CSE 100: Algorithm Design and Analysis
Chapter 08: Sorting in Linear Time

Sungjin Im

University of California, Merced

Last Update: 02-24-2023

Thinking is the hardest work there is, which is probably the reason why so few engage in it.

Henry Ford

Outline

- Comparison sort
  - Any comparison sort has a running time $\Omega(n \log n)$
- Non-comparison sort
  - Counting sort
  - Radix sort
  - Bucket sort

Comparison sorts

- Based only on comparisons between the input elements. In other words, the sorted order is completely determined by comparisons between the input elements.
- Eg. Insertion sort, Merge sort, Heap sort, ...

Theorem

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

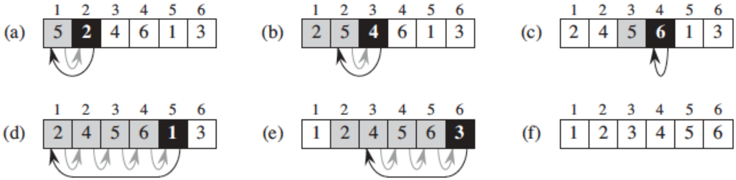
Revisiting Insertion Sort

The Sorting Problem:

Instance: 5, 2, 4, 6, 1, 3.
An idea (Insertion Sort): Let's first sort the first number.
Add the 2nd number to get the sorted order of the first two numbers.
Add the 3rd number to get the sorted order of the first three numbers.
...
Execution:
(5)
(2) 5
2 (4) 5
2 4 5 (6)
(1) 2 4 5 6
1 2 (3) 4 5 6

Revisiting Insertion Sort

Assume that the input is given as an array $A[1 \cdots n]$.



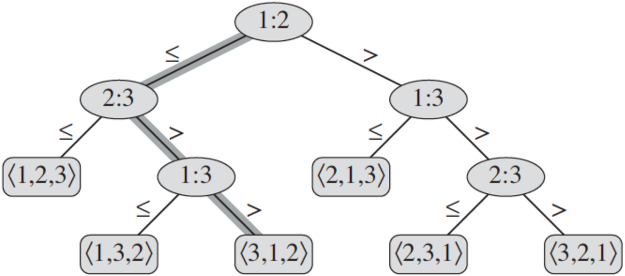
Describing insertion sort via decision tree

Decision tree

- Can represent any specific comparison sort algorithm for all inputs of a given size.
- Only focuses on comparisons abstracting away other details such as control, data movement, and memory management.

Describing insertion sort via decision tree

Decision tree for Insertion sort on three elements



The path in bold shows the decisions made for the inputs with ordering $a_3 \leq a_1 \leq a_2$.

Lower bounds for sorting

Decision tree

- Internal node:

Lower bounds for sorting

Decision tree

- Internal node: comparison
- Leaf node:

Lower bounds for sorting

Decision tree

- Internal node: comparison
- Leaf node: a permutation of elements (which was established by the sorting algorithm)

Lower bounds for sorting

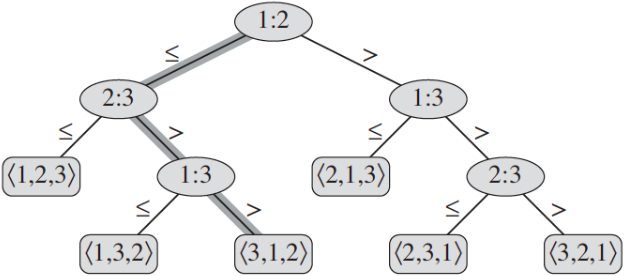
Decision tree

- Internal node: comparison
- Leaf node: a permutation of elements (which was established by the sorting algorithm)
- Each of the $n!$ permutations must appear at least once as one of the leaves

Lower bounds for sorting

Decision tree

- Internal node: comparison
- Leaf node: a permutation of elements (which was established by the sorting algorithm)
- Each of the $n!$ permutations must appear at least once as one of the leaves
- The length of each path = # of comparisons made along the path



Lower bonds for sorting

Theorem

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

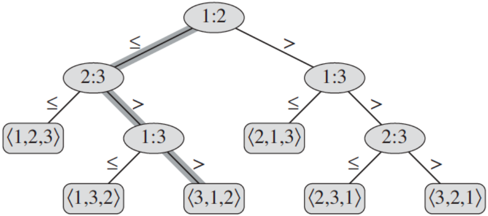
Lower bonds for sorting

Theorem

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

Proof.

h: the decision tree height.
$n! \leq (\#\text{ of leaves})$
$(\#\text{ of leaves}) \leq 2^h$
Therefore, we have $n! \leq 2^h$, which gives $h = \Omega(n \log n)$.
$(2^h \geq (n/2)^{n/2}.)$ □



Lower bonds for sorting

Corollary

Heapsort and Mergesort are asymptotically optimal comparison sorts.

