Notes on Sorting

Sorting with hash tables and hash function

Can we use hashing for sorting? Not in general case --- the items we hash on need not have total order semantics.

One of the things that makes hashing interesting is that it gives us efficient searching without requiring underlying sorting.

Effective hashes on strings need not involve the usual order on strings (lexicographical). For example, with `HASH_TABLE_SIZE = 1000` and `hashViaShift` (see notes on hashing), `hashViaShift("cat")==596` and `hashViaShift("act")==612`. So, in our hash table, "cat" will come before "act" (despite that with normal string ordering, "act" < "cat"):

```
 ______
|______|   0
|______|   1
|______|
| ...  |
|------|
|"cat" |  596
|------|
| ...  |
|------|
|"act" |  612
|------|
| ...  |
|______|
```

However, we can use hashing to achieve sorting in cases where our hash function is *monotone*:

if x < y then hash(x) < hash(y).

In other words, a monotone hash function will place items with keys that have a relative "lower" ordering early (near top) in the table and items with relatively greater ordering later (closer to the bottom) in the table.

Quiz: which of the following functions are monotone?

1.
   ```
   int hash(int i)
   {
    return (i % 100);
   }
   ```

   No, `hash(22) > hash (101).`

2.
   ```
   int foo(int i)
   {
    return (i + 1);
   }
   ```

   Yes: if x < y then x+1 < y+1

Distribution sort

As a very simple case, suppose we want to sort a list of positive integers. Let M be the max of those integers. We can create a boolean array B of Max elements, initially with each entry set to false.

- For each i in our list, set B[i] = true
- Now we can produce the sorted list of integers by walking through B and outputting only the indices for which B[i] is true.

```
input: 4 3 6 1 2 9

  |-------|
0 | false |
  |-------|
1 | true  |
  |-------|
2 | true  |
  |-------|
3 | true  |
  |-------|
4 | true  |
  |-------|
5 | false |
  |-------|
6 | true  |
  |-------|
7 | false |
```

```
  |-------|
8 | false |
  |-------|
9 | ture  |
  |-------|

output: 1 2 3 4 6 9
```

This is the simplest kind of hash sort (the underlying hash function is the trivial identity function) known as a *distribution sort*. (It is also a simple form of a *counting sort*.)

Distribution sort:

- input: an array A of n items that have total order semantics.
- action: place items of A into a hash_table using a monotone hash function. (can think of the item as being its own key)
- result: A is replaced by the items from the occupied slots in the order in which they occur in the hash table.
- preconditions:
    - assume item uniqueness
    - hash table must be at least as large as input array
    - hash function must be a valid hash function:
        - it returns arrays indices in range 0 ... HASH_TABLE_SIZE-1
        - it's *monotone*: if A[i] < A[j] then hash(A[i]) < hash(A[j])

pseudocode

```
initialize hash table so all slots marked as vacant

for(i=0; i < n; i++) {
  insert A[i] into table at location hash(A[i])
  (mark slot as occupied)
}

for(i=0,j=0; j < n; i++) {
  if table[i] is occupied {
    A[j] = item stored at table[i]
    j++
  }
}
```

What is the complexity of this method? O(M+n) where M is the max and n is the number of items in the array. Of course, with this simple formulation, we have no way of dealing with collisions, so we also must stipulate that the items to be sorted be unique. So M must be at least as big as n. So this sort is O(M). This is reasonable if M is roughly the same size as n.

Bucket sort

Is it practical to use distribution sort to sort the following numbers?

```
65536 1024 16777216 32 32768
```

Counting sort is not efficient if we have large numbers. However, with a little luck (i.e. an effective hash function) we can extend hash sorting to work efficiently on more general input.

We use chained hashing and think of each entry in the hash table as a bin or bucket (this kind of sorting is often called *bucket sorting*). The hash table is implemented such that the first bucket contains an ordered collection of the smallest numbers, the next bucket contains an ordered collection of the next smallest numbers and so on so that the last bucket contains an ordered collection of the largest numbers. For example:

We can easily produce a sorted list of the numbers in the following hash table, by linking each chain together in order:

```
  |---|
0 |   |
  |---|
1 |  -|-- 13 - 18
  |---|
2 |  -|-- 24 - 27 - 29
  |---|
3 |   |
  |---|
4 |  -|-- 41
  |---|

sorted list: 13 18 24 27 29 41
```

A monotone hash function that would group the data as such might be:

```
int hash(int i) { return (i / 10); }
```

(This will work as long as our table is no smaller than a tenth the size of the largest number.)

What do we need to make this into an effective sorting technique?

- Use a hash table of size k (where k is on the order of n). Each slot consists of a (possibly empty) ordered collection of items.
- Use a monotone hash function.
- If there is a collision - insert the item into the ordered collection at that hash address in such a way as to maintain that order.
- Choose a hash function that distributes the data evenly.

Maintaining ordered collections at each hash table entry may slow down the insertion process - worst-case O(n) if we use ordered lists, O(lg(n)) if we use ordered trees, making the sort time O(n^2) or O(n*(lg(n)). However, if the hash function distributes the data well, then on average - there will be very few elements in each bucket, so the effective time for the sort will be O(n).

