Numerical Probabilistic Algorithm**
2. **Monte-carlo Algorithm**
3. **Las-Vegas Algorithm**
4. **Sherwood Algorithm**

Sherwood Algorithm

- Algorithms which always return a result and the **correct result**, but where a random element increases the efficiency, by avoiding or reducing the probability of worst-case behaviour.
- This is useful for algorithms which have a poor worst-case behaviour but a good average-case behaviour, and in particular can be used where embedding an algorithm in an application may lead to increased worst-case behaviour.
- Quicksort is a sorting algorithm with worst case behaviour $O(N^2)$ but average case behaviour of $O(N \times \log(N))$. The problem is that we cannot guarantee very uneven splits of the list by choosing pivots.
- Indeed, systematic choice of pivots (e.g. the first element), can lead, in certain applications (e.g. when almost sorted lists tend to be supplied), to worst-case behaviour. This is undesirable!

Randomized Quicksort

- **Randomized Quick Sort** is an extension of Quick Sort in which the pivot element is chosen randomly.



Randomized Quicksort

Given a set S of n numbers.

1. If $|S| \leq 1$, output the elements of S and stop.
2. Choose a pivot element y uniformly at random from S.
3. Determine the set S_1 of elements $\leq y$, and the set S_2 of elements $> y$.
4. Recursively apply to S_1, output the pivot element y, then recursively apply to S_2.

Randomized Quicksort

IDEA: Partition around a random element.

- Running time is independent of the input order. It depends only on the sequence s of random numbers.
- No assumptions need to be made about the input distribution.
- No specific input elicits the worst-case behavior.
- The worst case is determined only by the sequence **s** of random numbers.
- Expected runtime is the expected value of the runtime random variable of a randomized algorithm. It effectively "averages" over all sequences of random numbers.
- De facto both analyses are very similar. However in practice the randomized algorithm ensures that not one single input elicits worst case behavior.

Randomized Quick Sort

```
**Algorithm** RQuickSort($p, q$)
// Sorts the elements $a[p], \ldots, a[q]$ which reside in the global
// array $a[1:n]$ into ascending order. $a[n+1]$ is considered to
// be defined and must be $\geq$ all the elements in $a[1:n]$.
{
    **if** ($p < q$) **then**
    {
        **if** (($q - p$) $> 5$) **then**
            Interchange($a$, Random() **mod** $(q - p + 1) + p, p$);
        $j :=$ Partition($a, p, q + 1$);
            // $j$ is the position of the partitioning element.
        RQuickSort($p, j - 1$);
        RQuickSort($j + 1, q$);
    }
}
```

Partition Algorithm

```
Algorithm Partition($a, m, p$)
// Within $a[m], a[m+1], \ldots, a[p-1]$ the elements are
// rearranged in such a manner that if initially $t = a[m]$,
// then after completion $a[q] = t$ for some $q$ between $m$
// and $p-1$, $a[k] \le t$ for $m \le k < q$, and $a[k] \ge t$
// for $q < k < p$. $q$ is returned. Set $a[p] = \infty$.
{
    $v := a[m]$; $i := m$; $j := p$;
    repeat
    {
        repeat
            $i := i + 1$;
        until ($a[i] \ge v$);

        repeat
            $j := j - 1$;
        until ($a[j] \le v$);

        if ($i < j$) then Interchange($a, i, j$);

    } until ($i \ge j$);

    $a[m] := a[j]$; $a[j] := v$; return $j$;
}
```

Interchange Algorithm

```
Algorithm Interchange($a, i, j$)
// Exchange $a[i]$ with $a[j]$.
{
    $p := a[i]$;
    $a[i] := a[j]$; $a[j] := p$;
}
```

Randomized Quick Sort

- Randomized Quick Sort **RQuickSort** runs on a expected **O(N log N)** time.
- Every call the randomizer Random takes a certain amount of time. If there is a few elements to sort, the time taken by the randomizer may be comparable to the rest of the computation.
- In the algorithm **RQuickSort** the randomizer is invoked only if (p-q) > **x**, But **x** is not the magic number. This number to be determined empirically.
- Using a uniform randomizer selecting any element as pivot in a array of **N** element is equally likely i.e. **1/N .** Hence the time complexity **RQuickSort** is similar to the average case analysis of **Quicksort.**

