Design and Analysis of Algorithms

Divide and Conquer

National University of Computer and Emerging Sciences, Islamabad

Merge Sort (Divide and Conquer)

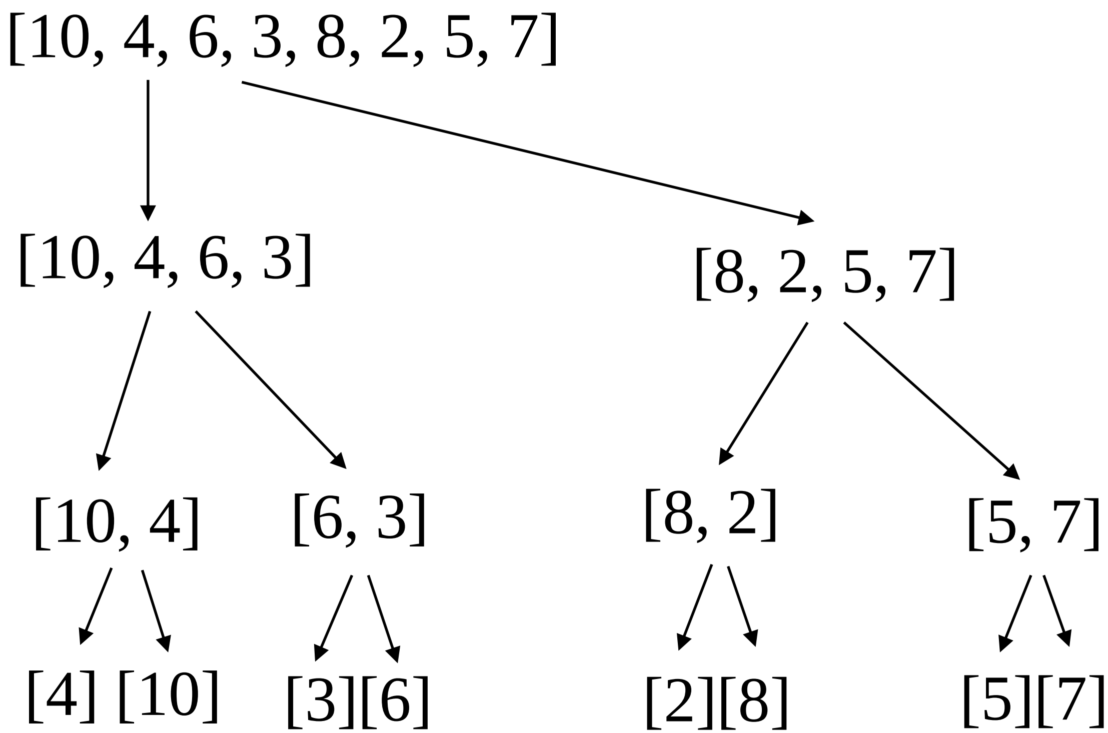
MERGE-SORT $A[1 \,..\, n]$

1. If $n = 1$, done.
2. Recursively sort $A[\,1 \,..\, \lceil n/2 \rceil\,]$ and $A[\,\lceil n/2 \rceil + 1 \,..\, n\,]$.
3. "*Merge*" the 2 sorted lists.

