Quick sort

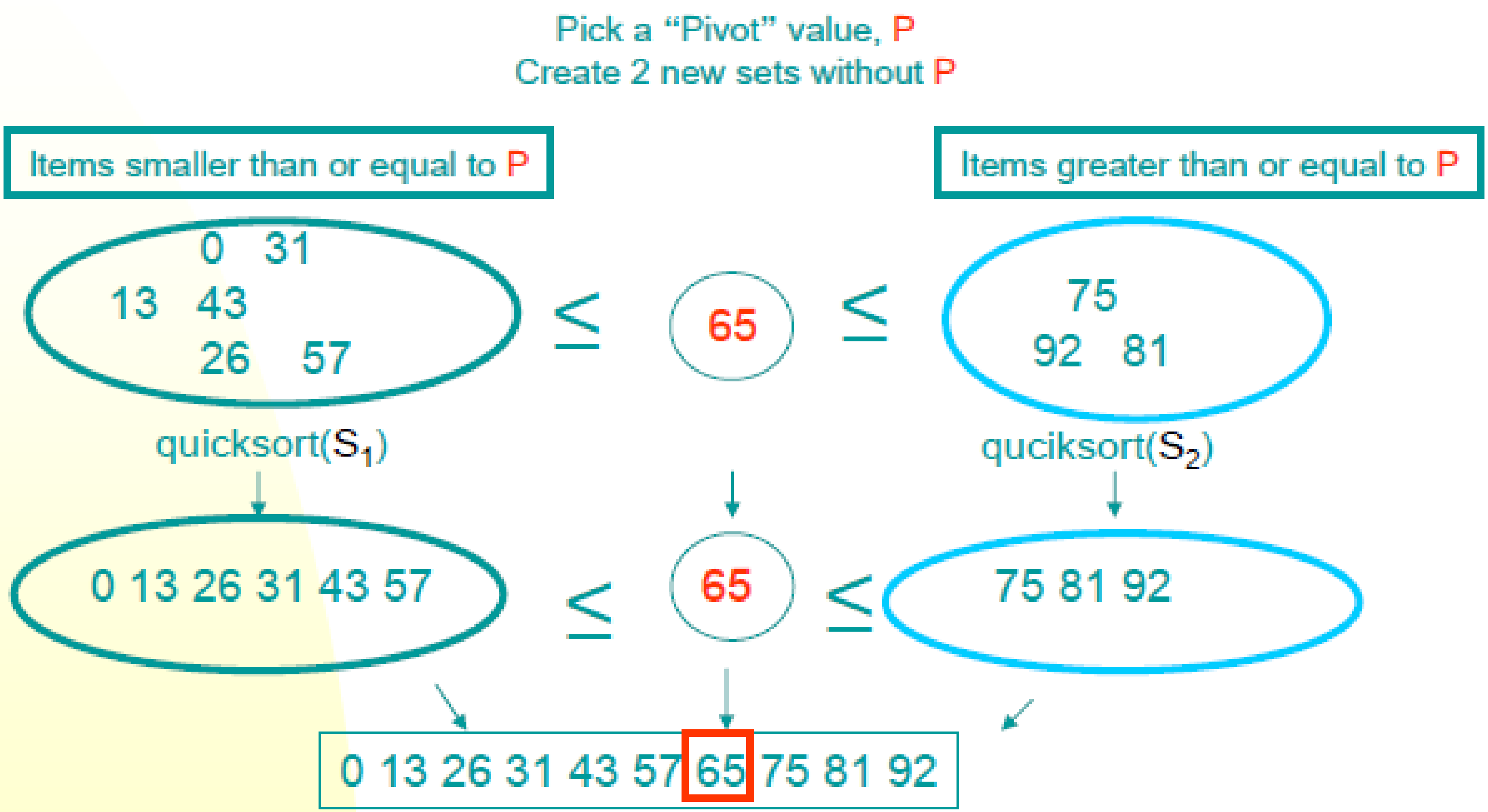
Dr. Bibhudatta Sahoo

Communication & Computing Group

Department of CSE, NIT Rourkela

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

Basic idea



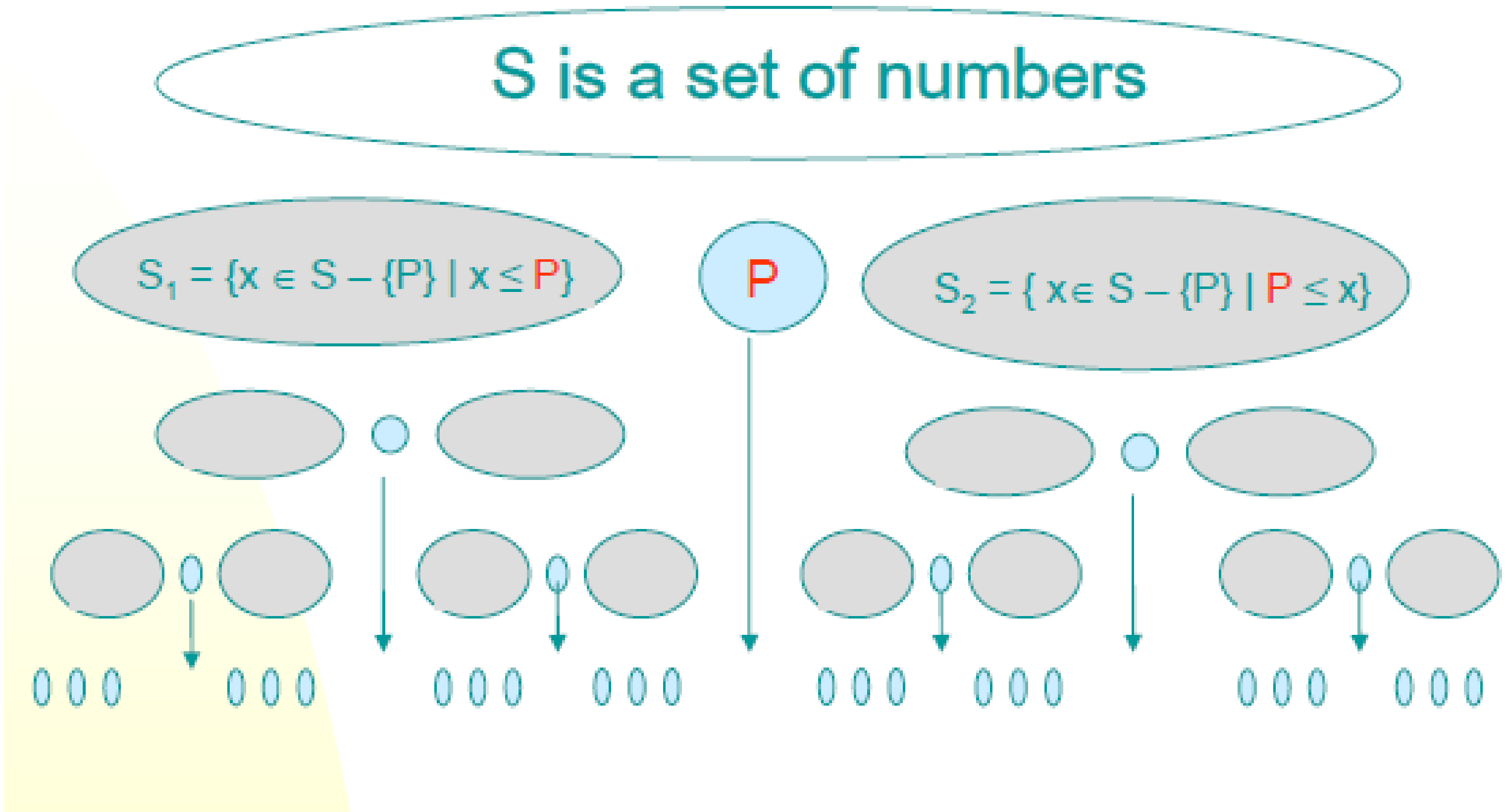
Pick a "Pivot" value, P
Create 2 new sets without P
Items smaller than or equal to P
Items greater than or equal to P
0 31
13 43
26 57
≤
65
≤
75
92 81
quicksort(S1)
quciksort(S2)
0 13 26 31 43 57
≤
65
≤
75 81 92
0 13 26 31 43 57 65 75 81 92

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.



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.

