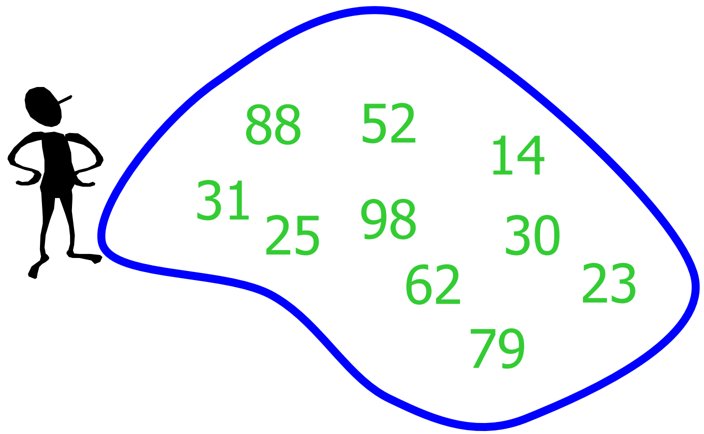
Algorithms Analysis

Quicksort

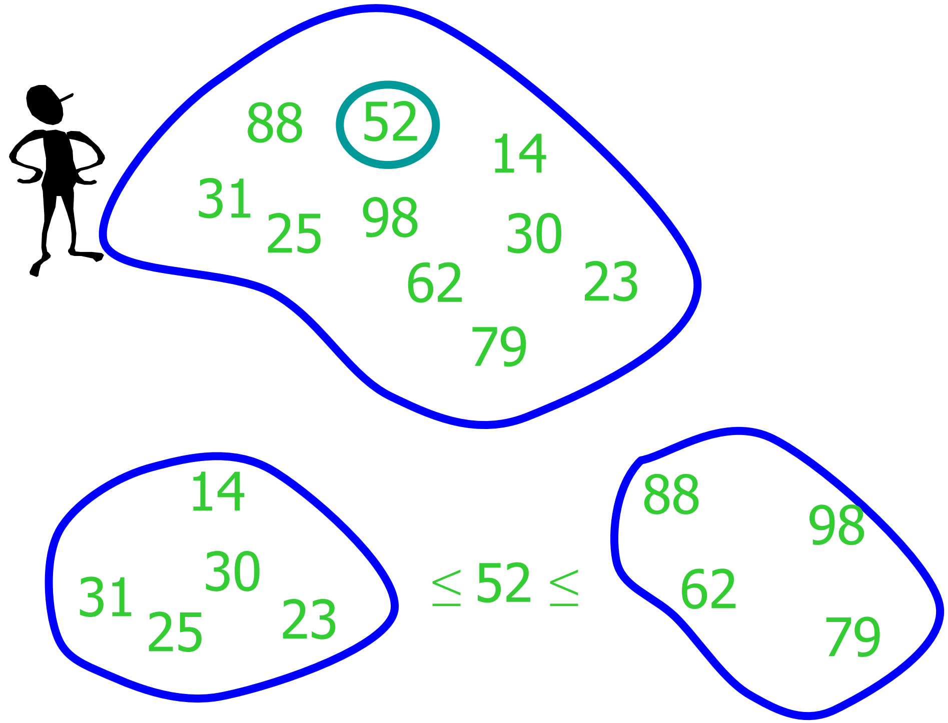
Quick Sort



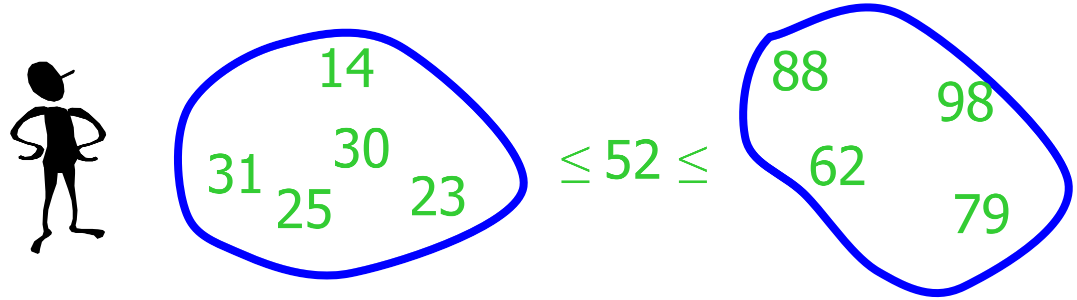
Divide and Conquer

Quick Sort

Partition set into two using randomly chosen pivot



Quick Sort



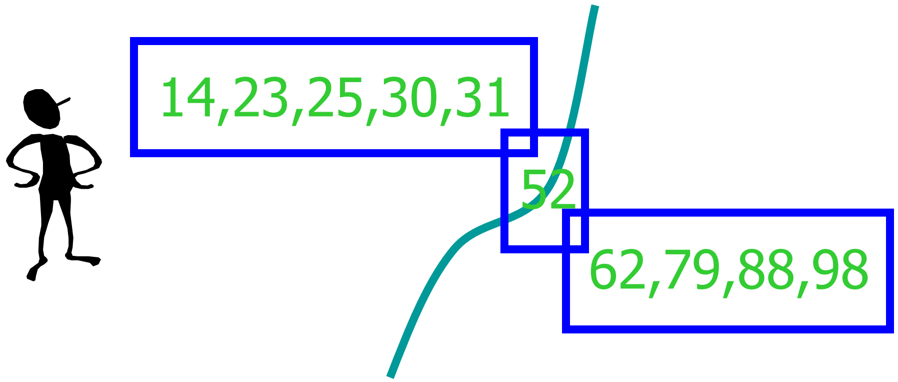
sort the first half.

14,23,25,30,31

sort the second half.

62,79,98,88

Quick Sort



Glue pieces together.

14,23,25,30,31,52,62,79,88,98

Quicksort

- Quicksort pros [advantage]:
  - Sorts in place
  - Sorts $O(n \lg n)$ in the average case
  - Very efficient in practice , it's quick

- Quicksort cons [disadvantage]:
  - Sorts $O(n^2)$ in the worst case
  - And the worst case doesn't happen often … sorted

Quicksort

- Another divide-and-conquer algorithm:
- *Divide*: $A[p \ldots r]$ is partitioned (rearranged) into two nonempty subarrays $A[p \ldots q\text{-}1]$ and $A[q+1 \ldots r]$ s.t. each element of $A[p \ldots q\text{-}1]$ is less than or equal to each element of $A[q+1 \ldots r]$. Index q is computed here, called **pivot**.
- *Conquer*: two subarrays are sorted by recursive calls to quicksort.
- *Combine*: unlike merge sort, no work needed since the subarrays are sorted in place already.

Quicksort

- The basic algorithm to sort an array A consists of the following four easy steps:
  - If the number of elements in A is 0 or 1, then return
  - Pick any element $\boldsymbol{v}$ in A. This is called the ***pivot***
