Unit 3.2 Sorts

Algorithms

EE/NTHU

Apr. 7, 2020

Sorting Problem

- Given a set of n elements, arrange the elements in a nondecreasing order.
- For simplicity, we assume the input is an array $A[1:n]$ of n elements.
- The brute force approach to solve the sorting problem has been demonstrated in the `SelectionSort` algorithm.
- The time complexity is $\mathcal{O}(n^2)$ due to two nested `for` loops.
- In this unit, we will study more sorting algorithms such that appropriate sorting algorithms can be applied for specific problems to gain the best efficiency.

Merge Sort

- Merge Sort is a good example of divide and conquer approach.

Algorithm 3.2.1. Merge Sort

```
// Sort A[low : high] into nondecreasing order.
// Input: array A, int low, high
// Output: A rearranged.
Algorithm MergeSort(A, low, high)
{
    if (low < high) then {
        mid := ⌊(low + high)/2⌋ ;
        MergeSort(A, low, mid) ;
        MergeSort(A, mid + 1, high) ;
        Merge(A, low, mid, high) ;
    }
}
```

- This algorithm should be invoked by MergeSort$(A, 1, n)$ in the main function.
- The following algorithm assumes a global array $B[1 : n]$ of n elements and uses it as a temporary storage.

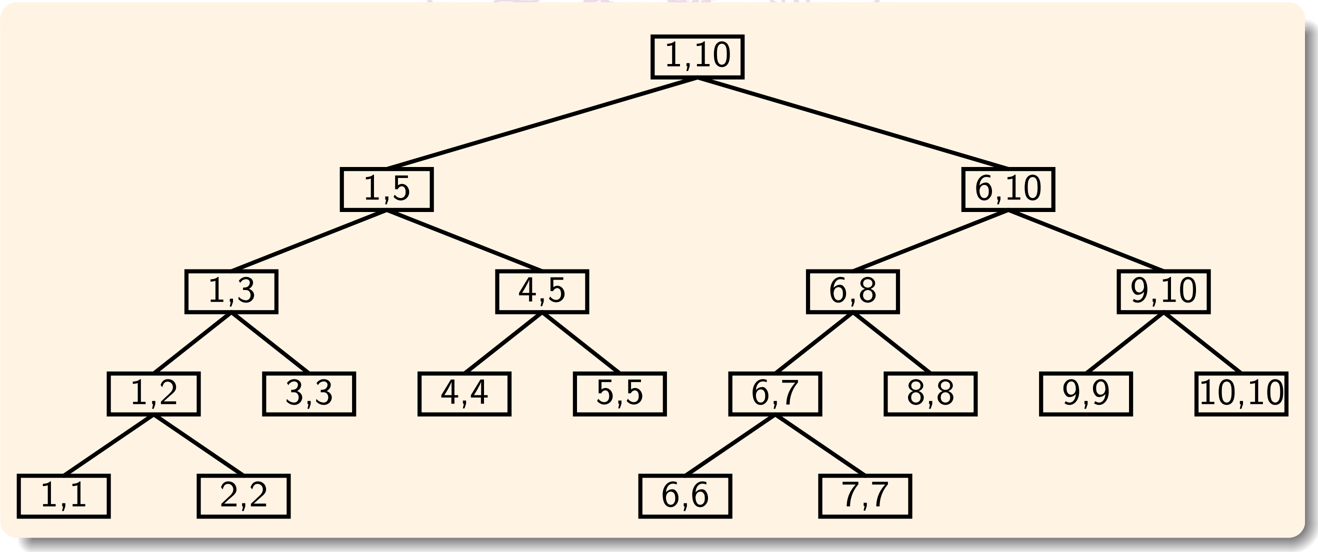
Merge Sort – Merge

Algorithm 3.2.2. Merge Process

```
// Merge sorted A[low : mid] and A[mid + 1 : high] to nondecreasing order.
// Input: A[low : high], int low, mid, high
// Output: A[low : high] sorted.
Algorithm Merge(A, low, mid, high)
{
    h := low ; i = low ; j := mid + 1 ; // Initialize looping indices.
    while ((h ≤ mid) and (j ≤ high)) do { // Store smaller one to B[i].
        if (A[h] ≤ A[j]) then { // A[h] is smaller.
            B[i] := A[h] ; h := h + 1 ;
        } else { // A[j] is smaller.
            B[i] := A[j] ; j := j + 1 ;
        }
        i := i + 1 ;
    }
    if (h > mid) then // A[j : high] remaining.
        for k := j to high do {
            B[i] = A[k] ; i := i + 1 ;
        }
    else // A[h : mid] remaining.
        for k := h to mid do {
            B[i] := A[k] ; i := i + 1 ;
        }
    for k := low to high do A[k] := B[k] ; // Copy B to A.
}
```

