CSCI203

Week 7 – Lecture B

Looking at Sorting (Again)

- We have already seen a number of sorting algorithms with varying efficiency.
  - Insertion Sort:
  - Selection Sort:
  - Bubble Sort:
    - All $O(n^2)$
  - Quick Sort:
  - Heap Sort:
  - Merge Sort:
    - All $O(n \log n)$
- Can we do better than $O(n \log n)$ operations?

Linear Time Sorting?

- What if we could sort *n* items in O(*n*)?
- We can!
- Sometimes…
- …provided the keys we wish to sort satisfy certain conditions:
  - The keys are integers;
  - Each key fits in a single word of memory;
  - All keys are smaller than some upper bound, *k*;
  - *k* satisfies a specific relationship to *n*.
    - (To be revealed later).

Counting Sort

Counting Sort

- Given these conditions we can sort our array as follows:
  - ```
    key: array[1..n] of integer
    a: array[1..k] of integer
    for i in 1..k
       a[i]=0
    rof
    for i in 1..n
       a[key[i]]++
    rof
    for i in 1..k
       for j in 1..a[i]
           print i
       rof
    rof
    ```
    ```

Counting Sort

- This algorithm takes:
  - O(k) operations to initialize the array;
  - O(n) operations to store the keys;
  - O($n+k$) operations to output the result.
- Provided $k \in O(n)$ this algorithm is also in $O(n)$.
- You might object that, if there is data associated with each key, it has been lost.
- You would be right!

Counting Sort Improved.

- We can address this objection by replacing our integer array with an array of lists.
  - ```
    key: array[1..n] of integer
    a: array[1..k] of list[]
    for i in 1..k
       a[i]=[]
    rof
    for i in 1..n
       append data to a[key[i]]
    rof
    for i in 1..k
       for j in 1..length[a[i]]
           print i,a[i][j]
       rof
    rof
    ```
- Now, we have retained the data associated with each key.
  - In the order they appeared in the original data set.
    ```

Counting Sort

- This, revised, version of counting sort is still $O(n+k)$.
  - $O(n)$ if $k \in O(n)$.
- If $k \notin O(n)$ this will not be the case.
- If $k >> n$ the algorithm is $O(k)$.
- Can we find a way to sort n integers in $O(n)$ time even if k is much larger than n?
- Yes—provided $k=n^{O(1)}$.
  - We say k is polynomial in n.

Radix Sort

Radix Sort

- Radix sort works by treating the integer keys as numbers in some arbitrary base, b.
- E.g. *key*=123 decimal
  - = 1111011 base 2
  - = 173 base 8
  - = 7B base 16
  - = 443 base 5
  - = 78 base 17
  - etc.
- Note: if the maximum value of *key* is k, the number of digits in a key, $d \leq \lceil \log_b k \rceil$
- We will sort our keys using the digits in our selected base in order, as follows…

Radix Sort

- Our algorithm is as follows:
  - **`sort the data by the least significant digit of the key`**
  - **`sort the data by the next least significant digit of the key`**
  - **…**
  - **`sort the data by the most significant digit of the key`**
- We use counting sort for each of the sorts in the above sequence.
- Remember: counting sort preserves prior order.
  - We say counting sort is a ***stable*** sort.

Radix Sort

- We can analyse radix sort as follows:
  - At each step we perform counting sort on n keys with k represented in base b.
  - Each sort is $O(n+b)$.
  - There are d sort steps, where $d=\lceil \log_b k \rceil$
  - Thus, our algorithm is $O((n+b).\log_b k)$
  - Now, if we set $b=n$, we get a complexity of $O((n+n).\log_n k)$
  - If k is polynomial in n, $\log_n k$ is a constant…
  - …and our algorithm is $O(n)$.

An Example

- Consider the following set of numbers:
- 467, 362, 753, 178, 610, 800, 250, 138, 708, 426, 692, 426, 187, 965, 346, 257, 684, 575, 350, 594, n=20.
- Let us set the base, b, to 10.
- We will sort them by each decimal digit from least to most significant:

Key	d_2	d_1	d_0
467	4	6	7
362	3	6	2
753	7	5	3
178	1	7	8
610	6	1	0
800	8	0	0
250	2	5	0
138	1	3	8
708	7	0	8
426	4	2	6
692	6	9	2
426	4	2	6
187	1	8	7
965	9	6	5
346	3	4	6
257	2	5	7
684	6	8	4
575	5	7	5
350	3	5	0
594	5	9	4

Sort on d_0

Key	d_2	d_1	d_0
610	6	1	0
800	8	0	0
250	2	5	0
350	3	5	0
362	3	6	2
692	6	9	2
753	7	5	3
684	6	8	4
594	5	9	4
965	9	6	5
575	5	7	5
426	4	2	6
426	4	2	6
346	3	4	6
467	4	6	7
187	1	8	7
257	2	5	7
178	1	7	8
138	1	3	8
708	7	0	8

Key	d_2	d_1	d_0
610	6	1	0
800	8	0	0
250	2	5	0
350	3	5	0
362	3	6	2
692	6	9	2
753	7	5	3
684	6	8	4
594	5	9	4
965	9	6	5
575	5	7	5
426	4	2	6
426	4	2	6
346	3	4	6
467	4	6	7
187	1	8	7
257	2	5	7
178	1	7	8
138	1	3	8
708	7	0	8

Now sort on d_1

Key	d_2	d_1	d_0
800	8	0	0
708	7	0	8
610	6	1	0
426	4	2	6
426	4	2	6
138	1	3	8
346	3	4	6
250	2	5	0
350	3	5	0
753	7	5	3
257	2	5	7
362	3	6	2
965	9	6	5
467	4	6	7
575	5	7	5
178	1	7	8
684	6	8	4
187	1	8	7
692	6	9	2
594	5	9	4

Key	d_2	d_1	d_0
800	8	0	0
708	7	0	8
610	6	1	0
426	4	2	6
426	4	2	6
138	1	3	8
346	3	4	6
250	2	5	0
350	3	5	0
753	7	5	3
257	2	5	7
362	3	6	2
965	9	6	5
467	4	6	7
575	5	7	5
178	1	7	8
684	6	8	4
187	1	8	7
692	6	9	2
594	5	9	4

Finally, sort on d_2

Key	d_2	d_1	d_0
138	1	3	8
178	1	7	8
187	1	8	7
250	2	5	0
257	2	5	7
346	3	4	6
350	3	5	0
362	3	6	2
426	4	2	6
426	4	2	6
467	4	6	7
575	5	7	5
594	5	9	4
610	6	1	0
684	6	8	4
692	6	9	2
708	7	0	8
753	7	5	3
800	8	0	0
965	9	6	5

And we are done!

Analysis of Radix Sort

Analysis

- Although radix sort is O(n), we cannot immediately conclude that it will be faster than an order $n \log n$ sort such as merge sort.
- The critical issue is the size of $\log_2 n$ compared to $\log_n k$.
- If we assume 64-bit integers $k = 2^{64}$.
- Let us look at the relationship for different values of n.

Analysis

- The following table compares $\log_2 n$ with $\log_n k$:

n	$\log_2 n$	$\log_n k$
10	3.32	19.27
100	6.64	9.63
1,000	9.97	6.42
10,000	13.29	4.82
100,000	16.61	3.85
1,000,000	19.93	3.21

- As you can see, radix sort wins as soon as n is somewhere between one hundred and one thousand.
- In fact, the break-even point is n=256.
- For 32-bit integers break-even is n=51.