CS 61B Data Structures and Programming Methodology

Aug 4, 2008

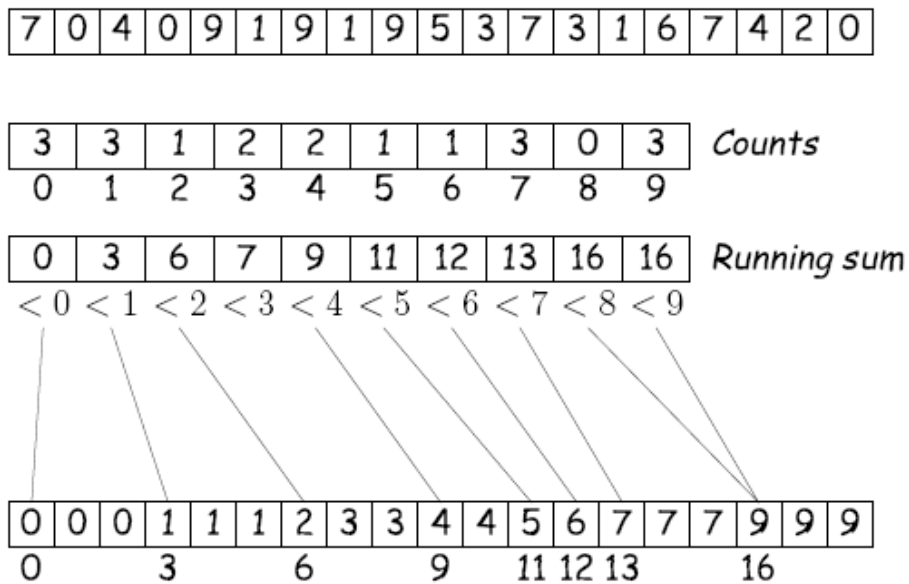
David Sun

Linear Sorting Algorithms

- But suppose can do more than comparing keys.
- What if we know *something* about the keys that are being sorted:
  - For example, how can we sort a set of n integer keys whose values range from 0 to kn , for some small constant k?
- One approach: for each element x, determine the number of elements *less* than x. We can place x immediately into its right position.
  - If M_p = #items with value < p, then in sorted order, the jth item with value p must be $\#M_p + j$.
- Gives linear-time algorithm.

Counting Sort

- Suppose all items are between 0 and 9:



- "Counts" array gives number of occurrences of each item.
- "Running sum" gives cumulative count of item. Each value which tells us where to put each item:
  - The first instance of key *k* goes into slot *m*, where *m* is the number of key instances that are < *k*.

```
int[] countingSort (int[] x, int maxKey) {
  int[] counts = new int[maxKey];
  int[] runningSum = new int[maxKey];
  int i, j, c, total;

  //takes theta(n)
  for (i = 0; i < x.length; i++) {
    counts[x[i]]++;
  }
  total = 0;
  //takes theta(k)
  for (j = 0; j < counts.length; j++) {
    runningSum[j] = total;
    total = total + counts[j];
  }
  //takes theta(n)
  for (i = 0; i < x.length; i++) {
    y[runningSum[x[i]] = x[i];
    runningSum[x[i]]++;
  }
```

Running Time of Counting Sort

- *Theta(n + k)* where *n* is the size of the input and *k* is the length of the counting array.
  - Takes *Theta(n)* time to set up the *count* array.
  - Takes *Theta(k)* time to build the *Running Sum* array.
  - Takes *Theta(n)* time to assign items in the input array into the output array using the *Running Sum* array.
- In order for this algorithm to be efficient, *k* must not be much larger than *n*.
  - If *k* is in *O(n)*, then the running time is *Theta(n).*

Comments

- The indices of the *counting* array must run from the *minimum* to the *maximum* value in the input to be able to index directly with the values of the input.
- Otherwise, shift the values of the input so that the minimum value of the input matches the smallest index of the counting array.
- Counting sort is stable: numbers with the same value appear in the output array in the same order as they do in the input.

