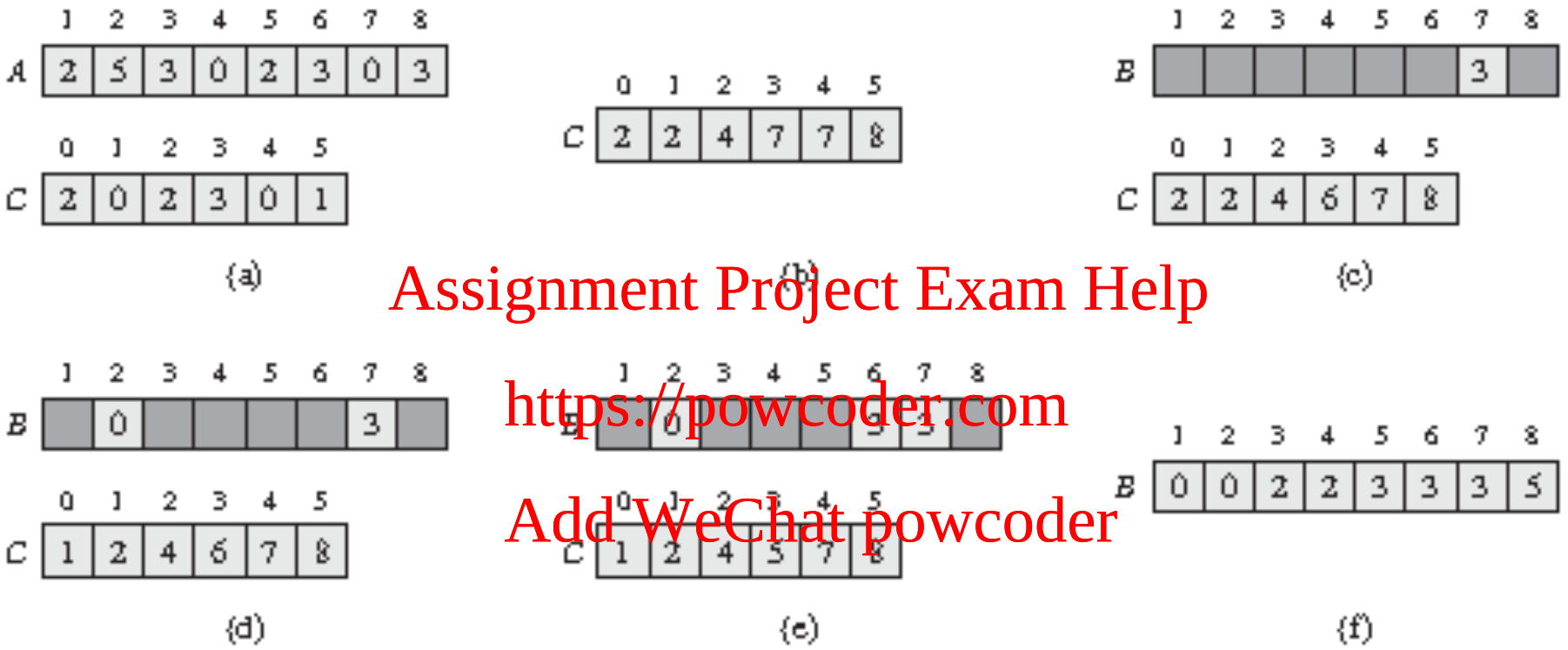
Counting-Sort (*A*, *B*, *k*)



- (a): Array *A* and the auxiliary array *C* after line 5
- (b): Array after line 7
- (c) ~ (e): The change values of array *B* and auxiliary array *C* after iteration, 1, 2, and 3.
- (f): The final sorted output array *B*.

Algorithm Analysis

- The overall time is $O(n+k)$. When we have $k=O(n)$, the worst case is $O(n)$.
  - for-loop of lines 2-3 takes time $O(k)$
  - for-loop of lines 4-5 takes time $O(n)$
  - for-loop of lines 6-7 takes time $O(k)$
  - for-loop of lines 8-10 takes time $O(n)$
- Stable, but not in place.
- No comparisons made: it uses actual values of the elements to index into an array.

