COT 6405 Introduction to Theory of Algorithms

Topic 10. Linear Time Sorting

How fast can we sort?

- The sorting algorithms we learned so far
  - Insertion Sort, Merge Sort, Heap Sort, and Quicksort
- How fast are they?
  - Insertion sort O(n^2)
  - Merge Sort O(nlgn)
  - Heap Sort O(nlgn)
  - Quicksort O(nlgn)

Common property

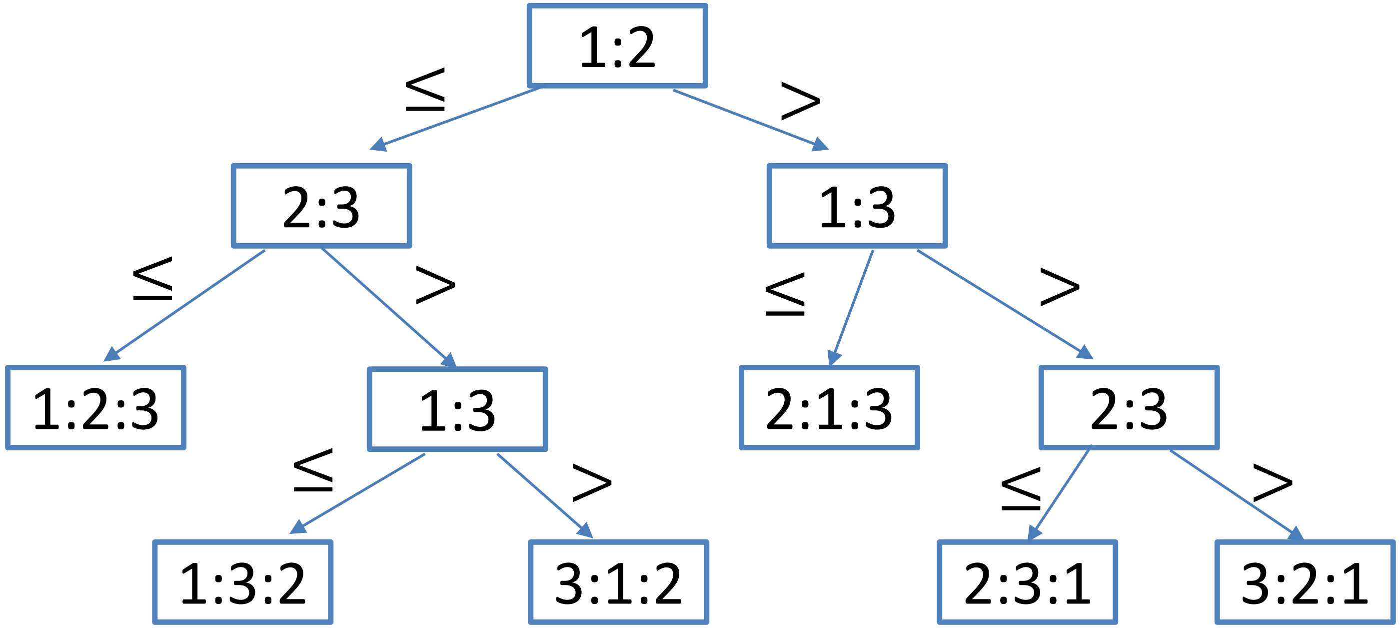
- Use only comparisons between elements to gain order information about an input sequence
- Comparison sort
  - Given two elements a_i and a_j, we perform one of the following tests to determine their relative order
  - $a_i < a_j$, $a_i \leq a_j$, $a_i = a_j$, $a_i \geq a_j$, $a_i > a_j$

Decision trees

- We can view comparison sorts abstractly in terms of decision trees
  - A decision tree is a binary tree that represents the comparisons between elements
  - Each node on the tree is a comparison of *i*:*j*, i.e., a_i v.s. a_j

