SE2205: Algorithms and Data Structures for Object-Oriented Design

Sorting Algorithms

Dr. Pirathayini Srikantha

Western University

April 3, 2019

Readings/References

- Standish (13)

Table of Contents

Table of Contents

Introduction

- **Sorting** has been heavily studied in the literature as it applies to a large number of problems and there are many ways to solve these
- **Objective**: Given an array A of n unsorted keys, sort these in ascending order
- Two main classes of sorting that will be examined are: **comparison-based** and **address calculation**
- **Comparison-based** sorting: Keys to be sorted are compared with one another and reorganized appropriately
  - Basic sorting: bubble sort, insertion sort
  - Priority queue sorting: selection sort, heap sort
  - Divide and conquer sorting: merge sort and quick sort
- **Address-calculation based** sorting: Keys are mapped to addresses close to the final resting position of the sorted key
  - Radix sort

Introduction

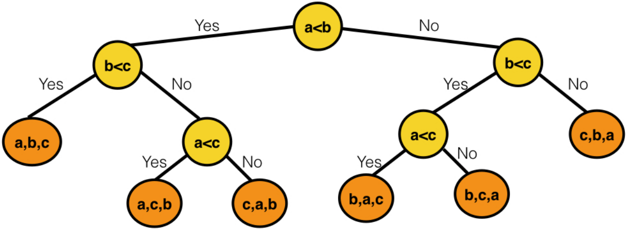
Minimum Complexity of Comparison-Based Sorting

Minimum Complexity of Comparison-Based Sorting

- Suppose that n keys are to be **sorted**
- There are n **possibilities** for the first position in the sorted array
- Since one key has been selected for the first spot, there are $n-1$ options for the second position
- Repeating this for all n positions, the number of possible ordering of keys are $n = n * (n-1) * (n-2) * \ldots * 1 = n!$
- Hence there are $n!$ **permutations** or possibilities of key-orderings
- Assuming that the keys are **distinct**, only one out of this $n!$ possibilities will result in the **sorted ordering**

Minimum Complexity of Comparison-Based Sorting

- Comparison-key sorting can be represented as a binary tree (see below for an example of a **comparison** tree for $A = \{a, b, c\}$)
- Note that the number of leafs are $n! = 6$ as these are all possible combinations of 3 distinct keys
- The **height** of the tree determines the **complexity** of the algorithm
- The minimum height h of a tree with k leafs is $h \geq \lfloor log(k) \rfloor$
- An **efficient** comparison-based algorithm will maintain this height



Minimum Complexity of Comparison-Based Sorting

- In order to simplify $h_{min} = \lfloor log(n!) \rfloor$, can use the **Stirling's** formula

$$ln(n!) \approx (n + \frac{1}{2}) ln(n) + O(n)$$

- Using the stirling's formula, it can be shown that

$$O(ln(n!)) = O(nlog(n))$$

- This is the **minimal complexity** of the comparison-based method (i.e. lower bound)
- This is **not** the case for all comparison-based sorting algorithms as we will see next

Table of Contents

Basic Sorting

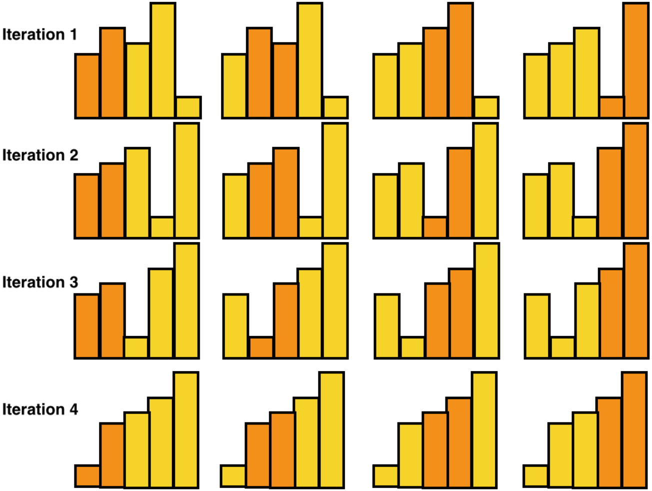
- **Bubble sort** and **insertion sort** are classified under the heading *basic sort* as these algorithms are simple to implement
- As you will see next, one major problem with these algorithms is that these do not have **good performance efficiencies**
- Bubble sort is **notorious** for how bad its performance generally is (except for the cases where the array is almost sorted)

