Introduction to Algorithms

George Voutsadakis[1]

[1]Mathematics and Computer Science
Lake Superior State University

LSSU Math 400

1. Sorting in Linear Time
   - Lower Bounds for Sorting
   - Counting Sort
   - Radix Sort
   - Bucket Sort

Comparison Sorts and $n \log n$ Bound

- We have now introduced several algorithms that can sort n numbers in $O(n \log n)$ time.
  - Merge sort and heapsort achieve this upper bound in the worst case.
  - Quicksort achieves it on average.
- Moreover, for each of these algorithms, we can produce a sequence of n input numbers that causes the algorithm to run in $\Theta(n \log n)$ time.
- These algorithms share the property that the sorted order they determine is based only on comparisons between the input elements.

  Such sorting algorithms are called **comparison sorts**.
- We show next that any comparison sort must make $\Omega(n \log n)$ comparisons in the worst case to sort n elements.
- Thus, merge sort and heapsort are asymptotically optimal.
- Then we examine three sorting algorithms that use operations other than comparisons and run in linear time.

Subsection 1

Lower Bounds for Sorting

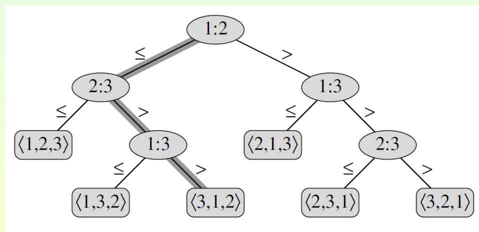
Comparisons Performed by a Comparison Sort

- In a comparison sort, we use only comparisons between elements to gain order information about an input sequence $\langle a_1, a_2, \ldots, a_n \rangle$ i.e., given two elements a_i and a_j, we perform one of the tests $a_i < a_j$, $a_i \leq a_j$, $a_i = a_j$, $a_i \geq a_j$, or $a_i > a_j$ to determine their relative order.
- We may not inspect the values of the elements or gain order information about them in any other way.
- If all the input elements are distinct, comparisons of the form $a_i = a_j$ are useless, so we can assume that no comparisons of this form are made.
- We also note that the comparisons $a_i \leq a_j$, $a_i \geq a_j$, $a_i > a_j$, and $a_i < a_j$ are all equivalent in that they yield identical information about the relative order of a_i and a_j.

  We therefore assume that all comparisons have the form $a_i \leq a_j$.

The Decision Tree Model

- A **decision tree** is a full binary tree that represents the comparisons between elements that are performed by a particular sorting algorithm.
- The figure shows the decision tree corresponding to a sorting algorithm operating on an input sequence of three elements.



- In a decision tree, each internal node is annotated by $i : j$ for some i and j in the range $1 \leq i, j \leq n$, where n is the number of elements in the input sequence.
- Each leaf is annotated by a permutation $\langle \pi(1), \pi(2), \ldots, \pi(n) \rangle$.
- The execution of the sorting algorithm corresponds to tracing a path from the root of the decision tree to a leaf.

Correctness of a Sorting Algorithm

- At each internal node, a comparison $a_i \leq a_j$ is made.
  - The left subtree then dictates subsequent comparisons for $a_i \leq a_j$.
  - The right subtree dictates subsequent comparisons for $a_i > a_j$.
- When we come to a leaf, the sorting algorithm has established the ordering $a_{\pi(1)} \leq a_{\pi(2)} \leq \cdots \leq a_{\pi(n)}$.
- Because any correct sorting algorithm must be able to produce each permutation of its input, a necessary condition for a comparison sort to be correct is that:
  - Each of the $n!$ permutations on n elements must appear as one of the leaves of the decision tree; and
  - each of these leaves must be reachable from the root by a path corresponding to an actual execution of the comparison sort. (We refer to such leaves as "reachable".)
- Thus, we shall consider only decision trees in which each permutation appears as a reachable leaf.