  - Partition A-{$\boldsymbol{v}$} (the remaining elements in A) into two disjoint groups:
    - $A_1 = \{\boldsymbol{x} \in A\text{-}\{\boldsymbol{v}\} \mid \boldsymbol{x} \leq \boldsymbol{v}\}$, and
    - $A_2 = \{\boldsymbol{x} \in A\text{-}\{\boldsymbol{v}\} \mid \boldsymbol{x} \geq \boldsymbol{v}\}$
  - return
    - { quicksort(A_1) followed by $\boldsymbol{v}$ followed by quicksort(A_2)}

Quicksort

- Small instance has $n \leq 1$
  - Every small instance is a sorted instance

- To sort a large instance:
  - select a pivot element from out of the ***n*** elements

- Partition the ***n*** elements into 3 groups left, middle and right
  - The middle group contains only the pivot element
  - All elements in the left group are ≤ pivot
  - All elements in the right group are ≥ pivot

- Sort left and right groups recursively

- Answer is sorted left group, followed by middle group followed by sorted right group

Quicksort Code

```
P: first element
r: last element
Quicksort(A, p, r)
{
    if (p < r)
    {
        q = Partition(A, p, r)
        Quicksort(A, p  , q-1)
        Quicksort(A, q+1, r)
    }
}
```

- Initial call is **Quicksort**(A, 1, n), where n in the length of A

Partition

- Clearly, all the action takes place in the **`partition()`** function
  - Rearranges the subarray in place
  - End result:
    - Two subarrays
    - All values in first subarray ≤ all values in second
  - Returns the **index** of the "pivot" element separating the two subarrays

