Università di Napoli Federico II – Scuola Politecnica e delle Scienze di Base
Corso di Laurea Magistrale in Ingegneria Informatica

Corso di Algoritmi e Strutture Dati

Ordinamento in tempo lineare

Algoritmi di ordinamento

- Gli algoritmi di ordinamento che abbiamo visto sono basati sul confronto tra elementi (***comparison sort***)
- Dimostriamo che, per algoritmi basati sul confronto, nel caso peggiore sono necessari $\Omega(n\lg n)$ confronti per ordinare n elementi

- Merge sort e heap sort sono asintoticamente ottimi

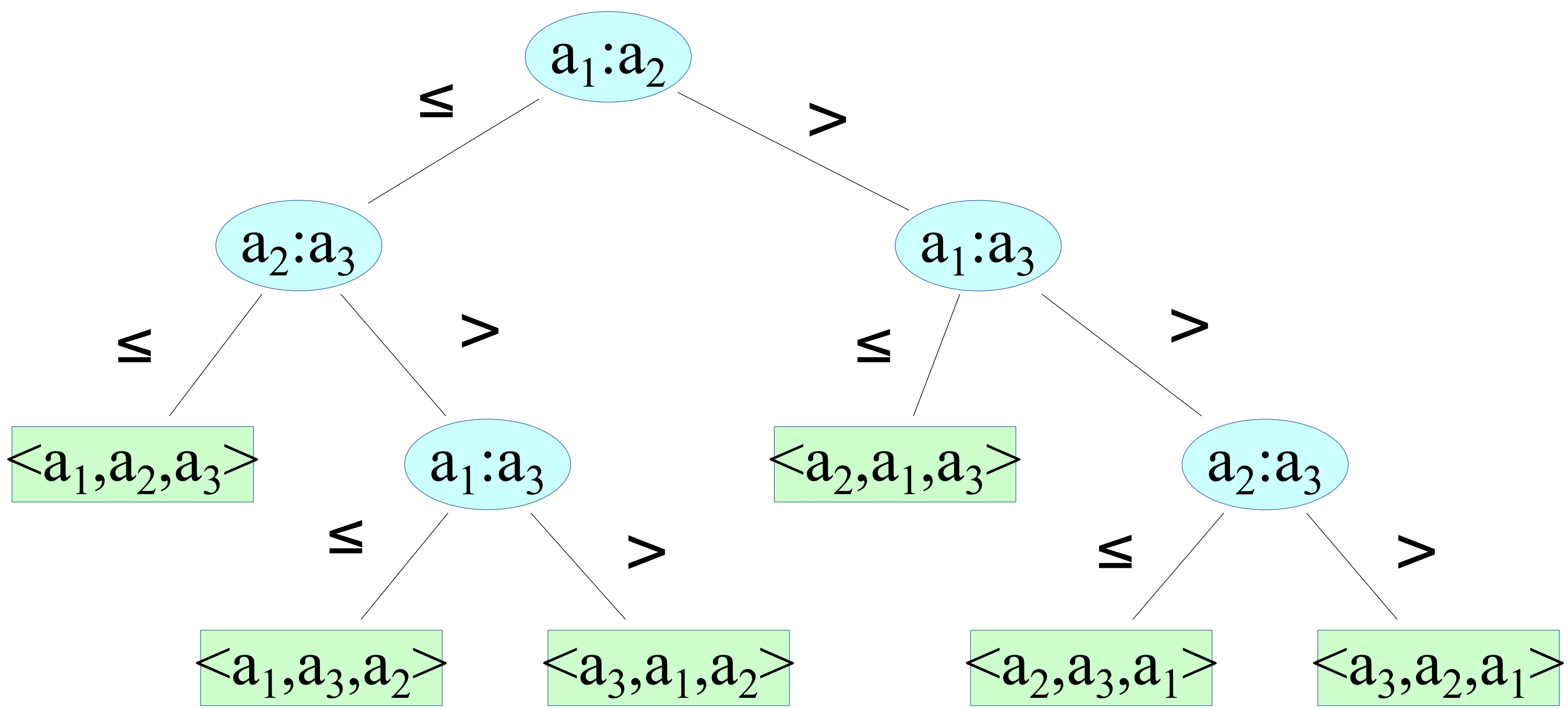
- Quick sort invece nel caso peggiore ottiene $\Theta(n^2)$

- Vedremo anche algoritmi, **non** basati sul confronto, che richiedono un tempo lineare