Bubble Sort

Bubble Sort Basics

- The basic idea behind bubble sort is the following
- The algorithm makes repeated passes through the unsorted array A and **bubbles up** larger keys until no more bubbling up is possible
- The bubble sort algorithm is composed of the following main steps:
  1. Pass through the entire array and exchange adjacent elements when the left element is larger than the right element
  2. Repeat the above until the algorithm reaches a point where no more exchanges are necessary

Bubble Sort: Example



Bubble Sort: Implementation

```
public void bubbleSort(int[] A){
    int n=A.length, temp;
    boolean done=false;
    do{
        done=true;
        for (int i=0; i<n-1; i++) {
            if (A[i]>A[i+1]) {
                temp=A[i];
                A[i]=A[i+1];
                A[i+1]=temp;
                done=false;
            }
        }
    }while(!done);
}
```

1 2

1: Passing through the entire array bubbling up elements

2: Repeat the bubbling up until no more exchanges are possible

Complexity of Bubble Sort

- Cost of 1: an
- Cost of 2: $(an + b)n$
- Overall worst case cost of bubble sort:

$$f(n) = an^2 + bn + c$$

- Exercise: Formally prove that $f(n) = O(g(n))$ where $g(n) = n^2$

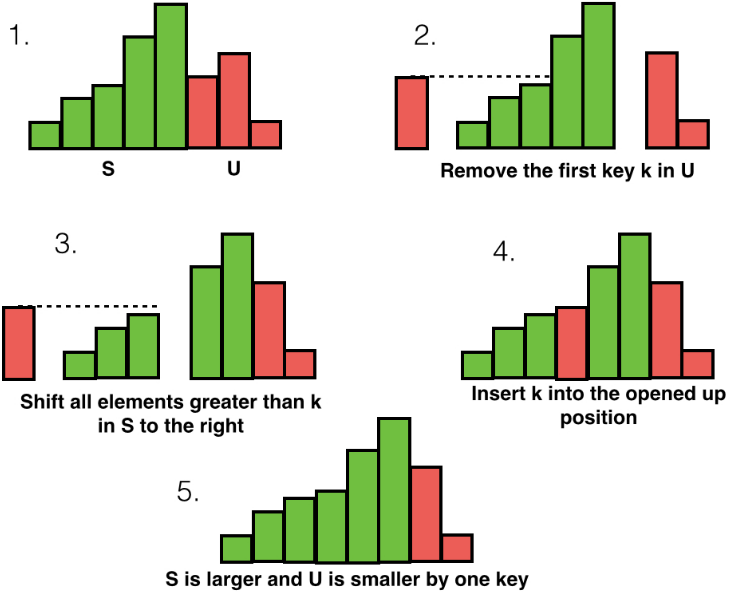
```
public void bubbleSort(int[] A){
    int n=A.length, temp;
    boolean done=false;
    do{
        done=true;
        for (int i=0; i<n-1; i++) {
            if (A[i]>A[i+1]) {
                temp=A[i];              1    2
                A[i]=A[i+1];
                A[i+1]=temp;
                done=false;
            }
        }
    }while(!done);
}
```

Insertion Sort

Insertion Sort

- **Insertion sort** maintains two sub-arrays S and U within the original array A
- S is located on the left portion of A and contains all elements that are **sorted** in ascending order
- U is located on the right portion of A and contains all elements that are **unsorted**
- The general steps in insertion sort are the following:
    1. Remove the left-most key k from U creating a space/hole
    2. Keep comparing k with elements in S until an element that is larger than k is encountered
    3. Shift all keys in S from that point one space to the right
    4. Insert k in the spot that has opened up
    5. Keep repeating the above until S encompasses all elements in A

Insertion Sort: An Example



Insertion Sort: Implementation

```
public void insertionSort(int[] A){
    int k,j,n=A.length;
    boolean done = false;

    for (int i=1; i<n; i++) {
        k=A[i];
        j=i;                                    1
        if(A[j-1]<k)
          done=true;
        else
          done=false;
        while (!done) {
            A[j]=A[j-1];
            j--;                                        5
            if (j>0){
              if(A[j-1]<k)                    2,3
                done=true;
              else
                done=false;
            }
            else{
                done=true;                    4
            }
        }
        A[j]=k;
    }
}
```

1: Remove the left most key k from the unsorted array U encompassing all elements in A at the beginning
2,3: Shift all elements in the sorted array that are greater than k to the right
4: Insert k into the space that has opened up
5: Repeat until all elements are in S

