Sorting in Linear Time

CSE 5311: Design and Analysis of Algorithms

Alex Dillhoff

The University of Texas at Arlington

Introduction

Establishing a Lower Bound on Comparison Sorts

Counting Sort

Radix Sort

Bucket Sort

Introduction

Introduction

All sorting algorithms discussed up to this point are **comparison based**.

It may be intuitive to think that sorting cannot be done without a comparison.

If you have no way to evaluate the relative ordering of two different objects, how can you possibly arrange them in any order?

It turns out that comparison based sorts cannot possibly reach linear time.

Any comparison sort **must** make $\boxed{\Omega(n \lg n)}$ comparisons in the worst case.

Establishing a Lower Bound on Comparison Sorts

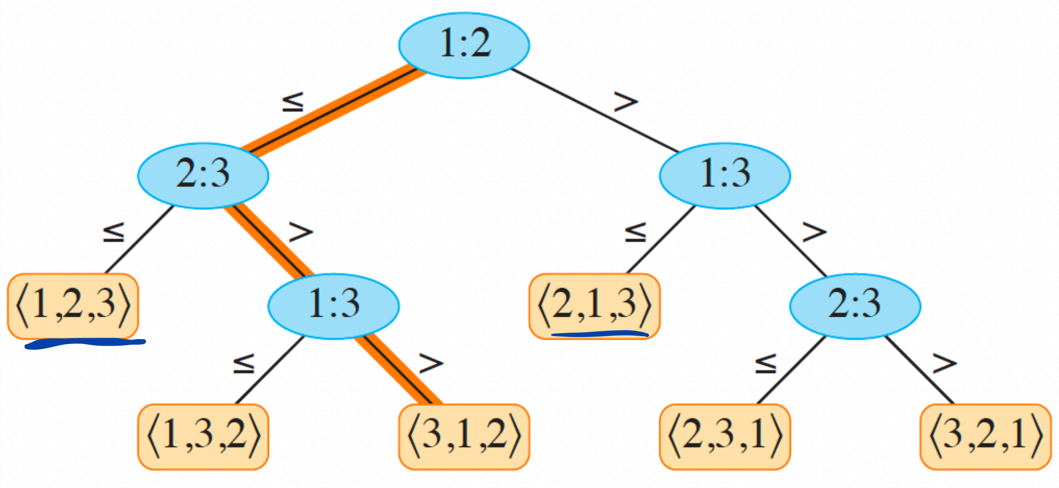
Establishing a Lower Bound

The basis of the proof is to consider that all comparison sorts can be viewed as a decision tree.

Each leaf represents a unique permutation of the input array.

If there are n elements in the input array, there are $n!$ possible permutations.

Establishing a Lower Bound



A decision tree for a comparison sort on a 3-element array.

Establishing a Lower Bound

Interpreting the tree

- Each node compares two values as $a : b$.
- If $a \leq b$, the left path is taken.
- The worst case of a comparison sort can be determined by the height of the tree.

Establishing a Lower Bound

Consider a binary tree of height h with l reachable leaves.

- Each of the $n!$ permutations occurs as one of the leaves, so $n! \leq l$ since there may be duplicate permutations in the leaves.

Establishing a Lower Bound

Consider a binary tree of height h with l reachable leaves.

- Each of the $n!$ permutations occurs as one of the leaves, so $n! \leq l$ since there may be duplicate permutations in the leaves.
- A binary tree with height h has no more than 2^h leaves, so $n! \leq l \leq 2^h$.

Establishing a Lower Bound

Consider a binary tree of height h with l reachable leaves.

- Each of the $n!$ permutations occurs as one of the leaves, so $n! \leq l$ since there may be duplicate permutations in the leaves.

  lg lg
- A binary tree with height h has no more than 2^h leaves, so $n! \leq l \leq 2^h$.
- Taking the logarithm of this inequality implies that $h \geq \lg n!$.

Establishing a Lower Bound

Consider a binary tree of height h with l reachable leaves.