Using Trees to Lower Bound Comparison Sorts

- The length of the longest simple path from the root of a decision tree to any of its reachable leaves represents the worst-case number of comparisons that the corresponding sorting algorithm performs.
- Consequently, the worst-case number of comparisons for a given comparison sort algorithm equals the height of its decision tree.
- A lower bound on the heights of all decision trees in which each permutation appears as a reachable leaf is therefore a lower bound on the running time of any comparison sort algorithm.

A Lower Bound for the Worst Case

Theorem

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

- From the preceding discussion, it suffices to determine the height of a decision tree in which each permutation appears as a reachable leaf. Consider a decision tree of height h with ℓ reachable leaves corresponding to a comparison sort on n elements. Because each of the $n!$ permutations of the input appears as some leaf, we have $n! \leq \ell$. Since a binary tree of height h has no more than 2^h leaves, we have $n! \leq \ell \leq 2^h$. Taking logs, $h \geq \log(n!) = \Omega(n \log n)$.

Corollary

Heapsort and merge sort are asymptotically optimal comparison sorts.

- The $O(n \log n)$ upper bounds on the running times for heapsort and merge sort match our $\Omega(n \log n)$ worst-case lower bound.

Subsection 2

Counting Sort

Basic Idea and Input

- 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 $\Theta(n)$ time.
- Counting sort determines, for each input element x, the number of elements less than x.
- It uses this information to place element x directly into its position in the output array.
- For example, if 17 elements are less than x, then x belongs in output position 18.

Modification and Data Structures

- We must modify the preceding scheme slightly to handle the situation in which several elements have the same value, since we do not want to put them all in the same position.
- In the code for counting sort, we assume that the input is an array $A[1 \ldots n]$.

  So $A.\text{length} = n$.
- We require two other arrays:
  - The array $B[1 \ldots n]$ holds the sorted output;
  - The array $C[0 \ldots k]$ provides temporary working storage.

The Counting Sort Algorithm

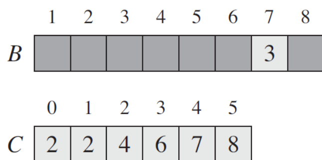
COUNTINGSORT(A, B, k)

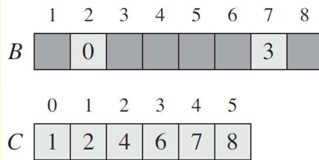
```
1.  let C[0...k] be a new array
2.  for i = 0 to k
3.      C[i] = 0
4.  for j = 1 to A.length
5.      C[A[j]] = C[A[j]] + 1
6.  // C[i] now contains the number of elements equal to i.
7.  for i = 1 to k
8.      C[i] = C[i] + C[i - 1]
9.  // C[i] now contains the number of elements less than or equal to i.
10. for j = A.length downto 1
11.     B[C[A[j]]] = A[j]
12.     C[A[j]] = C[A[j]] - 1
```

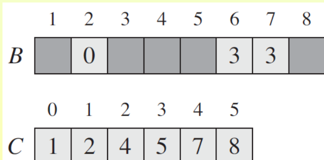
How Counting Sort Works

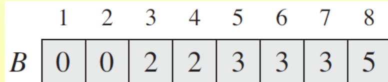
- The for loop of Lines 2-3 initializes the array C to all zeros;
- The for loop of Lines 4-5 inspects each input element.
  - If the value of an input element is i, we increment C[i].
    Thus, after Line 5, C[i] holds the number of input elements equal to i for each integer $i = 0, 1, \ldots, k$.
- Lines 7-8 determine for each $i = 0, 1, \ldots, k$ how many input elements are less than or equal to i by keeping a running sum of the array C.
- Finally, the for loop of Lines 10-12 places each element A[j] into its correct sorted position in the output array B.
  - If all n elements are distinct, then when we first enter Line 10, for each A[j], the value C[A[j]] is the correct final position of A[j] in the output array, since there are C[A[j]] elements less than or equal to A[j].
  - Because the elements might not be distinct, we decrement C[A[j]] each time we place a value A[j] into the B array. Decrementing C[A[j]] causes the next input element with a value equal to A[j], if one exists, to go to the position immediately before A[j] in the output array.