Merge Sort – Divide-and-Merge Recursion

- The following tree shows the divide-and-merge recursion.
  - Array A is assumed to have 10 elements.



Merge Sort – Example

- Example

$A = \{$	310,	285,	179,	652,	351,	423,	861,	254,	450,	520	$\}$
	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	

- A is partitioned into two sets and each set is sorted into nondecreasing order
  - This process is carried out through the recursive calls

$A = \{$	179,	285,	310,	351,	652,	254,	423,	450,	520,	861	$\}$
	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	

- Merging the two sets together

$A = \{$	$\underline{179}$,	285,	310,	351,	652,	$\underline{254}$,	423,	450,	520,	861	$\}$
	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	
$B = \{$	179,										$\}$

$A = \{$	179,	$\underline{285}$,	310,	351,	652,	$\underline{254}$,	423,	450,	520,	861	$\}$
	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	
$B = \{$	179,	254,									$\}$

Merge Sort – Example II

$A = \{$	179,	$\underline{285}$,	310,	351,	652,	254,	$\underline{423}$,	450,	520,	861	$\}$
	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	
$B = \{$	179,	254,	285,								$\}$
$A = \{$	179,	285,	$\underline{310}$,	351,	652,	254,	$\underline{423}$,	450,	520,	861	$\}$
	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	
$B = \{$	179,	254,	285,	310,							$\}$
$A = \{$	179,	285,	310,	$\underline{351}$,	652,	254,	$\underline{423}$,	450,	520,	861	$\}$
	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	
$B = \{$	179,	254,	285,	310,	351,						$\}$
$A = \{$	179,	285,	310,	351,	$\underline{652}$,	254,	$\underline{423}$,	450,	520,	861	$\}$
	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	
$B = \{$	179,	254,	285,	310,	351,	423,					$\}$
$A = \{$	179,	285,	310,	351,	$\underline{652}$,	254,	423,	$\underline{450}$,	520,	861	$\}$
	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	
$B = \{$	179,	254,	285,	310,	351,	423,	450,				$\}$

Merge Sort – Example III

$A = \{$	179,	285,	310,	351,	$\underline{652}$,	254,	423,	450,	$\underline{520}$,	861	$\}$
	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	
$B = \{$	179,	254,	285,	310,	351,	423,	450,	520,			$\}$
$A = \{$	179,	285,	310,	351,	$\underline{652}$,	254,	423,	450,	520,	$\underline{861}$	$\}$
	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	
$B = \{$	179,	254,	285,	310,	351,	423,	450,	520,	652,		$\}$
$A = \{$	179,	285,	310,	351,	652,	254,	423,	450,	520,	$\underline{861}$	$\}$
	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	
$B = \{$	179,	254,	285,	310,	351,	423,	450,	520,	652,	861	$\}$

- Then B is copied into A to get the sorted result.

Merge Sort – Complexity

- Let $T(n)$ be the computing time of merge sort applying to a data set of n elements, then

$$T(n) = \begin{cases} a, & n = 1, a \text{ is a constant,} \\ 2\,T(n/2) + c \cdot n, & n > 1, c \text{ is a constant.} \end{cases} \tag{3.2.1}$$

- If $n = 2^k$, then

$$\begin{aligned} T(n) &= 2\Big(2\,T(n/4)) + c \cdot n/2\Big) + c \cdot n \\ &= 4\,T(n/4) + 2c \cdot n \\ &= 4\Big(2\,T(n/8) + c \cdot n/4\Big) + 2c \cdot n \\ &= 2^k\,T(1) + k \cdot c \cdot n \\ &= a \cdot n + c \cdot n \cdot \lg n \end{aligned} \tag{3.2.2}$$