Complexity of Insertion Sort

- Cost of 1: a, Cost of 2,3: bi and Cost of 4: c
- Cost of 5: $\sum_{i=1}^{n-1}(a + bi + c)$
- Cost of insertion sort algorithm:

$$f(n) = \sum_{i=1}^{n-1}(a + bi + c) + d$$

- Exercise: Formally prove that $f(n) = O(g(n))$ where $g(n) = n^2$

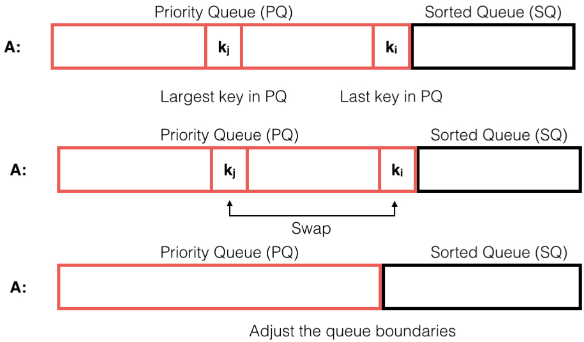
```
public void insertionSort(int[] A){
    int k,j,n=A.length;
    boolean done = false;

    for (int i=1; i<n; i++) {
        k=A[i];
        j=i;
        if(A[j-1]<k)                        1
          done=true;
        else
          done=false;
        while (!done) {
            A[j]=A[j-1];
            j--;                                   5
            if (j>0){
              if(A[j-1]<k)                  2,3
                done=true;
              else
                done=false;
            }
            else{
                done=true;                  4
            }
        }
        A[j]=k;
    }
```

Table of Contents

Priority Queue Based Sorting

- **Priority Queue**: Defined to be a queue in which a delete removes the largest element in the queue
- In priority queue based sorting, the main array A is divided into **two sub-arrays**: priority queue (PQ) and sorted queue (SQ)
- At every iteration:
  - An element from PQ is removed and inserted to the end of the PQ
  - The boundary of PQ is reduced and the boundary of SQ is enlarged



Selection Sort

Selection Sort

Priority Queue Based: Selection Sort

- The only restriction for a PQ is that a deletion must remove the largest node in the array
- Selection sort is one type of PQ-based sorting in which keys in PQ have no specific order:
  1. In order to locate the largest element $A[j]$ in the PQ region (i.e. A[0] to A[i]), it is necessary to make $i + 1$ comparisons
  2. Then, the largest element is swapped with key at the end of PQ (i.e. $A[j] \Leftrightarrow A[i]$)
  3. The boundary of PQ is reduced by a decrement on i

```
public void selectionSort(int[] A){
    int i=A.length-1;
    int j,temp,k;
    while (i>0) {
        j=i;
        for (k=0; k<=i; k++) {
            if (A[j]<A[k]) {
                j=k;
            }
        }
        temp=A[j];
        A[j]=A[i];
        A[i]=temp;
        i--;
    }
}
```

Locating the largest element in indices 0 ... i

Swapping the largest element at j in [0 ... i] with element in i

Reducing the boundary on the unsorted array

Complexity of Selection Sort

- Cost of 1: $(i+1)a$
- Cost of 2: $\sum_{i=1}^{n-1}[(i+1)a+b]$
- Cost of 3: $\sum_{i=1}^{n-1}[(i+1)a+b]+c$

$$
\begin{aligned}
f(n) &= a\sum_{i=1}^{n-1} i + (a+b)(n-1) + c \\
&= a\frac{(n-1)n}{2} + (b+a)(n-1) + c
\end{aligned}
$$

- Exercise: Formally prove that $f(n) = O(g(n))$ where $g(n) = n^2$

```
public void selectionSort(int[] A){
    int i=A.length-1;
    int j,temp,k;
    while (i>0) {
        j=i;
        for (k=0; k<=i; k++) {
            if (A[j]<A[k]) {
                j=k;
            }
        }
        temp=A[j];
        A[j]=A[i];
        A[i]=temp;
        i--;
    }
}
```