- Each of the $n!$ permutations occurs as one of the leaves, so $n! \leq l$ since there may be duplicate permutations in the leaves.
- A binary tree with height h has no more than 2^h leaves, so $n! \leq l \leq 2^h$.
- Taking the logarithm of this inequality implies that $h \geq \lg n!$.
- Since $\lg n! = \underline{\Theta(n \lg n)}$, and is a lower bound on the height of the tree, then any comparison sort must make $\Omega(n \lg n)$ comparisons.

Counting Sort

Counting Sort

100 2^{20}

Counting sort can sort an array of integers in $O(n + k)$ time, where $k \geq 0$ is the largest integer in the set.

It works by counting the number of elements less than or equal to each element x.

0-50

Counting Sort

```
def counting_sort(A, k):
    n = len(A)
    B = [0 for i in range(n)]
    C = [0 for i in range(k+1)]

    for i in range(n):
        C[A[i]] += 1

    for i in range(1, k):
        C[i] = C[i] + C[i-1]

    for i in range(n - 1, -1, -1):
        B[C[A[i]]-1] = A[i]
        C[A[i]] = C[A[i]] - 1

    return B
```

$2_1, 5, 3_1, 0_1, 2_2, 3_2, 0_2, 3_3$

$B = [0_1, 0_2, 2_1, 2_2, 3_1, 3_2, 3_3, 5]$

$C = [1, 2, 4, 6, 7, 8]$

$i = 5$

$A[5] = 3_2$

$C[3] = 6$

$B[5]$

Counting Sort

Walkthrough

- The first two loops establish the number of elements less than or equal to i for each element i.
- The main sticking point in understanding this algorithm is the last loop.
- It starts at the very end of loop, placing the last element from A into the output array B in its correct position as determined by C.

Example

$A = \{2, 5, 5, 3, 4\}$.

- After the second loop, $C = \{0, 0, 1, 2, 3, 5\}$.

Counting Sort

Example

$A = \{2, 5, 5, 3, 4\}$.

- After the second loop, $C = \{0, 0, 1, 2, 3, 5\}$.
- On the first iteration of the last loop, $A[4] = 4$ is used as the index into C, which yields 3 since the value 4 is greater than or equal to 3 elements in the original array.
- It is then placed in the correct spot $B[3 - 1] = 4$.

Counting Sort

Class Exercise: Sort the array $A = \{2, 5, 3, 0, 2, 3, 0, 3\}$ using counting sort.

Do similar values maintain their relative order?

yes => stable

Radix Sort

Radix Sort

Dating back to 1887 by Herman Hollerith’s work on tabulating machines

- Places numbers in one of k bins based on their **radix**, or the number of unique digits.
- It was used for sorting punch cards via multi-column sorting.
- It works by iteratively sorting a series of inputs based on a column starting with the least-significant digit.

Radix Sort

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

An example of radix sort sorting a list of integers.

$d\Theta(n+k)$

Radix Sort

$d \approx n$ n^2

```
def radix_sort(A, d):
    for i in range(d):
        A = counting_sort(A, len(A), 9)
    return A
```

Analysis

- Counting sort is $\Theta(n + k)$.

Analysis

- Counting sort is $\Theta(n + k)$.
- Radix sort calls it d times.

Radix Sort

Analysis

- Counting sort is $\Theta(n+k)$.
- Radix sort calls it d times.
- Therefore, the time complexity of radix sort is $\Theta(d(n+k))$.

Radix Sort

Analysis

- Counting sort is $\Theta(n + k)$.
- Radix sort calls it d times.
- Therefore, the time complexity of radix sort is $\Theta(d(\overset{n+n}{n + k}))$.
- If $k = O(n)$ then the time complexity is $\Theta(dn)$.

Complex Keys

.27 → .27
.36 → .34
.34 → .36

What if the data is not just a single integer, but a complex key or series of keys?

What if the data is not just a single integer, but a complex key or series of keys?

The keys themselves can be broken up into digits.

Complex Keys

2^{32}

$\Theta(d(n+k))$

$k \leq n$

Consider a 32-bit word.

