**Sorting in Linear Time**

Data Structures and Algorithms
Andrei Bulatov

---

**The Decision Tree Model**

Observe that for different input permutations, even if they differ only in the order of elements, our algorithm must perform different actions. Therefore, we view its work as recognizing a permutation. It is convenient to represent this process as a decision tree.

---

**Decision Trees**

Let the input sequence contain $n$ elements. We assume they are numbers from $1$ to $n$. Therefore, the input sequence is a permutation $(\pi(1), \pi(2), ..., \pi(n))$.

Each internal node is labeled by $i:j$ for some $i$ and $j$ in the range $1 \leq i, j \leq n$.

Execution of the algorithm is a path from the root to a leaf. At each internal node a comparison $a_i \leq a_j$ is made. Depending on the outcome either comparisons on the left or the right subtree are made.

When we come to a leaf, the algorithm recognizes a particular ordering.

---

**Lower Bound**

**Theorem**

Any comparison sort algorithm requires $\Omega(n \log n)$ comparisons in the worst case.

**Proof**

There are $n!$ permutations of $n$ elements.

The algorithm should recognize all of them.

Therefore the decision tree contains $k \geq n!$ leaves.

The complete binary tree has $2^h$ leaves where $h$ is the height of the tree, the length of the longest root-to-leaf path.

$n! \leq k \leq 2^h$

$h \geq \log(n!) = \Omega(n \log n)$

Finally, note that the height of the tree is running time in the worst case.

---

**Counting Sort**

Suppose we are allowed to do more than just comparisons.

We also assume that the input numbers are in the range $0$ to $k$.

We give an algorithm that works in $\Theta(n)$ time provided $k = O(n)$.

The idea is: for each input element $x$ to count how many elements are less than $x$, and use this information to place $x$ to the right place in the output sequence.

We use $A[1..n]$ the input array

$B[1..n]$ the output array

$C[0..k]$ auxiliary storage.

Note that we do not assume that all elements are different.
**Counting Sort: Algorithm**

```
Counting-Sort(A, B, k)
for i = 0 to k do
    set C[i] := 0
for j = 1 to n do
    set C[A[j]] := C[A[j]] + 1  /* C[i] contains the number of elements equal to i */
for i = 1 to k do
    set C[i] := C[i] + C[i-1]  /* C[i] contains the number of elements less than or equal to i */
for j = n downto 1 do
    set C[A[j]] := C[A[j]] - 1
endfor
```

**Counting Sort: Soundness and Running Time**

**Proof**

- **Running time:**
  Each of the 4 for loops is iterated at most \( \max\{n, k\} \) times.

- **Soundness:**
  Trace the content of arrays.

**Theorem**

Counting Sort correctly sorts a sequence of \( n \) elements in \( O(n) \) time, provided \( k = O(n) \).

**Stable Sorting**

For the next algorithm it is important how Counting Sort and other sorting algorithms deal with equal elements. A sorting algorithm is called **stable** if it preserves the order of equal elements in the input sequence, that is, if \( a_i = a_j \) and \( i < j \) in the input sentence, then \( a_i \) goes first in the output sequence.

**Lemma**

Counting Sort is stable.

**Proof**

Suppose that \( A[i] = A[j] \) and \( i < j \). Then in the last for loop we first output \( A[j] \) then \( A[i] \). Moreover, between the outputs \( C[A[j]] = C[A[i]] \) is decremented. Therefore \( A[j] \) is placed into \( B \) with smaller index than \( A[i] \).

**Radix Sort**

If the elements of the input sequence are \( d \)-digit integers, we can first sort the according to the most significant digit, then the second most significant digit, etc. This, however, requires a lot of copying and auxiliary storage.

Radix Sort:

Sort in place according to the least significant digit but use a stable sorting algorithm!!

| 329 | 72 | 39 |
| 457 | 35 | 33 |
| 657 | 43 | 41 |
| 856 | 45 | 41 |
| 456 | 49 | 47 |
| 720 | 32 | 73 |
| 355 | 83 | 83 |

**Radix Sort: Algorithm**

```
Radix-Sort(A, d)
for i = 0 to d do
    use a stable sorting algorithm to sort array A on digit i
endfor
```

**Theorem**

Radix Sort correctly sorts a sequence of \( n \) \( d \)-digit numbers in which each digit can take up to \( k \) different values in \( O(d(n+k)) \) time.
Radix Sort: Analysis

Running time:
RadixSort considers d digits in turn
each pass takes O(n + k) time (when using, say, Counting Sort)

Soundness:
By induction on the number d of digits
Base Case:  d = 1  Counting Sort just sorts everything
Induction Step: Suppose algorithm works correctly for d – 1
Radix sort on d-digit numbers is equivalent to d runs of radix sort on
smaller d-1 numbers, followed by sorting on digit d.
By Induction Hypothesis Radix Sort sorts correctly on lower d – 1
digits