Key subroutine: MERGE

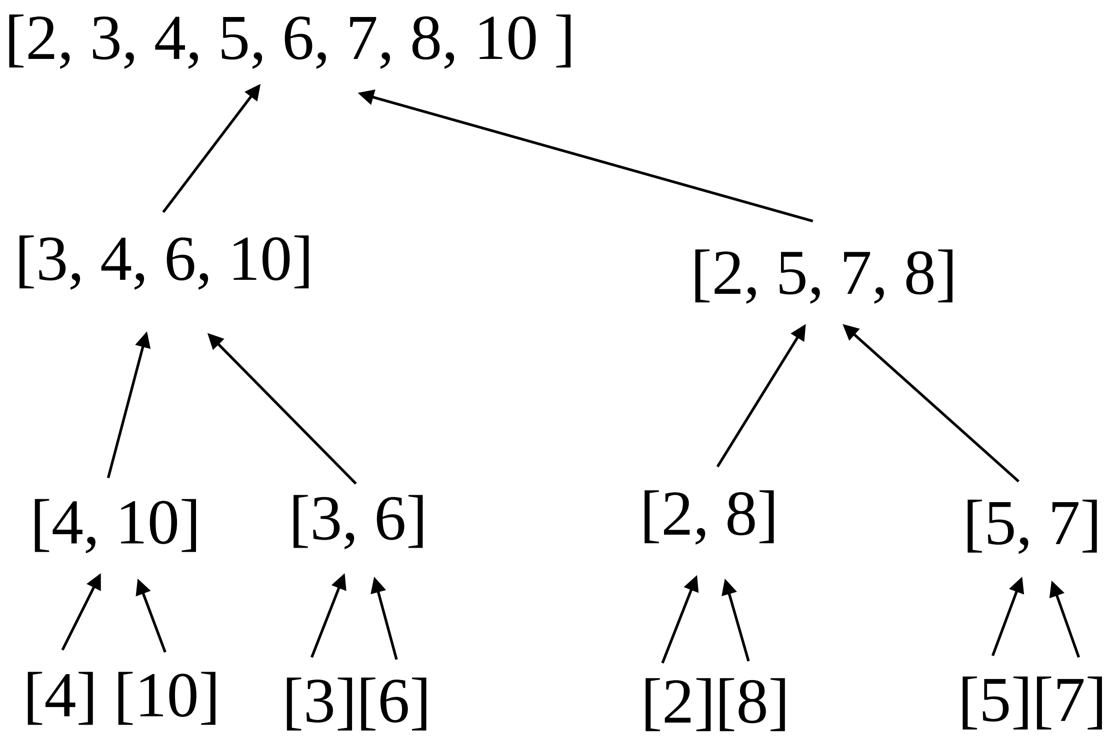
Example

- **Partition into lists of size n/2**



Example Cont'd

- **Merge**



Analysis of Merge Sort

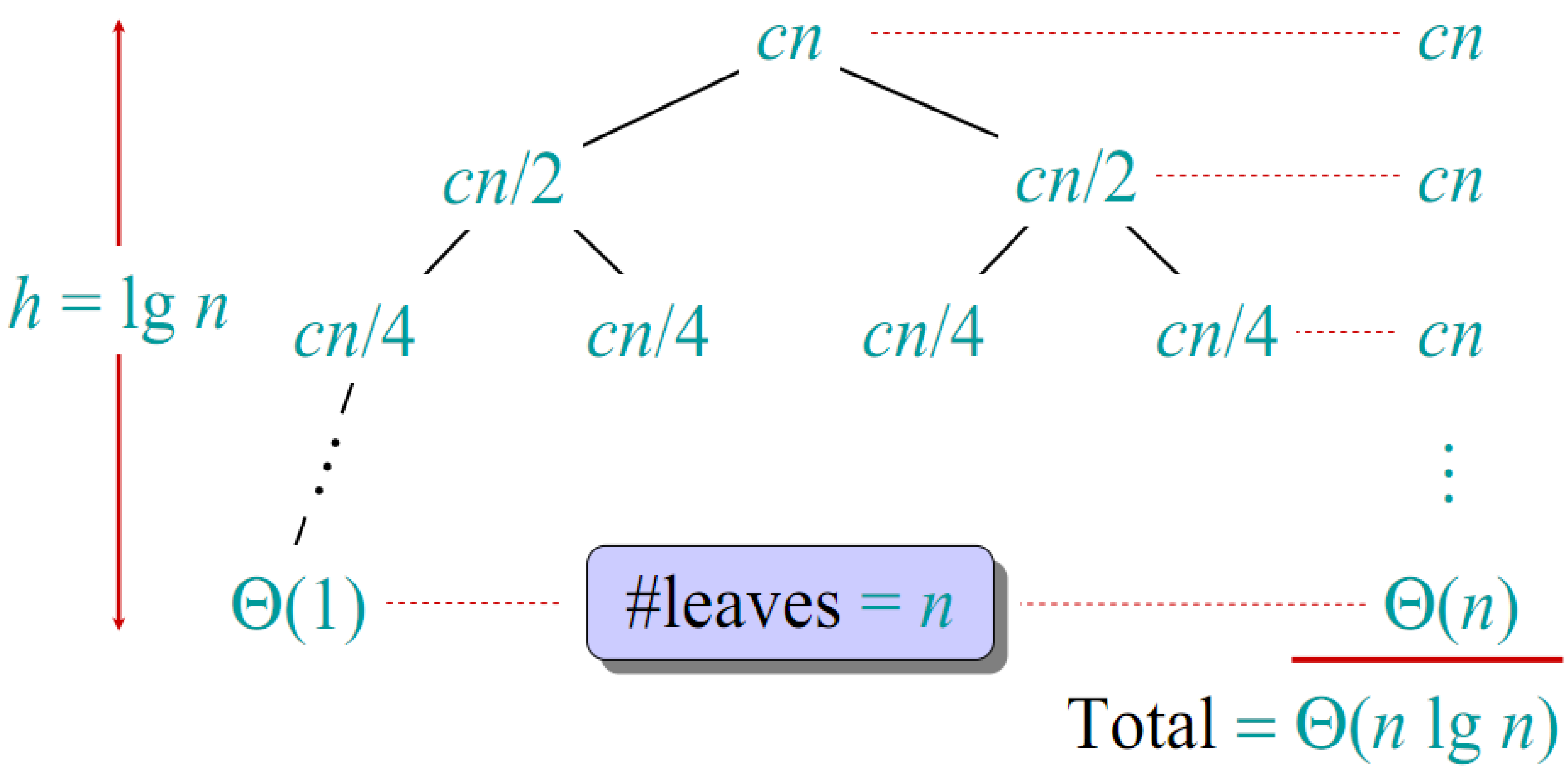
	$T(n)$	MERGE-SORT $A[1\,..\,n]$
	$\Theta(1)$	1. If $n = 1$, done.
	$2T(n/2)$	2. Recursively sort $A[\,1\,..\,\lceil n/2\rceil\,]$ and $A[\,\lceil n/2\rceil+1\,..\,n\,]$.
Abuse		
	$\Theta(n)$	3. ***"Merge"*** the 2 sorted lists

Sloppiness: Should be $T(\lceil n/2\rceil) + T(\lfloor n/2\rfloor)$, but it turns out not to matter asymptotically.

$$T(n) = \begin{cases} \Theta(1) \text{ if } n = 1; \\ 2T(n/2) + \Theta(n) \text{ if } n > 1. \end{cases}$$

Visual Representation of the Recurrence for Merge Sort

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



Time Complexity (Using Master Theorem)