Counting Sort Animation

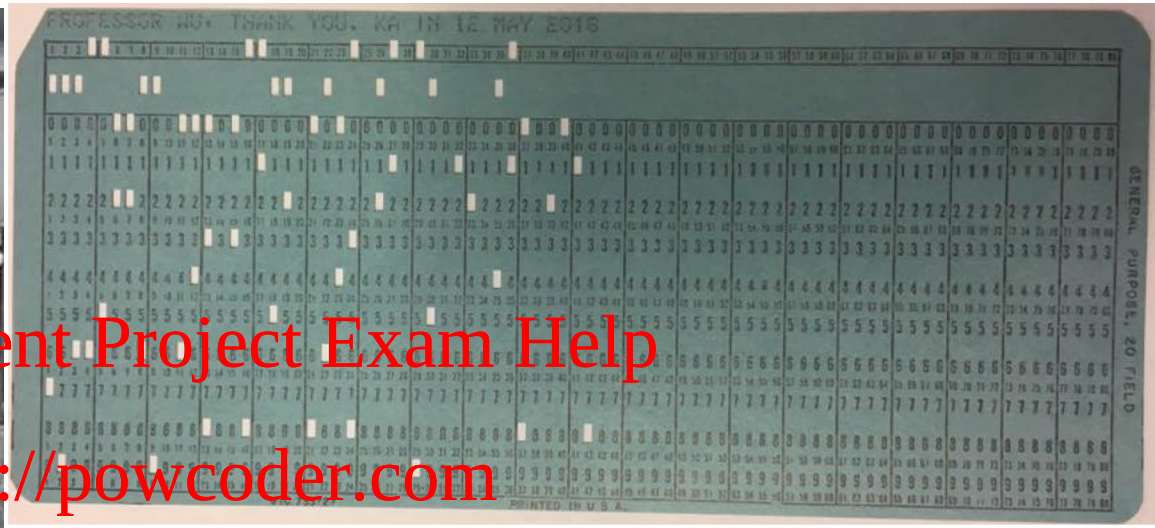
Counting Sort

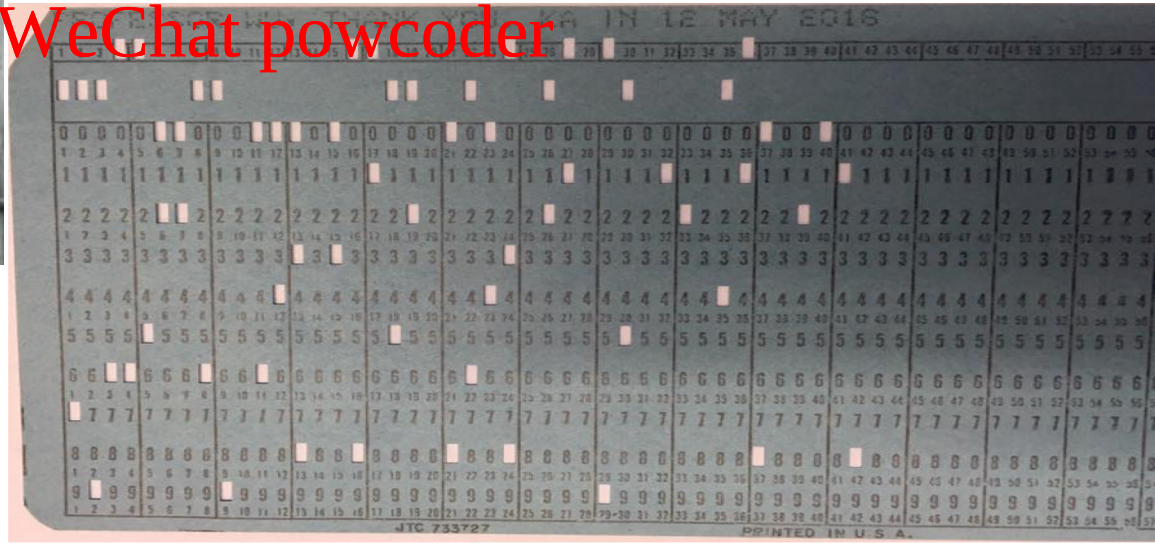
- Cool! *Why don't we always use counting sort?*
- Because it depends on range *k* of elements
- *Could we use counting sort to sort 32 bit integers? Why or why not?*
  - Answer: no, *k* too large (2^{32} = 4,294,967,296)
- 16-bit?
  - Probably not.
- 8-bit?
  - Maybe, depending on *n*.
- **4-bit?**
  - Probably, (unless *n* is really small).