Bucket Sort

- Similar to *Counting Sort*, uses the fact that
  - the keys are distributed with in some small range of values. e.g. from 0 to *q-1*, and
  - the number of items *n* is larger than, or nearly as large as *q*, i.e. *q* is in *O(n)*.
- Basic version:
  1. *Initialize*: Allocate an array of *q queues*, numbered from 0 to *q-1*, called buckets.
  2. *Scatter*: We walk through the list of input items, and enqueue an item with key *i* into queue *i*.
  3. *Gather*: When we're done, we concatenate all the queues together in order.

Running Time of Bucket Sort

- *Theta(q + n)* time - in the best case and in the worst case.
  - It takes *Theta(q)* time to initialize the buckets in the beginning
  - It takes *Theta(q)* to concatenate the queues in the buckets together in the end.
  - It takes Theta(n) time to put all the items in their buckets.

Generalized Bucket Sort

1. *Initialize*: Allocate an *array* of *m buckets*, numbered from 0 to *m-1.* If the values of input array ranges from *1* to *q,* each bucket *i* will contain input values from *i*q/m to i*(q/m+1) -1*.
2. *Scatter*: Walk through the list of input items, and add an item with key *i* into bucket *i div q*.
3. *Sort: S*ort the each of the buckets individually by picking another sorting algorithm or recursively apply Bucket Sort.
4. *Gather*: Concatenate all the queues together in order.

Comments

- Bucket sort works well when the input values are drawn from a uniform distribution.
- If clustering occurs, some buckets will get more values than others, and the running time will be dominated by these “heavy” buckets.
- If each bucket has a size of 1, then bucket sort is essentially the same as counting sort.
- If the number of buckets is 2, then bucket sort is essentially the same as quick sort, where the pivot is chosen perfectly.

Radix Sort

- How to sort 1,000 items in the range from 0 to 99,999,999?
  1. Using bucket/counting sort: spend too much time initializing the buckets.
  2. Provide *10* buckets (instead of 100 million) and sort on the first digit only using bucket sort. Then sort each queue recursively on the second digit; then sort the resulting queues on the third digit, and so on.
     - This tends to break the set of input items into smaller and smaller subsets, each of which will be sorted relatively inefficiently.

Radix Sort

- The way it works:
  - Keep all the numbers together in one big pile throughout the sort.
  - Use counting/bucket sort to sort on the least significant digit (last digit), then the next-to-last, and so on up to the most significant digit.
- Why does this work?
  - Each pass is implemented using counting sort or bucket sort and both algorithms are *stable*!
  - After sorting on the last digital, all the values in the input that differ only in the last digit are put into their correct *relative* order.
  - After sorting on the second last digit, all the values that differ only in the last and second last digits are put into their correct *relative* order.

Example

Sort on 1s: 720 450 771 822 332 504 925 5 955 825 777 858 28 829

Sort on 10s: 504 5 720 822 925 825 28 829 332 450 955 858 771 777

Sort on 100s: 5 28 332 450 504 720 771 777 822 825 829 858 925 955

Radix Sort

- Radix sort will likely be faster if we sorted on two digits, or even three digits, at a time.
- The radix is denoted by q – all the numbers in the input are treated as base-q numbers:
  - Sorting in two digits means to use a radix of $q = 100$.
  - Sorting in three digits means using a radix of $q = 1000$.
- On computers, it's more natural to choose a power-of-two radix, like $q = 256$.
  - they are easier to extract from a key – just pull out the eight bits that we need by using bit operators.

Running time of Radix Sort

- Each pass of the radix sort uses counting/bucket sort: *Theta(n+q).*
- How many passes must we perform?
  - Each pass inspects a single digit of the input value, treated as a *based-q* number. This corresponds to $log_2\ q$ bits.
  - If all the numbers can be represented in *b* bits, the number of passes is ceiling(b / $\log_2$ q).
- The running time of radix sort is in *O((n+q)* ceiling(b / $\log_2$ q)).