Non-comparison sorts

- Counting sort
- Radix sort
- Bucket sort

Non-comparison sorts

Counting sort

Input: $A[1 \cdots n]$ with an additional parameter k.
Assumption: $0 \leq A[1], \cdots, A[n] \leq k$.

Non-comparison sorts

Counting sort

Input: $A[1 \cdots n]$ with an additional parameter k.
Assumption: $0 \leq A[1], \cdots, A[n] \leq k$.

Output: $B[1 \cdots n]$, sorted.
Temporary working storage: $C[0 \cdots k]$.

Counting sort is **stable**: keys with the same value appear in output in the same order as they did in input.

Non-comparison sorts

Counting sort

```
COUNTING-SORT(A, B, k)
let C[0..k] be a new array
for i = 0 to k
    C[i] = 0
for j = 1 to A.length
    C[A[j]] = C[A[j]] + 1
// C[i] now contains the number of elements equal to i.
for i = 1 to k
    C[i] = C[i] + C[i - 1]
// C[i] now contains the number of elements less than or equal to i.
for j = A.length downto 1
    B[C[A[j]]] = A[j]
    C[A[j]] = C[A[j]] - 1
```

Non-comparison sorts

Counting sort

	1	2	3	4	5	6	7	8
A	2	5	3	0	2	3	0	3

	0	1	2	3	4	5
C	2	0	2	3	0	1

Non-comparison sorts

Counting sort

	1	2	3	4	5	6	7	8
A	2	5	3	0	2	3	0	3

	0	1	2	3	4	5
C	2	0	2	3	0	1

$C[i]$ contains $\#$ of elements $= i$.

	0	1	2	3	4	5
C	2	2	4	7	7	8

Non-comparison sorts

Counting sort

	1	2	3	4	5	6	7	8
A	2	5	3	0	2	3	0	3

	0	1	2	3	4	5
C	2	0	2	3	0	1

$C[i]$ contains # of elements $= i$.

	0	1	2	3	4	5
C	2	2	4	7	7	8

$C[i]$ contains # of elements $\leq i$.

	1	2	3	4	5	6	7	8
B	0	0	2	2	3	3	3	5

Non-comparison sorts

Counting sort

```
COUNTING-SORT(A, B, k)
let C[0..k] be a new array
for i = 0 to k
    C[i] = 0
for j = 1 to A.length
    C[A[j]] = C[A[j]] + 1
// C[i] now contains the number of elements equal to i.
for i = 1 to k
    C[i] = C[i] + C[i - 1]
// C[i] now contains the number of elements less than or equal to i.
for j = A.length downto 1
    B[C[A[j]]] = A[j]
    C[A[j]] = C[A[j]] - 1
```

Non-comparison sorts

Counting sort

```
COUNTING-SORT(A, B, k)
let C[0..k] be a new array
for i = 0 to k
    C[i] = 0
for j = 1 to A.length
    C[A[j]] = C[A[j]] + 1
// C[i] now contains the number of elements equal to i.
for i = 1 to k
    C[i] = C[i] + C[i - 1]
// C[i] now contains the number of elements less than or equal to i.
for j = A.length downto 1
    B[C[A[j]]] = A[j]
    C[A[j]] = C[A[j]] - 1
```

Running time: $\Theta(k + n)$.

Non-comparison sorts

Radix sort

- Useful for sorting keys of a small number of digits.
- Can sort records keyed by multiple fields. e.g. $\langle$ year, month, date $\rangle$.

Note: Radix-sort is not an in-place sorting algorithm.

Non-comparison sorts

Radix sort

How would you sort the following numbers (as a human)?

326
453
608
835
751
435
704
690

Probably sorting based on the most significant bit first, and then on the second most significant bit.

Non-comparison sorts

Radix sort

How would you sort the following numbers (as a human)?

326	326	326
453	453	435
608	435	453
835	608	608
751	690	690
435	751	704
704	704	751
690	835	835

Probably sorting based on the most significant bit first, and then on the second most significant bit.

Non-comparison sorts

Radix sort

How would you sort the following numbers (as a human)?

326	326	326
453	453	435
608	435	453
835	608	608
751	690	690
435	751	704
704	704	751
690	835	835

Probably sorting based on the most significant bit first, and then on the second most significant bit.
It works but you have to be carful with 'boundaries.'