Algoritmi di ordinamento

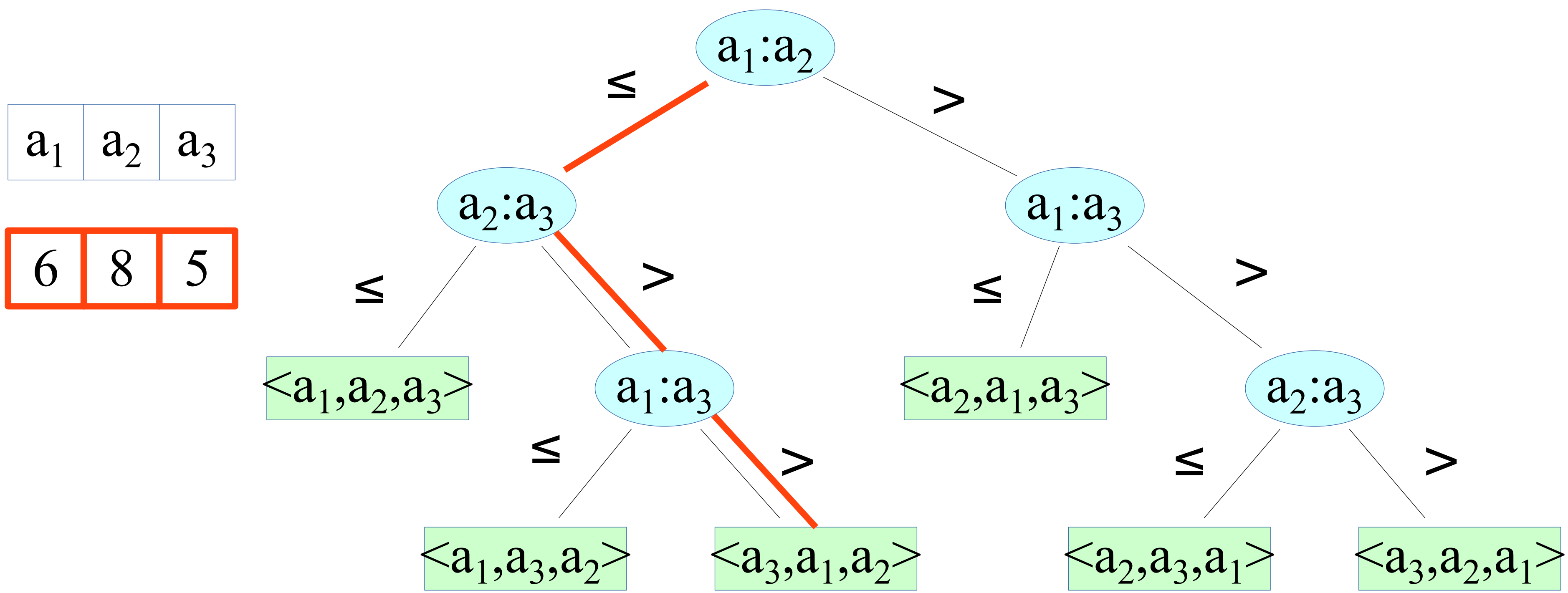
- Le operazioni di un algoritmo di ordinamento basato su confronti possono essere descritte mediante un albero di decisione
  - Ogni nodo indica una coppia di elementi da confrontare
  - Le foglie indicano i possibili ordinamenti risultanti dai confronti
- Es. Insertion Sort

a_1	a_2	a_3



Algoritmi di ordinamento

- L'esecuzione di un algoritmo corrisponde a tracciare un percorso dalla radice ad una foglia
- Ci devono essere *almeno* n! foglie *raggiungibili* dalla radice
  - n! è il numero di permutazioni di n elementi



Limite inferiore per il caso peggiore

- Il numero di confronti effettuati da un algoritmo nel caso peggiore è dato dalla <u>lunghezza del più lungo percorso dalla radice dell'albero di decisione ad una foglia</u>
    - *Ovvero dall'altezza dell'albero di decisione*
- Un limite inferiore per il tempo di esecuzione nel caso peggiore di un qualunque algoritmo basato su confronti è dato dal limite inferiore per l'altezza di tutti gli alberi di decisione in cui ogni permutazione appare come una foglia raggiungibile dalla radice

Limite inferiore per il caso peggiore

- Teorema
- *Qualunque algoritmo di ordinamento basato su confronti richiede $\Omega(n\lg n)$ confronti nel caso peggiore*

- Indichiamo con **h** l'altezza e con **l** il numero di foglie raggiungibili dell'albero di decisione

- $n! \leq l \leq 2^h$ (un albero binario di altezza h ha al più 2^h foglie)

- Prendendo i logaritmi (funzione crescente)