$T(1) = 1$ **for N =1**

$T(N) = T(\mathrm{i}) + \mathrm{T(N\text{-}i\text{-}1)} + cN,$ **for** $N > 1$

Time complexity of Randomized Quick Sort

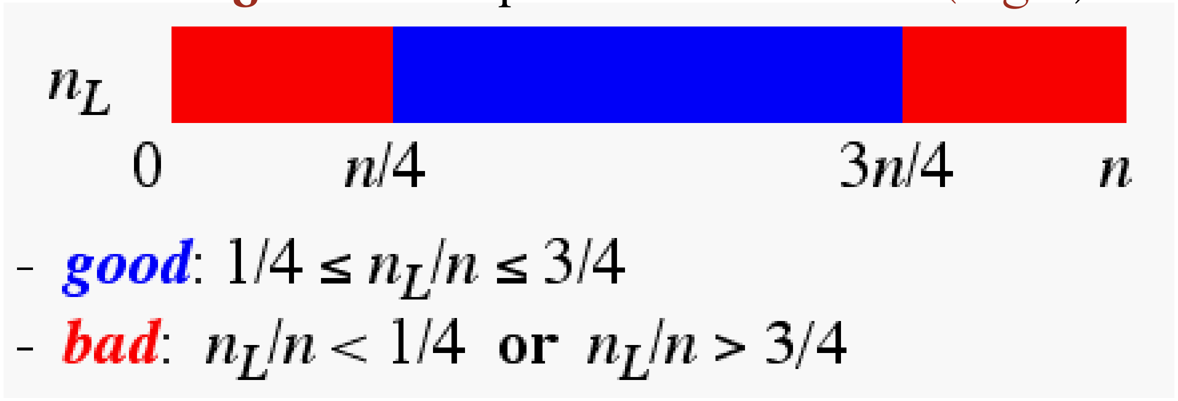
$T(0) = 1$ for $N = 1$

$T(N) = T(i) + T(N-i-1) + cN,$ for $N > 1$

$\Rightarrow T(N) = O(\log N)$

Randomized Quick-Sort

- Select the pivot as a ***random*** element of the sequence.
- The expected running time of randomized quick-sort on a sequence of size n is O(n log n).
- The time spent at a level of the quick-sort tree is O(n)
- We show that the ***expected height*** of the quick-sort tree is O(log n)
- good vs. bad pivots



- ***good***: $1/4 \le n_L/n \le 3/4$
- ***bad***: $n_L/n < 1/4$ **or** $n_L/n > 3/4$

- The probability of a good pivot is 1/2, thus we expect k/2 good pivots out of k pivots
- After a good pivot the size of each child sequence is at most 3/4 the size of the parent sequence
- After h pivots, we expect $(3/4)^{h/2}$ n elements
- The expected height h of the quick-sort tree is at most: $2 \log_{4/3} n$

A generalized version of randomized algorithm for sorting

```
Algorithm RSort($a, n$)
// Sort the elements $a[1 : n]$.
{
    Randomly sample $s$ elements from $a[\,]$;
    Sort this sample;
    Partition the input using the sorted sample as partition keys;
    Sort each part separately;
}
```

What is 3-Way Quicksort?

- In simple QuickSort algorithm, we select an element as pivot, partition the array around pivot and recur for subarrays on left and right of pivot.
- Consider an array which has many redundant elements. For example, {1, 4, 2, 4, 2, 4, 1, 2, 4, 1, 2, 2, 2, 2, 4, 1, 4, 4, 4}. If 4 is picked as pivot in Simple QuickSort, we fix only one 4 and recursively process remaining occurrences.
- In 3 Way QuickSort, an array arr[l..r] is divided in 3 parts:

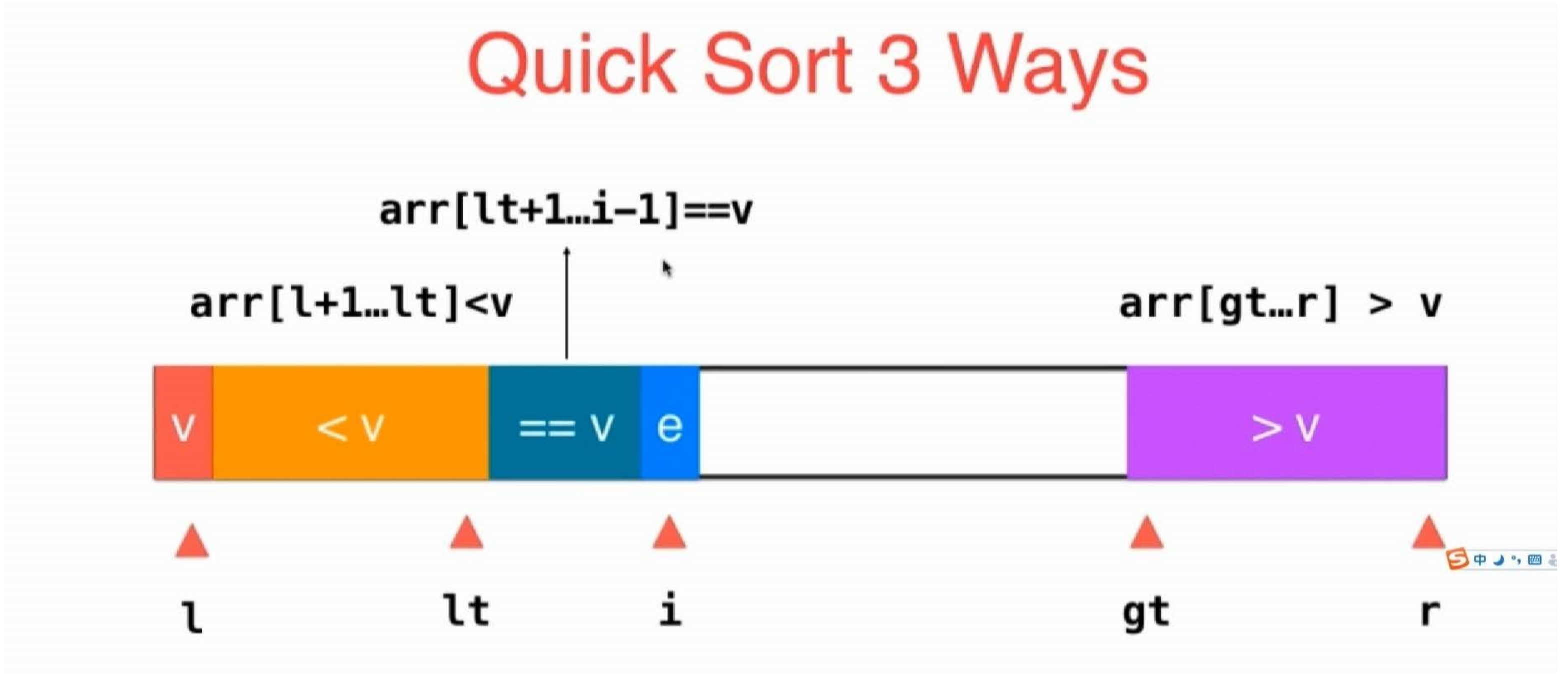
  a) arr[l..i] elements less than pivot.

  b) arr[i+1..j-1] elements equal to pivot.

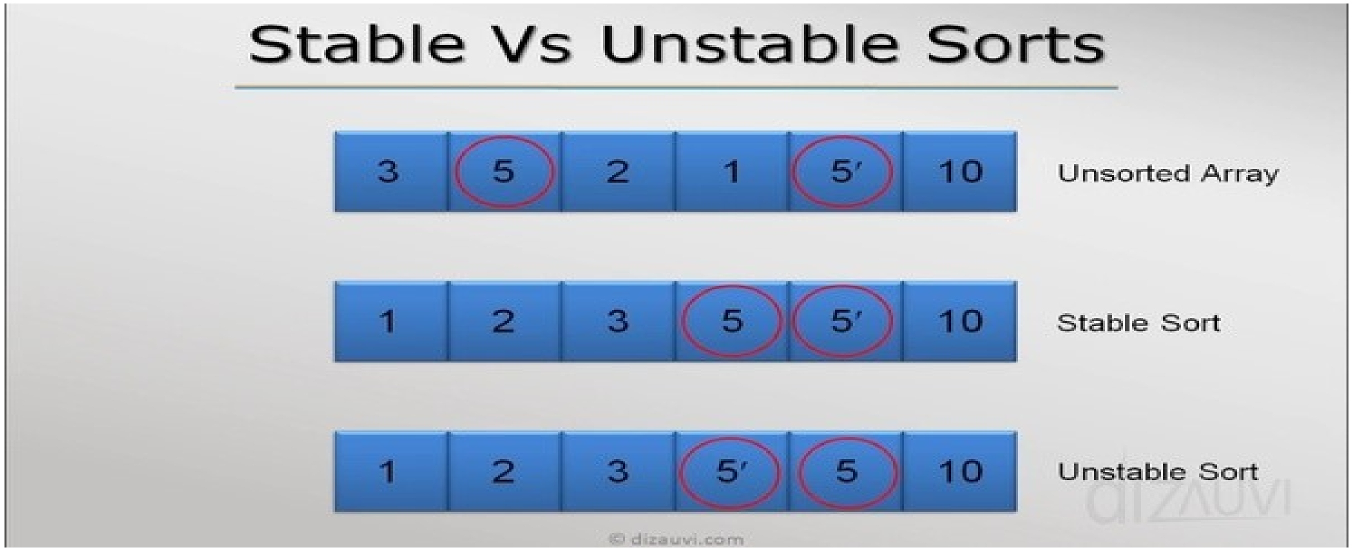
  c) arr[j..r] elements greater than pivot.