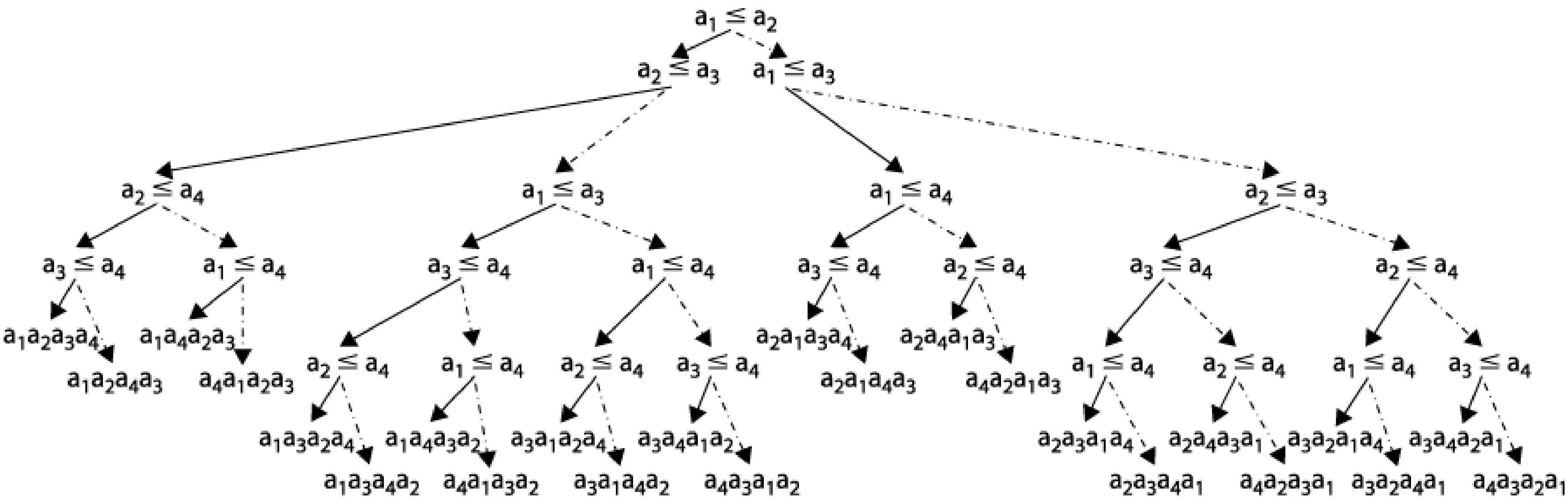
Constructing the decision tree

- Given an input sequence $\{a_1, a_2, a_3\}$



Decision tree for an input set of four elements

Given an input sequence $\{a_1, a_2, a_3, a_4\}$



Decision trees (cont'd)

- What do the leaves represent?
  - The leaf node in the tree indicates the sorted ordering
- How many leaves must be there for an input of size *n*
  - Each of the n! permutations on n elements must appear as one of the leaves of the decision tree

Lemma

- Any binary tree of height h has $\leq 2^h$ leaves
- In other words:
  - i = number of leaves
  - h = height
  - Then, $i \leq 2^h$
- How to prove this?

Theorem 8.1

- Any comparison sort algorithm requires $\Omega(nlgn)$ comparisons.
- How to prove?
  - By proving that the height of the decision tree is $\Omega(nlgn)$
  - What's the # of leaves of a decision tree? l = ?
  - A decision tree is a binary tree. What's the maximum # of leaves of a general binary tree?

    l_{max} = ?

Proof

- $l = n!$ and $l_{max} = 2^h$
- Clearly, the # of leaves of a decision tree is less than or equal to the maximum # of leaves in a general binary tree
- So we have: $n! \leq 2^h$
- Taking logarithms: $\lg(n!) \leq h$

Proof (cont'd)

- Stirling's approximation tells us:

$$n! > \left(\frac{n}{e}\right)^n$$

- Thus, $h \geq \lg(n!)$

$$\begin{aligned} h &\geq \lg\left(\frac{n}{e}\right)^n \\ &= n\lg n - n\lg e \\ &= \Omega(n\lg n) \end{aligned}$$

Sorting in linear time

- Counting sort
  - No direct comparisons between elements!
  - Depends on assumption about the numbers being sorted
    - We assume numbers are in the range [0.. k]
  - The algorithm is NOT "in place"
    - Input: A[1..n], where A[j] $\in$ {0, 2, 3, ..., k}
    - Output: B[1..n], sorted
    - Auxiliary counter storage: Array C[0..k]
    - notice: A[], B[], and C[] → <u>not sorting in place</u>