Illustration of Counting Sort









Time Requirements

- The for loop of Lines 2-3 takes time $\Theta(k)$.
- The for loop of Lines 4-5 takes time $\Theta(n)$.
- The for loop of Lines 7-8 takes time $\Theta(k)$.
- The for loop of Lines 10-12 takes time $\Theta(n)$.

  Thus, the overall time is $\Theta(n)$.
- In practice, we usually use counting sort when we have $k = O(n)$, in which case the running time is $\Theta(n)$.

Remarks

- Counting sort beats the lower bound of $\Omega(n \log n)$ because it is not a comparison sort.
- No comparisons between input elements occur anywhere in the code.
- Instead, counting sort uses the actual values of the elements to index into an array.
- 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.

  I.e., ties are broken between two numbers by the rule that whichever number appears first in the input array appears first in the output array.
- Counting sort's stability is important because counting sort is often used as a subroutine in radix sort.

  For radix sort to work correctly, counting sort must be stable.

Subsection 3

Radix Sort

Punch Cards

- Radix sort is the algorithm used by the card-sorting machines you now find only in computer museums.
- The cards have 80 columns, and in each column a machine can punch a hole in one of 12 places.
- The sorter can be mechanically "programmed" to examine a given column of each card in a deck and distribute the card into one of 12 bins depending on which place has been punched.
- An operator can then gather the cards bin by bin, so that cards with the first place punched are on top of cards with the second place punched, and so on.

Numbers in Decimal

- For decimal digits, each column uses only 10 places.
- A d-digit number would then occupy a field of d columns.
- Since the card sorter can look at only one column at a time, the problem of sorting n cards on a d-digit number requires a sorting algorithm.
- Intuitively, you might sort numbers on their most significant digit, sort each of the resulting bins recursively, and then combine the decks in order.
- Unfortunately, since the cards in 9 of the 10 bins must be put aside to sort each of the bins, this procedure generates many intermediate piles of cards that you would have to keep track of.

Idea of Radix Sort

- Radix sort solves the problem of card sorting by sorting on the least significant digit first.
- The algorithm then combines the cards into a single deck, with the cards in the 0 bin preceding the cards in the 1 bin preceding the cards in the 2 bin, and so on.
- Then it sorts the entire deck again on the second-least significant digit and recombines the deck in a like manner.
- The process continues until the cards have been sorted on all *d* digits.
- In order for radix sort to work correctly, the digit sorts must be stable. The sort performed by a card sorter is stable, but the operator has to be wary about not changing the order of the cards as they come out of a bin, even though all the cards in a bin have the same digit in the chosen column.

Input		Sorted on digit 1		Sorted on digit 2		Sorted on digit 3
329		720		720		329
457		355		329		355
657		436		436		436
839	⇒	457	⇒	839	⇒	457
436		657		355		657
720		329		457		720
355		839		657		839

The Procedure RadixSort

- The following procedure for radix sort assumes that each element in the n-element array A has d digits, where digit 1 is the lowest-order digit and digit d is the highest-order digit.

RadixSort(A, d)

```
1. for i = 1 to d
2.     use a stable sort to sort array A on digit i
```

Correctness and Performance of RadixSort

Lemma

Given n d-digit numbers in which each digit can take on up to k possible values, RadixSort correctly sorts these numbers in $\Theta(d(n+k))$ time if the stable sort it uses takes $\Theta(n+k)$ time.

- The correctness of radix sort follows by induction on the column being sorted.

  The analysis of the running time depends on the stable sort used as the intermediate sorting algorithm. When each digit is in the range 0 to $k-1$ (so that it can take on k possible values), and k is not too large, counting sort is the obvious choice.

  - Each pass over n d-digit numbers then takes time $\Theta(n+k)$.
  - There are d passes.

  So the total time for radix sort is $\Theta(d(n+k))$.

Flexibility in Breaking Keys into Digits

- When d is constant and $k = O(n)$, radix sort can run in linear time.
- More generally, we have some flexibility in how to break each key into digits.