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

Quicksort

- **To sort a[left...right]:**

1. if left < right:

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

    all a[left...p-1] are less than a[p], and

    all a[p+1...right] are >= a[p]

    1.2. **Quicksort** a[left...p-1]

    1.3. **Quicksort** a[p+1...right]

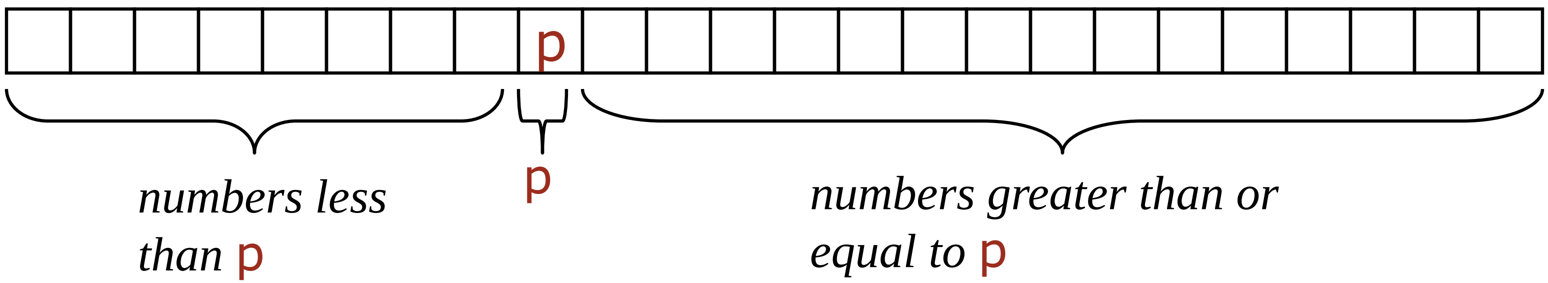
2. Terminate

Algorithm QuickSort

```
Algorithm QuickSort(p, q)
// Sorts the elements a[p], ..., 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 ≥ 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.
    }
}
```

Partitioning

- A key step in the Quicksort algorithm is partitioning the array
  - We choose some (any) number p in the array to use as a pivot
  - We partition the array into three parts:



Partitioning

- Choose an array value (say, the first) to use as the pivot
- Starting from the left end, find the first element that is greater than or equal to the pivot
- Searching backward from the right end, find the first element that is less than the pivot
- Interchange (swap) these two elements
- Repeat, searching from where we left off, until done

Partitioning

- **To partition a[left...right]:**

1. Set pivot = a[left], l = left + 1, r = right;

2. while l < r, do

2.1. while l < right & a[l] < pivot , set l = l + 1

2.2. while r > left & a[r] >= pivot , set r = r - 1

2.3. if l < r, swap a[l] and a[r]

3. Set a[left] = a[r], a[r] = pivot

4. Terminate

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

The quicksort method (in Java)

```
static void quicksort(int[] array, int left, int right) {
    if (left < right) {
        int p = partition(array, left, right);
        quicksort(array, left, p - 1);
        quicksort(array, p + 1, right);
    }
}
```

The partition method (Java)

```
static int partition(int[] a, int left, int right) {
    int p = a[left], l = left + 1, r = right;
    while (l < r) {
        while (l < right && a[l] < p) l++;
        while (r > left && a[r] >= p) r--;
        if (l < r) {
            int temp = a[l]; a[l] = a[r]; a[r] = temp;
        }
    }
    a[left] = a[r];
    a[r] = p;
    return r;
}
```

