1.00 Lecture 34

Sorting 2

Reading for next time: Big Java 21.1-21.3

Counting sort

- O(n) sort when keys are small integers
- Procedure:
  - Count the number of records with a given key
  - Calculate the number of records with keys that are less than given key by adding up their counts
  - This tells you where the final position of your keys will be
  - Move keys to final positions and you're done
- You never compare any value pairs!
  - This is special case of radix sorting

Counting sort example

Input		Count:	#	Cum		Output	
i	a	0	0	0		i	b
0	3	1	0	0		0	2
1	5	2	1	1	*-1 offset*	1	3
2	2	3	2	3	→	2	3
3	3	4	0	3		3	5
		5	1	4			

size

range

Move values to output vector or array

Steps in the CountingSort Algorithm

1. Copy the numbers to be sorted to a temporary array.
2. Initialize an array indexed by key values (a count of keys) to 0 (Java initializes arrays to 0 automatically)
3. Iterate over the array to be sorted counting the frequency of each key.
4. Calculate the cumulative count for each key value, k. The first element, k=0, is the same as the frequency of key k. The second, k=1, is the sum of the frequency for k=0 and k=1. The third is the sum of the second plus the frequency for k=2. And so on.

Steps in the CountingSort Algorithm, 2

5. The first element of the cumulative count contains the number of elements of the original array with values <= 0. the second, those <= 1. They lay out blocks of values in the sorted array.
6. Starting with the last element in the original array and working back to the first, look up its key in the cumulative count to find its destination in the sorted array. Its destination = `count value - 1`.
7. Decrement the entry in the cumulative count so the next element is not stored on top of the first, if there are more than one at the same key value (e.g., two 3s)

CountingSort

`Range = 8 (maximum element value)`

4	3	7	6	4	8	3	5	2

`Frequencies (no of keys w/each value)`

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

`Cumulative count`

0	1	2	3	4	5	6	7	8
0	0	1	3	5	6	7	8	9

`Sorted output`

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

There are no keys 0, 1

Key 2 occurs 1 time and should occupy position 0.

Key 3 occurs 2 times and should occupy positions 1 and 2.

Key 4 occurs 2 times and should occupy positions 3 and 4.

Etc.

(Our code actually writes values from last to first)

CountingSort

`Range = 8 (maximum element value)`

4	3	7	6	4	8	3	5	2

`Count (no of keys w/each value)`

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

`Cumulative count`

0	1	2	3	4	5	6	7	8
0	0	1	3	5	6	7	8	9

`Sorted output`

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

There are no keys 0, 1

Key 2 occurs 1 time and should occupy position 0.

Key 3 occurs 2 times and should occupy positions 1 and 2.

Key 4 occurs 2 times and should occupy positions 3 and 4.

Etc.

(Our code actually writes values from last to first)

CountingSort Questions

- **Download and double click the `CountingSort2004.jar` file to run the simulation.**
  - **It works the same way as the others.**
- **Experiment with the simulation.**
  - **Type the numbers to be sorted first.**
    - **Hit carriage return after entering each number**
    - **Use single digit numbers to keep the range small**
  - **Then enter the range (the largest number you entered) in the range field and press the `start` button.**
  - **Use `stepinto` to trace the code.**
- **Does counting sort have a best case or worst case?**
- **Does it sort in O(1), O(n), O(n log n) or O(n^2) time? Why?**

Counting Sort

```
import javax.swing.*;
public class CountingSort {  // A few changes from the demo
   private int sortRange;
   public CountingSort(int range) { sortRange = range;}
   public void sort(int[] d, int[] v) {
        int[] value = new int[d.length];
        int[] extraStuff= new int[d.length];     // ‘Value’ field
        int[] count = new int[sortRange];
        int i, size = d.length;
        System.arraycopy(d, 0, value, 0, size);  // Copy arrays
        System.arraycopy(v, 0, extraStuff, 0, size);
        for (i = 0; i < size; i++)
                count[value[i]]++;
        for (i = 1; i < sortRange; i++)
                count[i] += count[i - 1];
        for (i = size - 1; i >= 0; i--) {
                v[count[value[i]] - 1] = extraStuff[i]; //Only chg
                d[count[value[i]] - 1] = value[i];
                count[value[i]]--;
        }
   }
```

Counting Sort, p.2

```
   public static void main(String[] args) {
        String input =
               JOptionPane.showInputDialog("Enter no. elements:");
        int size = Integer.parseInt(input);
        int[] keyData = new int[size];       // What will be sorted
        int[] valueData = new int[size];     // Associated data
        int range = 1000;
        CountingSort s = new CountingSort(range);
        for (int i = 0; i < size; i++) {     // Random data
            keyData[i] = (int) (range * Math.random());
            valueData[i] = keyData[i] + 7;
        }
        s.sort(keyData, valueData);   // Call with “parallel arrays”
        System.out.println("Done");
        if (size <= 1000)
            for (int i = 0; i < size; i++)
               System.out.println(keyData[i] + " " + valueData[i]);
        System.exit(0);
   }
}
```

