

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 $\{0, \dots, u-1\}$ (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 $\Theta(n)$ 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.

```

1 def direct_access_sort(A):
2     "Sort A assuming items have distinct non-negative keys"
3     u = 1 + max([x.key for x in A])      # O(n) find maximum key
4     D = [None] * u                      # O(u) direct access array
5     for x in A:                          # O(n) insert items
6         D[x.key] = x
7     i = 0
8     for key in range(u):                  # O(u) read out items in order
9         if D[key] is not None:
10             A[i] = D[key]
11             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.

```

1 def counting_sort(A):
2     "Sort A assuming items have non-negative keys"
3     u = 1 + max([x.key for x in A])      # O(n) find maximum key
4     D = [[] for i in range(u)]          # O(u) direct access array of chains
5     for x in A:                          # O(n) insert into chain at x.key
6         D[x.key].append(x)
7     i = 0
8     for chain in D:                      # O(u) read out items in order
9         for x in chain:
10            A[i] = x
11            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.

```

1 def counting_sort(A):
2     "Sort A assuming items have non-negative keys"
3     u = 1 + max([x.key for x in A])      # O(n) find maximum key
4     D = [0] * u                          # O(u) direct access array
5     for x in A:                          # O(n) count keys
6         D[x.key] += 1
7     for k in range(1, u):                # O(u) cumulative sums
8         D[k] += D[k - 1]
9     for x in list(reversed(A)):          # O(n) move items into place
10        A[D[x.key] - 1] = x
11        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. $x.k_1, x.k_2, x.k_3, \dots$), 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!

```

1 def radix_sort(A):
2     "Sort A assuming items have non-negative keys"
3     n = len(A)
4     u = 1 + max([x.key for x in A])          # O(n) find maximum key
5     # take log_2 of u and n using bit_length
6     # and apply base change formula to get log_n(u)
7     c = 1 + (u.bit_length() // n.bit_length())
8     class Obj: pass
9     D = [Obj() for a in A]
10    for i in range(n):                        # O(nc) make digit tuples
11        D[i].digits = []
12        D[i].item = A[i]
13        high = A[i].key
14        for j in range(c):                    # O(c) make digit tuple
15            high, low = divmod(high, n)
16            D[i].digits.append(low)
17    for i in range(c):                        # O(nc) sort each digit
18        for j in range(n):                    # O(n) assign key i to tuples
19            D[j].key = D[j].digits[i]
20        counting_sort(D)                      # O(n) sort on digit i
21    for i in range(n):                        # O(n) output to A
22        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 $[-n^2, \dots, n^3]$.

Solution: Add n^2 to each number so integers are all positive, apply Radix sort, and then subtract n^2 from each element of the output.

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 $\Theta(n)$.

Solution: Use tuple sort to repeatedly sort the strings by each character from right to left with counting sort, using the integers $\{0, \dots, 25\}$ to represent the English alphabet. There are k rounds of counting sort, and each round takes $\Theta(n + 26) = \Theta(n)$ time, thus the algorithm runs in $\Theta(nk)$ time. This running time is linear because the input size is $\Theta(nk)$.

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.

Solution: We assume the number of possible characters k to be small compared to the number of Kerberos usernames, $k \ll n$. A Kerberos username will be similar to another when it is different only by a single character. Suppose a similar pair differs at character i . If we sort the usernames character-by-character, but prioritize character i last below all others, similar usernames differing at character i will be adjacent to each other in the sort as all other characters in a similar pair are the same by definition. We can perform such a sort in worst-case $O(n + k)$ time, by applying radix sort with a counting sort subroutine on the letters, prioritizing character i last. Then scan through the sorted list comparing adjacent usernames. If two adjacent usernames are identical except at character i , then they are similar. To keep track of whether usernames have been found to be similar, append the character 0 to the end of each username. If a scan finds a pair that are similar, change their appended character to 1. We repeat this process eight times, each time prioritizing character $i \in 0, \dots, 7$ last. After all eight radix sorts and scans have been complete, loop through the sorted list in linear time and append to an output list any username for which a 1 is appended. We perform a constant number of radix sorts and linear scans, so the entire algorithm takes worst-case linear time.