Choosing the Radix

- How should we choose the number of queues q?
- Choose q to be in O(n), so each pass of bucket sort or counting sort takes O(n) time.
- However, we want q to be large enough to keep the number of passes small.
- Therefore, let's choose q to be approximately n. With this choice, the number of passes is in $O(1 + b / \log_2 n)$, and radix sort takes $O(n + b / \log_2 n)$

Graphs

- A graph *G* consists of :
  - a set *V* of vertices
  - a set *E* of edges that each connect a pair of vertices together
- Formalism: a graph *G* with vertex set *V* and edge set *E* is denoted as $G = (V, E)$.
- Directed graph:
  - Every edge *e* is directed from a vertex *v* to a vertex w, denoted as $e = (v, w)$ (the order matters).
  - The vertex *v* is called the origin of *e*, and *w* is the destination of *e*.
- Undirected graph:
  - Each edges *e* has no favored direction, and is denoted as $e = (v, w)$.

Graphs

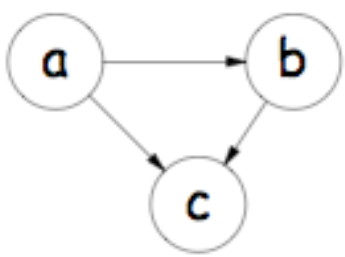
- Multiple copies of an edge are forbidden, but a directed graph may contain both (v, w) and (w, v).
- Both types of graph can have ***self-edge***s of the form (v, v), which connect a vertex to itself.
- A ***path*** is a sequence of vertices such that each adjacent pair of vertices is connected by an edge. For directed graphs, the edges on a path must all be aligned with the direction of the path. The ***length of a path*** is the number of *edges* it traverses.
- The ***distance*** from one vertex to another is the length of the shortest path from one to the other.

Graphs

- A graph is ***strongly connected*** if there is a path from any vertex to any other vertex.
  - For undirected graphs, this is just called *connected.*
- The ***degree*** of a vertex is the number of edges incident on that vertex.
- A vertex in a directed graph has
  - an ***indegree*** - the number of edges directed toward it and
  - an ***outdegree*** - the number of edges directed away.

Graph Representation

- Adjacent Matrix:
  - a $|V|$-*by*-$|V|$ array of boolean values (where $|V|$ is the number of vertices in the graph).
  - Each row and column represents a vertex of the graph.
  - Set the value at row *i*, column *j* to *true* if *(i, j)* is an edge of the graph.
  - If the graph is undirected , the adjacency matrix is symmetric: *(i, j)* has the same value as *(j, i).*



$$\begin{array}{c c} & \begin{array}{ccc} 1 & 2 & 3 \end{array} \\ \begin{array}{c} 1 \\ 2 \\ 3 \end{array} & \begin{bmatrix} 0 & 1 & 1 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix} \end{array}$$

Graph Representation

- The maximum possible number of edges is $|V|^2$ for a directed graph, and slightly more than half that for an undirected graph.
  - Cost of storage is quadratic in the number of vertices.
  - In many applications, however, the number of edges is much less than *Theta*($|V|^2$).
- A graph is called ***sparse*** if it has far *fewer* edges than the maximum possible number vertices .
  - Number of edges is asymptotically smaller than $|V|^2$.

Graph Representation

- Adjacency List:
  - A more memory-efficient data structure for sparse graphs.
  - An adjacency list is a collection of linked lists. Each vertex *v* maintains a linked list of the edges directed out from *v*.
  - Cost of representation is *Theta(|V| + |E|)*.
  - An adjacency list is more space- and time-efficient than an adjacency matrix
  - Less efficient than adjacency matrix for a ***complete graph*** -- for every vertex *u* and every vertex *v*, *(u, v)* is an edge of the graph.
  - Use a hash table to map non-integer vertices to linked lists. Each entry in the hash table uses a vertex name as a key, and a List as the associated value.

Readings

- Objects, Abstractions, Data Structures and Design
  - Chapter 12.