Counting sort

```
CountingSort(A, B, k)
    for i= 0 to k    // counter initialization
          C[i]= 0;
    for j= 1 to A.length
          C[A[j]] += 1;
    for i= 1 to k    // aggregate counters
          C[i] = C[i] + C[i-1];
    for j= A.length downto 1 //move results
          B[C[A[j]]] = A[j];
          C[A[j]] -= 1;
```

A counting sort example

Numbers are in the range [0.. 5]

A

2	5	3	0	2	3	0	3

k = 5

C

0	1	2	3	4	5
0	0	0	0	0	0

```
CountingSort(A, B, k)
    for i= 0 to k     // counter initialization
          C[i]= 0;
    for j= 1 to A.length
          C[A[j]] += 1;
    for i= 1 to k     // aggregate counters
          C[i] = C[i] + C[i-1];
    for j= A.length downto 1 //move results
          B[C[A[j]]] = A[j];
          C[A[j]] -= 1;
```

Filling the C array

A	2	5	3	0	2	3	0	3

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

```
CountingSort(A, B, k)
   for i= 0 to k    // counter initialization
        C[i]= 0;
   for j= 1 to A.length // counting each number
        C[A[j]] += 1;
   for i= 1 to k    // aggregate counters
        C[i] = C[i] + C[i-1];
   for j= A.length downto 1 //move results
        B[C[A[j]]] = A[j];
        C[A[j]] -= 1;
```

Filling the C array (Cont’d)

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

```
CountingSort(A, B, k)
   for i= 0 to k    // counter initialization
         C[i]= 0;
   for j= 1 to A.length
         C[A[j]] += 1;
   for i= 1 to k    // aggregate counters
         C[i] = C[i] + C[i-1];
   for j= A.length downto 1 //move results
         B[C[A[j]]] = A[j];
         C[A[j]] -= 1;
```

Sorting the numbers

A	2	5	3	0	2	3	0	3

	1	2	3	4	5	6	7	8
B								

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

```
CountingSort(A, B, k)
   for i= 0 to k    // counter initialization
         C[i]= 0;
   for j= 1 to A.length
         C[A[j]] += 1;
   for i= 1 to k    // aggregate counters
         C[i] = C[i] + C[i-1];
   for j= A.length downto 1 //move results
         B[C[A[j]]] = A[j];
         C[A[j]] -= 1;
```

Sorting the numbers

A	2	5	3	0	2	3	0	3

	1	2	3	4	5	6	7	8
B		0					3	

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

```
CountingSort(A, B, k)
   for i= 0 to k    // counter initialization
        C[i]= 0;
   for j= 1 to A.length
        C[A[j]] += 1;
   for i= 1 to k    // aggregate counters
        C[i] = C[i] + C[i-1];
   for j= A.length downto 1 //move results
        B[C[A[j]]] = A[j];
        C[A[j]] -= 1;
```

Sorting the numbers

A	2	5	3	0	2	3	0	3

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

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

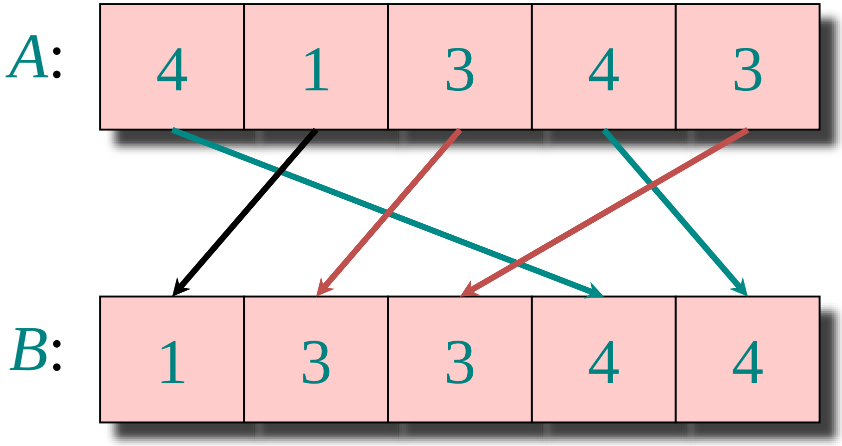
```
CountingSort(A, B, k)
   for i= 0 to k    // counter initialization
        C[i]= 0;
   for j= 1 to A.length
        C[A[j]] += 1;
   for i= 1 to k    // aggregate counters
        C[i] = C[i] + C[i-1];
   for j= A.length downto 1 //move results
        B[C[A[j]]] = A[j];
        C[A[j]] -= 1;
```