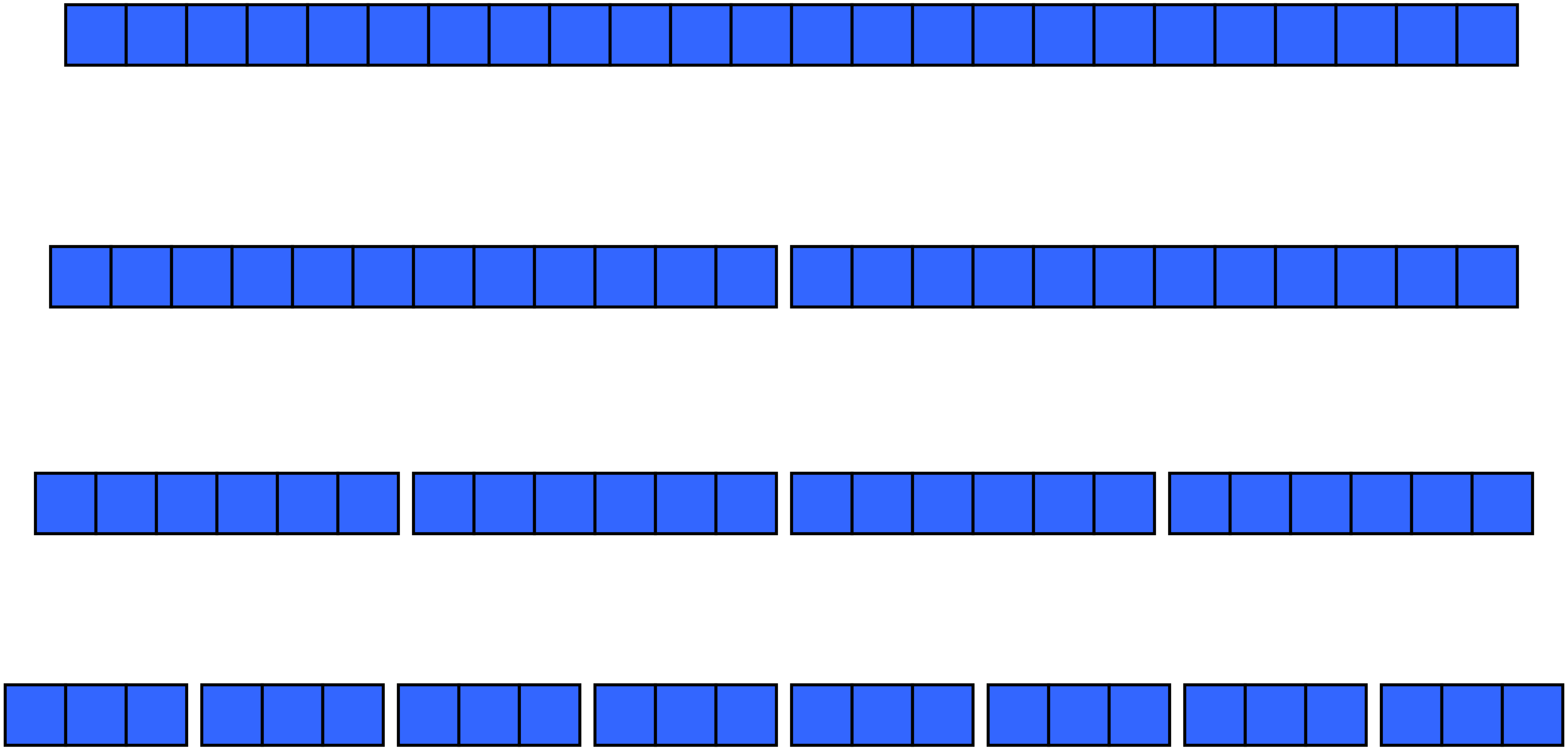
Recurrence relation for Quick Sort

```
            i := i + 1;
        }
        if (h > mid) then
            for k := j to high do
            {
                b[i] := a[k]; i := i + 1;
            }
        else
            for k := h to mid do
            {
                b[i] := a[k]; i := i + 1;
            }
        for k := low to high do a[k] := b[k];
}
```

Best case analysis of Quicksort

- Suppose each partition operation divides the array almost exactly in half
- Then the depth of the recursion in $\log_2 n$
  - Because that's how many times we can halve n
- However, there are many recursions!
  - How can we figure this out?
  - We note that
    - Each partition is linear over its subarray
    - All the partitions at one level cover the array

Partitioning at various levels



Best case analysis of Quicksort

$T(0) = T(1) = 0 \quad (base\ case)$
$T(N) = 2T(N/2) + N$

Solving the RR:

$$\frac{T(N)}{N} = \frac{N}{N} + \frac{2T(N/2)}{N}$$

Note: Divide both side of recurrence relation by N

$$\frac{T(N)}{N} = 1 + \frac{T(N/2)}{N/2}$$

$$\frac{T(N/2)}{N/2} = 1 + \frac{T(N/4)}{N/4}$$

$$\frac{T(N/4)}{N/4} = 1 + \frac{T(N/8)}{N/8}$$

...

$$\frac{T(\frac{N}{N/2})}{\frac{N}{N/2}} = 1 + \frac{T(\frac{N}{N})}{\frac{N}{N}} = 1 + \frac{T(1)}{1}$$

Best case analysis of Quicksort

$$\frac{T(\frac{N}{N/2})}{\frac{N}{N/2}} = 1 + \frac{T(\frac{N}{N})}{\frac{N}{N}} = 1 + \frac{T(1)}{1}$$