3-Way Quicksort (Dutch National Flag)

- The 3-way partition variation of quick sort has slightly higher overhead compared to the standard 2-way partition version.
- Both have the same best, typical, and worst case time bounds, but this version is highly adaptive in the very common case of sorting with few unique keys.



- **Is QuickSort stable?** The default implementation is not stable. However any sorting algorithm can be made stable by considering indexes as comparison parameter.



- **Is QuickSort In-place?** As per the broad definition of in-place algorithm it qualifies as an in-place sorting algorithm as it uses extra space only for storing recursive function calls but not for manipulating the input.

Why Quick Sort is preferred over MergeSort for sorting Arrays ?

- Quick Sort in its general form is an in-place sort (i.e. it doesn’t require any extra storage) whereas merge sort requires O(N) extra storage, N denoting the array size which may be quite expensive. Allocating and de-allocating the extra space used for merge sort increases the running time of the algorithm. Comparing average complexity we find that both type of sorts have O(NlogN) average complexity but the constants differ. For arrays, merge sort loses due to the use of extra O(N) storage space.
- Most practical implementations of Quick Sort use randomized version. The randomized version has expected time complexity of O(nLogn). The worst case is possible in randomized version also, but worst case doesn’t occur for a particular pattern (like sorted array) and randomized Quick Sort works well in practice.
- Quick Sort is also a cache friendly sorting algorithm as it has good locality of reference when used for arrays.
- Quick Sort is also tail recursive, therefore tail call optimizations is done.

Why Merge Sort is preferred over Quicksort for Linked Lists?

- In case of linked lists the case is different mainly due to difference in memory allocation of arrays and linked lists. Unlike arrays, linked list nodes may not be adjacent in memory. Unlike array, in linked list, we can insert items in the middle in O(1) extra space and O(1) time. Therefore merge operation of merge sort can be implemented without extra space for linked lists.
- In arrays, we can do random access as elements are continuous in memory. Let us say we have an integer (4-byte) array A and let the address of A[0] be x then to access A[i], we can directly access the memory at (x + i*4). Unlike arrays, we can not do random access in linked list. Quick Sort requires a lot of this kind of access. In linked list to access i'th index, we have to travel each and every node from the head to i'th node as we don't have continuous block of memory. Therefore, the overhead increases for quick sort. Merge sort accesses data sequentially and the need of random access is low.

External quicksort

- An external sorting algorithm based on quicksort is presented. The file to be sorted is kept on a disk and only those blocks are fetched into the main memory which are currently needed. At each time, a block is kept in the main memory, if the expected space-time cost of holding it until its next use is smaller than the expected space-time cost of removing it and fetching it again.

- A.Inkeri Verkamo, External quicksort,Performance Evaluation, Volume 8, Issue 4, 1988, Pages 271-288, ISSN 0166-5316, https://doi.org/10.1016/0166-5316(88)90029-6.
- (http://www.sciencedirect.com/science/article/pii/0166531688900296)

Thanks for Your Attention!

Exercises

Exercises

1. Briefly describe the basic idea of quick sort. What is the complexity of quick sort? Analyze the worst-case complexity solving the recurrence relation.
2. Briefly describe the basic idea of quick sort. Analyze the best-case complexity solving the recurrence relation.
3. Compare quick sort with merge sort and heap sort.
4. What are the advantages and disadvantages of quick sort? Which applications are not suitable for quick sort and why?
5. Why is Quicksort better than other sorting algorithms in practice?
6. How to implement QuickSort for Linked Lists?
7. Show that the running time of QUICKSORT is $(n \log n)$ when all elements of array A have the same value.
8. Show that the running time of QUICKSORT is (n^2) when the array A is sorted in non-increasing order.