- To sort n of these words with $b = 32$ bits per word, break the words into $r = 8$ bit digits.
- This yields $d = \lceil b/r \rceil = 4$ digits.
- The largest value for each digit is then $k = 2^r - 1 = \boxed{255.}$
- Plugging these values into the analysis from above yields $\Theta((b/r)(n + 2^r))$.

Complex Keys

What is the best choice of r?

Complex Keys

What is the best choice of r?

- As r increases, 2^r increases.
- As it decreases, $\frac{b}{r}$ increases.
- The best choice depends on whether $b < \lfloor \lg n \rfloor$.

Complex Keys

$b=32$ $r=8$

If $b < \lfloor \lg n \rfloor$, then $r \leq b$ implies $(n + 2^r) = \Theta(n)$ since $2^{\lg n} = n$.

If $b \geq \lfloor \lg n \rfloor$, then we should choose $r \approx \lg n$.

This would yield $\Theta((b/\lg n)(n + n)) = \Theta(bn/\lg n)$

Another perspective…

Choosing $r < \lg n$ implies $\frac{b}{r} > \frac{b}{\lg n}$; the $n + 2^r$ term doesn't increase.

Choosing $r > \lg n$ implies an increase in $n + 2^r$.

Complex Keys

If we are given 2^{16} 32-bit words, we should use $r = \lg 2^{16} = 16$ bits.

This would result in $\lceil \frac{32}{16} \rceil = 2$ passes.

Comparison to Quicksort and Merge sort

Consider an input of 1 million (2^{20}) 32-bit integers.

- **Radix sort:** 2 passes, each with $O(n)$ time.
- **Quicksort:**

Comparison to Quicksort and Merge sort

Consider an input of 1 million (2^{20}) 32-bit integers.

- **Radix sort:** 2 passes, each with $O(n)$ time.
- **Quicksort:** $\lg n = 20$ passes, each with $O(n)$ time.
- **Merge sort:**

Comparison to Quicksort and Merge sort

Consider an input of 1 million (2^{20}) 32-bit integers.

- **Radix sort:** 2 passes, each with $O(n)$ time.
- **Quicksort:** $\lg n = 20$ passes, each with $O(n)$ time.
- **Merge sort:** $\lg n = 20$ passes, each with $O(n)$ time.

Example: Sorting Names

98 = B

Use radix sort to sort the following list of names: ”Beethoven”, ”Bach”, ”Mozart”, ”Chopin”, ”Liszt”, ”Schubert”, ”Haydn”, ”Brahms”, ”Wagner”, ”Tchaikovsky”.

Example: Sorting Names

First, we need to figure out how to encode the names as integers.

- If we convert the input to lowercase, we only have to deal with $k = \underline{26}$ unique characters.

  $2^5 = 32$

- This only requires 5 bits.
- Since each name has varying length, we can use a sentinel value of 0 to pad the shorter names.
- That is, 0 represents a padding character and the alphabet starts at 1.

Example: Sorting Names

Original Name	Encoded Name
Beethoven	[2, 5, 5, 20, 8, 15, 22, 5, 14, 0, 0]
Bach	[2, 1, 3, 8, 0, 0, 0, 0, 0, 0, 0]
Mozart	[13, 15, 26, 1, 18, 20, 0, 0, 0, 0, 0]
Chopin	[3, 8, 15, 16, 9, 14, 0, 0, 0, 0, 0]
Liszt	[12, 9, 19, 26, 20, 0, 0, 0, 0, 0, 0]
Schubert	[19, 3, 8, 21, 2, 5, 18, 20, 0, 0, 0]
Haydn	[8, 1, 25, 4, 14, 0, 0, 0, 0, 0, 0]
Brahms	[2, 18, 1, 8, 13, 19, 0, 0, 0, 0, 0]
Wagner	[23, 1, 7, 14, 5, 18, 0, 0, 0, 0, 0]
Tchaikovsky	[20, 3, 8, 1, 9, 11, 15, 22, 19, 11, 25]

