

Linear Sorting

Reading: *CLRS* Ch. 8, *CLR* Ch. 9

The Best Worst-Case Running Time for Comparison-Based Sorts

What is the minimum height of a binary tree with L leaves?

Given n distinct elements, how many different arrays can we make with these elements (with no duplicates)? (For example, consider $\{1, 2, 3\}$.)

Can view comparison-based sorts as *binary decision trees* that process sets of arrays over n distinct elements. For example:

Note that each leaf of the tree must hold a singleton array.

What is the minimum height of a decision tree for arrays of length n ?

(Note that $\lg(n!) = n \cdot \lg(n)$ via Stirling's equation or calculus (see *CLRS* Exercise 8.1-2/*CLR* Exercise 9.1-2).)

Digression: Finding the Odd Nugget

Suppose you are given n nuggets, all of which have exactly the same weight, except for one “odd” one, which may be slightly lighter or heavier than the rest. You are allowed to perform experiments in which you compare the weights of two collections with k nuggets each: one collection may be $<$, $=$, or $>$ the other. How many experiments do you need to find the odd nugget?

- Show via a counting argument that 3 experiments is not sufficient for finding the odd nugget when $n = 14$.

- By case analysis on the first experiment, show via a counting argument that 3 experiments is not sufficient for finding the odd nugget when $n = 13$.

- It is possible to find the odd nugget with 3 experiments when $n = 12$. This is left as an exercise for the reader.

Linear Sorting Algorithms

All comparison-based sorting algorithms have worst-case time $\Omega(n \cdot \lg(n))$.

Yet there are linear sorting algorithms – algorithms with worst-case time $\Theta(n)$. How can this be? They're not comparison-based! They take advantage of some feature of the input. E.g.:

- The elements are integers in a restricted range, or easily convertible to integers in a restricted range (e.g., the letters of the alphabet).
- The elements have a special probability distribution.

Integer Bucket Sort

Problem: sort n integers taken from the range $1..k$.

Example: 4 2 6 3 2 4 2 6 1 2 4

Solution:

```
Integer-Bucket-Sort(A,B,k)
  ▷ A is input array of length n
  ▷ containing keys in range [1..k].
  ▷ B is output array of length n.
  Count ← new array(k)
  for i ← 1 to k do
    Count[i] ← 0
  ▷ Find count of each key in A.
  for j ← 1 to length[A] do
    Count[A[j]] ← Count[A[j]] + 1
  ▷ Read out result.
  j ← 1
  for p ← 1 to k do
    for q ← 1 to Count[p]
      B[j] ← p
      j ← j + 1
```

Can A be the same as B in this algorithm?

Counting Sort (a.k.a. Distribution Counting)

Integer bucket sort has a problem when there is satellite data. E.g., suppose (k, n) pairs a key k with a satellite integer n :

(4,1) (2,1) (6,1) (3,1) (2,2) (4,2) (2,3) (6,2) (1,1) (2,4) (4,3)

(Here, it happens that the satellite data for a particular key is sorted left to right, but that's just "coincidence".)

```
Counting-Sort(A,B,k)
  ▷ A is input array of length n
  ▷ containing keys in range [1..k].
  ▷ B is output array of length n.
  Count ← new array(k)
  for i ← 1 to k do
    Count[i] ← 0
  ▷ Find count of each key in A.
  for j ← 1 to length[A] do
    Count[A[j]] ← Count[A[j]] + 1
  ▷ Find partial sums of key counts in A.
  for i ← 2 to k do
    Count[i] ← Count[i] + Count[i-1]
  ▷ Put elements in final position.
  for j ← length[A] downto 1 do
    B[Count[A[j]]] ← A[j]
    Count[A[j]] ← Count[A[j]] - 1
```

Can A be the same as B in this algorithm?

How much time does Counting-Sort need?

How much space does Counting-Sort need?

Is Counting-Sort stable?

Would Counting-Sort be stable if the **for** $j \dots$ loop counted up rather than down?

General Bucket Sort

Can't use counting sort if keys aren't integers in a given range. What if keys are real numbers in a given range?

If keys are evenly distributed can use a general bucket sort:

Step 1: Given n elements, make an array of n buckets, each of which contains an empty list.

Step 2: Insert each element into the list of the bucket chosen by matching its key with the result of dividing the range into n equal-sized intervals

Step 3: Use any $O(n^2)$ sorting algorithm to sort the resulting lists in the buckets.

Step 4: Read the elements from the lists in order back into the input array.

Example: Sorting 10 numbers between 0 and 1:

.63
.75
.16
.65
.61
.17
.89
.92
.28
.56

- In the best case, there is one element per bucket, and running time is $\Theta(n)$.
- In the worst case, all elements end up in same bucket and running time is $\Theta(n^2)$.
- Assuming evenly distributed keys, can use probability analysis to show average case running time is $\Theta(n)$. (See *CLRS/CLR* for details.)

Radix Sort

Problem: Suppose you have a machine that can perform a stable sort on the i th digit of a d -digit number. How can you use the machine to sort a “pile” of n d -digit numbers?

Example:

443
124
232
431
132
123
321
211
121
441
122
442

To avoid lots of intermediate piles, process digits right-to-left, not left-to-right!

```
Radix-Sort(A, d)
  for i ← 1 to d do
    Stable-Sort(A, i) ▷ use digit i for comparison
```

Any stable sort will do. **Counting-Sort** is a good candidate since it's $\Theta(n + k)$.

What is the running time for sorting n d -digit numbers, where digits are in the range $[1..k]$?

Note:

- If d is constant and $k = O(n)$, then running time of **Radix-Sort** is $\Theta(n)$.
- In many applications, $d = \log_k(n)$, so the sort ends up being $\Theta(n \cdot \log_k(n))$ for a given k .