Non-comparison sorts

Radix sort

Radix sort starts *from the least significant bit*!

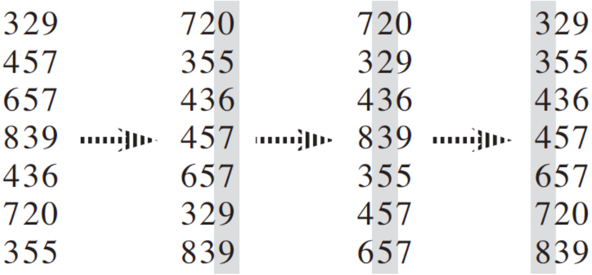
```
RADIX-SORT(A, d)
for i = 1 to d
    use a stable sort to sort array A on digit i
```

Non-comparison sorts

Radix sort

Radix sort starts *from the least significant bit*!

```
RADIX-SORT(A, d)
for i = 1 to d
    use a stable sort to sort array A on digit i
```



Non-comparison sorts

Radix sort

Loop Invariant: At the start of the For loop, keys are sorted in non-decreasing order of their last $i - 1$ digits.

```
RADIX-SORT(A, d)
for i = 1 to d
    use a stable sort to sort array A on digit i
```

Non-comparison sorts

Radix sort

Loop Invariant: At the start of the For loop, keys are sorted in non-decreasing order of their last $i - 1$ digits.

RADIX-SORT(A, d)

for $i = 1$ **to** d
    use a stable sort to sort array A on digit i

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

Non-comparison sorts

Radix sort

What happens if i starts from the most significant digit?

```
RADIX-SORT(A, d)
for i = 1 to d
    use a stable sort to sort array A on digit i
```

Non-comparison sorts

Radix sort

How much time does Radix sort need to sort n d-digit numbers where each digit can take on up to k possible values, say from 0 to $k - 1$?

Non-comparison sorts

Radix sort

How much time does Radix sort need to sort n d-digit numbers where each digit can take on up to k possible values, say from 0 to $k-1$?

$\Theta(d(n+k))$ if the stable sort it uses takes $\Theta(n+k)$ time.

How much time does Radix sort need to sort n d-digit numbers where each digit can take on up to k possible values, say from 0 to $k-1$?
$\Theta(d(n+k))$ if the stable sort it uses takes $\Theta(n+k)$ time.

Question. Sort n $10 \lg n$-digit binary numbers as fast as possible. Assume that a bit operation takes $O(1)$ time.

Non-comparison sorts

Radix sort

How much time does Radix sort need to sort n d-digit numbers where each digit can take on up to k possible values, say from 0 to $k-1$?
$\Theta(d(n+k))$ if the stable sort it uses takes $\Theta(n+k)$ time.

Question. Sort n $10\lg n$-digit binary numbers as fast as possible. Assume that a bit operation takes $O(1)$ time.

- Merge sort or heap sort:

Non-comparison sorts

Radix sort

How much time does Radix sort need to sort n d-digit numbers where each digit can take on up to k possible values, say from 0 to $k-1$?
$\Theta(d(n+k))$ if the stable sort it uses takes $\Theta(n+k)$ time.

Question. Sort n $10\lg n$-digit binary numbers as fast as possible. Assume that a bit operation takes $O(1)$ time.

- Merge sort or heap sort: $O(n \log n)$

Non-comparison sorts

Radix sort

How much time does Radix sort need to sort n d-digit numbers where each digit can take on up to k possible values, say from 0 to $k-1$?
$\Theta(d(n+k))$ if the stable sort it uses takes $\Theta(n+k)$ time.

Question. Sort n $10\lg n$-digit binary numbers as fast as possible. Assume that a bit operation takes $O(1)$ time.

- Merge sort or heap sort: $O(n \log n)$
- Radix sort:

Non-comparison sorts

Radix sort

How much time does Radix sort need to sort n d-digit numbers where each digit can take on up to k possible values, say from 0 to $k-1$?
$\Theta(d(n+k))$ if the stable sort it uses takes $\Theta(n+k)$ time.

Question. Sort n $10\lg n$-digit binary numbers as fast as possible. Assume that a bit operation takes $O(1)$ time.

- Merge sort or heap sort: $O(n\log n)$
- Radix sort: $((10\log n)\cdot(n+2)) = O(n\log n)$.

Non-comparison sorts

Radix sort

How much time does Radix sort need to sort n d-digit numbers where each digit can take on up to k possible values, say from 0 to $k-1$?
$\Theta(d(n+k))$ if the stable sort it uses takes $\Theta(n+k)$ time.