Counting sort

- Total time: O(n + k)
  - Usually, k = O(n) → k < c n
  - Thus counting sort runs in O(n) time
- But sorting is Ω(n lg n) ! Contradiction?
  - No contradiction--this is not a comparison sort (in fact, there are *no* comparisons at all!)
  - Notice that this algorithm is stable
    - The elements with the same value is in the same order as the original
    - index i < j, $a_i = a_j$ → new index i' < j'

Stable sorting

Counting sort is a stable sort: it preserves the input order among equal elements.



Counting Sort

- 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)
  - We need huge arrays, e.g., C[4,294,967,296]?
  - k >> n → O(n+k) = O(k)

Radix Sort

- Intuitively, we may sort on the most significant digit (MSD), then the second msd, etc.
- Recursive MSD radix sort:
  - Take the k-th most significant digit (MSD)
  - Sort based on that digit, grouping same digit elements into one bucket
  - In each bucket, start with the next digit and sort recursively
  - Finally, concatenate the buckets in order

An example of a forward recursive MSD radix sort

- Original sequence: 170, 045, 075, 090, 002, 024, 802, 066
- 1st pass- Sorting by most significant digit (100's):
  - Zero bucket: 045, 075, 090, 002, 024, 066
  - One bucket: 170
  - Eight bucket: 802

An example (cont’d)

- 2nd pass- Sorting by next most significant digit (10’s), only needed by numbers in zero bucket:
  - 045, 075, 090, 002, 024, 066
  - Zero bucket: 002
  - Twenties bucket: 024
  - Forties bucket: 045
  - Sixties bucket: 066
  - Seventies bucket: 075
  - Nineties bucket: 090

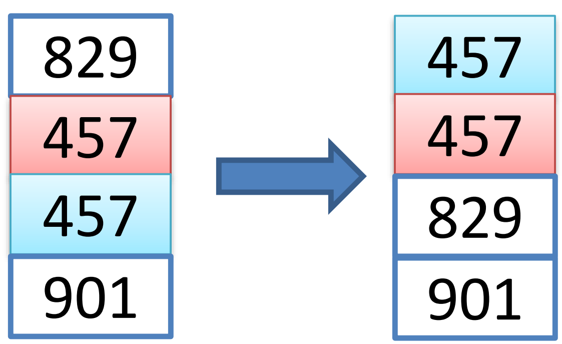
An example (cont’d)

- 3rd pass- Sorting by least significant digit (1’s): no need because there are no tens buckets with more than one number.
- 4th pass- The sorted zero hundreds buckets are concatenated and joined in sequence to give 002, 024, 045, 066, 075, 090, 170, 802
    - Zero bucket: 002
    - Twenties bucket: 024
    - Forties bucket: 045
    - Sixties bucket: 066
    - Seventies bucket: 075
    - Nineties bucket: 090

    - Zero bucket: 045, 075, 090, 002, 024, 066
    - One bucket: 170
    - Eight bucket: 802

Most Significant Digit (MSD) Radix Sort

- Problem:
  - lots of intermediate piles of cards to keep track of
    - *d* digits –> T(n) = O(*dn*)
    - Needs to use many buckets to store intermediate results, each is a linked list of size up to *n*
  - MSD sort does not necessarily preserve the original order of duplicate keys
    - Depending on how we sort the bucket



Least significant digit (LSD) Radix Sort

- Key idea: sort the least significant digit first
- Assume we have d-digit numbers in A

```
RadixSort(A, d)
        for i= 1 to d
            StableSort(A) on digit i
```

Example: LSD Radix Sorting

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

- Can we prove it works?
- Sketch of an inductive argument (induction on the number of passes)
  - Assume lower-order digits {j: j < i } are sorted
  - Show that sorting next digit i leaves array correctly sorted
    - If two digits at position i are different, ordering numbers by that digit is correct (lower-order digits are irrelevant)
    - If they are the same, numbers are already sorted on the lower-order digits. Since we use a stable sort, the numbers stay in the right order

Questions?