- **Recurrence Relation**

  T(n)=2T(n/2) + n

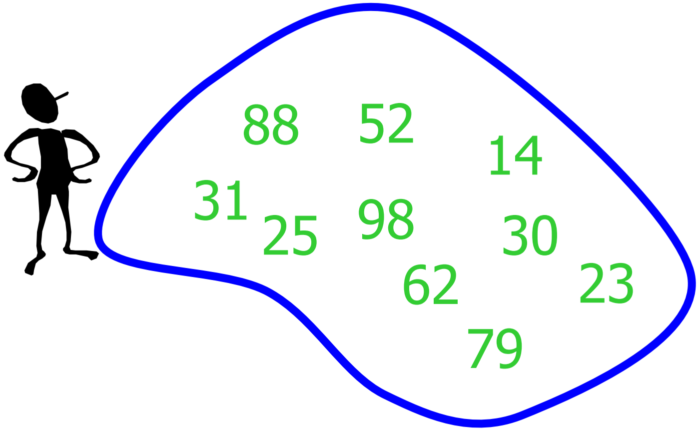
Using Master Theorem applying case 2:

$$\Theta\left(n^{\log_b a}\log n\right)$$

So time complexity is O(nlogn)

- $\Theta(n\lg n)$ grows more slowly than $\Theta(n^2)$
- Therefore, merge sort asymptotically beats insertion sort in the worst case.
- In practice, merge sort beats insertion sort for n >=3

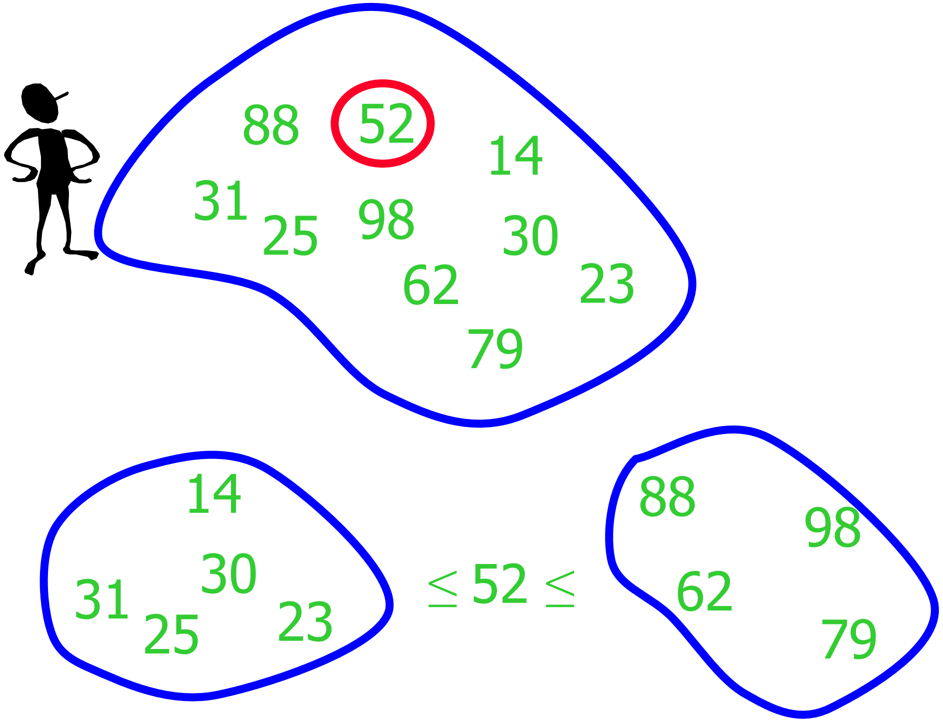
Quick Sort



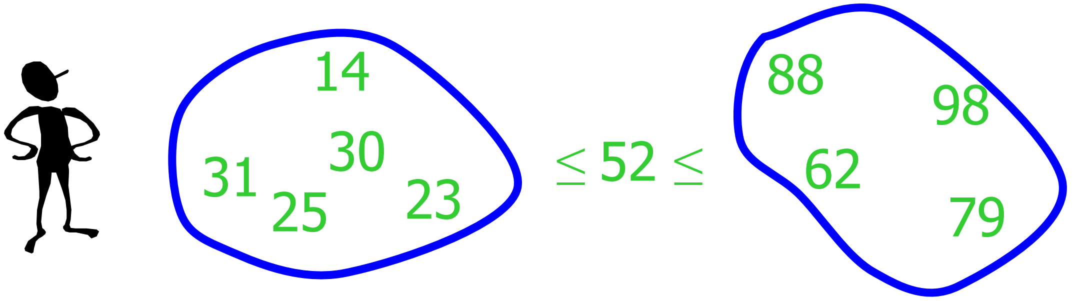
Divide and Conquer

Quick Sort

Partition set into two using randomly chosen pivot



Quick Sort



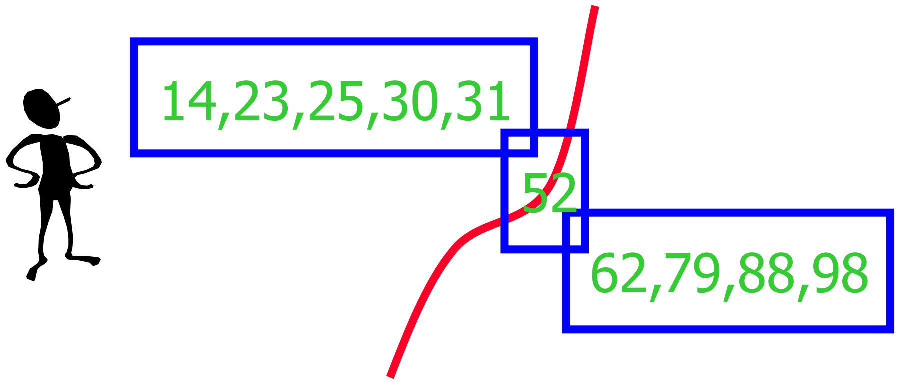
sort the first half.

sort the second half.