Question. Sort n $10 \lg n$-digit binary numbers as fast as possible. Assume that a bit operation takes $O(1)$ time.

- Merge sort or heap sort: $O(n \log n)$
- Radix sort: $((10 \log n) \cdot (n+2)) = O(n \log n)$.
- Radix sort (sort n 10-digit numbers where each digit can take up to n different values):

Non-comparison sorts

Radix sort

How much time does Radix sort need to sort n d-digit numbers where each digit can take on up to k possible values, say from 0 to $k-1$?
$\Theta(d(n+k))$ if the stable sort it uses takes $\Theta(n+k)$ time.

Question. Sort n $10\lg n$-digit binary numbers as fast as possible. Assume that a bit operation takes $O(1)$ time.

- Merge sort or heap sort: $O(n\log n)$
- Radix sort: $((10\log n)\cdot(n+2)) = O(n\log n)$.
- Radix sort (sort n 10-digit numbers where each digit can take up to n different values): $O(10(n+n)) = O(n)$.

Non-comparison sorts

Radix sort

How much time does Radix sort need to sort n d-digit numbers where each digit can take on up to k possible values, say from 0 to $k-1$?
$\Theta(d(n+k))$ if the stable sort it uses takes $\Theta(n+k)$ time.

Question. Sort n $10\lg n$-digit binary numbers as fast as possible. Assume that a bit operation takes $O(1)$ time.

- Merge sort or heap sort: $O(n \log n)$
- Radix sort: $((10\log n)\cdot(n+2)) = O(n\log n)$.
- Radix sort (sort n 10-digit numbers where each digit can take up to n different values): $O(10(n+n)) = O(n)$.
- Radix sort (sort n 1-digit numbers where each digit can take up to $2^{10\lg n}$ different values):

How much time does Radix sort need to sort n d-digit numbers where each digit can take on up to k possible values, say from 0 to $k-1$?
$\Theta(d(n+k))$ if the stable sort it uses takes $\Theta(n+k)$ time.

Question. Sort n $10\lg n$-digit binary numbers as fast as possible. Assume that a bit operation takes $O(1)$ time.

- Merge sort or heap sort: $O(n \log n)$
- Radix sort: $((10 \log n) \cdot (n+2)) = O(n \log n)$.
- Radix sort (sort n 10-digit numbers where each digit can take up to n different values): $O(10(n+n)) = O(n)$.
- Radix sort (sort n 1-digit numbers where each digit can take up to $2^{10\lg n}$ different values): $O(1 \cdot (n+n^{10})) = O(n^{10})$.

Non-comparison sorts

Radix sort

One can sort n $O(\log n)$-bit numbers in $O(n)$ time assuming that each bit operation takes $O(1)$ time.

Non-comparison sorts

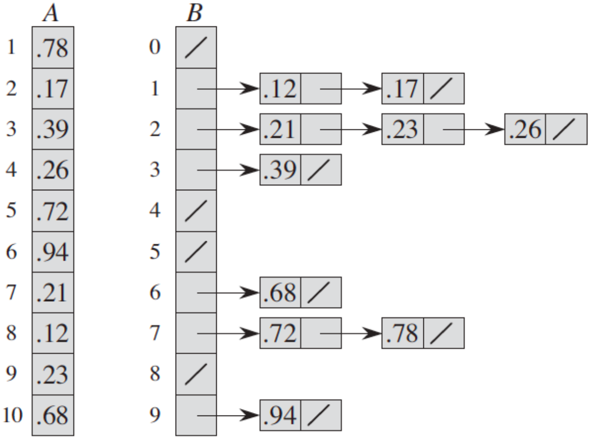
Bucket sort

Input: $A[1 \cdots n]$ where $0 \leq A[i] < 1$ for all i.

- Requires an auxiliary array $B[0 \cdots n-1]$ of linked lists (buckets).
- Average running time is $O(n)$ under the assumption that each key $A[i]$ is sampled from $[0, 1)$ uniformly at random independent of other keys.

Non-comparison sorts

Bucket sort



Non-comparison sorts

Bucket sort

```
BUCKET-SORT(A)
let B[0..n-1] be a new array
n = A.length
for i = 0 to n - 1
    make B[i] an empty list
for i = 1 to n
    insert A[i] into list B[⌊nA[i]⌋]
for i = 0 to n - 1
    sort list B[i] with insertion sort
concatenate the lists B[0], B[1], ..., B[n - 1] together in order
```