Applications of sorting

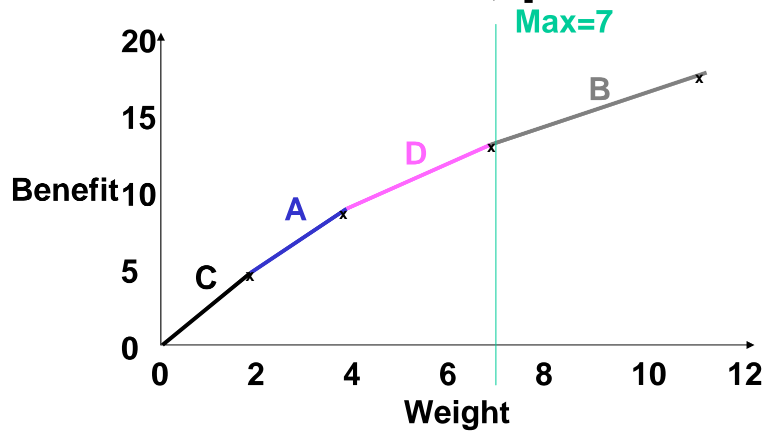
- Business applications are obvious:
  - Reports sorted by customer, department, etc.
- Many technical applications use sorting:
  - Satellite payload
  - Optimal maintenance policy (e.g., pavement)
  - Processor scheduling
  - Generating graphs (networks)
- One estimate is that 25% of all CPU time is spent on sorting
  - And the other 75% is spent on IM and games?

Satellite payload

Experiment	Benefit	Weight	Ben/Wgt
A	4	2	2.0
B	6	4	1.5
C	5	2	2.5
D	5	3	1.7

Assume satellite can carry max weight= 7

Satellite, p.2



Sort in Benefit/Weight ratio. Maximum derivative gives optimum value. Gives solution for all maximum weights M. ('Greedy algorithm')

Last item in solution may be fractional. Often this is acceptable. If not, we use more sophisticated optimization methods as in PS9

Exercise: Sorting, p. 1

- **Finish class Satellite, which selects experiments to be taken aloft:**
- **We create an Experiment static nested class, like insertion sort's Item, to store the benefit/weight ratio (key) and weight (value)**
  - Experiment data members: ratio, weight
  - Write a constructor with two arguments
  - Experiment, to be sortable, must implement Comparable:
  - Implement public int compareTo(Object o) { … }
    - Cast o to an Experiment other
    - Return –1 if other.ratio < ratio
    - Return 0 if other.ratio == ratio
    - Return 1 if other.ratio > ratio

Exercise, p. 1

```
public class Satellite {
    private static class Experiment implements Comparable {
        public double ratio;
        public int weight;

        // Write constructor for Experiment

        public int compareTo(Object o) {
            // Complete this method:
            // Cast o to an Experiment other
            // Return -1 if other.ratio < ratio
            // Return 0 if other.ratio == ratio
            // Return 1 if other.ratio > ratio
        }
    }
```

Exercise: Sorting, p. 2

- In Satellite's main():
  - We created the 4 Experiments from the earlier slide
  - We put the experiments in an array of Experiments
  - You should call a sort method to put the experiments in benefit/weight order
    - Invoke InsertionSortTest.sort() or QuicksortTest.sort() or …
    - Copy the class from last lecture's project to this lecture
  - Set the weight limit= 7 (or ask the user to input it)
  - Loop through the sorted experiments, and select the best ones to go on the satellite that fit within the weight limit
    - Keep track of cumulative weight, cumulative benefit
  - Print out your solution:
    - Experiments to go on the satellite
    - Total weight
    - Total benefit

Exercise, p.2

```
public static void main(String[] args) {
  Experiment a= new Experiment(2.0, 2);
  Experiment b= new Experiment(1.5, 4);
  Experiment c= new Experiment(2.5, 2);
  Experiment d= new Experiment(1.66667, 3);
  Experiment[] e= {a, b, c, d};

  // Invoke a sort method

  // Set maximum weight= 7 (or get it from user via JOptionPane)
  // Initialize cumulative weight, cumulative benefit= 0

  // Loop thru the sorted experiments
  //   Accumulate the weight until the max weight is reached
  //   Accumulate the benefit as each experiment is added
  //     Compute benefit as ratio*weight
  //   Print out each experiment added to the payload
  //   Break out of the loop when max weight is reached

  // Print the total benefit
```

Processor scheduling

- **You have a set of n jobs to run on a processor (CPU) or machine**
- **Each job i has a deadline d_i and profit p_i**
- **There is one processor or machine**
- **Each job takes 1 unit of time (simplification)**
- **You earn the profit if and only if the job is completed by its deadline**
  - **"Profit" can be the priority of the computer job in a real time system that discards tasks that cannot be completed by their deadline**

Processor scheduling example

Number of jobs n=5. Time slots 1, 2, 3.

Job (i)	Profit	Deadline	Profit/Time
A	100	2	100
B	19	1	19
C	27	2	27
D	25	1	25
E	15	3	15

Solution is {C, A, E} for profit of 142

Processor scheduling algorithm

- Sort jobs by profit/time ratio (slope or derivative): A, C, D, B, E
- Place each job at latest time that meets its deadline
  - Nothing is gained by scheduling it earlier, and scheduling it earlier could prevent another more profitable job from being done