Recitation 5

Direct Access Array Sort

In Lecture 5 we discussed how analyzing the decision tree of deterministic comparison sorting, we arrive at a lower bound of $\Omega(n \log n)$. However, if we are **not** limited to comparison operations, it is possible to beat the $\Omega(n \log n)$ bound. If the items to be sorted have **unique** keys from a bounded positive range $\{0, \ldots, u-1\}$ (so $n \leq u$), we can sort them simply by using a direct access array. Construct a direct access array with size u and insert each item x into index $x.key$. Then simply read through the direct access array from left to right returning items as they are found. Inserting takes time $\Theta(n)$ time while initializing and scanning the direct access array takes $\Theta(u)$ time, so this sorting algorithm runs in $\Theta(n+u)$ time. If $u = O(n)$, then this algorithm is linear! Unfortunately, this sorting algorithm has two drawbacks: first, it cannot handle duplicate keys, and second, it cannot handle large key ranges.

```
def direct_access_sort(A):
    "Sort A assuming items have distinct non-negative keys"
    u = 1 + max([x.key for x in A])         # O(n) find maximum key
    D = [None] * u                          # O(u) direct access array
    for x in A:                             # O(n) insert items
        D[x.key] = x
    i = 0
    for key in range(u):                    # O(u) read out items in order
        if D[key] is not None:
            A[i] = D[key]
            i += 1
```

Counting Sort

To solve the first problem, we simply link a chain to each direct access array index, just like in hashing. When multiple items have the same key, we store them both in the chain associated with their key. Later, it will be important that this algorithm be **stable**: that items with duplicate keys appear in the same order in the output as the input. Thus, we choose chains that will support a sequence **queue interface** to keep items in order, inserting to the end of the queue, and then returning items back in the order that they were inserted.

```python
def counting_sort(A):
    "Sort A assuming items have non-negative keys"
    u = 1 + max([x.key for x in A])          # O(n) find maximum key
    D = [[] for i in range(u)]               # O(u) direct access array of chains
    for x in A:                              # O(n) insert into chain at x.key
        D[x.key].append(x)
    i = 0
    for chain in D:                          # O(u) read out items in order
        for x in chain:
            A[i] = x
            i += 1
```

Counting sort takes $O(u)$ time to initialize the chains of the direct access array, $O(n)$ time to insert all the elements, and then $O(u)$ time to scan back through the direct access array to return the items; so the algorithm runs in $O(n + u)$ time. Again, when $u = O(n)$, then counting sort runs in linear time, but this time allowing duplicate keys.

There's another implementation of counting sort which just keeps track of how many of each key map to each index, and then moves each item only once, rather the implementation above which moves each item into a chain and then back into place. The implementation below computes the final index location of each item via cumulative sums.

```python
def counting_sort(A):
    "Sort A assuming items have non-negative keys"
    u = 1 + max([x.key for x in A])          # O(n) find maximum key
    D = [0] * u                              # O(u) direct access array
    for x in A:                              # O(n) count keys
        D[x.key] += 1
    for k in range(1, u):                    # O(u) cumulative sums
        D[k] += D[k - 1]
    for x in list(reversed(A)):              # O(n) move items into place
        A[D[x.key] - 1] = x
        D[x.key] -= 1
```

Now what if we want to sort keys from a larger integer range? Our strategy will be to break up integer keys into parts, and then sort each part! In order to do that, we will need a sorting strategy to sort tuples, i.e. multiple parts.

Tuple Sort

Suppose we want to sort tuples, each containing many different keys (e.g. $x.k_1, x.k_2, x.k_3, \ldots$), so that the sort is lexicographic with respect to some ordering of the keys (e.g. that key k_1 is more important than key k_2 is more important than key k_3, etc.). Then **tuple sort** uses a stable sorting algorithm as a subroutine to repeatedly sort the objects, first according to the **least important key**, then the second least important key, all the way up to most important key, thus lexicographically sorting the objects. Tuple sort is similar to how one might sort on multiple rows of a spreadsheet by different columns. However, tuple sort will only be correct if the sorting from previous rounds are maintained in future rounds. In particular, tuple sort requires the subroutine sorting algorithms be stable.

Radix Sort

Now, to increase the range of integer sets that we can sort in linear time, we break each integer up into its multiples of powers of n, representing each item key its sequence of digits when represented in base n. If the integers are non-negative and the largest integer in the set is u, then this base n number will have $\lceil \log_n u \rceil$ digits. We can think of these digit representations as tuples and sort them with tuple sort by sorting on each digit in order from least significant to most significant digit using counting sort. This combination of tuple sort and counting sort is called radix sort. If the largest integer in the set $u \leq n^c$, then radix sort runs in $O(nc)$ time. Thus, if c is constant, then radix sort also runs in linear time!

```
def radix_sort(A):
    "Sort A assuming items have non-negative keys"
    n = len(A)
    u = 1 + max([x.key for x in A])              # O(n) find maximum key
    # take log_2 of u and n using bit_length
    # and apply base change formula to get log_n(u)
    c = 1 + (u.bit_length() // n.bit_length())
    class Obj: pass
    D = [Obj() for a in A]
    for i in range(n):                           # O(nc) make digit tuples
        D[i].digits = []
        D[i].item = A[i]
        high = A[i].key
        for j in range(c):                       # O(c) make digit tuple
            high, low = divmod(high, n)
            D[i].digits.append(low)
    for i in range(c):                           # O(nc) sort each digit
        for j in range(n):                       # O(n) assign key i to tuples
            D[j].key = D[j].digits[i]
        counting_sort(D)                         # O(n) sort on digit i
    for i in range(n):                           # O(n) output to A
        A[i] = D[i].item
```

We've made a CoffeeScript Counting/Radix sort visualizer which you can find here:

`https://codepen.io/mit6006/pen/LgZgrd`

Exercises

1) Sort the following integers using a base-10 radix sort.

$$(329, 457, 657, 839, 436, 720, 355) \longrightarrow (329, 355, 436, 457, 657, 720, 839)$$

2) Describe a linear time algorithm to sort n integers from the range $[-n^2, \ldots, n^3]$.

3) Describe a linear time algorithm to sort a set n of strings, each having k English characters. Remember this should be linear time in the **size of the input** - which is not necessarily $\Theta(n)$.

4) You're given an exactly 8 character long alphanumeric list of n kerberos usernames. Two kerberos are considered to be *similar* if they differ by exactly one character. For instance, `fun6006` and `fun6046` are similar, but `fun6006` and `6006fun` are not similar. Describe a worst-case $O(n)$ algorithm to return a list of all kerberoses that are similar to at least one other kerberos in the list.