14,23,25,30,31

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**
  - **And the worst case doesn't happen often … sorted**

- **Quicksort cons [disadvantage]:**
  - **Sorts $O(n^2)$ in the worst case**

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 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)
    }
}
```

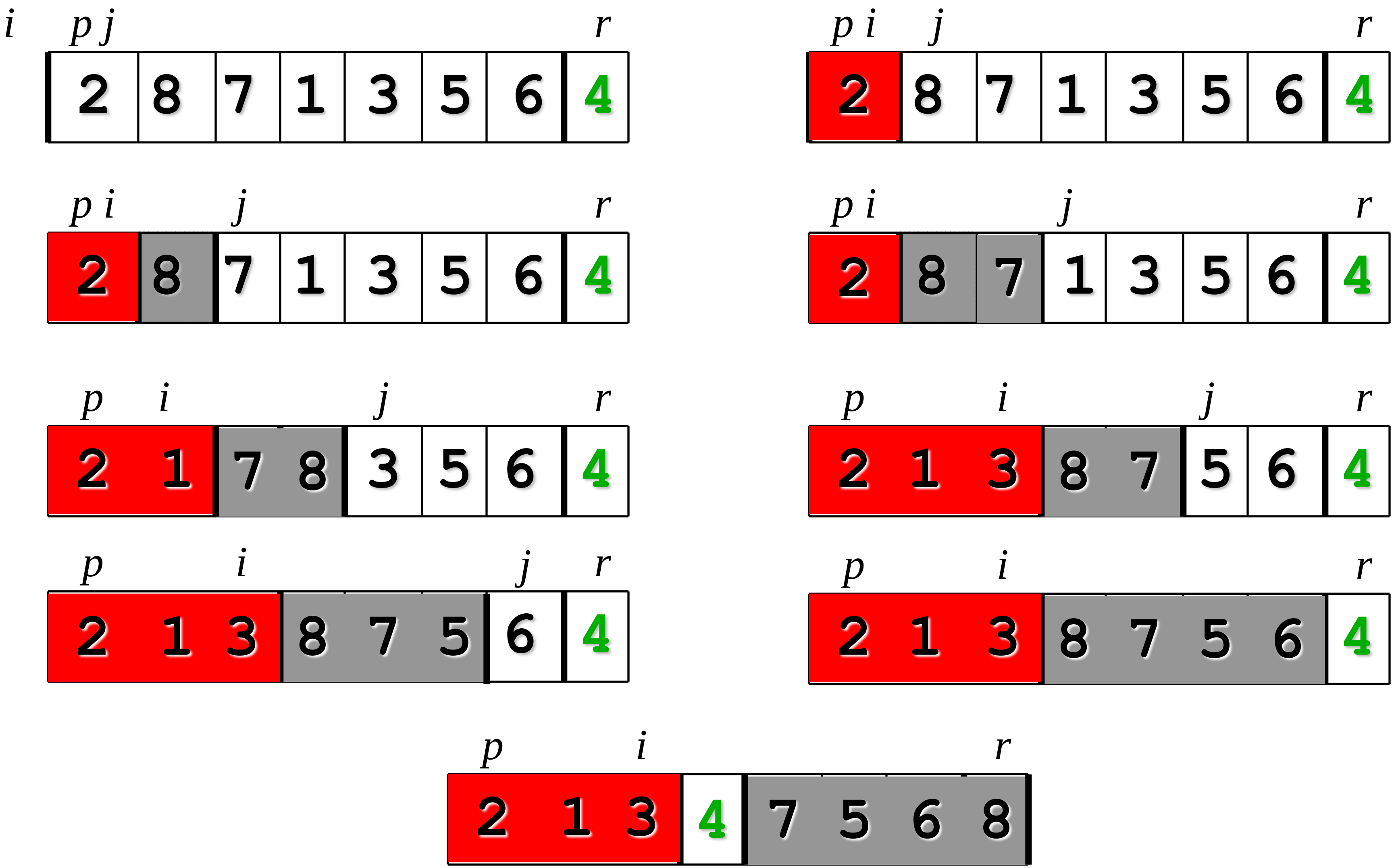
- **Initial call is Quicksort(*A*, 1, *n*), where *n* in the length of *A***

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 Example

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



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**

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

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

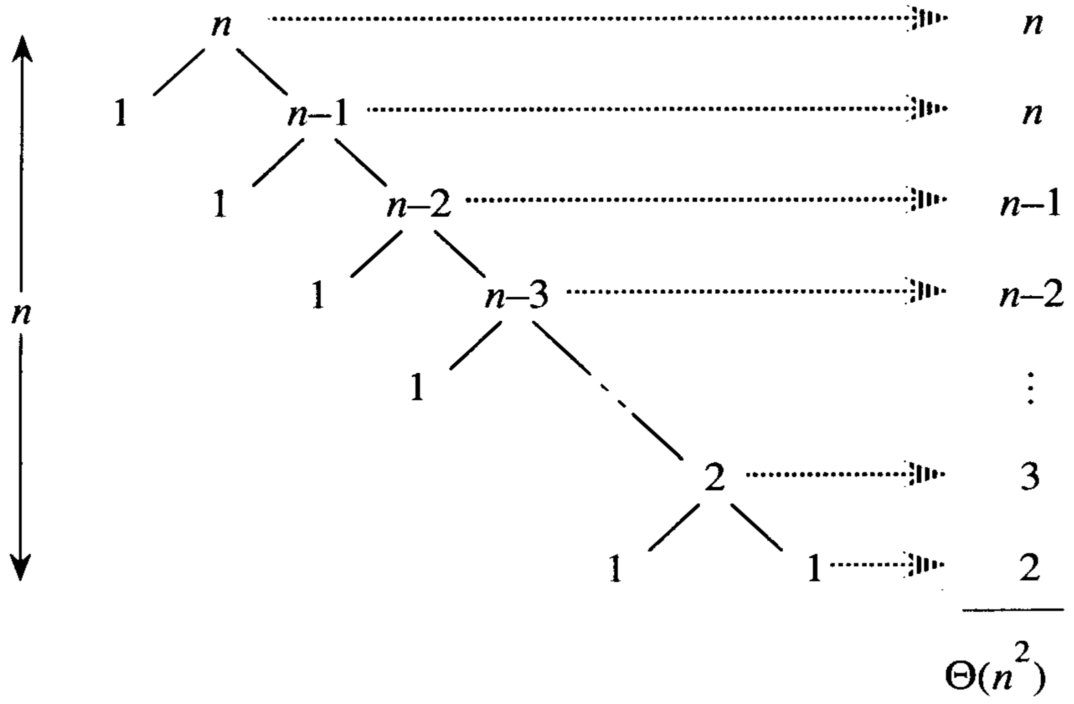


Figure 8.2 A recursion tree for QUICKSORT in which the PARTITION procedure always puts only a single element on one side of the partition (the worst case). The resulting running time is $\Theta(n^2)$.

Worst Case Partitioning

- **Worst-Case Performance (unbalanced):**
  - **$T(n) = T(0) + T(n\text{-}1) + \Theta(n)$**
    - **partitioning takes $\Theta(n)$**