Example: Sorting Names

No changes are made in the first 2 iterations. Iteration 3 yields:

Original Name	Encoded Name
Bach	[2, 1, 3, 8, 0, 0, 0, 0, 0, 0, 0]
Mozart	[13, 15, 26, 1, 18, 20, 0, 0, 0, 0, 0]
Chopin	[3, 8, 15, 16, 9, 14, 0, 0, 0, 0, 0]
Liszt	[12, 9, 19, 26, 20, 0, 0, 0, 0, 0, 0]
Schubert	[19, 3, 8, 21, 2, 5, 18, 20, 0, 0, 0]
Haydn	[8, 1, 25, 4, 14, 0, 0, 0, 0, 0, 0]
Brahms	[2, 18, 1, 8, 13, 19, 0, 0, 0, 0, 0]
Wagner	[23, 1, 7, 14, 5, 18, 0, 0, 0, 0, 0]
Beethoven	[2, 5, 5, 20, 8, 15, 22, 5, 14, 0, 0]
Tchaikovsky	[20, 3, 8, 1, 9, 11, 15, 22, 19, 11, 25]

Example: Sorting Names

Iteration 4

Original Name	Encoded Name
Bach	[2, 1, 3, 8, 0, 0, 0, 0, 0, 0, 0]
Mozart	[13, 15, 26, 1, 18, 20, 0, 0, 0, 0, 0]
Chopin	[3, 8, 15, 16, 9, 14, 0, 0, 0, 0, 0]
Liszt	[12, 9, 19, 26, 20, 0, 0, 0, 0, 0, 0]
Haydn	[8, 1, 25, 4, 14, 0, 0, 0, 0, 0, 0]
Brahms	[2, 18, 1, 8, 13, 19, 0, 0, 0, 0, 0]
Wagner	[23, 1, 7, 14, 5, 18, 0, 0, 0, 0, 0]
Schubert	[19, 3, 8, 21, 2, 5, 18, 20, 0, 0, 0]
Beethoven	[2, 5, 5, 20, 8, 15, 22, 5, 14, 0, 0]
Tchaikovsky	[20, 3, 8, 1, 9, 11, 15, 22, 19, 11, 25]

Example: Sorting Names

Iteration 5

Original Name	Encoded Name
Bach	[2, 1, 3, 8, 0, 0, 0, 0, 0, 0, 0]
Liszt	[12, 9, 19, 26, 20, 0, 0, 0, 0, 0, 0]
Haydn	[8, 1, 25, 4, 14, 0, 0, 0, 0, 0, 0]
Schubert	[19, 3, 8, 21, 2, 5, 18, 20, 0, 0, 0]
Tchaikovsky	[20, 3, 8, 1, 9, 11, 15, 22, 19, 11, 25]
Chopin	[3, 8, 15, 16, 9, 14, 0, 0, 0, 0, 0]
Beethoven	[2, 5, 5, 20, 8, 15, 22, 5, 14, 0, 0]
Wagner	[23, 1, 7, 14, 5, 18, 0, 0, 0, 0, 0]
Brahms	[2, 18, 1, 8, 13, 19, 0, 0, 0, 0, 0]
Mozart	[13, 15, 26, 1, 18, 20, 0, 0, 0, 0, 0]

Example: Sorting Names

Iteration 6

Original Name	Encoded Name
Bach	[2, 1, 3, 8, 0, 0, 0, 0, 0, 0, 0]
Schubert	[19, 3, 8, 21, 2, 5, 18, 20, 0, 0, 0]
Wagner	[23, 1, 7, 14, 5, 18, 0, 0, 0, 0, 0]
Beethoven	[2, 5, 5, 20, 8, 15, 22, 5, 14, 0, 0]
Tchaikovsky	[20, 3, 8, 1, 9, 11, 15, 22, 19, 11, 25]
Chopin	[3, 8, 15, 16, 9, 14, 0, 0, 0, 0, 0]
Brahms	[2, 18, 1, 8, 13, 19, 0, 0, 0, 0, 0]
Haydn	[8, 1, 25, 4, 14, 0, 0, 0, 0, 0, 0]
Mozart	[13, 15, 26, 1, 18, 20, 0, 0, 0, 0, 0]
Liszt	[12, 9, 19, 26, 20, 0, 0, 0, 0, 0, 0]