Partition Code

```
Partition(A, p, r)
{
    x = A[r]    // x is pivot
    i = p - 1
    for j = p to r – 1
    {
        do if A[j] <= x
         then
        {
          i = i + 1
          exchange A[i] ↔ A[j]
        }
    }
    exchange A[i+1] ↔ A[r]
    return i+1
}
```

partition() runs in O(n) time

i is the pointer of left side where I will keep the value/ where last small value ended

j is my movement pointer during iteration

Partition Example

$$A = \{2, 8, 7, 1, 3, 5, 6, 4\}$$

i	*p j*							*r*
	2	8	7	1	3	5	6	4

p i	*j*						*r*
2	8	7	1	3	5	6	4

p i		*j*					*r*
2	8	7	1	3	5	6	4

p i			*j*				*r*
2	8	7	1	3	5	6	4

p	*i*			*j*			*r*
2	1	7	8	3	5	6	4

p		*i*			*j*		*r*
2	1	3	8	7	5	6	4

p		*i*				*j*	*r*
2	1	3	8	7	5	6	4

p		*i*					*r*
2	1	3	8	7	5	6	4

p		*i*					*r*
2	1	3	4	7	5	6	8

r is my pivoting index

Partition Example Explanation

- Red shaded elements are in the first partition with values $\leq x$ (pivot)
- Gray shaded elements are in the second partition with values $\geq x$ (pivot)
- The unshaded elements have no yet been put in one of the first two partitions
- The final white element is the pivot

Choice Of Pivot

Three ways to choose the pivot:

- Pivot is **rightmost** element in list that is to be sorted
  - When sorting *A*[6:20], use *A*[20] as the pivot
  - Textbook implementation does this

- **Randomly** select one of the elements to be sorted as the pivot
  - When sorting *A*[6:20], generate a random number *r* in the range [6, 20]
  - Use *A*[*r*] as the pivot

Choice Of Pivot

- **Median-of-Three** rule - from the leftmost, middle, and rightmost elements of the list to be sorted, select the one with median key as the pivot
  - When sorting *A*[6:20], examine *A*[6], *A*[13] ((6+20)/2), and *A*[20]
  - Select the element with median (i.e., middle) key
  - If *A*[6].key = 30, *A*[13].key = 2, and *A*[20].key = 10, *A*[20] becomes the pivot
  - If *A*[6].key = 3, *A*[13].key = 2, and *A*[20].key = 10, *A*[6] becomes the pivot

Worst Case Partitioning

- The running time of quicksort depends on whether the partitioning is **balanced** or not.
- $\Theta(n)$ time to partition an array of n elements
- Let $T(n)$ be the time needed to sort n elements
- $T(0) = T(1) = c$, where c is a constant
- When $n > 1$,
  - $T(n) = T(|\text{left}|) + T(|\text{right}|) + \Theta(n)$
- $T(n)$ is maximum (worst-case) when <u>either |left| = 0 or |right| = 0 following each partitioning</u>