  $= \Theta(n^2)$

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

- **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

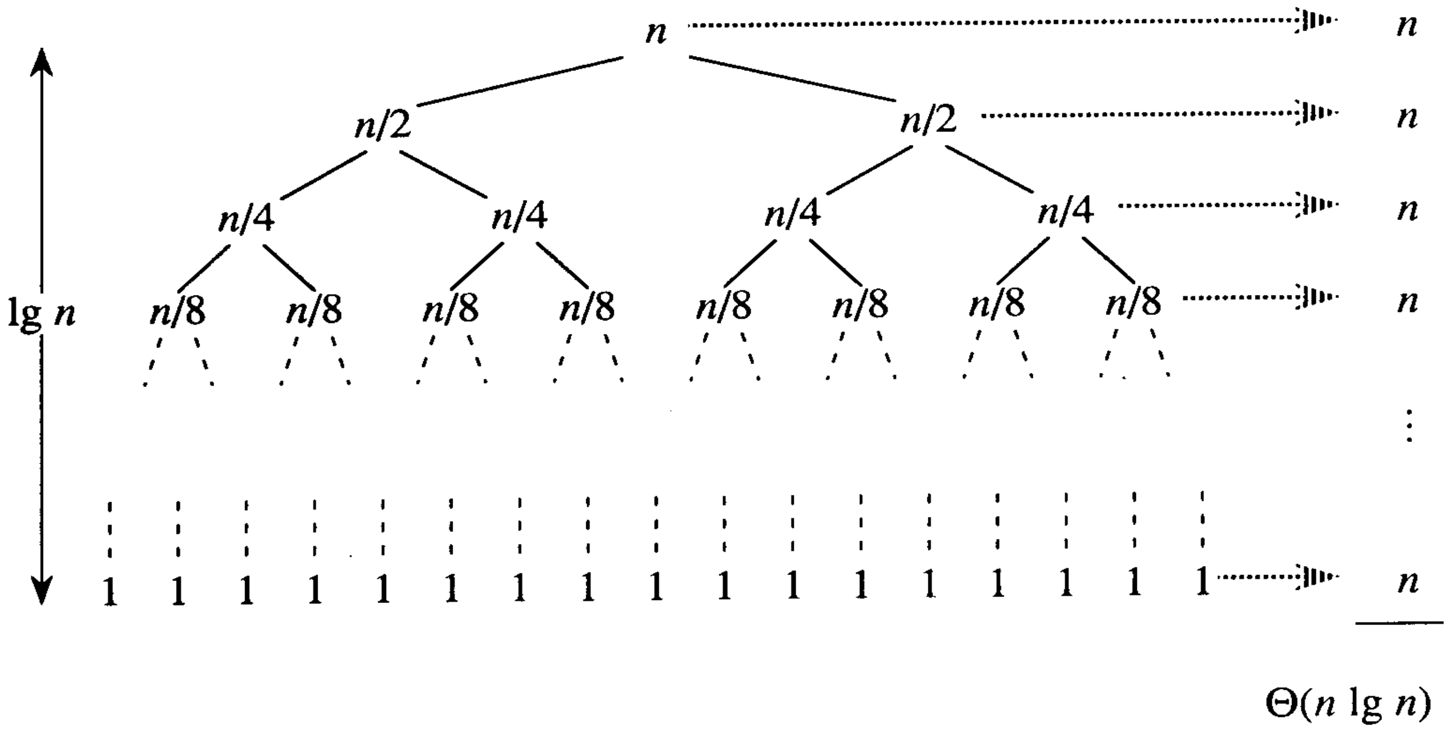


Figure 8.3 A recursion tree for QUICKSORT in which PARTITION always balances the two sides of the partition equally (the best case). The resulting running time is $\Theta(n \lg n)$.

Average Case

- **Assuming random input, average-case running time is much closer to $\Theta(n \lg n)$ than $\Theta(n^2)$**

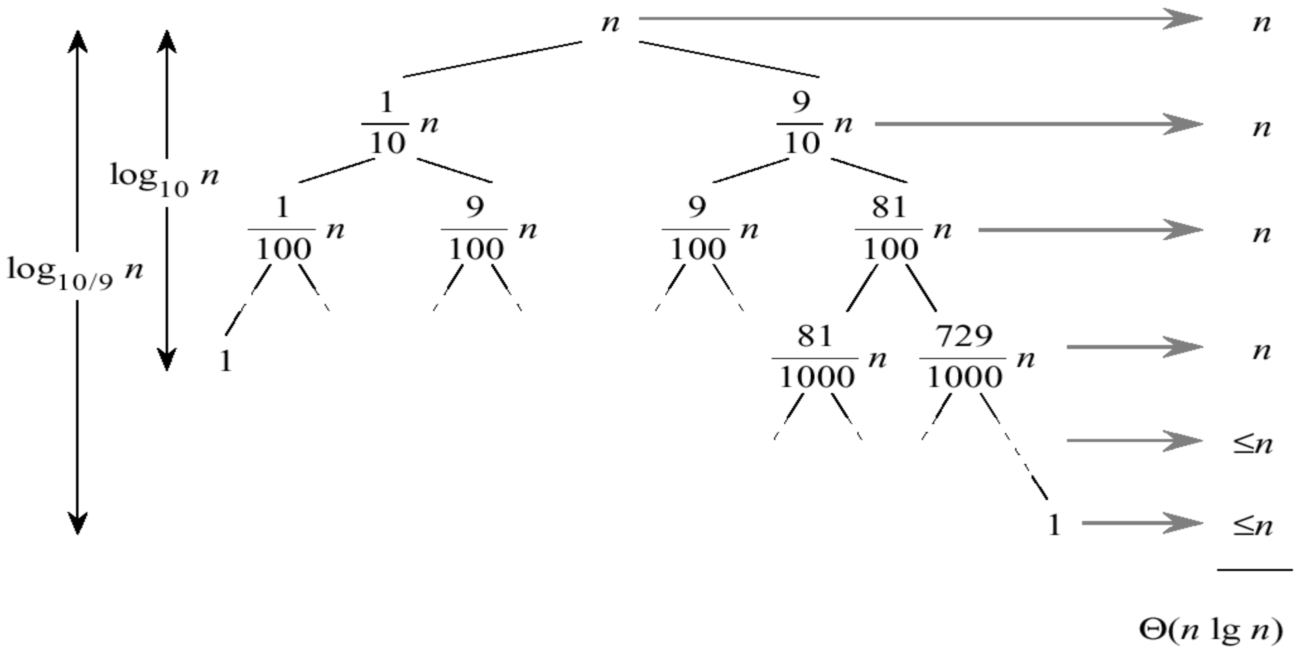
- **First, a more intuitive explanation/example:**
  - **Suppose that partition() always produces a 9-to-1 proportional split. This looks quite unbalanced!**
  - **The recurrence is thus:**

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

[Using recursion tree method to solve]

Average Case

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



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.**

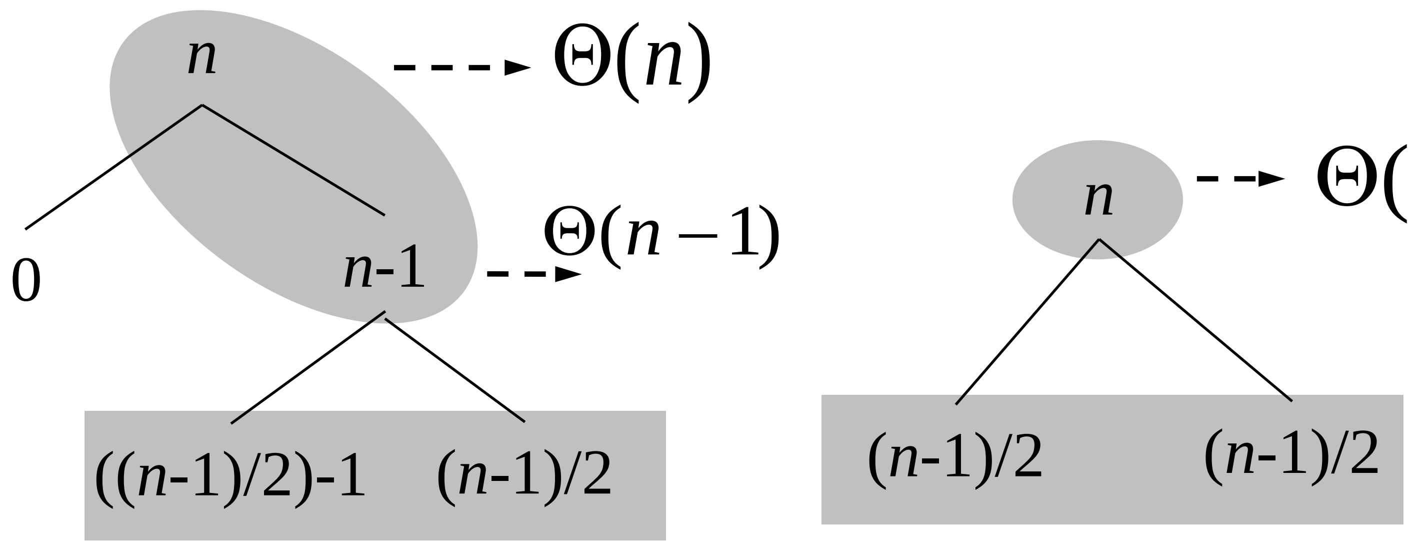
- **The recursion terminates at depth $\log_{10/9} n = \Theta(\lg n)$.**

- **The total average cost of quicksort is therefore O(n lg n).**

$$\log_b(x) = \frac{\log_k(x)}{\log_k(b)}.$$