- If $2^k < n \le 2^{k+1}$, then $T(n) \le T(2^{k+1})$. Therefore,

$$T(n) = \mathcal{O}(n \lg n). \tag{3.2.3}$$

Merge Sort – Improvement

- The Merge Sort, Algorithm (3.2.1), continues to divide the input into smaller subsets until each subset contains only one element.
- The CPU time mostly spent on recursive function calls
  - With each function doing very little operations
- This inefficiency can be improved as the following

Algorithm 3.2.3. Improved Merge Sort

```
    // Sort A[low : high] into nondecreasing order with better efficiency.
    // Input: A, int low, high
    // Output: rearranged A.
Algorithm MergeSort1(A, low, high)
{
    if (high − low < 15) then // When A is small, perform insertion sort.
        return InsertionSort(A, low, high) ;
    else { // For large A, divide-and-conquer merge sort.
        mid := ⌊ (low + high)/2⌋ ;
        MergeSort(A, low, mid) ;
        MergeSort(A, mid + 1, high) ;
        Merge(A, low, mid, high) ;
    }
}
```

Insertion Sort

Algorithm 3.2.4. Insertion Sort

```
   // Sort A[low : high] into nondecreasing order.
   // Input: A, int low, high
   // Output: rearranged A.
1  Algorithm InsertionSort(A, low, high)
2  {
3      for j := low + 1 to high do { // Check for every low < j ≤ high
4          item := A[j]; // Compare A[i] and A[j], i < j.
5          i := j − 1;
6          while ((i ≥ low) and (item < A[i])) do { // Make room for item = A[j]
7              A[i + 1] := A[i];
8              i := i − 1;
9          }
10         A[i + 1] = item; // Store item.
11     }
12 }
```


Insertion Sort, Complexity

- Line 6 can be executed at most $j - low$ times, thus the time complexity is

$$T(n) = \sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1 = \mathcal{O}(n^2). \tag{3.2.4}$$

  - The best-case complexity is $\Theta(n)$.
- Though the InsertionSort has high computation time complexity, for small n this function executes very fast.
  - Note that the number 15 can be fine tuned to gain better efficiency.
- Thus, Algorithm MergeSort1 is usually more efficient.

Quick Sort

- Another divide and conquer approach in sorting an array.

Algorithm 3.2.5. Quick Sort

```
// Sort A[low : high] into nondecreasing order.
// Input: A[low : high], int low, high
// Output: A[low : high] sorted.
Algorithm QuickSort(A, low, high)
{
    if (low < high) then {
        mid := Partition(A, low, high + 1) ;
        QuickSort(A, low, mid − 1) ;
        QuickSort(A, mid + 1, high) ;
    }
}
```


Quick Sort, II

Algorithm 3.2.7. Partition

```
// Partition A into A[low : mid − 1] ≤ A[mid] and A[mid + 1 : high] ≥ A[mid].
// Input: A, int low, high
// Output: j that A[low : j − 1] ≤ A[j] ≤ A[j + 1 : high].
Algorithm Partition(A, low, high)
{
    v := A[low] ; i := low; j := high; // Initialize
    repeat { // Check for all elements.
        repeat i := i + 1 ; until (A[i] ≥ v) ; // Find i such that A[i] ≥ v.
        repeat j := j − 1 ; until (A[j] ≤ v) ; // Find j such that A[j] ≤ v.
        if (i < j) then Swap(A, i, j) ; // Exchange A[i] and A[j].
    } until (i ≥ j) ;
    A[low] := A[j] ; A[j] = v; // Move v to the right position.
    return j;
}
Algorithm Swap(A, i, j)
{
    t := A[i] ; A[i] := A[j] ; A[j] := t;
}
```

Partition Example

[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	i	j
65	70	75	80	85	60	55	50	45	$+\infty$	2	9
65	45	75	80	85	60	55	50	70	$+\infty$	3	8
65	45	50	80	85	60	55	75	70	$+\infty$	4	7
65	45	50	55	85	60	80	75	70	$+\infty$	5	6
65	45	50	55	60	85	80	75	70	$+\infty$	6	5
60	45	50	55	65	85	80	75	70	$+\infty$		