Worst Case Partitioning

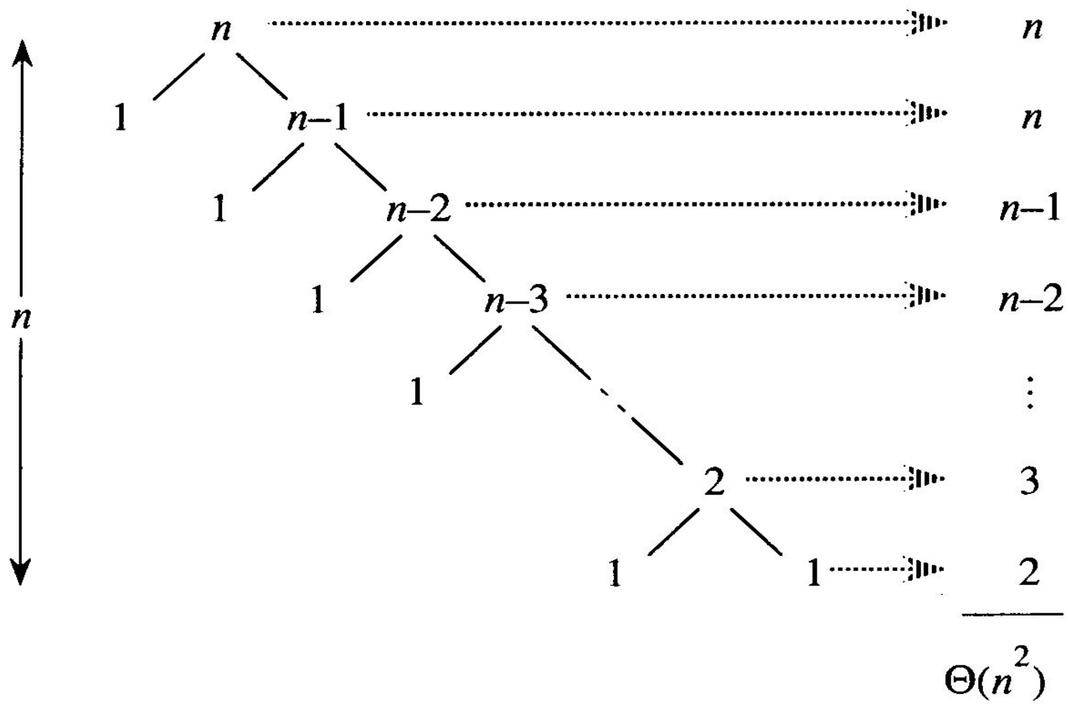


Figure 8.2 A recursion tree for QUICKSORT in which the PARTITION procedure always puts only a single element on one side of the partition (the worst case). The resulting running time is $\Theta(n^2)$.

Worst Case Partitioning

- **Worst-Case** Performance (unbalanced):
  - $T(n) = T(1) + T(n\text{-}1) + \Theta(n)$
    - partitioning takes $\Theta(n)$

  $= [2 + 3 + 4 + \ldots + n\text{-}1 + n\ ] + n =$

  $= [\sum_{k = 2 \text{ to } n} k\ ] + n = \Theta(n^2)$

$$\sum_{k=1}^{n} k = 1 + 2 + \ldots + n = n(n+1)/2 = \Theta(n^2)$$

- This occurs when
  - the input is **completely sorted**
- or when
  - the pivot is always the **smallest (largest)** element

Best Case Partition

- When the partitioning procedure produces two regions of size ***n*/2**, we get the a **balanced** partition with **best case** performance:
  - $T(n) = 2T(n/2) + \Theta(n) = \Theta(n \lg n)$