Radix Sort

- *How did IBM get rich originally?*
- Answer: punched card readers for census tabulation in early 1900's.
  - In particular, a *card sorter* that could sort cards into different bins
    - Each column can be punched in 12 places
    - Decimal digits use 10 places
  - Problem: only one column can be sorted on at a time

IBM Punched Card





Radix Sort

- Used to sort on card-sorters:
- Do a stable sort on each column, one column at a time.
- The human operator is part of the algorithm!

- ***Key idea:*** sort on the "least significant digit" first and on the remaining digits in sequential order. The sorting method used to sort each digit must be "stable".

  - If we start with the "most significant digit", we'll need extra storage. (Why? Try it!)
  - Problem: lots of intermediate piles of cards

An Example

Input		*After sorting on LSD*		*After sorting on middle digit*		*After sorting on MSD*
392		631		928		356
356		392		631		392
446		532		532		446
928	⇒	495	⇒	446	⇒	495
631		356		356		532
532		446		392		631
495		928		495		928
		↑		↑		↑

Sorting Animation **Radix Sort using Counting Sort**

Radix-Sort(*A*, *d*)

RadixSort(A, d)

1. *for i ← 1 to d*
2. *do use a stable sort to sort array A on digit i*

Correctness of Radix Sort

- By induction on the number of digits sorted.
- Assume that radix sort works for $d - 1$ digits.
- Show that it works for d digits.
- Radix sort of d digits ≡ radix sort of the low-order $d - 1$ digits followed by a sort on digit d .

Algorithm Analysis

- Each pass over n d-digit numbers then takes time $\Theta(n+k)$. (Assuming counting sort is used for each pass.)
- There are d passes, so the total time for radix sort is $\Theta(d\,(n+k))$.
- When d is a constant and $k = O(n)$, radix sort runs in linear time.
- Radix sort, if uses counting sort as the intermediate stable sort, does not sort in place.
  - If primary memory storage is an issue, quicksort or other sorting methods may be preferable.

Correctness of Radix Sort

By induction hypothesis, the sort of the low-order $d - 1$ digits works, so just before the sort on digit d , the elements are in order according to their low-order $d - 1$ digits. The sort on digit d will order the elements by their d^{th} digit.

Consider two elements, a and b, with d^{th} digits a_d and b_d:

- If $a_d < b_d$, the sort will place a before b, since $a < b$ regardless of the low-order digits.

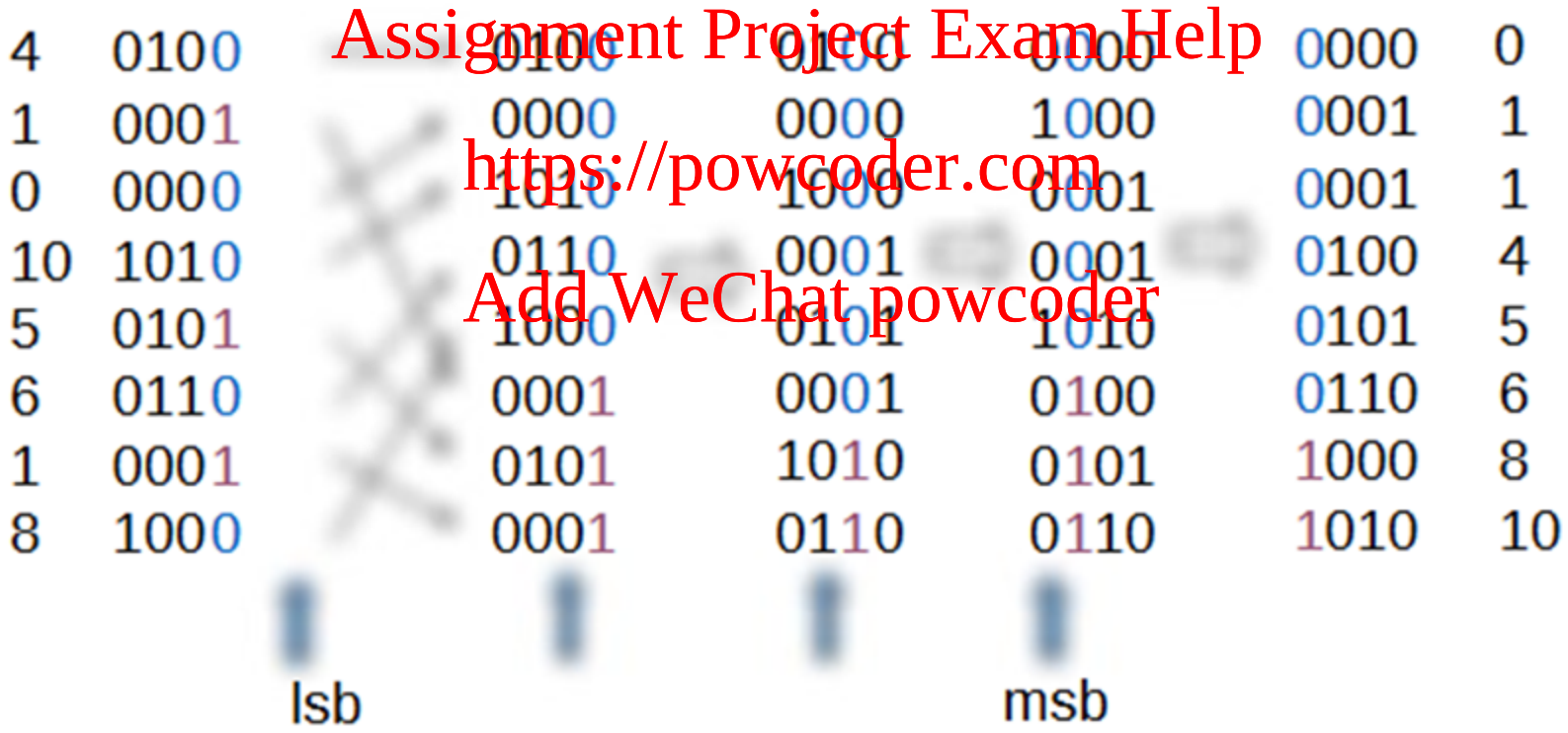
- If $a_d > b_d$, the sort will place a after b, since $a > b$ regardless of the low-order digits.
- If $a_d = b_d$, the sort will leave a and b in the same order, since the sort is stable. But that order is already correct, since the correct order of is determined by the low-order digits when their d^{th} digits are equal.

