Linear-time Sorting

(read Section 8 on Sorting in Linear Time in Cormen, Leiserson, Rivest, Stein)

We have seen two algorithms which both sort \( n \) items in time proportional to \( n \log_2 n \). Can we be sure that there are no faster algorithms? We cannot, but we can show that every algorithm that sorts by comparing pairs of items takes at least some constant times \( \log_2 n \) comparisons.

After proving this statement, we will discuss an algorithm that sorts in time \( O(n) \).

**Decision Tree.** We use a tree to visualize a comparison-based algorithm. Assuming all items are different, each comparison has only two possible outcomes: \( a < b \) or \( a > b \). With this assumption, we can draw every comparison-based sorting algorithm as a binary tree in which each internal node is a comparison and each external node is a permutation of the input items. As an example, consider insertion sort, which runs fast for sorted and almost sorted input sequences and slow for random ones.

```void INSERTIONSORT(int n)
    A[0] = -\infty;
    for \( j = 2 \) to \( n \) do
        \( i = j \); \text{ while} A[i] < A[i - 1] \text{ do}
        \text{SWAP}(i - 1, i); \( i -- \)
    endfor.
```

The decision tree of insertion sort for three items \( a_i = A[i] \) for \( i = 1, 2, 3 \) is shown in Figure 10.

**Lower Bound.** The decision tree picture suggests we think of an algorithm that sorts \( n \) items as an algorithm that searches among \( n! \) permutations. The size of the tree relates to the size of the problem. Define the **depth** of a node \( \mu \) as the length of the path from the root to \( \mu \). The **height** of the tree is the maximum depth of any node, \( h = \max_\mu d(\mu) \), and the node that defines the maximum is necessarily external. The height is also the worst-case number of comparisons required by the algorithm. Instead of the maximum we may consider the average length of a path from the root to an external node. Define the **external path length** as the sum of depths over all external nodes, \( L = \sum d(\mu) \). The average number of comparisons taken by the algorithm is the average depth, which is \( L \) divided by the number of external nodes. We state a few properties relating size to height and external path length.

1. The decision tree of a deterministic sorting algorithm has \( n! \geq (\frac{n}{e})^n \) external nodes.
2. The number of external nodes is at most \( 2^h \).
3. The external path length is at least \( \frac{n!}{\log_2 n!} \).

To see (i) approximate the natural logarithm of \( n! \) using integration: \( \ln n! = \sum_{i=1}^{n} \ln i \geq \int_{x=1}^{n} \ln x \, dx = [x \ln x - x]_1^n = n \ln n - n - 1 \). Therefore \( n! \geq e^{n \ln n - n} = (\frac{n}{e})^n \). Putting (i) and (ii) together we get \( 2^h \geq n! \), and taking logarithms we get \( h \geq \log_2 n! \geq n \log_2 n - n \log_2 e \). In words, the worst-case number of comparisons is at least \( n \log_2 n \) minus a linear term. Finally, we prove (iii) by
The grandchildren, etc.) with an external node. It follows that

ing out the items, we collect them from back to front.

Bucket Sort.

particular class, namely all comparison-based algorithms,
sometimes available.

summarize the results.

number of comparisons is

the external path length is a minimum only if the tree is

depth at least

exchanging an internal node (together with its children,
grand-children, etc.) with an external node. It follows that

Theorem. Every comparison-based algorithm sorting

items needs \( \Omega(n \log n) \) comparisons, both in the

worst and the average case.

The \( \Omega(n \log n) \) lower bound applies to all algorithms in a

particular class, namely all comparison-based algorithms,

which do not take advantage of extra information which is

sometimes available.

Bucket Sort. We consider the case in which this extra

information is a small and known collection from which

the items are drawn. This collection is referred to as the

universe. For example characters are encoded in one byte

each, and the universe of characters consists of all \( 2^8 = 256 \)
different bytes. Another example is the universe of 52

playing cards.

The idea of bucket sort is to reserve an array \( B[0..N-1] \)
of buckets, where \( N \) is the size of the universe. Each

bucket is a labeled stack that can hold any number of

items. Bucket sort uses a bucket only for items equal to

the label of the bucket. The items are sorted in two steps:

spreading and collecting. To spread, we assume the items

are initially given on a stack \( S \).

void SPREAD(Buckets B, Stack S)

while not isEmpty(S) do

PUSH(Pop(S), B[f(a)])

endwhile.

Here \( f : U \rightarrow \{0, 1, \ldots, N - 1\} \) is a bijection that maps

the universe, \( U \), to a set of card \( U \) integers. After spreading

out the items, we collect them from back to front.

void COLLECT(Buckets B, Stack S)

for i = N - 1 downto 0 do

while not isEmpty(B[i]) do

PUSH(Pop(B[i]), S)

endwhile

endfor.

Note that each time we move the contents of one stack to

another we reverse the ordering. Since bucket sort uses

two steps, the second step reverses the reversal and re-

stores the original ordering. In other words, bucket sort

preserves the order of identical items. An algorithm that

has this property is said to be stable.

Function SPREAD takes constant time per item and

Function COLLECT takes constant time per item and per

bucket. This adds up to time \( O(n + N) \), where \( n \) is the

number of items and \( N \) is the size of the universe.

Radix Sort. Frequently, a single item consists of several

components or keys. Examples are names consisting of

first name and surname or playing cards given by suit and

rank. Lexicographical ordering means all items are sorted

by first key, and items with identical first key are sorted

by second key, and so on. This suggests further rounds of

bucket sort for each bucket filled in the first round. Alter-

natively, we can repeatedly bucket sort the entire set from

least to most significant key. Let \( k \) be the number of keys

per item indexed in an order of decreasing significance.

void RADIXSORT(Buckets B, Stack S)

for i = k - 1 downto 0 do

SPREAD(B, S, i); COLLECT(B, S)

endfor.

Note that we introduced a new parameter, \( i \), that controls

the bijection mapping items to integers. In other words,

there is a bijection \( f_i \) for each \( 0 \leq i \leq k - 1 \), and if

Function SPREAD invokes \( f_i \) then items are spread out ac-

cording to the \( i \)-th key.

As an example consider sorting 32-bit integers. We can

choose \( N = 2^8 \) and \( k = 4 \) and sort in four rounds us-

ing an array of 256 buckets indexed by all possible bytes.

The example illustrated in Figure 11 chooses \( N = 4 \) and

sorts 3-digit integers written in base 4. Note that the stack

\( S \) after round two is sorted by the second digit, and all

items with identical second digit are sorted by the third
digit. Similarly, the stack \( S \) after round three is sorted by

the first digit, all items with identical first digit are sorted

by the second digit, and all items with identical first and

second digits are sorted by the third digit. This is true be-

cause bucket sort and therefore every round of radix sort

is stable.
Figure 11: From top to bottom: the buckets after spreading and the stack after collecting in the first, second, third round. Observe that $S$ contains the contents of each bucket in reverse order.