1 2 3

Heap Sort

Heap Sort

Priority Queue Based: Heap Sort

- When **heap sort** is used, the PQ portion of the array A is organized into a heap (**sequential representation** of a heap)
- Heap sort is one type of PQ-based sorting in which keys in PQ have an order:
  1. Since the first element $A[1]$ of the heap $A[1] \ldots A[i]$ is the largest, $A[1]$ is swapped with $A[i]$
  2. The boundary of PQ is reduced by a decrement on i
  3. In order to restore the heap property of $A[1] \ldots A[i]$, element in $A[1]$ is bubbled down

```
public void heapSort(int[] A){
    int i=A.length;
    int j=1,temp,k;
    while (i>1) {
        temp=A[j];
        A[j]=A[i];          1
        A[i]=temp;
        i--;                    2
        bubbleDown(A,i);            3
    }
}
```

```
public void bubbleDown(int[] A){
    int parentIndex=1, childIndex=2, temp;
    int n=A.length;
    while (childIndex<=n) {
        if (childIndex<n) {
            if (A[childIndex]<A[childIndex+1]) {
                childIndex=childIndex+1;
            }
        }
        if (A[parentIndex]<A[childIndex]) {
            temp=A[parentIndex];
            A[parentIndex]=A[childIndex];
            A[childIndex]=temp;
        }
        childIndex=childIndex*2;
    }
}
```

Complexity of Heap Sort

- Cost of 1: $a \lfloor log(i) \rfloor$
- Cost of 2: $\sum_{i=2}^{n} (a \lfloor log(i) \rfloor + b)$
- Cost of 3: $\sum_{i=2}^{n} (a \lfloor log(i) \rfloor + b) + c$
- Using $\sum_{i=1}^{n} \lfloor log(i) \rfloor = (n+1) \lfloor log(n+1) \rfloor - 2^{(\lfloor log(n+1) \rfloor + 1)} + 2$:

$$
\begin{aligned}
f(n) = & \sum_{i=2}^{n} (a \lfloor log(i) \rfloor + b) + c \\
& = a(n+1) \lfloor log(n+1) \rfloor - 2^{(\lfloor log(n+1) \rfloor + 1)} + 2 + b(n-2) + c
\end{aligned}
$$

- Exercise: Formally prove that $f(n) = O(g(n))$ where $g(n) = nlog(n)$

```
public void heapSort(int[] A){
    int i=A.length;
    int j=1,temp,k;
    while (i>1) {
        temp=A[j];
        A[j]=A[i];
        A[i]=temp;
        i--;
        bubbleDown(A,i);          1   2   3
    }
}
```

Table of Contents

Divide and Conquer

General structure of sorting methods such as **merge sort** and **quick sort** that use divide and conquer techniques:

1. **Divide** the array into two sub-arrays
2. **Sort** the two sub-arrays
3. **Combine** the sorted sub-arrays into a single array

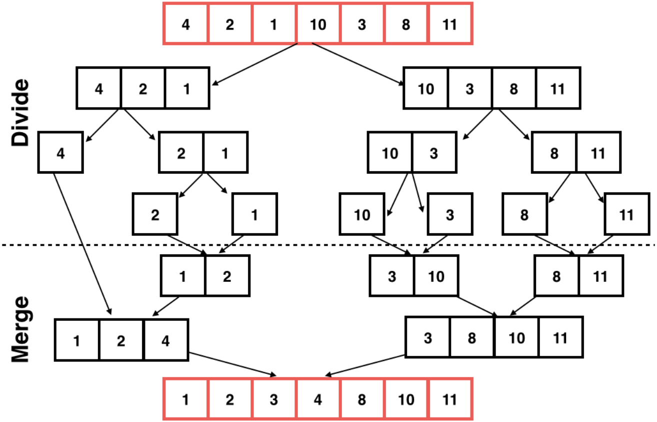
Merge Sort

Merge Sort

Divide and Conquer: Merge Sort

General algorithm:

1. Divide the array into two halves
2. Sort the left half and sort the right half
3. Merge the sorted left half with the sorted right half



Divide and Conquer: Merge Sort

```
public void merge(int[] A, int l, int m, int r){
    int n1=m-l+1;
    int n2=r-m,k=0, i, j;
    int[] A1=new int[n1]; int[] A2=new int[n2];

    for (i=0; i<n1; i++) {
        A1[i]=A[l+i];
    }
```

1

```
    for (i=0; i<n2; i++) {
        A2[i]=A[m+i+1];
    }
```

2

```
    i=0;
    j=0;
    while (i<n1 && j<n2) {
        if (A1[i]<A2[j]) {
            A[l+k]=A1[i];
            i++;
        }
        else{
            A[l+k]=A2[j];
            j++;
        }
        k++;
    }
```

3

```
    while (i<n1) {
        A[l+k]=A1[i];
        i++;
    }
    while (j<n2) {
        A[l+k]=A2[j];
        j++;
    }
}
```