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**
    - **0, ((n-1)/2)-1, (n-1)/2**
  - **Combined cost of splits = $\Theta(n) + \Theta(n\text{-}1) = \Theta(n)$**



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 obtain good average-case performance over all inputs.

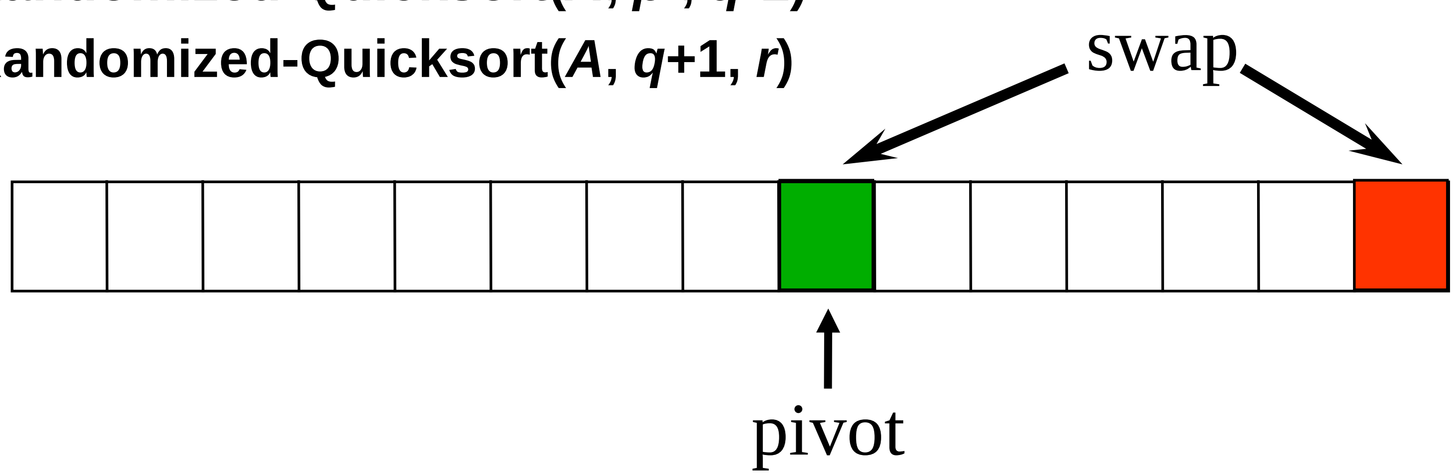
Randomized Quicksort

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*-1)**
4. **Randomized-Quicksort(*A*, *q*+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

Linear Time Sorting

- **Count Sort**
- **Radix Sort**
- **Bucket Sort**

Counting Sort

- Counting Sort was invented by H.H.Seward in 1954.
- All the sorting algorithms introduced so far share an interesting property : the sorted order they determine is based only on comparisons between the input elements.

Assumptions of Counting Sort:

- Counting sort assumes that each of the input element is an Integer and lies in the range 1 to k, for some integer k.

When k = *O(n)* then the sort runs in *O(n)* time

Determine how many elements are less than an element *x*

Then place *x* directly in its correct position

Counting Sort Algorithm