Lemma

Given n b-bit numbers and any positive integer $r \leq b$, RadixSort correctly sorts these numbers in $\Theta(\frac{b}{r}(n + 2^r))$ time if the stable sort it uses takes $\Theta(n + k)$ time for inputs in the range 0 to k.

- For a value $r \leq b$, we view each key as having $d = \lceil \frac{b}{r} \rceil$ digits of r bits each. Each digit is an integer in the range 0 to $2^r - 1$. So, we can use counting sort with $k = 2^r - 1$. For example, we can view a 32-bit word as having four 8-bit digits, so that $b = 32, r = 8, k = 255$ and $d = \frac{b}{r} = 4$. Each pass of counting sort takes time $\Theta(n + k) = \Theta(n + 2^r)$. There are d passes, for a total running time of $\Theta(d(n + 2^r)) = \Theta(\frac{b}{r}(n + 2^r))$.

Choosing r to Minimize Running Time

- For given values of n and b, we wish to choose the value of r, with $r \leq b$, that minimizes the expression $\frac{b}{r}(n + 2^r)$.
  - If $b < \lfloor \log n \rfloor$, then for any value of $r \leq b$, we have that $n + 2^r = \Theta(n)$. Thus, choosing $r = b$ yields a running time of $\frac{b}{b}(n + 2^b) = \Theta(n)$, which is asymptotically optimal.
  - If $b \geq \lfloor \log n \rfloor$, then choosing $r = \lfloor \log n \rfloor$ gives the best time to within a constant factor:
    - Choosing $r = \lfloor \log n \rfloor$ yields a running time of $\Theta(\frac{bn}{\log n})$.
    - As we increase r above $\lfloor \log n \rfloor$, the 2^r term in the numerator increases faster than the r term in the denominator. So increasing r above $\lfloor \log n \rfloor$ yields a running time of $\Omega(\frac{bn}{\log n})$.
    - If instead we were to decrease r below $\lfloor \log n \rfloor$, then the $\frac{b}{r}$ term increases and the $n + 2^r$ term remains at $\Theta(n)$.

Subsection 4

Bucket Sort

Idea Behind Bucket Sort

- Bucket sort assumes that the input is drawn from a uniform distribution and has an average-case running time of $O(n)$.
- Like counting sort, bucket sort is fast because it assumes something about the input:
  - Counting sort assumes that the input consists of integers in a small range;
  - Bucket sort assumes that the input is generated by a random process that distributes elements uniformly and independently over the interval $[0, 1)$.
- Bucket sort does the following:
  - It divides the interval $[0, 1)$ into n equal-sized subintervals, or buckets, and then distributes the n input numbers into the buckets. Since the inputs are uniformly and independently distributed over $[0, 1)$, we do not expect many numbers to fall into each bucket.
  - To produce the output, we simply sort the numbers in each bucket and then go through the buckets in order, listing the elements in each.

The Procedure Bucket Sort

- We assume the input is an n-element array A and that each element A[i] in the array satisfies $0 \leq \text{A}[i] < 1$.
- The code requires an auxiliary array $\text{B}[0 \dots n-1]$ of linked lists (buckets) and assumes that there is a mechanism for maintaining such lists.

BucketSort(A)

1. $n = \text{A.length}$
2. Let $\text{B}[0 \dots n-1]$ be a new array
3. for $i = 0$ to $n-1$
4.   make B[i] an empty list
5. for $i = 1$ to n
6.   insert A[i] into list $\text{B}[\lfloor n\text{A}[i] \rfloor]$
7. for $i = 0$ to $n-1$
8.   sort list B[i] with insertion sort
9. concatenate the lists $\text{B}[0], \text{B}[1], \dots, \text{B}[n-1]$ together in order

Correctness of Bucket Sort

- Consider two elements $A[i]$ and $A[j]$.

  Assume, without loss of generality, that $A[i] \leq A[j]$.

  Since $\lfloor nA[i] \rfloor \leq \lfloor nA[j] \rfloor$, element $A[i]$ is placed either into the same bucket as $A[j]$ or into a bucket with a lower index.