4

1) Copy the left half of the array into A1

2) Copy the right half of the array into A2

3) Compare values in A1 and A2 one by one and copy the smaller value into A for min(n1, n2) iterations

4) Copy the remaining elements in A1 or A2 into the rest of A

O(n)

```
public void mergeSort(int[] A, int l, int r){
    int m=(l+r)/2;
    if (r-l>=1) {
        mergeSort(A, l, m);
        mergeSort(A, m+1, r);
        merge(A, l, m, r);
    }
}
```

Complexity of Merge Sort

- Cost of 1: 2f(n/2)
- Cost of 2: bn +c
- Cost of merge sort algorithm:

$$f(n) = 2f(n/2) + bn + c + d$$

- Exercise: Formally prove that $f(n) = O(g(n))$ where $g(n) = nlog(n)$

```
public void mergeSort(int[] A, int l, int r){
    int m=(l+r)/2;
    if (r-l>=1) {
        mergeSort(A, l, m);          1
        mergeSort(A, m+1, r);
        merge(A, l, m, r);           2
    }
}
```

Divide and Conquer: Quick Sort

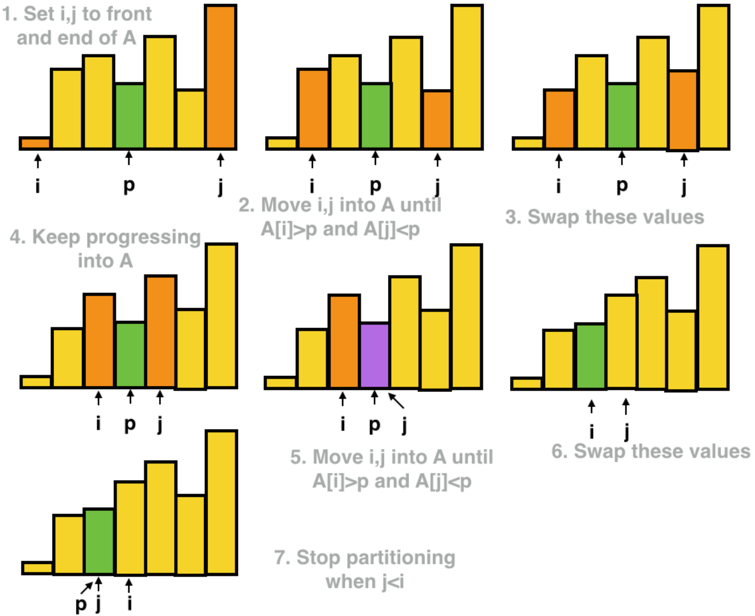
Quick Sort

Divide and Conquer: Quick Sort

General Algorithm:

1. **Partition** the array from m to n into two parts (all elements on one side of the array will be lesser than pivot and the others will be greater than pivot)
    - Select an element p to pivot
    - Mark the front and back of the array as i and j
    - Progress i and j inwards until $A[i] > p$ and $A[j] < p$
    - Swap these elements
    - Repeat above until $i > j$, where elements in A from m to j are less than p and i to n are greater
2. Apply quick sort to the left sub-array
3. Apply quick sort to the right sub-array
4. Repeat above until the indices m and n are no longer feasible (i.e. $m > n$)

Divide and Conquer: Quick Sort Partitioning



Divide and Conquer: Quick Sort Implementation

```
public void quickSort(int[] A, int m, int n){
    int i,j;
    if (m<n) {
        i=m;
        j=n;
        partition(A, i,j);       1
        quickSort(A, m, j);      2
        quickSort(A, i, n);
    }
}
```

3

```
public void partition(int[] A){
    int pivot=A[(i+j)/2],temp;
    do{
        while (A[i]<pivot)
            i++;
        while (A[j]>pivot)
            j--;
        if (i <= j) {
            temp=A[i];
            A[i]=A[j];
            A[j]=temp;
            i++;
            j--;
        }
    }while (i <= j);
}
```