So bucket sorting is very effective when either the data is evenly distributed over a range, OR we have a hash function that disperses it evenly.

Bucket sorting:

- input: an array A of n items that have total order semantics.
- action: place items of A into a chained hash_table of size n using a monotone hash function.
- result: A is replaced by the items from each chain in the table in the order in which they occur in the chain in the order in which they occur in the hash table.
- preconditions:
    - hash function must be a valid hash function:
        - it returns arrays indices in range 0 ... HASH_TABLE_SIZE-1
        - it's *monotone*: if A[i] < A[j] then hash(A[i]) < hash(A[j])

pseudocode

```
initialize hash table so all slots have empty collections

for(i=0; i < n; i++) {
  insert A[i] into collection at table[hash(A[i]])
}

for(i=0,j=0; j < n; i++) {
  for each item in the collection at  table[i] {
    A[j] = item
    j++
  }
}
```

Example:

```
input : [7096,6051,553,1969,14205,9651,4194,12180,14721,13458,
         7580,14920,2796,8344,11360,8168,10971,3851,1770,3122,
          165,9557,8109,2844,14652]
n = 25
min = 165
max = 14920
range = 14920 - 165 + 1 = 14756
bucketSize = (range + n - 1) / n = 14780 / 25 = 591
hash(i) = (i - min) / bucketSize

 0 : 165 -- 553
 1 :
 2 : 1770
 3 : 1969
 4 : 2796 -- 2844
 5 : 3122
 6 : 3851 -- 4194
 7 :
 8 :
 9 : 6051
10 :
11 : 7096
12 : 7580
13 : 8109 -- 8168 -- 8344
14 :
15 : 9557
16 : 9651
17 :
18 : 10971 -- 11360
19 :
20 : 12180
21 :
22 : 13458
23 : 14205
24 : 14652 -- 14721 -- 14920

output : [ 165,553,1770,1969,2796,2844,3122,3851,4194,6051,
          7096,7580,8109,8168,8344,9557,9651,10971,11360,
          12180,13458,14205,14652,14721,14920]
```

Sorting review

What we thought we knew about sorting:

The fastest sorting algorithms have running-time complexity O(n*lg(n)).

We were mistaken. That's the best we can do for *comparison-based* sorting. With well distributed numerical data (or with a suitable hash function) we can sort directly into locations in an array (or table) and, at least on average, obtain O(n) sorting.

However, we'd like to be able to have effective sorting algorithms in the most general cases:

- data has total order semantics, but not necessarily numerically ordered (e.g. strings)
- data is not necessarily distributed evenly.

This brings us back to comparison-based sorts --- sorts that depend solely on our ability to determine whether one item is less than another. (No dependence on the kind of data.)

We have seen two quadratic (O(n^2)) algorithms:

- *selection sort* - always O(n^2). Find the largest element, the next largest element, etc.
- *insertion sort* - good on nearly sorted data. Mimics sorting hand of cards.

and two faster algorithm:

- *heap sort* (a.k.a. `pqSort`) - O(n*lg(n)). Relatively fast and easy - if we have a properly functioning heap.
- *binary tree sort* - O(n*lg(n)). Relatively fast and easy - if we have a properly functioning, nearly balanced, binary search trees.

Selection Sort

```
template <class T>
void selectionSort(T A[ ], size_t n)
{
  for (size_t i = 0; i < n; i++) {
     T min = A[i];
     size_t minIndex = i;
     for (size_t j = i+1; j < n; j++)
       if (A[j] < min ) {
         min = A[j];
         minIndex = j;
       }
     swap(A[i], A[minIndex]);
  }
}
```

For example, consider selection sort on the following five-element array of integers:

```
[28,92,97, 3, 0]

------

[ 0,92,97, 3,28]
[ 0, 3,97,92,28]
[ 0, 3,28,92,97]
[ 0, 3,28,92,97]
[ 0, 3,28,92,97]
```

Insertion Sort

```
template <class T>
void insertionSort(T A[], size_t n)
{
  for (size_t i = 1; i < n; i++) {
    Item key = A[i];
    int j = i-1;
    for(; j>=0 && A[j]>key; j--)
      swap(A[j],A[j+1]);
    A[j+1] = key;
  }
}
```

For example, consider insertion sort on the following eight-element array of integers:

```
[478,218,491,467,401,252,108,196]

------

[218,478]
[218,467,478]
[218,401,467,478]
[218,252,401,467,478]
[108,218,252,401,467,478]
```

```
[108,196,218,252,401,467,478]
[108,196,218,252,401,467,478,491]
```

Why so many sorting techniques?