  - If $A[i]$ and $A[j]$ are placed into the same bucket, then the for loop of Lines 7-8 puts them into the proper order.
  - If $A[i]$ and $A[j]$ are placed into different buckets, then Line 9 puts them into the proper order.

  Therefore, bucket sort works correctly.

The Running Time of Bucket Sort

- All lines except Line 8 take $\mathrm{O}(n)$ time in the worst case.
- To analyze the cost of the calls to insertion sort, let n_i be the random variable denoting the number of elements placed in bucket $\mathrm{B}[i]$. Since insertion sort runs in quadratic time, the running time of bucket sort is $T(n) = \Theta(n) + \sum_{i=0}^{n-1} \mathrm{O}\left(n_i^2\right)$. Taking expectations of both sides and using linearity, we have

$$\begin{aligned} E[T(n)] &= E\left[\Theta(n) + \sum_{i=0}^{n-1} \mathrm{O}\left(n_i^2\right)\right] \\ &= \Theta(n) + \sum_{i=0}^{n-1} E\left[\mathrm{O}\left(n_i^2\right)\right] \\ &= \Theta(n) + \sum_{i=0}^{n-1} \mathrm{O}\left(E\left[n_i^2\right]\right). \end{aligned}$$

$E[n_i^2] = 2 - \frac{1}{n}, i = 0, \ldots, n-1$

- We define indicator random variables $X_{ij} = \mathrm{I}\{\mathrm{A}[j] \text{ falls in bucket } i\}$, for $i = 0, 1, \ldots, n-1$ and $j = 0, 2, \ldots, n-1$. Thus, $n_i = \sum_{j=0}^{n-1} X_{ij}$.

  To compute $E[n_i^2]$, we expand the square and regroup:

$$\begin{aligned} E[n_i^2] &= E\left[\left(\sum_{j=0}^{n-1} X_{ij}\right)^2\right] = E\left[\sum_{j=0}^{n-1}\sum_{k=0}^{n-1} X_{ij}X_{ik}\right] \\ &= E\left[\sum_{j=0}^{n-1} X_{ij}^2 + \sum_{0\le j\le n-1}\sum_{\substack{0\le k\le n-1\\ k\ne j}} X_{ij}X_{ik}\right] \\ &= \sum_{j=0}^{n-1} E[X_{ij}^2] + \sum_{0\le j\le n-1}\sum_{\substack{0\le k\le n-1\\ k\ne j}} E[X_{ij}X_{ik}], \end{aligned}$$

  where the last line follows by linearity of expectation.

  - Indicator random variable X_{ij} is 1 with probability $\frac{1}{n}$ and 0 otherwise. Thus, $E[X_{ij}^2] = 1^2 \cdot \frac{1}{n} + 0^2 \cdot (1 - \frac{1}{n}) = \frac{1}{n}$.
  - When $k \neq j$, the variables X_{ij} and X_{ik} are independent. Thus, $E[X_{ij}X_{ik}] = E[X_{ij}]E[X_{ik}] = \frac{1}{n} \cdot \frac{1}{n} = \frac{1}{n^2}$.

Completing the Proof

- We get overall

$$\begin{aligned} E[n_i^2] &= \sum_{j=0}^{n-1} E[X_{ij}^2] + \sum_{0 \le j \le n-1} \sum_{\substack{0 \le k \le n-1 \\ k \ne j}} E[X_{ij} X_{ik}] \\ &= \sum_{j=0}^{n-1} \frac{1}{n} + \sum_{0 \le j \le n-1} \sum_{\substack{0 \le k \le n-1 \\ k \ne j}} \frac{1}{n^2} \\ &= n \cdot \frac{1}{n} + n(n-1) \cdot \frac{1}{n^2} = 1 + \frac{n-1}{n} = 2 - \frac{1}{n}. \end{aligned}$$

- Using this expected value, we conclude that the expected time for bucket sort is

$$\Theta(n) + n \cdot O\left(2 - \frac{1}{n}\right) = \Theta(n).$$

- Thus, the entire bucket sort algorithm runs in linear expected time.