same as

$$\frac{T(2)}{2} = 1 + \frac{T(1)}{1}$$

Note: T(1) = 0

Hence,

$$\frac{T(N)}{N} = 1 + 1 + 1 + \ldots 1$$

Note: log(N) terms

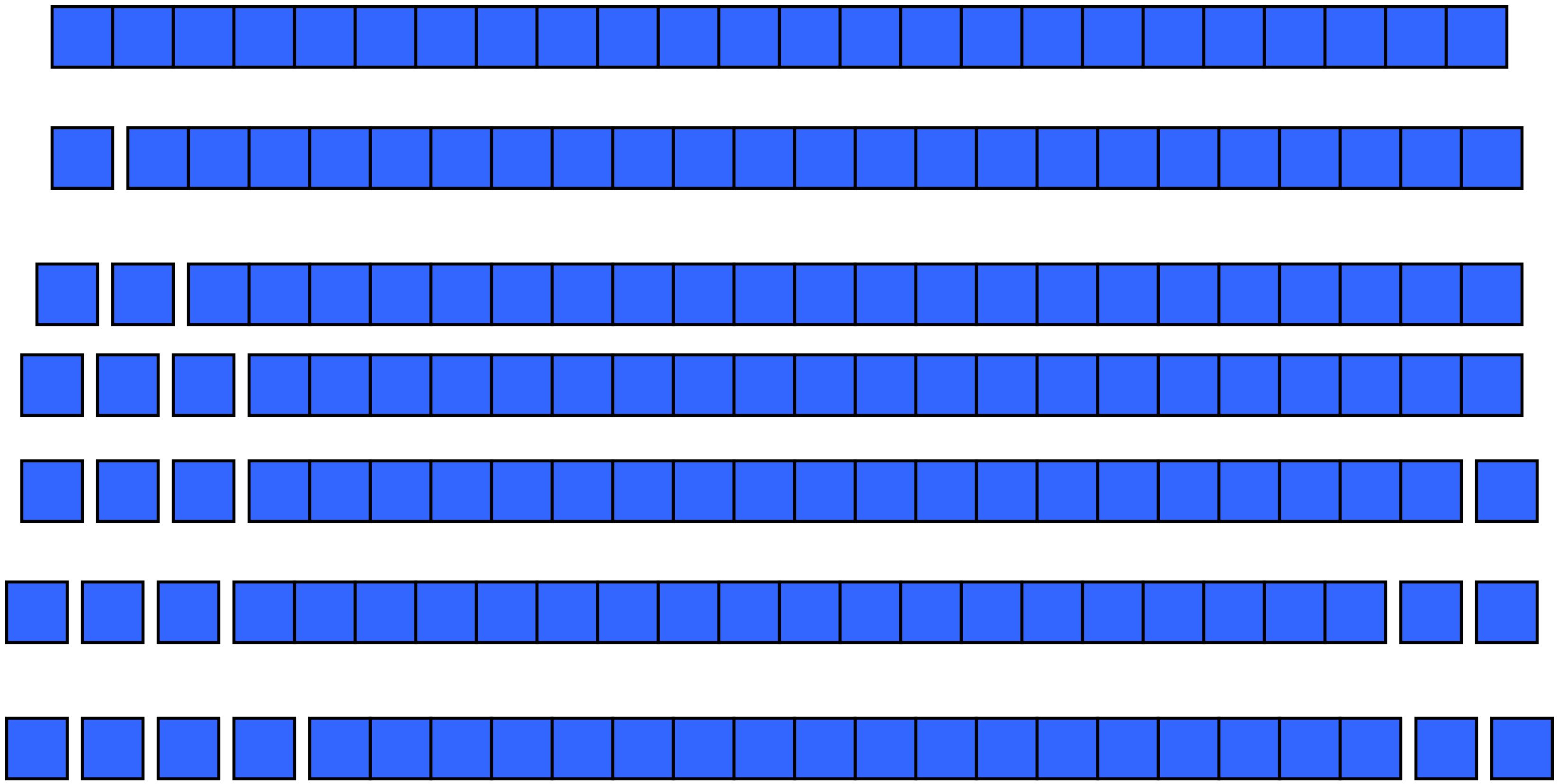
$$\frac{T(N)}{N} = \log N$$

$T(N) = N \log N$ *which is O(N logN)*

Worst case analysis of Quicksort

- In the worst case, partitioning always divides the size n array into these three parts:
  - A length one part, containing the pivot itself
  - A length zero part, and
  - A length n-1 part, containing everything else
- We don't recur on the zero-length part
- Recurring on the length n-1 part requires (in the worst case) recurring to depth n-1

Worst case partitioning



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 $\mathbf{T(N) = O(N^2)}$

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 $N-1$
- $T(N) = 2/N (\sum T(j)) + cN$
- Multiply by N
- $NT(N) = 2(\sum T(j)) + cN*N$
- To remove the summation, we rewrite the equation for N-1:
- $(N-1)T(N-1) = 2(\sum T(j)) + c(N-1)^2$, $j = 0$ 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:

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

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)**

How to choose the pivot P?