- Algorithm `Partition` returns $j = 5$.
- Note that

$$\begin{array}{ll} A[i] \le A[j], & \text{if } i < j, \\ A[i] \ge A[j], & \text{if } i > j. \end{array}$$

- Therefore, `QuickSort` can be applied to $A[low : j-1]$ and $A[j+1 : high]$
- Also note that $A[high+1] = \infty$ is assumed.
  - For the next recursion level
    - $A[j]$ serves as $a[high+1]$ in `QuickSort`$(A, low, j-1)$
    - $A[high+1]$ is still used in `QuickSort`$(A, j+1, high)$.

Quick Sort – Complexity

- Assume that element comparison dominates the CPU time
- The number of element comparisons in `Partition` algorithm is $high - low + 1$
- Worst-case complexity
  - At the top level, `Partition`$(A, 1, n+1)$ is called with $n+1$ comparisons
  - At the next level, the worst-case scenario has one of the partition with $n-1$ elements and n comparisons
  - Thus the total number of comparisons would be

$$C_W(n) = \sum_{i=2}^{n} (i+1) = \mathcal{O}(n^2) \qquad (3.2.5)$$

Quick Sort – Complexity, II

- Average-case complexity

$$C_A(n) = n + 1 + \frac{1}{n}\sum_{k=1}^{n}\Big(C_A(k-1) + C_A(n-k)\Big) \tag{3.2.6}$$

- Note that $C_A(0) = C_A(1) = 0$, and

$$nC_A(n) = n(n+1) + 2\Big(C_A(0) + C_A(1) + \cdots + C_A(n-1)\Big) \tag{3.2.7}$$

Replacing n by $n-1$, we have

$$(n-1)C_A(n-1) = n(n-1) + 2\Big(C_A(0) + C_A(1) + \cdots + C_A(n-2)\Big) \tag{3.2.8}$$

Subtract Eq. (3.2.8) from Eq. (3.2.7)

$$nC_A(n) - (n-1)C_A(n-1) = 2n + 2C_A(n-1)$$

$$nC_A(n) = (n+1)C_A(n-1) + 2n$$

$$\frac{C_A(n)}{n+1} = \frac{C_A(n-1)}{n} + \frac{2}{n+1} \tag{3.2.9}$$

Quick Sort – Complexity, III

$$\begin{aligned}
\frac{C_A(n)}{n+1} &= \frac{C_A(n-1)}{n} + \frac{2}{n+1} \\
&= \frac{C_A(n-2)}{n-1} + \frac{2}{n} + \frac{2}{n+1} \\
&= \frac{C_A(n-3)}{n-2} + \frac{2}{n-1} + \frac{2}{n} + \frac{2}{n+1} \\
&= \frac{C_A(1)}{2} + 2\sum_{k=3}^{n+1}\frac{1}{k} \\
&= 2\sum_{k=3}^{n+1}\frac{1}{k} \qquad (3.2.10)
\end{aligned}$$

$$\sum_{k=3}^{n+1}\frac{1}{k} \le \int_2^{n+1}\frac{1}{x}\,\mathrm{d}x = \log(n+1) - \log(2)$$

And, we have

$$C_A(n) \le 2(n+1)\Big(\log(n+1) - \log(2)\Big) = \mathcal{O}(n\log n) = \mathcal{O}(n\lg n) \tag{3.2.11}$$

Quick Sort – Space Complexity

- Note that for small n, the `InsertionSort` can be very fast and the `QuickSort` can be combined with `InsertionSort` to gain better performance (the same way as the `MergeSort` case).

- Let the stack space needed by the `QuickSort`$(A, low, high)$ is $S(n)$
- Worst-case: the number of recursion is $n-1$, thus

$$S_W(n) = 2 + S_W(n-1) = \mathcal{O}(n) \tag{3.2.12}$$

- Best-case:

$$S_B(n) = 2 + S_W(\lfloor (n-1)/2 \rfloor) = \mathcal{O}(\lg n) \tag{3.2.13}$$

- Average-case: it can be shown that

$$S_A(n) = \mathcal{O}(\lg n). \tag{3.2.14}$$

Randomized Quick Sort

- If the array A is already in order, then the `QuickSort` can have worst-case performance.
- The following randomized `QuickSort` can improve the performance.