- **Average** complexity is also $\Theta(n \lg n)$

Best Case Partitioning

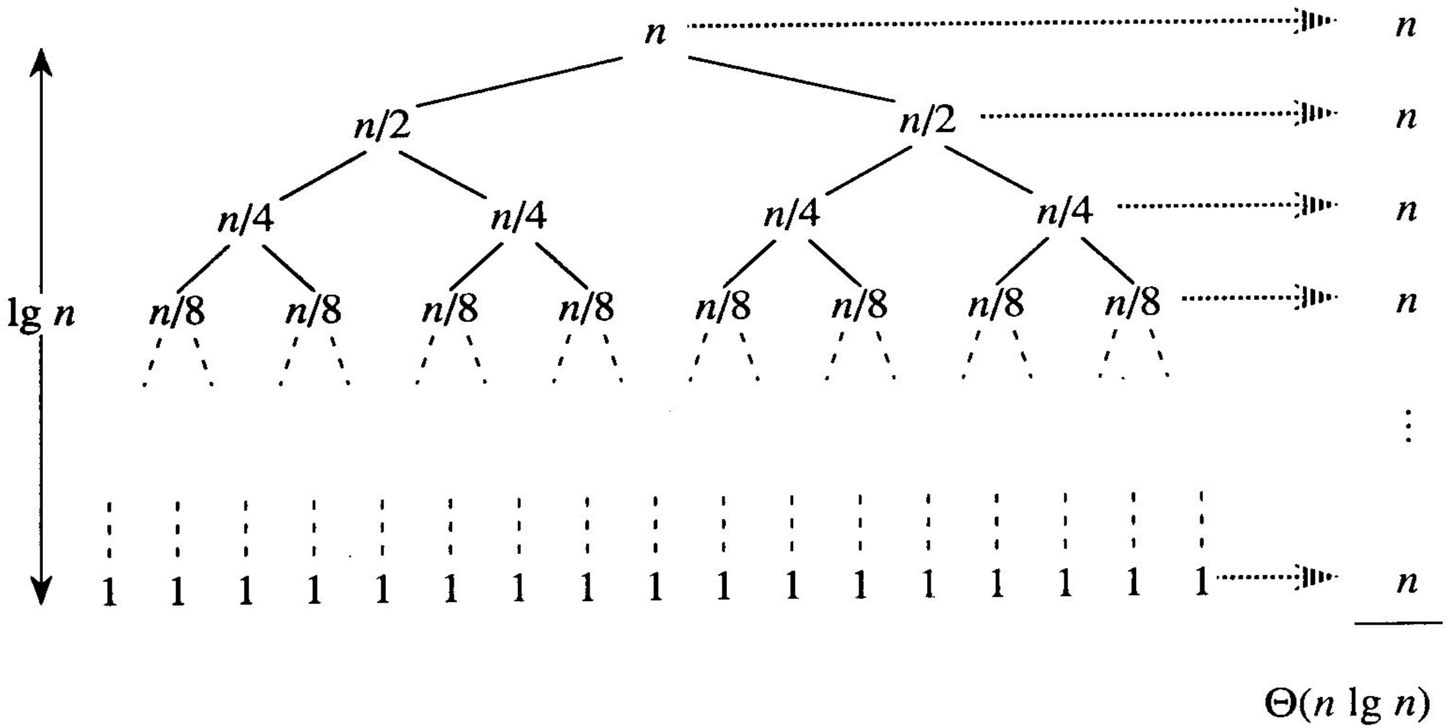


Figure 8.3 A recursion tree for QUICKSORT in which PARTITION always balances the two sides of the partition equally (the best case). The resulting running time is $\Theta(n \lg n)$.

Average Case

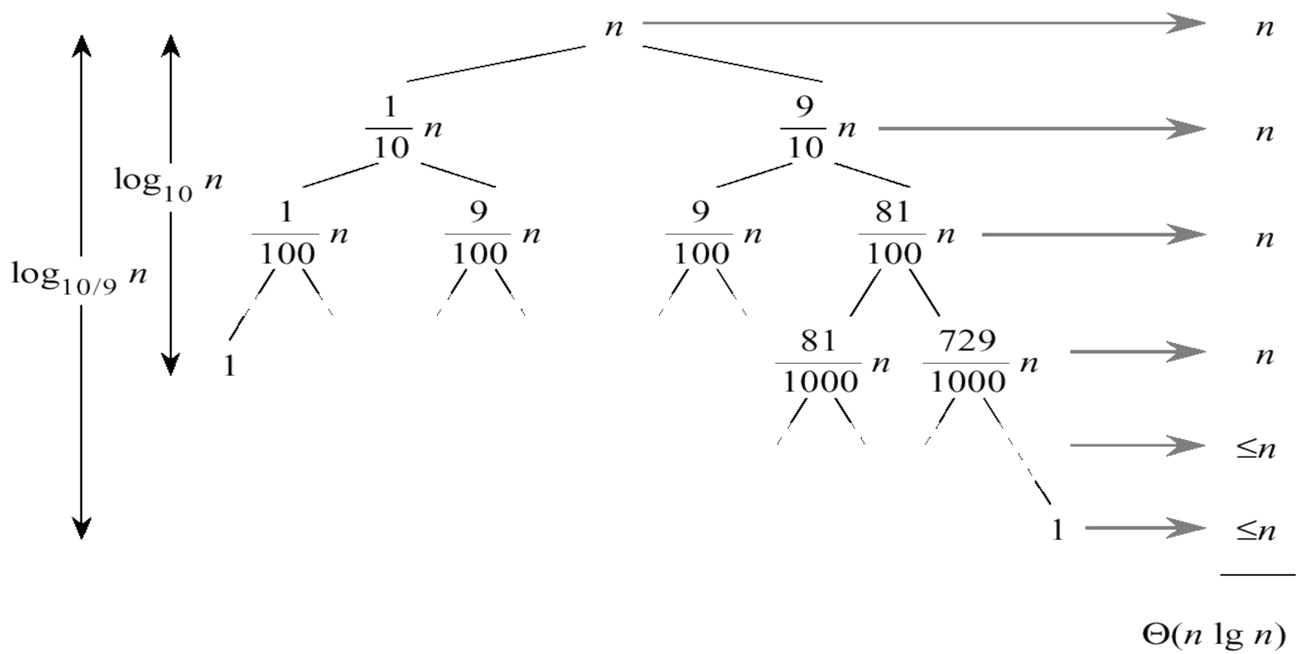
- Assuming random input, average-case running time is much closer to $\Theta(n \lg n)$ than $\Theta(n^2)$
- First, a more intuitive explanation/example:
  - Suppose that **partition()** always produces a 9-to-1 **proportional** split. This looks quite unbalanced!
  - The recurrence is thus:

    $T(n) = T(9n/10) + T(n/10) + \Theta(n) = \Theta(n \lg n)$?

[Using recursion tree method to solve]

Average Case

$$T(n) = T(n/10) + T(9n/10) + \Theta(n) = \Theta(n \log n)!$$



$\log_2 n = \log_{10} n / \log_{10} 2$

Average Case

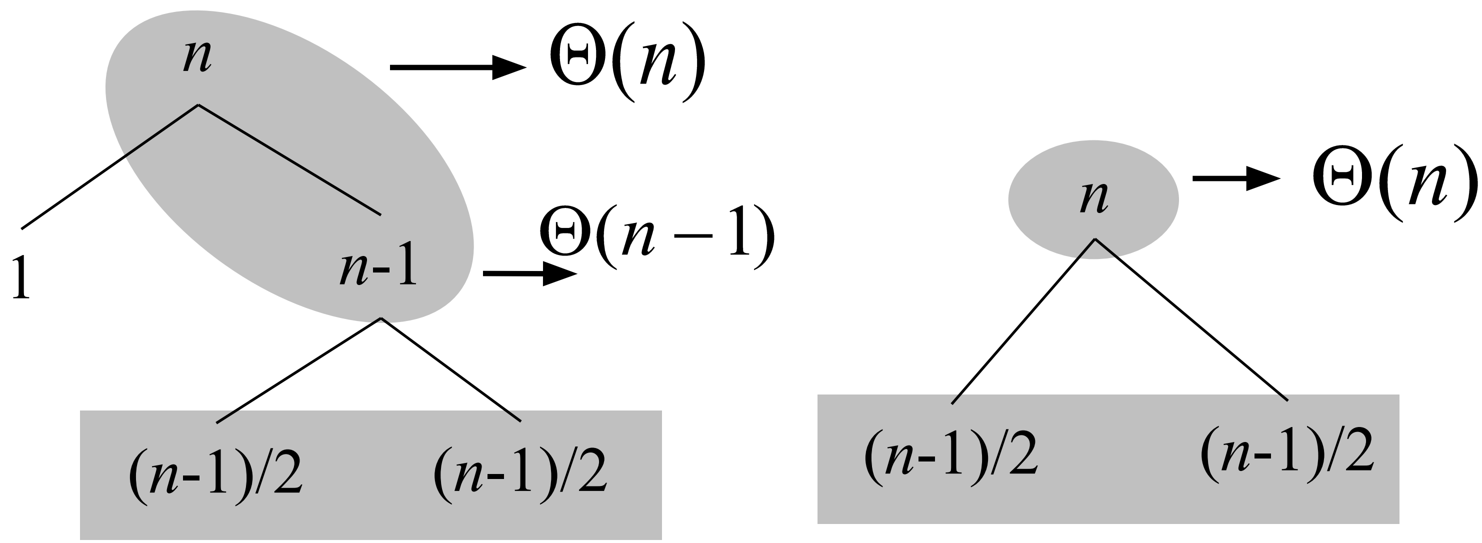
- Every level of the tree has cost cn, until a boundary condition is reached at depth $\log_{10} n = \Theta(\lg n)$, and then the levels have cost at most cn.

- The recursion terminates at depth $\log_{10/9} n = \Theta(\lg n)$.

- The total cost of quicksort is therefore $O(n \lg n)$.