Radix Sort: Analysis (cntd)

Before the last sort on digit d, all the numbers are properly sorted
accordingly their last d – 1 digits.
When sort on digit d, consider two elements a and b with d-th digits
a_d and b_d respectively
(1) If a_d < b_d then the algorithm will put a before b, which is
correct
(2) If a_d > b_d, the algorithm will put b before a, which is again
correct
(3) If a_d = b_d, the algorithm will leave a and b in the same order
as before, because it is stable.
This order is again correct, for the relative order of a and b
depends in this case on the lower d – 1 digits.
QED

Bucket Sort: The Idea

Counting Sort and Radix Sort achieve significant speed up against
comparison algorithms because they use certain assumptions about
the input numbers:
They are small integers, or integers of bounded size.
Bucket Sort uses an assumption about the distribution of these
numbers:
They are taken uniformly at random from [0; 1)
Then we:
Split [0;1) into n equal intervals (buckets)
Put every input element into the corresponding bucket
Sort each bucket
Concatenate the buckets

Bucket Sort: Algorithm

A[1..n] the input array
B[1..n] heads of buckets
i-th bucket is organized as a list with a pointer B[i] to the top of the
list

Bucket-Sort(A,d)
set n:=length(A)
for i=0 to n do
   insert A[i] into list B[ floor(n*A[i]) ]
endfor
for i=0 to n do
   sort list B[i] with insertion sort
concatenate the lists B[1],B[2],...,B[n]

Bucket Sort: Soundness

Theorem
Bucket Sort correctly sorts a sequence of numbers from the interval
[0;1)

Proof
Obvious

Bucket Sort: Running Time

Theorem
The expected running time of Bucket Sort is O(n)

Proof
Clearly, the first for loop takes Θ(n) time to complete
Each iteration of the second for loop contributes O(n^2) time where n_i
is the number of elements in the i-th bucket.
Therefore

\[ T(n) = \Theta(n) + \sum_{i=0}^{n-1} O(n_i^2) \]
Bucket Sort: Running Time

**Proof (cntd).**

Take the expectation of both sides

\[ E[T(n)] = E\left[\Theta(n) + \sum_{i=0}^{n-1} O(\sigma_i^2)\right] \]

\[ = \Theta(n) + E\left[\sum_{i=0}^{n-1} O(\sigma_i^2)\right] \]

\[ = \Theta(n) + \sum_{i=0}^{n-1} O(E[\sigma_i^2]) \]

We show that \( E[\sigma_i^2] = 2 - \frac{1}{n} \)

---

**Bucket Sort: Running Time**

**Proof (cntd).**

Define indicator random variables

\[ X_{ij} = 1 \text{ if and only if } A[j] \text{ falls into bucket } i \]

\[ \text{otherwise } X_{ij} = 0 \]

Thus

\[ n = \sum_{j=1}^{n} X_{ij} \]

We get

\[ E[\sigma_i^2] = E\left[\sum_{j=1}^{n} X_{ij}\right]^2 \]

\[ = E\left[\sum_{j=1}^{n} X_{ij}X_{ik}\right] = E\left[\sum_{j=1}^{n} X_{ij}^2 + \sum_{j=1}^{n} \sum_{j=1 \leq k \leq n, k \neq j} X_{ij}X_{ik}\right] \]

\[ = \sum_{i=1}^{n} E[X_{ij}^2] + \sum_{j=1 \leq k \leq n, k \neq j} \sum_{i=1}^{n} E[X_{ij}X_{ik}] \]

---

**Bucket Sort: Running Time**

**Proof (cntd).**

Then

\[ E[X_{ij}^2] = 1 \cdot \frac{1}{n} + 0 \cdot (1 - \frac{1}{n}) = \frac{1}{n} \]

and, since \( X_{ij} \) and \( X_{ik} \) are independent

\[ E[X_{ij}X_{ik}] = E[X_{ij}] \cdot E[X_{ik}] = \frac{1}{n} \cdot \frac{1}{n} = \frac{1}{n^2} \]

Finally

\[ E[\sigma_i^2] = \sum_{j=1}^{n} E[X_{ij}^2] + \sum_{j=1 \leq k \neq j} \frac{1}{n^2} \]

\[ = n \cdot \frac{1}{n} + n(n-1) \cdot \frac{1}{n^2} \]

\[ = 1 + \frac{n-1}{n} = 2 - \frac{1}{n} \]

---

**Bucket Sort: Running Time**

**Proof (cntd).**

For the running time we now have

\[ E[T(n)] = \Theta(n) + \sum_{i=0}^{n-1} O(E[\sigma_i^2]) \]

\[ = \Theta(n) + \sum_{i=0}^{n-1} O(2 - \frac{1}{n}) \]

\[ = \Theta(n) + O(n) = O(n) \]

QED