- Can we use any sorting algorithms instead of stable sorting in LSD Radix sorting?

Why stable sorting

- 6**5**7 6**5**8 469 595
- If the sorting algorithm is not stable
- First pass: 595 6**5**7 6**5**8 469
- Second pass: 6**5**8 6**5**7 469 595
- Third pass: 469 595 6**5**8 6**5**7

Radix Sort

- What sort will we use to sort on digits?
- Counting sort is obvious choice:
  - Sort n numbers on digits that range from $0..k$
  - Time: $O(n + k)$
- Each pass over n numbers with d digits takes time $O(n+k)$, so the total time $O(dn+dk)$
  - When d is constant and $k= O(n)$, takes $O(n)$ time

How to break words into digits?

- We have *n* word
- Each word is of *b* bits
- We break each word into *r*-bit digits, $d = \lceil b/r \rceil$
- Using counting sort, $k = 2^r - 1$
- E.g., 32-bit word, we break into 8-bit digits
  - $d = \lceil 32/8 \rceil = 4$, $k = 2^8 - 1 = 255$
- $T(n) = \Theta(\, d*(n+k)\,) = \Theta(b/r * (n + 2^r))$

How to choose r?

How to choose r? Balance b/r and $n + 2^r$. Choosing $r \approx \lg n$ gives us $\Theta\left(\frac{b}{\lg n}(n+n)\right) = \Theta(bn/\lg n)$.

Still in O(n)

- If we choose $r < \lg n$, then $b/r > b/\lg n$, and $n + 2^r$ term doesn't improve.
- If we choose $r > \lg n$, then $n + 2^r$ term gets big. Example: $r = 2\lg n \Rightarrow 2^r = 2^{2\lg n} = (2^{\lg n})^2 = n^2$.

Radix Sort Example

- Problem: sort 1 million 64-bit numbers
  - Treat as four-digit radix 2^{16} numbers
  - $r = 16$, $d = 4$, $n = 10^6$, and $b = 64$
  - $b/r * (n + 2^r) = 4{,}262{,}144 \approx 4n$
  - We can sort in just four passes with radix sort!
- Compares well with typical O(n lg n) comparison sort
  - Requires approximately lg n = 20 operations per number being sorted
  - So why would we ever use anything but radix sort?
  - Doesn't sort in place (why?)
  - Depends on implementation, e.g., quicksort uses cache better

Summary: 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, and Simple to code
    - Doesn’t sort in place
    - Depends on implementation, e.g., quicksort uses cache better
    - Cannot easily sort floating point numbers

Bucket Sort

- Assumes the input is generated by a random process that distributes elements uniformly over [0*,* 1*)*.
- ***Idea:***
  - Divide [0*,* 1*)* into *n* equal-sized *buckets*.
  - Distribute the *n* input values into the buckets.
  - Sort each bucket.
  - Then go through buckets in order, listing elements in each one.

Bucket Sort (cont'd)

- Input:
  - $A[1 \mathinner{.\,.} n]$, where $0 \le A[i] < 1$ for all i.
- Auxiliary array:
  - $B[0 \mathinner{.\,.} n-1]$ of linked lists, each list initially empty.

Bucket sort Implementation

BUCKET-SORT(A, n)