Example: Sorting Names

Iteration 7

Original Name	Encoded Name
Tchaikovsky	[20, 3, 8, 1, 9, 11, 15, 22, 19, 11, 25]
Mozart	[13, 15, 26, 1, 18, 20, 0, 0, 0, 0, 0]
Haydn	[8, 1, 25, 4, 14, 0, 0, 0, 0, 0, 0]
Bach	[2, 1, 3, 8, 0, 0, 0, 0, 0, 0, 0]
Brahms	[2, 18, 1, 8, 13, 19, 0, 0, 0, 0, 0]
Wagner	[23, 1, 7, 14, 5, 18, 0, 0, 0, 0, 0]
Chopin	[3, 8, 15, 16, 9, 14, 0, 0, 0, 0, 0]
Beethoven	[2, 5, 5, 20, 8, 15, 22, 5, 14, 0, 0]
Schubert	[19, 3, 8, 21, 2, 5, 18, 20, 0, 0, 0]
Liszt	[12, 9, 19, 26, 20, 0, 0, 0, 0, 0, 0]

Example: Sorting Names

Iteration 8

Original Name	Encoded Name
Brahms	[2, 18, 1, 8, 13, 19, 0, 0, 0, 0, 0]
Bach	[2, 1, 3, 8, 0, 0, 0, 0, 0, 0, 0]
Beethoven	[2, 5, 5, 20, 8, 15, 22, 5, 14, 0, 0]
Wagner	[23, 1, 7, 14, 5, 18, 0, 0, 0, 0, 0]
Tchaikovsky	[20, 3, 8, 1, 9, 11, 15, 22, 19, 11, 25]
Schubert	[19, 3, 8, 21, 2, 5, 18, 20, 0, 0, 0]
Chopin	[3, 8, 15, 16, 9, 14, 0, 0, 0, 0, 0]
Liszt	[12, 9, 19, 26, 20, 0, 0, 0, 0, 0, 0]
Haydn	[8, 1, 25, 4, 14, 0, 0, 0, 0, 0, 0]
Mozart	[13, 15, 26, 1, 18, 20, 0, 0, 0, 0, 0]

Example: Sorting Names

Iteration 9

Original Name	Encoded Name
Bach	[2, 1, 3, 8, 0, 0, 0, 0, 0, 0, 0]
Haydn	[8, 1, 25, 4, 14, 0, 0, 0, 0, 0, 0]
Wagner	[23, 1, 7, 14, 5, 18, 0, 0, 0, 0, 0]
Tchaikovsky	[20, 3, 8, 1, 9, 11, 15, 22, 19, 11, 25]
Schubert	[19, 3, 8, 21, 2, 5, 18, 20, 0, 0, 0]
Beethoven	[2, 5, 5, 20, 8, 15, 22, 5, 14, 0, 0]
Chopin	[3, 8, 15, 16, 9, 14, 0, 0, 0, 0, 0]
Liszt	[12, 9, 19, 26, 20, 0, 0, 0, 0, 0, 0]
Mozart	[13, 15, 26, 1, 18, 20, 0, 0, 0, 0, 0]
Brahms	[2, 18, 1, 8, 13, 19, 0, 0, 0, 0, 0]