1

1: Partition array so that all elements to the left of pivot is lesser than pivot and all elements to the right of the pivot is greater

i: Explore all keys in the array from m to n

2: Apply quick sort to the left and right of the partitioned arrays

3: Repeat this until m and n are no longer feasible

Complexity of Quick Sort

- Cost of 1: bn +c
  - At every iteration in the `do-while` loop, elements are swapped between the left and right partitions
  - In the `partition` function, counters `i` and `j` move into the array `A` while swapping elements with one another if necessary until these counters move past each other
  - The number of comparisons in the `partition` function is $bn + c$
- Cost of 2: $C(k-1)$ and $C(n-k)$
  - Given that the pivot key is located at k, the cost of applying quick sort to $A[0:k-1]$ and $A[k:n-1]$ is the above
- Cost of quick sort algorithm is:
  - Recurrent relation:

$$C(0) = 0,\ C(1) = 1$$
$$C(n,k) = bn + c + C(k-1) + C(n-k)$$

  - The above relation can be unrolled and expressed in the non-recurrent form via the Harmonic number formula:

$$H_n = \sum_{i=1}^{n} \frac{1}{i} = ln(n) + \gamma + O(\frac{1}{n}), \text{ where } \gamma = 0.57721566$$

  - Exercise: Formally prove that $f(n) = O(g(n))$ where $g(n) = nlog(n)$

Complexity of Quick Sort

- On **average** quick sort has a performance of $O(nlogn)$
- However, unlike other algorithms, the **position** of the pivot is important
- If the pivot key is **biased** (i.e. does not partition the array well), then the performance can degrade to $O(n^2)$

Table of Contents

Address Calculation Sorting

- All algorithms that have been covered so far are comparison-based
- As we saw earlier, the best performance of a comparison-based sorting algorithm is $O(nlogn)$
- With **address calculation sorting**, it is possible to sort keys by mapping these to a location close to their final resting point
- One specific example is **radix sort**

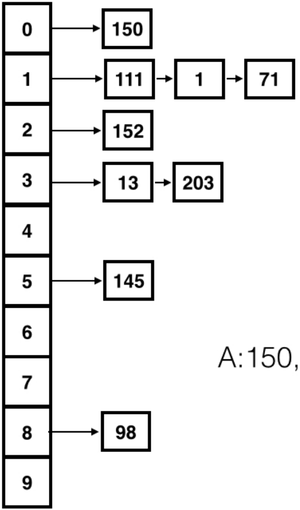
Radix Sort

- Typically keys of **integer** type are sorted with radix sort
- The number of passes through `A` is equal to the maximum number of significant digits of keys in this array
- At each pass, keys are stored at the end of a linked list associated with one of 10 **buckets** (each bucket represents a digit from $[0, \ldots, 9]$)
- General algorithm: Starting from the least significant digit i
    1. Examine digit i of all keys in A
    2. If a key's i^{th} significant digit is j, then map it to bucket j, store this key at the end of the linked list at bucket j and repeat this for all the keys
    3. Traverse linked lists located at each bucket starting from bucket 0, delete keys from left to right at each bucket and store these into array A in the order these are deleted
    4. Repeat the above for all digits in the keys

Radix Sort: An Example

Pass 1

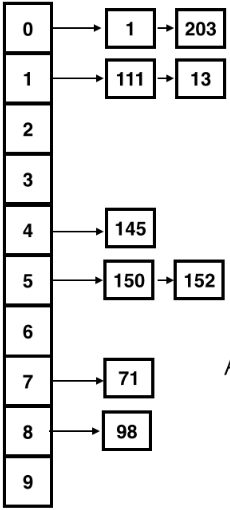
A:15**0**, 11**1**, 14**5**, 15**2**, 9**8**, 1**3**, **1**, 20**3**, 7**1**



A:150, 111, 1, 71, 152, 13, 203, 145, 98

Radix Sort: An Example

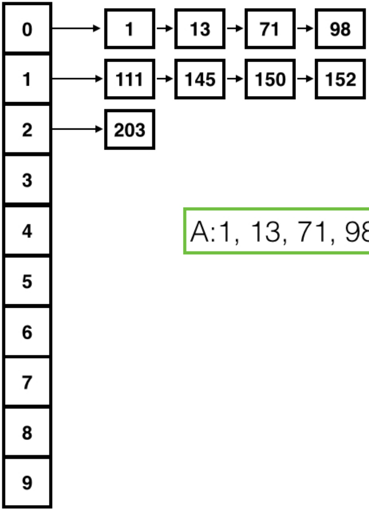
Pass 2 A:1**5**0, 1**1**1, **0**1, **7**1, 1**5**2, **1**3, 2**0**3, 1**4**5, **9**8



A:1, 203, 111, 13, 145, 150, 152, 71, 98

Radix Sort: An Example

Pass 3 A: **0**01, **2**03, **1**11, **0**13, **1**45, **1**50, **1**52, **0**71, **0**98



A:1, 13, 71, 98, 111, 145, 150, 152, 203