Advantages:

- One of the fastest algorithms on average.
- Does not need additional memory (the sorting takes place in the array - this is called **in-place** processing). Compare with mergesort: mergesort needs additional memory for merging.

Disadvantages:

- The worst-case complexity is $O(N^2)$

Applications:

- 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.

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.

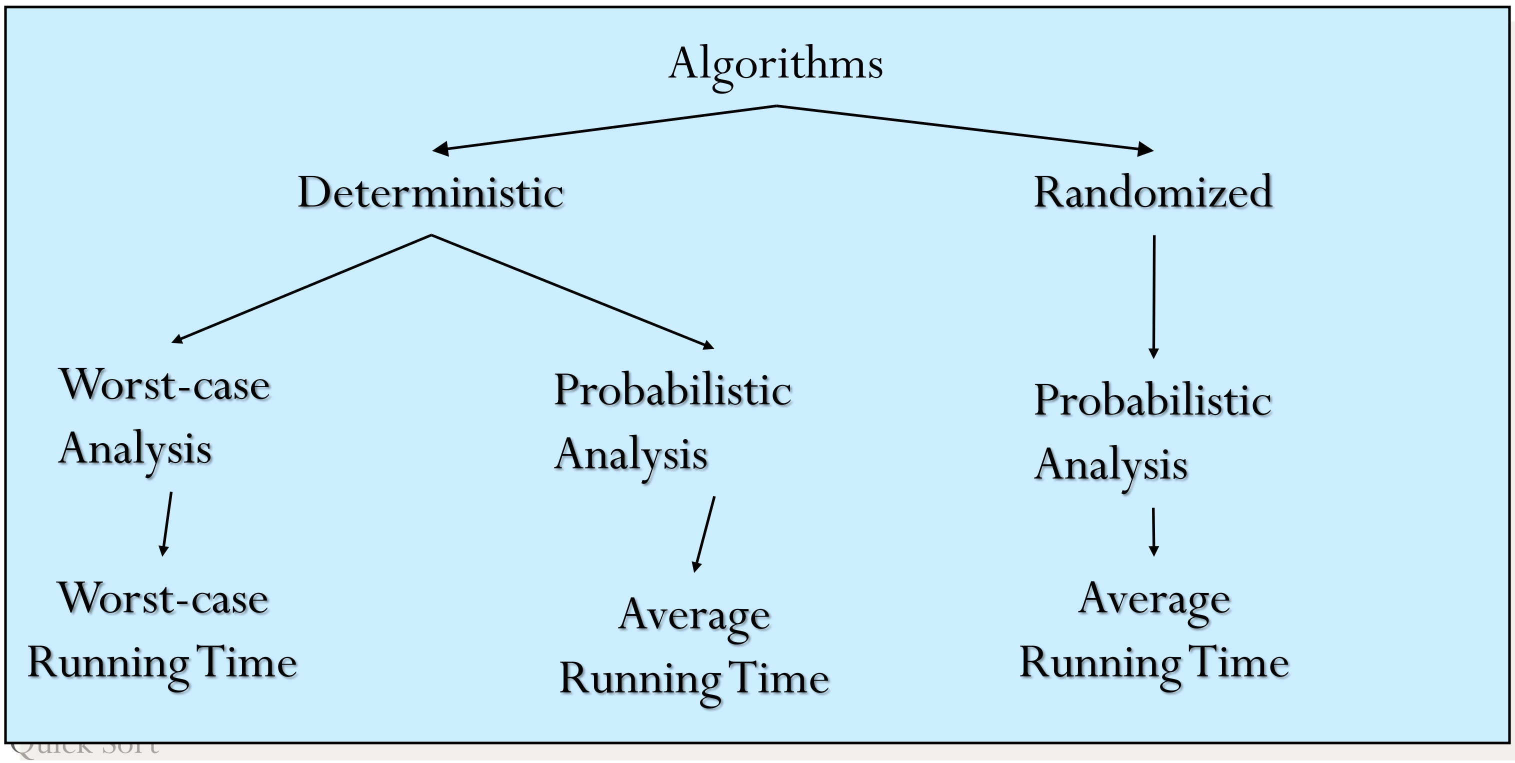
Summary of Sorting Algorithms

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

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.**



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: $2 \log_{4/3} n$

Thanks for Your Attention!

Exercises

Suggested Excercise

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?