Algorithm 3.2.8. Randomized Quick Sort

```
// Sort A[low : high] into nondecreasing order.
// Input: A[low : high], int low, high
// Output: A[low : high] sorted.
Algorithm RQuickSort(A, low, high)
{
    if (low < high) then {
        if ((high − low) > 5) then
            Swap(A, low + ( Random() mod (high − low + 1)), low) ;
        mid := Partition(A, low, high + 1) ;
        QuickSort(A, low, mid − 1) ;
        QuickSort(A, mid + 1, high) ;
    }
}
```

Comparison-based Sorts

- We have studied several sorting algorithms, and here are their time complexities
  - Selection Sort: $\Theta(n^2)$,
  - Heap sort: worst-case $\mathcal{O}(n \lg n)$,
  - Merge sort: worst-case $\mathcal{O}(n \lg n)$,
  - Quick sort: average-case $\mathcal{O}(n \lg n)$.
- All these algorithms use element comparisons as the basic operations to sort the array.
- It can be shown that using comparison based sorting algorithms the best time complexity one can get is $\mathcal{O}(n \lg n)$.
- In the following, we digress from the divide and conquer approach to show some linear time sorting algorithms.
- These algorithms do not use comparison operations and thus they can achieve even low time complexities.

Counting Sort

- The counting sort assumes the elements to be sorted are all integers in the range $[1 : k]$.
- Thus, if array $A[1 : n]$ contains the integer elements to be sorted, $1 \leq A[i] \leq k$, $1 \leq i \leq n$.
- Let $C[1 : k]$ be an array. Then we can place $A[i]$ into array $C[A[i]]$.
- After that is done, we simply trace C array once to get the sorted order.
- Example: $n = 8$, $A[1 : 8] = \{2, 5, 3, 1, 2, 3, 1, 3\}$ to be sorted. Then, we need a $C[1 : 5]$ array to perform counting sort.
  Placing $A[i]$ elements into C array, we have

			$A[8]$		
	$A[7]$	$A[5]$	$A[6]$		
	$A[4]$	$A[1]$	$A[3]$		$A[2]$
C array:	1	2	3	4	5

Thus, the sorting result is: $A[4]$, $A[7]$, $A[1]$, $A[5]$, $A[3]$, $A[6]$, $A[8]$, $A[2]$,
which is: 1, 1, 2, 2, 3, 3, 3, 5.

Counting Sort – Algorithm

Algorithm 3.2.9. Counting Sort.

```
   // Sort A[1 : n] and put results into B[1 : n]. Assume 1 ≤ A[i] ≤ k, ∀i.
   // Input: A, int n, k
   // Output: B contains sorted results.
1  Algorithm CountingSort(A, B, n, k)
2  {
3      for i := 1 to k do { // Initialize C to all 0.
4          C[i] := 0;
5      }
6      for i := 1 to n do { // Count # elements in C[A[i]].
7          C[A[i]] := C[A[i]] + 1;
8      }
9      for i := 1 to k do { // C[i] is the accumulate # of elements.
10         C[i] := C[i] + C[i − 1];
11     }
12     for i := n to 1 step −1 do { // Store sorted order in array B.
13         B[C[A[i]]] := A[i];
14         C[A[i]] := C[A[i]] − 1;
15     }
16 }
```


Counting Sort – Example

- Example: $n = 8$, $A[1:8] = \{2, 5, 3, 1, 2, 3, 1, 3\}$ to be sorted.