- $h \geq \lg(n!) = \Theta(n\lg n) \Rightarrow h = \Omega(n\lg n)$

Limite inferiore per il caso peggiore

h ≥ lg (n!) = Θ(nlg n) ⇒ h = Ω(nlg n)

$$\lg(n!) = \lg\big(n(n-1)(n-2)\ldots\big) = \sum_{i=1}^{n} \lg(i)$$

$$\geq \sum_{i=\frac{n}{2}}^{n} \lg(i)$$

$$\geq \sum_{i=n/2}^{n} \lg\left(\frac{n}{2}\right) = \sum_{i=n/2}^{n} \big(\lg(n) - lg(2)\big) = \sum_{i=n/2}^{n} (\lg(n) - 1)$$

$$= \frac{n}{2}\lg(n) - \frac{n}{2} = \Omega(nlg(n))$$

O, Equivalentemente, dall'approssimazione di Stirling:

$$n! \sim \sqrt{2\pi n}\left(\frac{n}{e}\right)^{n} > \left(\frac{n}{e}\right)^{n} \quad \Rightarrow \quad lg(n!) > n\big(lg(n) - \lg(e)\big) = \Omega(nlg(n))$$

Counting sort

- Assume che gli elementi da ordinare siano interi compresi tra 0 e k
- Approccio
  - Per ciascun intero i compreso tra 0 e k, si contano quanti elementi *pari* ad i ci sono nel vettore da ordinare
  - Per ciascun intero i compreso tra 0 e k, si determinano quanti elementi *minori o uguali* ad i ci sono nel vettore da ordinare
  - Ciò ci indica in che posizione deve stare ciascun elemento
- Counting sort utilizza due vettori di appoggio
  - B, di lunghezza n, che mantiene i valori ordinati
  - C, di lunghezza k+1, che indica le occorrenze di ciascun valore compreso tra 0 e k

Counting sort

- Es. Valori interi tra 0 e k=5

A	2	5	3	0	2	3	0	3

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

	0	1	2	3	4	5
C	0	2	2	4	7	7

<u>Counting-Sort</u> (A,B,k)

```
for i ← 0 to k
    do C[i] ← 0
for j ← 1 to length[A]
    do C[A[j]] ← C[A[j]]+1
// C[i] è il numero di occorrenze di i
for i ← 1 to k
    do C[i] ← C[i]+C[i-1]
// C[i] è il numero di elementi ≤ i
for j ← length[A] downto 1
    do B[C[A[j]]] ← A[j]
    C[A[j]] ← C[A[j]]-1
```

Counting sort

- Tempo di esecuzione $\Theta(n+k)$
- Nella pratica, counting sort si usa quando $k=O(n)$
  - In tal caso, il tempo di esecuzione è $\Theta(n)$
- **Non** è un algoritmo di ordinamento basato su confronti
- Non ordina sul posto
- È **stabile**
  - Gli elementi di pari valore si presentano nel vettore risultato nello stesso ordine in cui si trovano nel vettore di partenza

Radix sort

- Assume che gli elementi da ordinare siano rappresentati su d cifre
- Ordina i valori a partire dalla cifra meno significativa
- È indispensabile utilizzare per tale ordinamento un algoritmo stabile

329	720	720	329
457	355	329	355
657	436	436	436
839	457	839	457
436	657	355	657
720	329	457	720
355	839	657	839

Radix sort

<u>Radix-Sort</u> (A,d)
for i ← 1 to d
do use a stable sort to sort array A on digit i

La cifra di posto 1 è quella meno significativa

- La correttezza si può provare per induzione
- Se ciascuna cifra assume un numero limitato di valori (k), è opportuno scegliere counting sort
    - Tempo di esecuzione $\Theta(d(n+k))$
    - Tempo lineare se d è costante e $k=O(n)$
- Radix sort è spesso usato per ordinare informazioni aventi molteplici campi
    - es. per ordinare in base alla data, si possono effettuare tre ordinamenti stabili: in base al giorno, al mese, all'anno

Radix sort

- In generale, si ha la flessibilità di scegliere di ordinare in base a gruppi di cifre
- $\underbrace{0001}_{r}011011101010$ b bit (16), gruppi di $r \leq b$ bit (4)

- Il tempo di esecuzione di radix sort è $\Theta((b/r)(n+2^r))$
  - $d = \lceil b/r \rceil$ cifre, ciascuna di r bit $\Rightarrow$ $0 \leq d \leq 2^r-1$
  - $\Theta(d(n+k)) \rightarrow \Theta((b/r)(n+2^r))$

- Dati b e n, quale valore di r minimizza il tempo di esecuzione?