Example: Sorting Names

Iteration 10

Original Name	Encoded Name
Bach	[2, 1, 3, 8, 0, 0, 0, 0, 0, 0, 0]
Beethoven	[2, 5, 5, 20, 8, 15, 22, 5, 14, 0, 0]
Brahms	[2, 18, 1, 8, 13, 19, 0, 0, 0, 0, 0]
Chopin	[3, 8, 15, 16, 9, 14, 0, 0, 0, 0, 0]
Haydn	[8, 1, 25, 4, 14, 0, 0, 0, 0, 0, 0]
Liszt	[12, 9, 19, 26, 20, 0, 0, 0, 0, 0, 0]
Mozart	[13, 15, 26, 1, 18, 20, 0, 0, 0, 0, 0]
Schubert	[19, 3, 8, 21, 2, 5, 18, 20, 0, 0, 0]
Tchaikovsky	[20, 3, 8, 1, 9, 11, 15, 22, 19, 11, 25]
Wagner	[23, 1, 7, 14, 5, 18, 0, 0, 0, 0, 0]

$\Theta(11(10+26))$

$n = 2^{20}$

$k = 26$

Bucket Sort

Bucket Sort

As the name suggests, bucket sort distributes the input into a number of distinct buckets based on the input value.

- The key here is the assumption that the data is uniformly distributed.
- If the data were not uniformly distributed, then more elements would be concentrated.
- The uniformity ensures that a relatively equal number of data points are placed in each bucket.
- This is also a convenient assumption to have for a parallelized implementation.

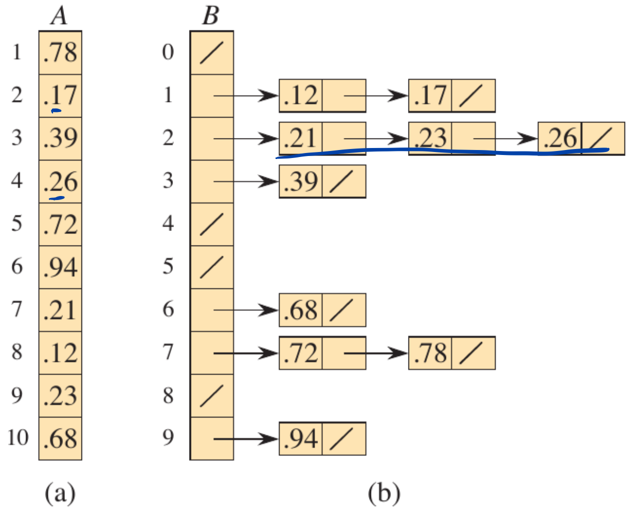
Bucket Sort

- Bucket sort places values into a bucket based on their most significant digits.
- Once the values are assigned, then a simple sort such as insertion sort is used to sort the values within each bucket.
- Once sorted, the buckets are concatenated together to produce the final output.

Under the assumption of uniformity, each bucket will contain no more than $1/n$ of the total elements.

This implies that each call to `insertion_sort` will take $O(1)$ time.

Bucket Sort



An example of bucket sort sorting a list of floats.

Bucket Sort

[.17, .26, .21, .12, .23]

```python
def bucket_sort(A):
    n = len(A)
    B = [[] for i in range(n)]
    for i in range(n):
        B[int(n * A[i])].append(A[i])
    for i in range(n):
        insertion_sort(B[i])
    return B
```

B = 0 [.17, .12
1 .26, .21, .23
2
3
4]

Initializing the array and placing each item into a bucket takes $\Theta(n)$ time.

The call to each insertion sort is $O(n^2)$.

The recurrence is given as

n_i^2 - # of elements in bucket i.

$$T(n) = \Theta(n) + \sum_{i=0}^{n-1} O(n_i^2).$$

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

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

Analysis

The key is to determine the expected value $E[n_i^2]$.

We will frame the problem as a binomial distribution, where a success occurs when an element goes into bucket i.

n is also # of buckets

- p is the probability of success: $p = \frac{1}{n}$
- q is the probability of failure: $q = 1 - \frac{1}{n}$.

Analysis

Under a binomial distribution, we have that $E[n_i] = np = n(1/n) = 1$ and $\text{Var}[n_i] = npq = 1 - 1/n$, where $p = 1/n$ and $q = 1 - 1/n$.

The expected value is then

$$E[n_i^2] = \text{Var}[n_i] + E[n_i]^2 = 1 - 1/n + 1 = 2 - 1/n.$$

This gives way to the fact that

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