Opening Questions

- If we have 17 distinct items to sort, what is the height of the decision tree that represents all possible orderings of the 17 items? The height of the decision tree is $\lceil \log_2(17!) \rceil = \lceil 48.337603311133 \rceil = 49$ and the possible orderings of the 17 items is 17!. However, $\log_2(n!) = \theta(n \lg n)$.

Comparison Sorts

The sorted order they determine is based only on comparisons between the input elements. That is, given two elements a_i and a_j, we perform one of the tests $a_i < a_j$, $a_i = a_j$, or $a_i > a_j$ to determine their relative order.

Claim: We must make at least $\Omega(n \lg n)$ comparisons in the general case.

The decision-tree model

- Abstraction of any comparison sort.
- Represents comparisons made by a specific sorting algorithm on inputs of a given size.
- Abstracts away everything else: control and data movement
- View the tree as if the algorithm splits in two at each node, based on the information it has determined up to that point; The tree models all possible execution traces.

Each internal node is labeled by indices of array elements **from their original positions**. Each leaf is labeled by the permutation of orders that the algorithm determines.

1. How many possible permutations of n items are there (at least one of them must be sorted)? In the decision tree model, how many leaves are there for n items? For a binary tree to have that many leaves, how many levels does it need? Why do we care about how many levels are in the decision tree? There are n! possible permutations. A binary tree would have 2^{height} levels.

Linear Time Sorting – Counting Sort

If we do not use comparison of keys to sort data, how can we sort data? It seems comparing items is necessary for sorting.

- Counting sort assumes that each of the n input elements is an integer in the range 0 to k, for some integer k. When $k = O(n)$, the sort runs in $T(n)$ time (best if the size of the range is $k \lll n$)
- For each input element x, count how many elements are equal to x then use this to count how many elements are $\leq x$. This information can be used to place element x directly into its position in the output array.