Radix Sort

- Problem: sort 1 million 64-bit numbers
  - If treat as four-digit radix 2^{16} numbers
  - Can sort in just four passes with radix sort!
  - Running time: $4(1 \text{ million} + 2^{16}) \approx 4$ million operatons
- Compares well with typical $O(n \lg n)$ comparison sort
  - Requires approx $\lg n = 20$ operations per number being sorted
  - Total running time ≈ 20 million operatons
- *So why would we ever use anything but radix sort?*

An Example

- *Sort N numbers, each with k bits*
- *E.g, input {4, 1, 0, 10, 5, 6, 1, 8}*



Another Example

- *English words sorting:*

COW	SEA	TAB	BAR
DOG	TEA	BAR	BIG
SEA	MOB	EAR	BOX
RUG	TAB	TAR	COW
ROW	RUG	SEA	DIG
MOB	DOG	TEA	DOG
BOX	DIG	DIG	EAR
TAB	BIG	BIG	FOX
BAR	BAR	MOB	MOB
EAR	EAR	DOG	NOW
TAR	TAR	COW	ROW
DIG	COW	ROW	RUG
BIG	ROW	NOW	SEA
TEA	NOW	BOX	TAB
NOW	BOX	FOX	TAR
FOX	FOX	RUG	TEA

Radix Sort

- In general, radix sort based on counting sort is
  - Fast
  - Asymptotically fast (i.e., O(*n*))
  - Simple to code
  - A good choice
- To think about: *Can radix sort be used on floating-point numbers?*