Line 8:

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

	1	2	3	4	5
C	2	2	3	0	1

Line 11:

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

	1	2	3	4	5
C	2	4	7	7	8

Line 15: $i = 8$

	1	2	3	4	5	6	7	8
B							3	

	1	2	3	4	5
C	2	4	6	7	8

Line 15: $i = 7$

	1	2	3	4	5	6	7	8
B		1					3	

	1	2	3	4	5
C	1	4	6	7	8

Line 15: $i = 6$

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

	1	2	3	4	5
C	1	4	5	7	8

Counting Sort – Example

Line 11:

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

	1	2	3	4	5
C	2	4	7	7	8

Line 15: $i = 5$

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

	1	2	3	4	5
C	1	3	5	7	8

Line 15: $i = 4$

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

	1	2	3	4	5
C	0	3	5	7	8

Line 15: $i = 3$

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

	1	2	3	4	5
C	0	3	4	7	8

Line 15: $i = 2$

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

	1	2	3	4	5
C	0	3	4	7	7

Line 15: $i = 1$

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

	1	2	3	4	5
C	0	2	4	7	7

Counting Sort – Complexity

- Four `for` loops in `CountingSort` algorithm
  - Lines 3-5, $\Theta(k)$,
  - Lines 6-8, $\Theta(n)$,
  - Lines 9-11, $\Theta(k)$,
  - Lines 12-15, $\Theta(n)$,
  - Overall time complexity, $\Theta(n + k)$
    - When $k = \mathcal{O}(n)$, then it is $\Theta(n)$.
- Thus, counting sort has the time complexity lower than $\mathcal{O}(n \lg n)$.
  - This is because that counting sort is not a comparison-based sorting algorithm.
- Algorithm `CountingSort` is stable.
  - A sorting algorithm is stable if the elements of the same value appear in the output in the same order as they do in the input array.

Radix Sort

- Given a set of n integers of d digits, then radix sort, which uses counting sort function, can perform the sorting efficiently.
- For the d digits, let the least significant digit be digit 1, and the most significant digit be digit d.

Algorithm 3.2.10. Radix sort.

```
// Sort the n d-digit integers in array A.
// Input: A, int n, d
// Output: sorted A.
Algorithm RadixSort(A, d)
{
    for i := 1 to d do {
        Sort array A by digit i using CountingSort ;
    }
}
```

- Note that any stable sort can be used in position of CountingSort and one gets the same result.

Radix Sort, Example

- Example

Original order	Sort digit 1	Sort digit 2	Sort digit 3
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

Lemma 3.2.11.

Given n d-digit numbers in which each digit can take on up to k possible values, then RadixSort correctly sorts these numbers in $\Theta(d \times (n + k))$ time since CountingSort takes $\Theta(n + k)$ time for each digit.

- This lemma can be easily generalized for any stable sort to be used in the place of CountingSort.

Radix Sort, Property

Lemma 3.2.12.

Given n b-bit numbers and any positive integer $r \le b$, RadixSort correctly sorts these numbers in $\Theta((b/r) \times (n + 2^r))$ time since CountingSort takes $\Theta(n + k)$ time for inputs in the range 0 to k.

Proof. For any $r \le b$, one can divide the b-bit numbers to $d = \lceil b/r \rceil$ digits of r bits each. Each digit is an integer in the range of 0 to $2^r - 1$, so one can use CountingSort with $k = 2^r$. Therefore, sorting those numbers takes $\Theta(d \times (n + 2^r)) = \Theta((b/r) \times (n + 2^r))$ time. □

- Again, any stable sort can be used in place of CountingSort.
- RadixSort, which is not comparison based algorithm, has lower complexity, $\Theta(n)$, while the best comparison based algorithm achieve $\mathcal{O}(n \lg n)$ time.
- In using RadixSort on sets with large size, memory swapping can be a limiting factor for performance. On the other hand, many comparison based algorithm using in-place sorts can have much fewer memory swapping.
- Computer hardware and compiler can impact on the performance of these algorithms.