Exercises

9. Banks often record transactions on an account in order of the times of the transactions, but many people like to receive their bank statements with checks listed in order by check number. People usually write checks in order by check number, and merchants usually cash them with reasonable dispatch. The problem of converting time-of-transaction ordering to check-number ordering is therefore the problem of sorting almost-sorted input. Argue that the procedure INSERTION-SORT would tend to beat the procedure QUICKSORT on this problem.

10. Suppose that the splits at every level of quicksort are in the proportion 1 - α to α, where $\mathbf{0 < \alpha \leq 1/2}$ is a constant. Show that the minimum depth of a leaf in the recursion tree is approximately ep **log n/log α** and the maximum depth is approximately **log n/log(1 - α).** (Don't worry about integer round-off.)

Exercises

11. Argue that for any constant $0 < \alpha \leq 1/2$, the probability is approximately $1 - 2\alpha$ that on a random input array, PARTITION produces a split more balanced than $1 - \alpha$ to α. For what value of are the odds even that the split is more balanced than less balanced?
12. Why do we analyze the average-case performance of a randomized algorithm and not its worst-case performance?
13. During the running of the procedure RANDOMIZED-QUICKSORT, how many calls are made to the random-number generator RANDOM in the worst case? How does the answer change in the best case?
14. Describe an implementation of the procedure RANDOM*(a, b)* that uses only fair coin flips. What is the expected running time of your procedure?
15. Give a $\Theta(n)$-time, randomized procedure that takes as input an array $A[1. . . n]$ and performs a random permutation on the array elements.