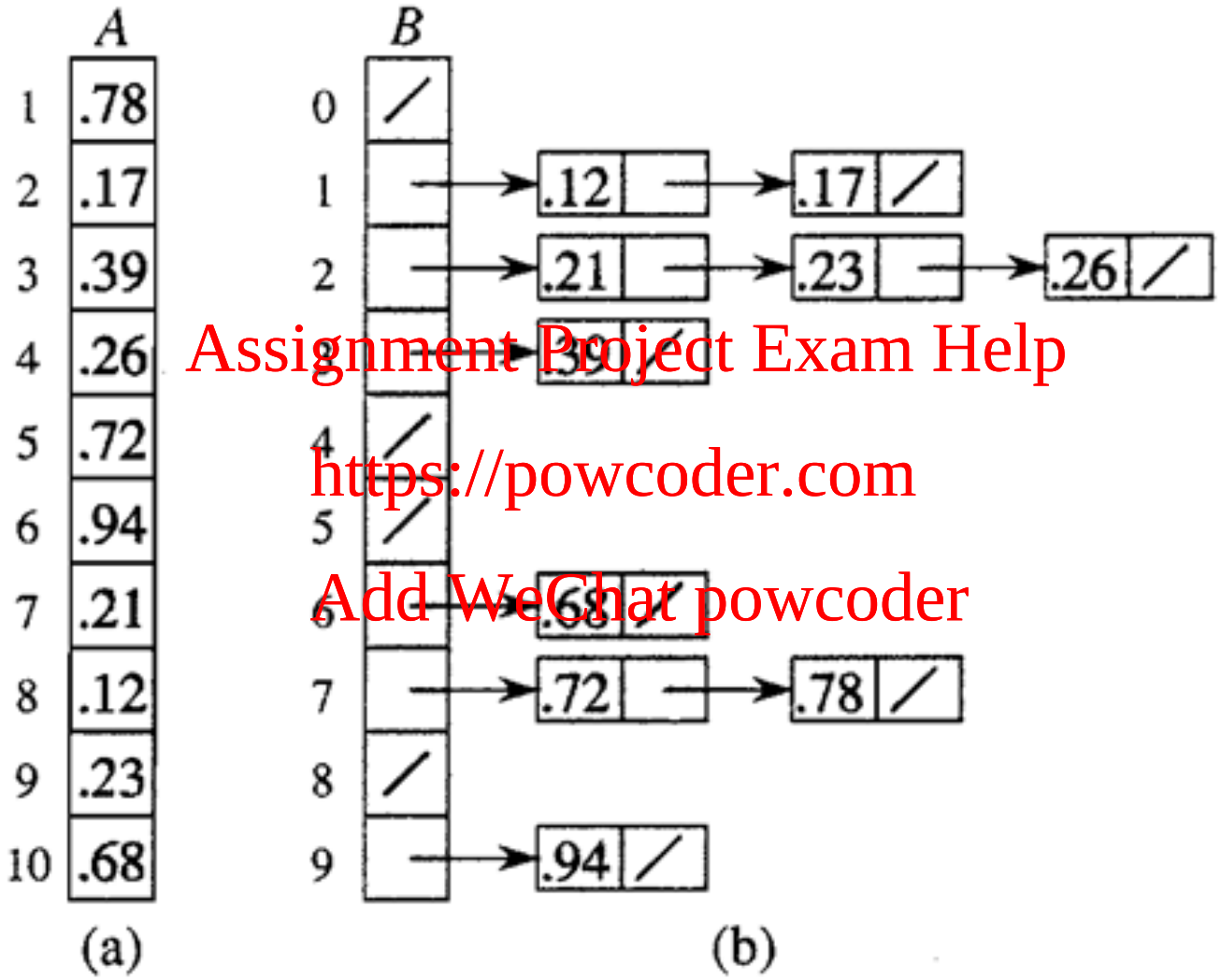
Summary: Radix Sort

- Radix sort:
  - Assumption: input has d digits ranging from 0 to k
  - Basic idea:
    - Sort elements by digit starting with *least* significant
    - Use a stable sort (like counting sort) for each stage
  - Each pass over n numbers with d digits takes time $O(n+k)$, so total time $O(dn+dk)$
    - When d is constant and $k=O(n)$, takes $O(n)$ time
  - Fast! Stable! Simple!
  - Doesn't sort in place

Bucket Sort

- Bucket sort
  - Assumption: input is n reals from [0, 1)
  - Basic idea:
    - Create n linked lists (*buckets*) to divide interval [0,1) **into equal subintervals of size $1/n$**
    - Add each input element to appropriate bucket and sort buckets with insertion sort
  - **Uniform input distribution** → O(1) bucket size
    - Therefore the expected total time is O(n)
  - These ideas will return when we study *hash tables*

An Example



Bucket-Sort (A)

Input: *$A[1..n]$, where $0 \le A[i] < 1$ for all i.*

Auxiliary array: $B[0..n-1]$ of linked lists, each list initially empty.