Bucket Sort

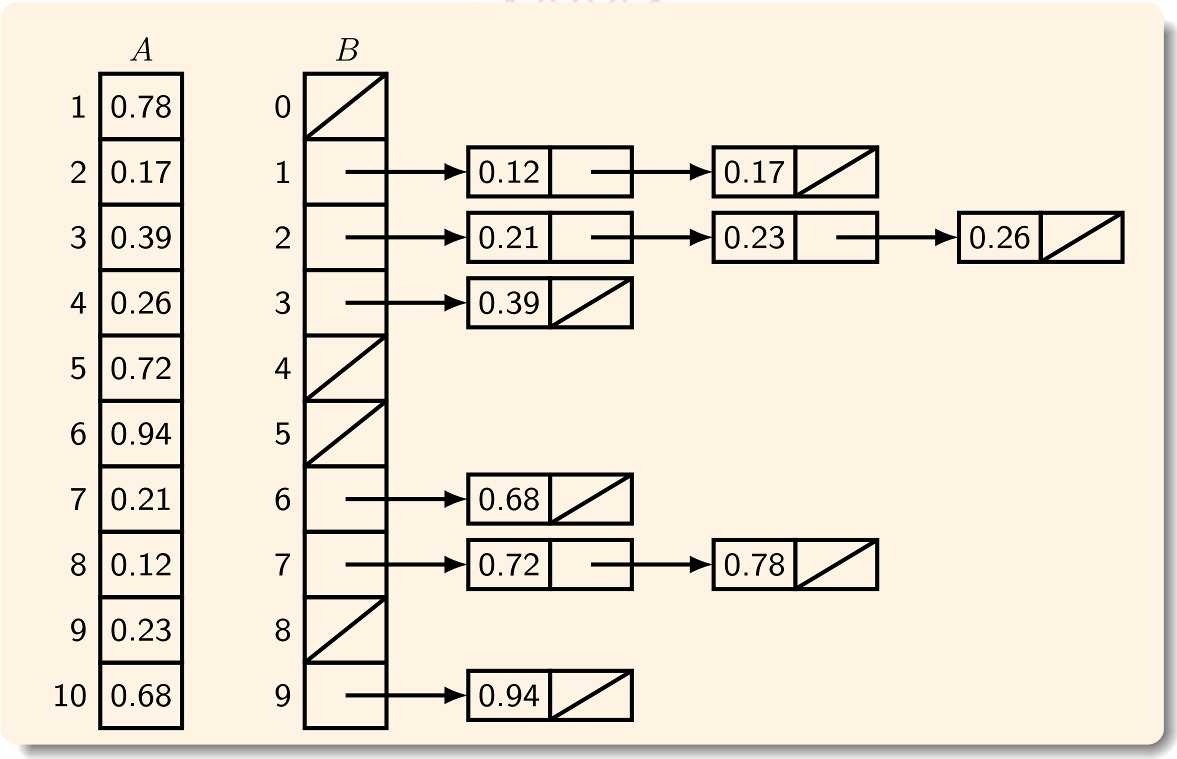
- BucketSort is an average-case $\mathcal{O}(n)$ sorting algorithm.
- BucketSort is not comparison based algorithm and it assumes the n numbers being sorted in uniformly distributed in the range $[0, 1)$.

Algorithm 3.2.13. Bucket Sort.

```
   // Sort n-element array A assuming A[i] is uniformly in [0,1).
   // Input: A, int n
   // Output: sorted A.
Algorithm BucketSort(A, n)
{
    Initialize array B[0 : n−1] to be all NULL;
    for i := 1 to n do // Append A[i] to B[⌊n×A[i]⌋]
        insertList(B[⌊n×A[i]⌋], A[i]);
    for i := 0 to n−1 do // Sort each B[i].
        insertionSort(B[i]);
    concatenate n lists B[0 : n−1] and store back to array A;
}
```

Bucket Sort, Example

- Example



Bucket Sort, Complexities

- In BucketSort
  - Line 3 executes n times
  - Loop in line 4 executes n times
  - Line 8 also executes n times
  - Line 6 loop executes n times
    - Each InsertionSort executes n_i^2 times
    - But, $E(n_i) = \mathcal{O}(1)$, therefore this loop executes $\mathcal{O}(n)$ times
- Overall time complexity: $\mathcal{O}(n)$.
- Space complexity is also $\mathcal{O}(n)$.
  - Array B is $\mathcal{O}(n)$,
  - Linked list is $\mathcal{O}(n)$.
- Note the assumption of the elements are uniformly distributed in a range.
- If the range is not $[0, 1)$, it can be scaled and the BucketSort can still perform well.

Summary

- Sorting problem.
- Comparison-based sorts.
  - Merge sort.
    - Improved merge sort.
  - Insertion sort.
  - Quick sort.
    - Randomized quick sort.
- Non-comparison based sorts.
  - Counting sort.
  - Radix sort.
  - Bucket sort.