Radix sort

- 0001011011101010 $\underbrace{}_{r}$ b bit (16), gruppi di $r \leq b$ bit (4)

- Tempo di esecuzione $\Theta((b/r)(n+2^r))$
- Se $b < \lfloor \lg n \rfloor$
  - $r \leq b < \lfloor \lg n \rfloor \Rightarrow 2^r < n \Rightarrow n+2^r = \Theta(n)$
  - Conviene scegliere il valore massimo di r per minimizzare b/r
  - Per r=b, il tempo di esecuzione è $\Theta(n)$
- Se $b \geq \lfloor \lg n \rfloor$, la scelta migliore è $r = \lfloor \lg n \rfloor \rightarrow \Theta(bn/\lg n)$
  - Per $r > \lfloor \lg n \rfloor$, 2^r cresce più rapidamente di n → tempo cresce
  - Per $r < \lfloor \lg n \rfloor$, b/r cresce e $n+2^r$ rimane $\Theta(n)$
- … ovvero $r = \min \{b, \lg n\}$

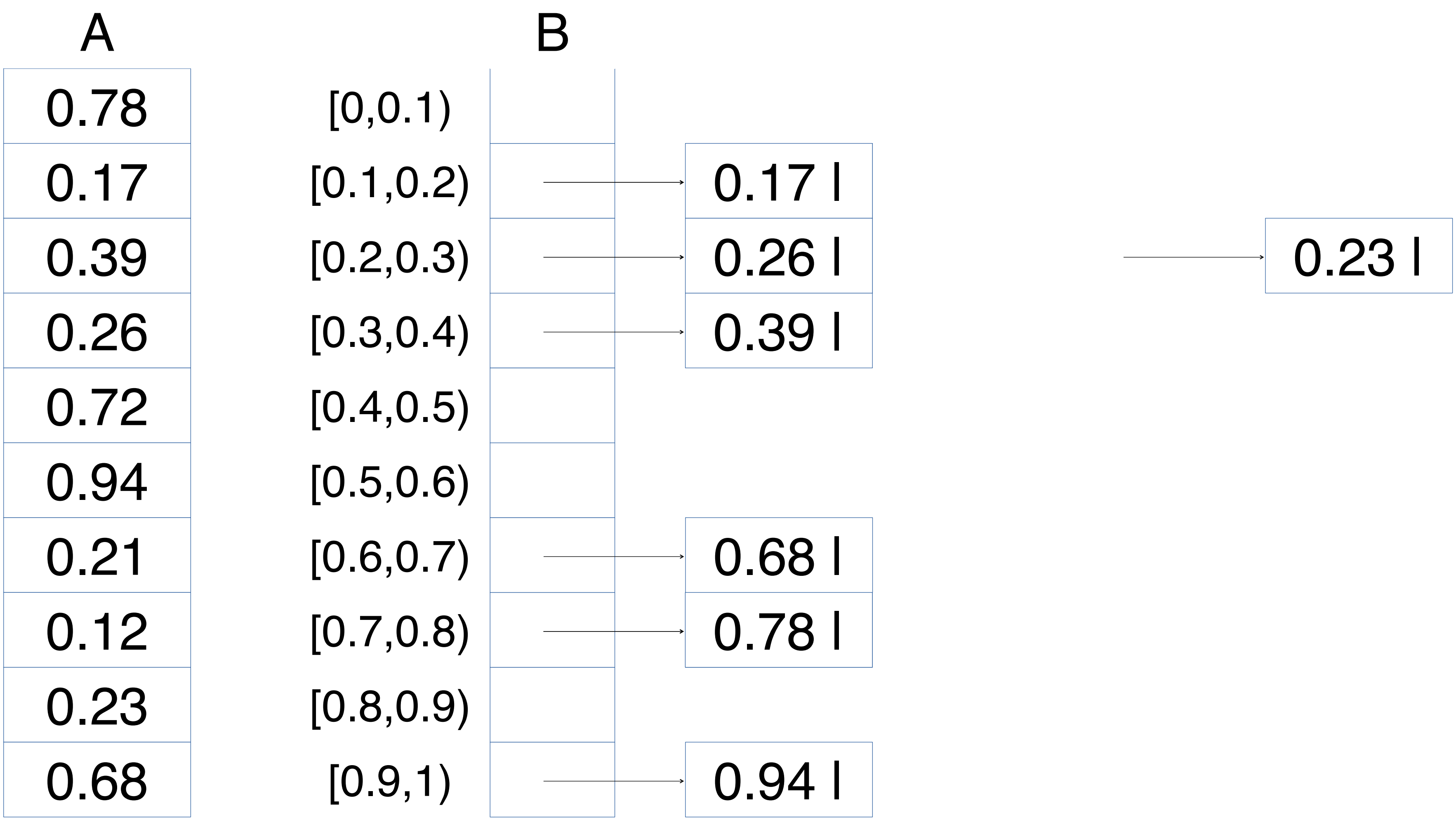
Radix sort

- Radix sort è preferibile ad un algoritmo basato su confronto come quick sort?
  - Se, come accade spesso, $b=O(\lg n)$ e $r \approx \lg n$, il tempo di esecuzione di radix sort è $\Theta(n)$, che *appare* migliore di $\Theta(n \lg n)$
  - I fattori costanti nascosti nella notazione Θ possono differire
- Radix sort può effettuare meno passi di quicksort, ma ogni passo di radix sort può impiegare più tempo
  - Quicksort può sfruttare le cache meglio di radix sort
  - Radix sort, quando usa counting sort, non ordina sul posto e quindi richiede memoria aggiuntiva

Bucket sort

- Assume che gli elementi da ordinare
  siano distribuiti uniformemente sull'intervallo [0,1)
- L'approccio consiste in
- Dividere l'intervallo [0,1) in n sottointervalli uguali e distribuire gli n elementi in tali sottointervalli
- Si devono ordinare gli elementi inseriti nello stesso sottointervallo
- Bucket sort richiede di gestire un array ausiliario B[0..n-1] di liste collegate

Bucket sort


A
B
0.78
0.17
0.39
0.26
0.72
0.94
0.21
0.12
0.23
0.68
[0,0.1)
[0.1,0.2)
[0.2,0.3)
[0.3,0.4)
[0.4,0.5)
[0.5,0.6)
[0.6,0.7)
[0.7,0.8)
[0.8,0.9)
[0.9,1)
0.17 /
0.26 /
0.23 /
0.39 /
0.68 /
0.78 /
0.94 /

Bucket sort

```