Average Case

- What happens if we bad-split root node, then good-split the resulting size (n-1) node?
  - We end up with three subarrays, size
    - 1, (n-1)/2, (n-1)/2
  - Combined cost of splits = n + n-1 = 2n -1 = $\Theta(n)$



Intuition for the Average Case

- Suppose, we alternate lucky and unlucky cases to get an average behavior

$$L(n) = 2U(n/2) + \Theta(n) \quad \text{lucky}$$
$$U(n) = L(n-1) + \Theta(n) \quad \text{unlucky}$$

we consequently get

$$\begin{aligned} L(n) &= 2(L(n/2-1) + \Theta(n/2)) + \Theta(n) \\ &= 2L(n/2-1) + \Theta(n) \\ &= \Theta(n \log n) \end{aligned}$$

The combination of good and bad splits would result in ***T*(*n*) = O (*n* lg *n*)**, but with slightly **larger constant** hidden by the O-notation.

Randomized Quicksort

- An algorithm is *randomized* if its behavior is determined not only by the input but also by values produced by a *random-number generator*.
- Exchange $A[r]$ with an element chosen at random from $A[p \ldots r]$ in **Partition**.
- This ensures that the pivot element is equally likely to be any of input elements.
- We can sometimes add randomization to an algorithm in order to <u>obtain good average-case</u> performance over all inputs.

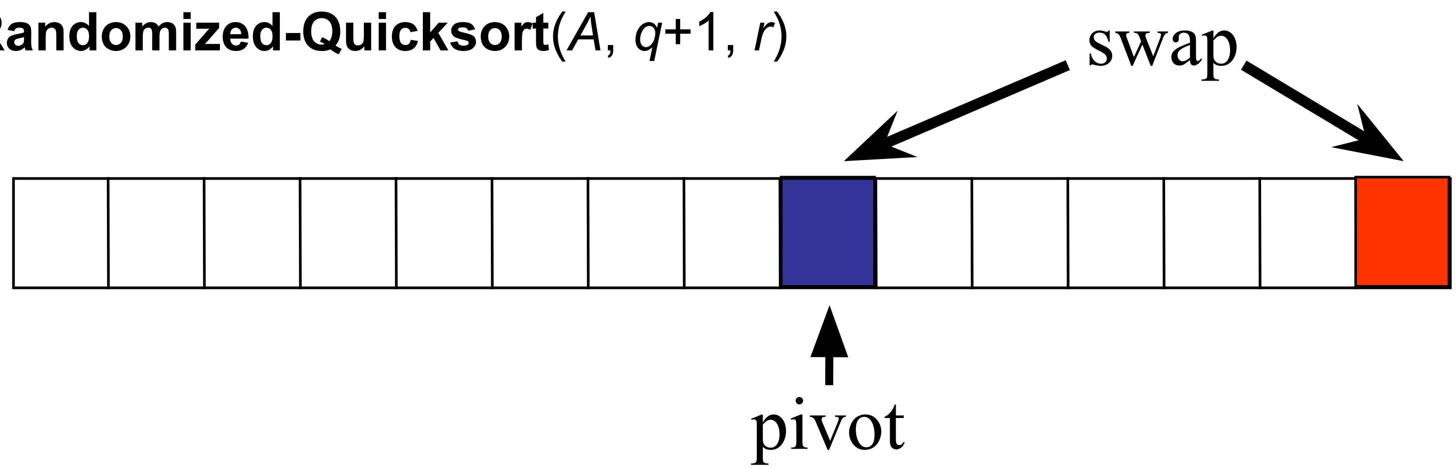
Randomized Quicksort

Randomized-Partition(*A, p, r*)

1. $i \leftarrow Random(p, r)$
2. exchange $A[r] \leftrightarrow A[i]$
3. **return Partition**(A, p, r)

Randomized-Quicksort(*A, p, r*)

1. **if** $p < r$
2.   **then** $q \leftarrow$ **Randomized-Partition**(A, p, r)
3.   **Randomized-Quicksort**$(A, p, q\text{-}1)$
4.   **Randomized-Quicksort**$(A, q\text{+}1, r)$



Review: Analyzing Quicksort

- *What will be the worst case for the algorithm?*
  - Partition is always unbalanced
- *What will be the best case for the algorithm?*
  - Partition is balanced

Summary: Quicksort

- In worst-case, efficiency is $\Theta(n^2)$
  - But easy to avoid the worst-case

- On average, efficiency is $\Theta(n \lg n)$

- Better space-complexity than mergesort.

- In practice, runs fast and widely used