Sorting depends on many things:

- kind of input
- size of input
- distribution of input: over how large a range? is input already (partially) sorted?
- avaliable memory/resources
- uniqueness: what happens if two different elements have the same key?
- hardware - some machines better than others for different types of sorts; in particular *parallelism* can be important

Divide and Conquer Algorithms

- divide a problem into smaller subproblems
- solve each subproblem (usually recursively)
- combine answers to subproblems to form overall answer

Merge Sort

- split input into two halves (form two lists L, R)
- sort smaller lists L and R
- *merge* the results of the two sorts

top-down example:

```
A : [   2,  98, 740, 769,  79, 318, 583,  48, 553,  30]

mergeSort(A)

L : [   2, 198, 740, 769,  79]
R : [ 318, 583,  48, 553,  30]

...

L : [   2,  79, 198, 740, 769]
R : [  30,  48, 318, 553, 583]

merge:

    [   2,  30,  48,  79, 198, 318, 553, 583, 740, 769]
```

Pseudocode

More specifically, merge sort involves two recursive calls to itself:

```
mergeSort array A of size n
{
  split A into two halves L and R
  if L has more than one element
    mergeSort L
  if R has more than one element
    mergeSort R
  merge L and R into A
}
```

a more detailed example

For example, consider merge sort on the following eight-element array of integers:

```
A : [ 507, 277, 284, 182, 158,  28, 183,  25]

------

mergeSort(A)
  A : [ 507, 277, 284, 182, 158,  28, 183,  25]
  L : [ 507, 277, 284, 182]
  R : [ 158,  28, 183,  25]

  mergeSort(L)
    A : [ 507, 277, 284, 182]
    L : [ 507, 277]
    R : [ 284, 182]

    mergeSort(L)
      A : [ 507, 277]
      L : [ 507]
      R : [ 277]
      merge(L,R)
      A : [ 277, 577]

    mergeSort(R)
      A : [ 284, 182]
      L : [ 284]
      R : [ 182]
      merge(L,R)
      A : [ 182, 284]

    merge(L,R)
    L : [ 277, 577]
    R : [ 182, 284]
    A : [ 182, 277, 284, 577]

  mergeSort(R)
    A : [ 158,  28, 183,  25]
    L : [ 158,  28]
    R : [ 183,  25]

    mergeSort(L)
      A : [ 158, 28]
      L : [ 158]
      R : [  28]
      merge(L,R)
      A : [  28, 158]

    mergeSort(R)
      A : [ 183, 25]
      L : [ 183]
      R : [  25]
      merge(L,R)
      A : [  25, 183]

    merge(L,R)
    L : [  28, 158]
    R : [  25, 183]
    A : [  25,  28, 158, 183]

  merge(L,R)
  L : [ 182, 277, 284, 507]
  R : [  25,  28, 158, 183]
  A : [  25,  28, 158, 182, 183, 277, 284, 507]
```

merge pseudocode

```
merge L and R into A
{
  i,j,k=0
  while i < length(L) and j < length(R) {
    if L[i] < R[j] {
      A[k] = L[i];
      i++;
      k++;
    }
    else {
      A[k] = R[j];
      j++;
      k++;
    }
  }
  if there's anything left in L put it at end of A
  else put whatever is left in R at end of A
}
```

For example:

```
L : [   2,  79, 198]
R : [  30, 318]

merge L and R into A:


       0     1     2              0     1
    -------------------        -------------
L : |   2 |  79 | 198 |    R : |  30 | 318 |
    -------------------        -------------
       ^                          ^

       0     1     2     3     4
    -------------------------------
A : |   2 |     |     |     |     |
    -------------------------------
       ^
       0     1     2             0     1
    -------------------       -------------
L : |   2 |  79 | 198 |   R : |  30 | 318 |
    -------------------       -------------
             ^                   ^

       0     1     2     3     4
    -------------------------------
A : |   2 |  30 |     |     |     |
    -------------------------------
             ^


       0     1     2             0     1
    -------------------       -------------
L : |   2 |  79 | 198 |   R : |  30 | 318 |
    -------------------       -------------
             ^                   ^

       0     1     2     3     4
    -------------------------------
A : |   2 |  30 |     |     |     |
    -------------------------------
             ^


       0     1     2             0     1
    -------------------       -------------
L : |   2 |  79 | 198 |   R : |  30 | 318 |
    -------------------       -------------
             ^                         ^

       0     1     2     3     4
    -------------------------------
A : |   2 |  30 |  79 |     |     |
    -------------------------------
                   ^


       0     1     2             0     1
    -------------------       -------------
L : |   2 |  79 | 198 |   R : |  30 | 318 |
    -------------------       -------------
                   ^                   ^

       0     1     2     3     4
    -------------------------------
A : |   2 |  30 |  79 | 198 |     |
    -------------------------------
                         ^


       0     1     2             0     1
    -------------------       -------------
L : |   2 |  79 | 198 |   R : |  30 | 318 |
    -------------------       -------------
                        ^              ^

       0     1     2     3     4
    -------------------------------
A : |   2 |  30 |  79 | 198 | 318 |
    -------------------------------
                               ^
```

complexity of `merge`

`merge` is O(n) where n is the total number of elements in L and R.

Complexity of merge sort

We can view `mergeSort` as a binary tree of `merge` operations:

- The leaves are the individual items in the array.
- In between each level of the tree we perform k merges (going up from 2*k children to their k parents).
- Each merge will involve n/k items and cost O(n/k); each "level" of the tree has cost O(n).
- The number of levels is just the height of the tree which is roughly lg(n)

So the complexity of `mergeSort` is: O(n*lg(n))