```
for  i <-- 1 to k
    do C[i]<--0                                k
for  j<-- 1 to length[A]
    do  C[A[j]]<-- C[A[j]] + 1                 n
for  I<-- 2 to k
    do  C[i]<-- C[i] + C[i-1]                  k
for  j<--length[A] downto 1
    do  B[C[A[j]]]<-- A[j]
          C[A[j]]<-- C[A[j]] - 1               n
```

- Example for Counting Sort:

	1	2	3	4	5	6	7	8
A	3	6	4	1	3’	4’	1’	4”

	1	2	3	4	5	6
C	2	0	2	3	0	1

	1	2	3	4	5	6
C	2	2	4	7	7	8

	1	2	3	4	5	6
C	2	2	4	7	7	8

	1	2	3	4	5	6	7	8
B							4”	

	1	2	3	4	5	6
C	2	2	4	**6**	7	8

	1	2	3	4	5	6	7	8
B		1’					4”	

	1	2	3	4	5	6
C	**1**	2	4	6	7	8

	1	2	3	4	5	6	7	8
B		1’				4’	4”	

- Example Cont…d

A

1	2	3	4	5	6	7	8
3	6	4	1	3’	4’	1’	4”

C

1	2	3	4	5	6
1	2	4	**5**	7	8

B

1	2	3	4	5	6	7	8
	1’		3’		4’	4”	

C

1	2	3	4	5	6
1	2	**3**	5	7	8

B

1	2	3	4	5	6	7	8
1	1’		3’		4’	4”	

C

1	2	3	4	5	6
0	2	3	5	7	8

B

1	2	3	4	5	6	7	8
1	1’		3’	4	4’	4”	

- Example Cont…d

A

1	2	3	4	5	6	7	8
3	6	4	1	3’	4’	1’	4”

C

1	2	3	4	5	6
0	2	3	**4**	7	8

B

1	2	3	4	5	6	7	8
1	1’		3’	4	4’	4”	6

C

1	2	3	4	5	6
0	2	3	4	7	**7**

B

1	2	3	4	5	6	7	8
1	1’	3	3’	4	4’	4”	6

C

1	2	3	4	5	6
0	2	2	4	7	7

Output Array **B**

1	2	3	4	5	6	7	8
1	1’	3	3’	4	4’	4”	6

Running Time of Counting Sort

Running Time $T(n) = O(k+n)$

Counting Sort beats the lower bound of $\Omega(n \lg n)$ as it is not a comparison sort.

Bucket Sort

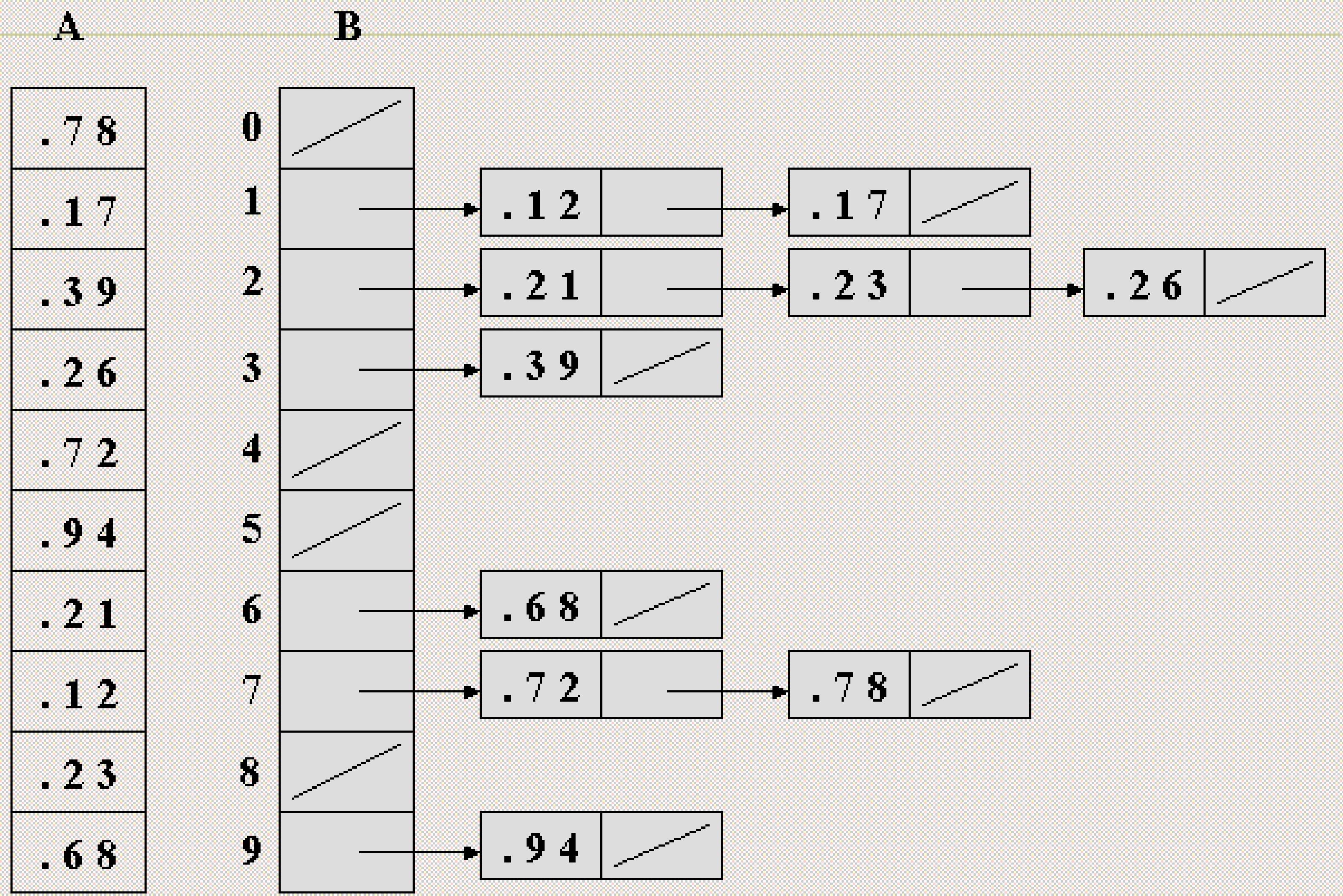
- Bucket sort runs in linear time on the average.

- Bucket sort assumes that the inputs are uniformly distributed over the interval[0,1).

- Basic idea:
  1. Divide the interval[0,1) into n equal-sized buckets.
  2. Distribute the n elements into the buckets.
  3. Sort the elements in each bucket.
  4. Concatenate the buckets in order.

N=10

Bucket i holds values in the interval [i / 10 , (i +1) / 10]



Pseudo Code

Bucket-Sort(A)