- Does not sort in place, needs 2nd array size n and 3rd array size k

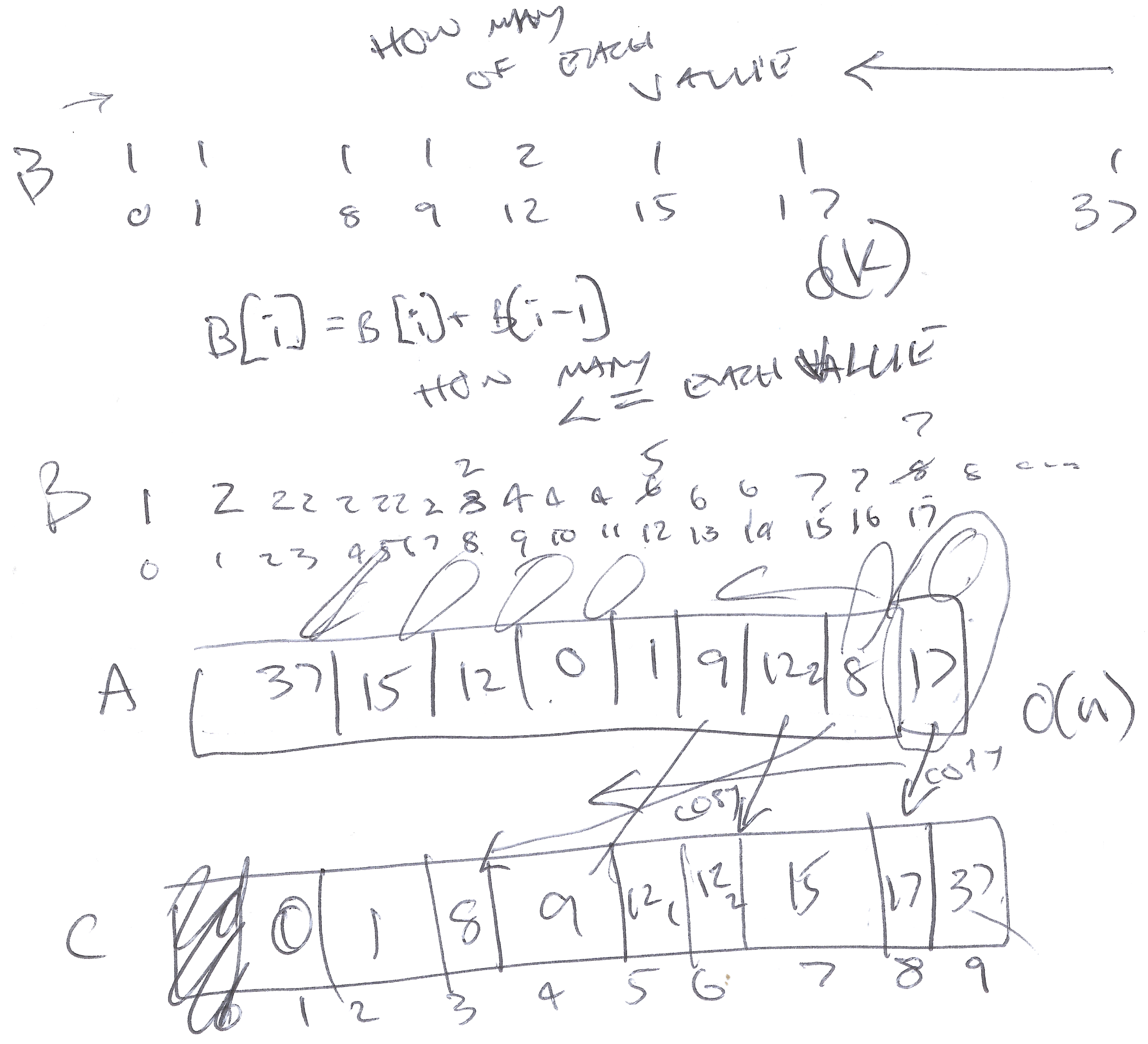

Figure 1: Counting sort demonstration.

2. Write pseudocode for counting sort.

Algorithm 1 Counting Sort

```
function CountingSort($A$)    ▷ $A$=original array, $B$=size-$k$ counters, $C$=size-$n$ sorted data
    for $i = 1$ to $n$ do
        $B[A[i]]$++                               ▷ counters
    end for
    for $j = 1$ to $k$ do
        $B[j] = B[j] + B[j-1]$                    ▷ $\leq$ counters
    end for
    for $i = n$ to 1 do
        $C[B[A[i]]] = A[i]$
        $B[A[i]]$--
    end for
end function
```

3. Demo counting sort on this data:

2	5	3	0	2	3	0	3

Counting sort is best on large amounts of data where the range of the data is small. Like many single digit numbers. However, we can use counting sort multiple times to sort multiple digit numbers, starting with the right most digit, etc. This is called Radix Sort.

Table 1: Radix Sort

329		720		720		329
457		355		329		355
657		436		436		436
839	$\rightarrow$	457	$\rightarrow$	839	$\rightarrow$	457
436		657		355		657
720		329		457		720
355		839		657		839

Each $O(k)$ where k is $0 \rightarrow 9$. This makes the overall $O = O(n + \text{digits} * k) = O(n + 10k)$, but k is insignificant to n, so it's essentially $O(n)$.

An important property of counting sort is that it is **stable**: numbers with the same value appear in the output array in the same order as they do in the input array. That is, it breaks ties between two numbers by the rule that whichever number appears first in the input array appears first in the output array. Normally, the property of stability is important only when satellite data are carried around with the element being sorted. Counting sort's ability is important for another reason: counting sort is often used as a subroutine in radix sort. As we shall see in the next section, in order for radix sort to work correctly, counting sort must be stable.

4. Which of these sorts can be stable:

- Insertion Sort
- Selection Sort ← you could potentially jump an item passed its duplicate
- Bubble Sort
- Merge Sort ← depends on the merge
- Quick Sort ← the paritioning might not be stable and an item can be jumped passed its duplicate.
- Heap Sort