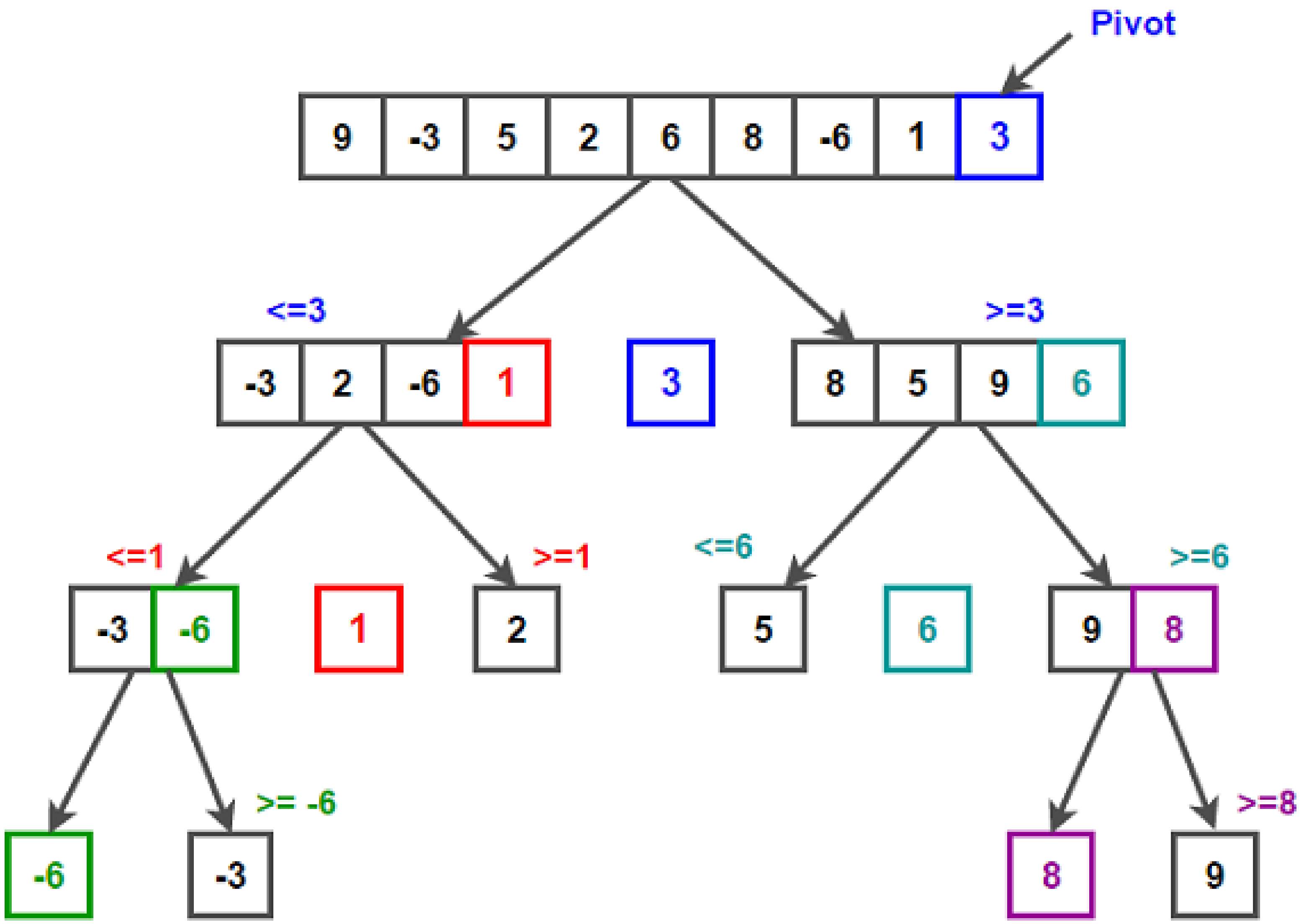
Exercises

16. Show that quicksort's best-case running time is (n logn).
17. Show that RANDOMIZED-QUICKSORT's expected running time is Ω(n log n).
18. The running time of quicksort can be improved in practice by taking advantage of the fast running time of insertion sort when its input is "nearly" sorted. When quicksort is called on a subarray with fewer than k elements, let it simply return without sorting the subarray. After the top-level call to quicksort returns, run insertion sort on the entire array to finish the sorting process. Argue that this sorting algorithm runs in $O(nk + n \log(n/k))$ expected time. How should k be picked, both in theory and in practice?
19. Consider modifying the PARTITION procedure by randomly picking three elements from array A and partitioning about their median. Approximate the probability of getting at worst an α-to-(1 -α) split, as a function of α in the range $0 < \alpha < 1$

Exercises

20. What value of q does PARTITION return when all elements in the array $A[p\,..\,r]$ have the same value?
21. Give a brief argument that the running time of PARTITION on a subarray of size n is $\Theta(n)$.
22. How would you modify QUICKSORT to sort in non-increasing order?
23. Present an analysis of quicksort when all keys are equal?

Quicksort selecting last item of input array as pivot



Worst case analysis: Quick Sort

$$T(N) = T(i) + T(N - i - 1) + cN$$

- If the pivot is the smallest element or largest element

$$\mathbf{T(N) = T(N\text{-}1) + cN,\ N > 1}$$

- Telescoping:

$$T(N-1) = T(N-2) + c(N-1)$$
$$T(N-2) = T(N-3) + c(N-2)$$
$$T(N-3) = T(N-4) + c(N-3)$$
$$\ldots$$
$$T(2) = T(1) + c.2$$

By adding above equaions $T(N) = 1 + c(N(N+1)/2 - 1)$

Therefore **$T(N) = O(N^2)$**

Average case analysis: Quick Sort

- The average value of T(i) is 1/N times the sum of T(0) through T(N-1)
- $1/N \ \sum T(j)$, $j = 0$ thru $N-1$
- $T(N) = 2/N \left(\sum T(j)\right) + cN$
- Multiply by N
- $NT(N) = 2\left(\sum T(j)\right) + cN*N$
- To remove the summation, we rewrite the equation for N-1:
- $(N-1)T(N-1) = 2\left(\sum T(j)\right) + c(N-1)^2$, $j = 0$ thru $N-2$
- and subtract:
- $NT(N) - (N-1)T(N-1) = 2T(N-1) + 2cN - c$
- Prepare for telescoping. Rearrange terms, drop the insignificant c:

$$NT(N) = (N+1)T(N-1) + 2cN$$

Average case analysis: Quick Sort

$$NT(N) = (N+1)T(N-1) + 2cN$$

Divide by N(N+1):

$T(N)/(N+1) = T(N-1)/N + 2c/(N+1)$

Telescope:

$$\begin{aligned} T(N)/(N+1) &= T(N-1)/N + 2c/(N+1) \\ T(N-1)/(N) &= T(N-2)/(N-1) + 2c/(N) \\ T(N-2)/(N-1) &= T(N-3)/(N-2) + 2c/(N-1) \\ &\ldots\ldots \\ T(2)/3 &= T(1)/2 + 2c/3 \end{aligned}$$

Add the equations and cross equal terms:

$T(N)/(N+1) = T(1)/2 + 2c \sum (1/j)$, $j = 3$ to $N+1$

$T(N) = (N+1)(1/2 + 2c \sum(1/j))$

- The sum $\sum (1/j)$, $j = 3$ to $N-1$, is about LogN
- Thus **T(N) = O(NlogN)**