BucketSort(A)

1. $n \leftarrow A.length$
2. Let $B[0..n-1]$ be a new array
3. **for** $i \leftarrow 0$ to $n-1$
4. make $B[i]$ an empty list
5. **for** $i \leftarrow 0$ to n
6. **do** insert $A[i]$ into list $B[\lfloor nA[i] \rfloor]$
7. **for** $i \leftarrow 0$ **to** $n-1$
8. **do** sort list $B[i]$ with insertion sort
9. concatenate the lists $B[i]$s together in order
10. **return** the concatenated lists

Bucket Sort Animation1 Bucket Sort Animation 2

Correctness of BucketSort

- Consider $A[i]$, $A[j]$. Assume w.o.l.o.g, $A[i] \le A[j]$.
- Then, $\lfloor n \times A[i] \rfloor \le \lfloor n \times A[j] \rfloor$.
- So, $A[i]$ is placed into the same bucket as $A[j]$ or into a bucket with a lower index.
  - If same bucket, insertion sort fixes up.
  - If earlier bucket, concatenation of lists fixes up.

Analysis

- Relies on no bucket getting too many values.
- All lines except insertion sorting in line 8 take $O(n)$ altogether in the worst case.
- Intuitively, if each bucket gets a constant number of elements, it takes $O(1)$ time to sort each bucket $\Rightarrow$ $O(n)$ sort time for all buckets.
- We "expect" each bucket to have few elements, since the average is 1 element per bucket.
- But we need to do a careful analysis.

Analysis – Contd.

- RV n_i = no. of elements placed in bucket $B[i]$.
- Insertion sort runs in quadratic time. Hence, time for bucket sort is:

$$T(n) = \Theta(n) + \sum_{i=0}^{n-1} O(n_i^2)$$

Taking expectations of both sides and using linearity of expectation, we have

$$E[T(n)] = E\left[\Theta(n) + \sum_{i=0}^{n-1} O(n_i^2)\right]$$

$$= \Theta(n) + \sum_{i=0}^{n-1} E[O(n_i^2)] \qquad \text{(by linearity of expectation)}$$

$$= \Theta(n) + \sum_{i=0}^{n-1} O(E[n_i^2]) \qquad (E[aX] = aE[X]) \qquad (8.1)$$

Analysis – Contd.

- **Claim:** $\mathrm{E}[n_i^2] = 2 - 1/n$. *(8.2)*
- **Proof:**
- Define indicator random variables.
  - $X_{ij} = \mathrm{I}\{A[j] \text{ falls in bucket } i\}$
  - $\Pr\{A[j] \text{ falls in bucket } i\} = 1/n$.
  - $n_i = \sum_{j=1}^{n} X_{ij}$

Analysis – Contd.

$$E[n_i^2] = E\left[\left(\sum_{j=1}^{n} X_{ij}\right)^2\right]$$

$$= E\left[\sum_{j=1}^{n}\sum_{k=1}^{n} X_{ij}X_{ik}\right]$$

$$= \mathrm{E}\left[\sum_{j=1}^{n} X_{ij}^2 + \sum_{\substack{1\le j\le n \\ j\neq k}}\sum_{1\le k\le n} X_{ij}X_{ik}\right]$$

$$= \sum_{j=1}^{n} E[X_{ij}^2] + \sum_{\substack{1\le j\le n \\ j\neq k}}\sum_{1\le k\le n} E[X_{ij}X_{ik}] \text{ , by linearity of expectation.} \qquad (8.3)$$

Analysis – Contd.

$$E[X_{ij}^2] = 0^2 \cdot \Pr\{A[j] \text{ doesn't fall in bucket } i\} +$$

$$1^2 \cdot \Pr\{A[j] \text{ falls in bucket } i\}$$

$$= 0 \cdot \left(1 - \frac{1}{n}\right) + 1 \cdot \frac{1}{n}$$

$$= \frac{1}{n}$$

$E[X_{ij} X_{ik}]$ for $j \neq k$:

Since $j \neq k$, X_{ij} and X_{ik} are independent random variables.

$$\Rightarrow E[X_{ij} X_{ik}] = E[X_{ij}]E[X_{ik}]$$

$$= \frac{1}{n} \cdot \frac{1}{n} = \frac{1}{n^2}$$

Analysis – Contd.

(8.3) is hence,

$$E[n_i^2] = \sum_{j=1}^{n} \frac{1}{n} + \sum_{1 \le j \le n} \sum_{\substack{1 \le k \le n \\ k \ne j}} \frac{1}{n^2}$$

$$= n \cdot \frac{1}{n} + n(n-1) \cdot \frac{1}{n^2}$$

$$= 1 + \frac{n-1}{n}$$

$$= 2 - \frac{1}{n}.$$

Substituting (8.2) in (8.1), we have,

$$E[T(n)] = \Theta(n) + \sum_{i=0}^{n-1} O(2 - 1/n)$$

$$= \Theta(n) + O(n)$$

$$= \Theta(n)$$