```
n ← length[A]
for i ← 1 to n
        do insert A[i] into list B [⌊nA[i]⌋]
for i ← 0 to n - 1
        do sort list B[i] with insertion sort
concatenate the lists B[0], B[1] ,…, B[n-1] together in order
```

Radix sort

This algorithm was used by old card-sorting machines.
(computers, not Black Jack)

Sorting on the least significant digit first, then the second,…

Only d passes through the array are required to sort.
d=the number of digits in every element

Pseudocode:

Radix-Sort (A,d)

for i ← 1 to d
    do use a <u>stable sort</u> to sort array A on digit i

Example:

329		720		720		329
457		355		329		355
657		436		436		436
839	$\Rightarrow$	457	$\Rightarrow$	839	$\Rightarrow$	457
436		657		355		657
720		329		457		720
355		839		657		839
		$\uparrow$		$\uparrow$		$\uparrow$
		1^{st} pass		2^{nd} pass		3^{rd} pass

T(n): Running time

Consider d as a constant

- For d-digit number, every digit is in the range from 1 to k
- When k is not too large, use counting sort as the <u>stable sort</u>
- Running time = running time of <u>stable sort</u> $\times$ d

$T(n) = d \times \Theta(n+k) = \Theta(dn+dk)$

k and d: constant

$T(n)= \Theta(n)$