for $i \leftarrow 1$ **to** n

  do insert $A[i]$ into list $B[\lfloor n \cdot A[i] \rfloor]$

for $i \leftarrow 0$ **to** $n - 1$

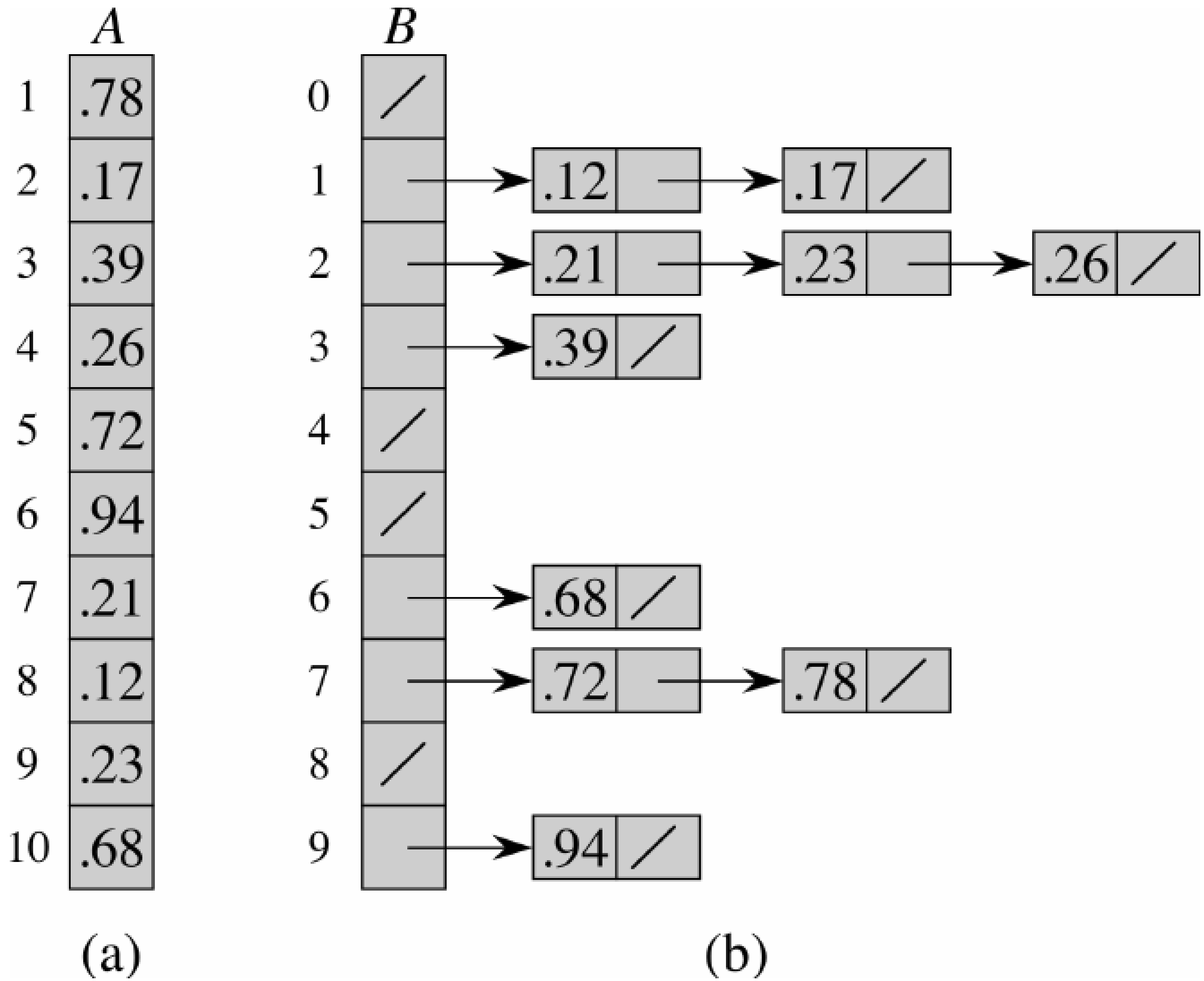
  do sort list $B[i]$ with insertion sort

concatenate lists $B[0], B[1], \ldots, B[n-1]$ together in order

return the concatenated lists

Easily compute the bucket index $\lfloor n \cdot A[\mathrm{i}] \rfloor$

Bucket sort with 10 buckets



Correctness

- Consider $A[i\,]$ and $A[\,j\,]$
  - Assume without loss of generality that $A[i\,] \le A[\,j\,]$
  - Then, bucket index $n \cdot A[i\,] \le n \cdot A[\,j\,]$
- 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

Informal Analysis

- All lines of algorithm except insertion sorting take $\Theta(n)$ altogether
- Since the inputs are uniformly and independently distributed over [0,1), we do not expect many numbers to fall into each bucket
- 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.