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[n-1] in order
```

- Correttezza
  - Se $A[i] \leq A[j]$, anche $\lfloor nA[i] \rfloor \leq \lfloor nA[j] \rfloor$ quindi $A[i]$ è posto nella stessa lista di $A[j]$ o in quella precedente
- Nel primo caso, il secondo ciclo for li mette nel giusto ordine
- Nel secondo caso, la concatenazione delle liste li mette nel giusto ordine

Bucket sort

- Tempo di esecuzione
  - L'inserimento degli elementi nelle liste (primo ciclo for) richiede Θ(n)
  - La concatenazione delle liste ordinate richiede Θ(n)
  - E l'ordinamento delle liste?
- Indicando con n_i la variabile aleatoria che denota il numero di elementi nella lista B[i]:

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

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

Bucket sort

- Indicando con X_{ij} la variabile aleatoria che vale 1 se A[j] finisce nella lista i e 0 altrimenti:

$$n_i = \sum_{j=1}^{n} X_{ij}$$

$$E\left[n_i^2\right] = E\left[\left(\sum_{j=1}^{n} X_{ij}\right)^2\right] = E\left[\sum_{j=1}^{n}\sum_{k=1}^{n} X_{ij}X_{ik}\right] = E\left[\sum_{j=1}^{n} X_{ij}^2 + \sum_{1\leqslant j\leqslant n}\ \sum_{\substack{1\leqslant k\leqslant n\\ k\neq j}} X_{ij}X_{ik}\right]$$

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

$$E\left[X_{ij}^2\right] = 1\cdot\frac{1}{n} + 0\cdot\left(1-\frac{1}{n}\right) = \frac{1}{n}$$

$$E\left[X_{ij}X_{ik}\right] = E\left[X_{ij}\right]E\left[X_{ik}\right] = \frac{1}{n}\cdot\frac{1}{n} = \frac{1}{n^2}$$

Bucket sort

$$E\left[n_i^2\right] = \sum_{j=1}^{n} \frac{1}{n} + \sum_{1 \leqslant j \leqslant n} \sum_{\substack{1 \leqslant k \leqslant n \\ k \neq j}} \frac{1}{n^2} = n \cdot \frac{1}{n} + n(n-1) \cdot \frac{1}{n^2} = 1 + \frac{n-1}{n} = 2 - \frac{1}{n}$$

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

- Bucket sort può ordinare in tempo lineare anche se i valori non hanno distribuzione uniforme
- È sufficiente che $\sum_{i=0}^{n-1} O\left(E\left[n_i^2\right]\right)$ cresca linearmente con n