Formal Analysis

- Define a random variable:

  n_i = the number of elements placed in bucket $B[i]$
- Because insertion sort runs in quadratic time, bucket sort time is

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

Formal Analysis (Cont'd)

Take expectations of both sides:

$$\begin{aligned} \mathrm{E}[T(n)] &= \mathrm{E}\left[\Theta(n) + \sum_{i=0}^{n-1} O(n_i^2)\right] \\ &= \Theta(n) + \sum_{i=0}^{n-1} \mathrm{E}[O(n_i^2)] && \text{(linearity of expectation)} \\ &= \Theta(n) + \sum_{i=0}^{n-1} O(\mathrm{E}[n_i^2]) && (\mathrm{E}[aX] = a\mathrm{E}[X]) \end{aligned}$$

n_i = the number of elements placed in bucket *B*[*i*]

n_i = the number of elements placed in bucket $B[i\,]$

Claim

$\mathrm{E}\,[n_i^2] = 2 - (1/n)$ for $i = 0, \ldots, n-1$.

Proof of claim

Define indicator random variables:

- $X_{ij} = \mathrm{I}\,\{A[j] \text{ falls in bucket } i\}$
- $\Pr\,\{A[j] \text{ falls in bucket } i\} = 1/n$
- $n_i = \sum_{j=1}^{n} X_{ij}$

$$X_{i,j} = \mathrm{I}\{A[j] \text{ falls in bucket } i\}.$$

$$= \begin{cases} 1 \text{ if } A[j] \text{ falls in bucket } i \\ 0 \text{ if } A[j] \text{ doesn't fall in bucket } i \end{cases}$$

The Claim

Then

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

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

$$= \sum_{j=1}^{n} \mathrm{E}\left[X_{ij}^2\right] + 2\sum_{j=1}^{n-1}\sum_{k=j+1}^{n} \mathrm{E}[X_{ij}X_{ik}] \quad \text{(linearity of expectation)}$$

$(x_1+x_2+x_3)(x_1+x_2+x_3)$

$= x_1^2 + x_1x_2 + x_1x_3$

$+ x_2^2 + x_1x_2 + x_2x_3$

$+ x_3^2 + x_1x_3 + x_2x_3$

$$\mathrm{E}\left[X_{ij}^2\right] = 0^2 \cdot \Pr\{A[j] \text{ doesn't fall in bucket } i\} + 1^2 \cdot \Pr\{A[j] \text{ falls in bucket } i\}$$

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

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

Analysis

$\mathrm{E}[X_{ij}X_{ik}]$ for $j \neq k$: Since $j \neq k$, X_{ij} and X_{ik} are independent random variables

$$\begin{aligned}\Rightarrow \mathrm{E}[X_{ij}X_{ik}] &= \mathrm{E}[X_{ij}]\,\mathrm{E}[X_{ik}] \\ &= \frac{1}{n}\cdot\frac{1}{n} \\ &= \frac{1}{n^2}\end{aligned}$$

Therefore:

$$\mathrm{E}[n_i^2] = \sum_{j=1}^{n}\frac{1}{n} + 2\sum_{j=1}^{n-1}\sum_{k=j+1}^{n}\frac{1}{n^2}$$

Analysis (Cont'd)

$$
\begin{aligned}
&= n \cdot \frac{1}{n} + 2\binom{n}{2}\frac{1}{n^2} \\
&= 1 + 2 \cdot \frac{n(n-1)}{2} \cdot \frac{1}{n^2} \\
&= 1 + \frac{n-1}{n} \\
&= 1 + 1 - \frac{1}{n} \\
&= 2 - \frac{1}{n} \qquad\qquad \blacksquare \text{ (claim)}
\end{aligned}
$$

Therefore:

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

Analysis conclusion

- This is a probabilistic analysis
  - We used probability to analyze an algorithm whose running time depends on the distribution of inputs.
- With bucket sort, if the input isn't drawn from a uniform distribution on [0*, 1)*, all bets are off
  - Performance-wise, but the algorithm is still correct

Bucket Sort Summary

- Assumption: input is *n* real #'s from [0, 1)
  - We can map other number into the range of [0, 1)
- Basic idea:
  - Create *n* linked lists (*buckets*) to divide interval [0,1) into 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)

Linear Sorting Common Mistakes

- Using counting sort, when memory is limited
  - The size of k → the size of C[0..k]
- Using bucket sort, when the input are not uniform distributed

Linear-time Sorting Summary

- We have learned three linear-time sorting algorithms
- Their assumptions on input
  - Counting sort → [0..k]
  - Radix sort → d digits
  - Bucket sort → uniform distribution [0, 1)