CSE 221: Algorithms

Sorting lower bounds and Linear time sorting

Mumit Khan
Fatema Tuz Zohora

Computer Science and Engineering
BRAC University

References

1. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, Second Edition*. The MIT Press, September 2001.
2. Erik Demaine and Charles Leiserson, *6.046J Introduction to Algorithms*. MIT OpenCourseWare, Fall 2005. Available from: `ocw.mit.edu/OcwWeb/Electrical-Engineering-and-Computer-Science/6-046JFall-2005/CourseHome/index.htm`

Last modified: February 25, 2013

Contents

1 Sorting lower bounds
- What's the best we can do?
- Lower bound

2 Sorting in linear time
- Counting sort
- Radix sort
- Conclusion

Contents

1. Sorting lower bounds
   - What's the best we can do?
   - Lower bound
2. Sorting in linear time
   - Counting sort
   - Radix sort
   - Conclusion

What's the best we can do?

- Bubble, selection, insertion, quicksort . . . $O(n^2)$
- Heapsort, mergesort . . . $O(n \lg n)$

What's the best we can do?

- Bubble, selection, insertion, quicksort . . . $O(n^2)$
- Heapsort, mergesort . . . $O(n \lg n)$

What's the best we can do?

- Bubble, selection, insertion, quicksort . . . $O(n^2)$
- Heapsort, mergesort . . . $O(n \lg n)$

What's the best we can do?

- Bubble, selection, insertion, quicksort . . . $O(n^2)$
- Heapsort, mergesort . . . $O(n \lg n)$

Question

Can a sorting algorithm do better than $O(n \lg n)$ in the worst-case?

What's the best we can do?

- Bubble, selection, insertion, quicksort . . . $O(n^2)$
- Heapsort, mergesort . . . $O(n \lg n)$

Question

Can a sorting algorithm do better than $O(n \lg n)$ in the worst-case?

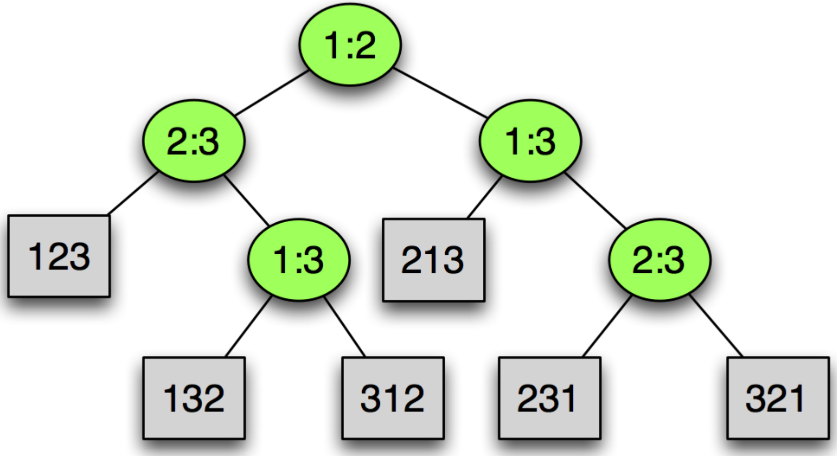
We can use a decision tree to answer this question.

Decision tree for comparison based sorting

- A model to represent any comparison based sorting algorithm
- For any comparison based sorting algorithm, if detailed operations are ignored and only the sequence of comparisons among the pair of elements is focused, then, for a input sequence, it follows a path in this tree from root to leaf to output correct solution
- Each node of this tree is labeled as (i, j) which indicates comparison between element i and element j
- If $i \leq j$ then left subtree is gives subsequent comparisons. Otherwise, right subtree is followed
- Each leaf represents a permutation of the input sequence

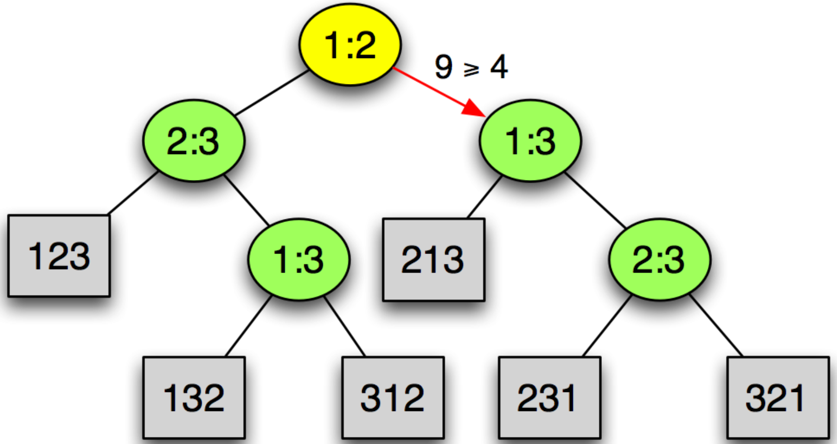
Decision tree for comparison based sorting

Sequence $A = \langle 9, 4, 6 \rangle$



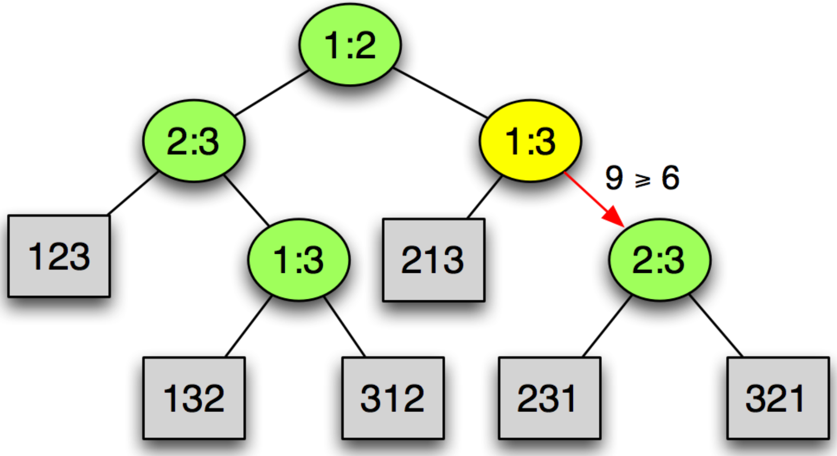
Decision tree for comparison based sorting

Sequence $A = \langle 9, 4, 6 \rangle$



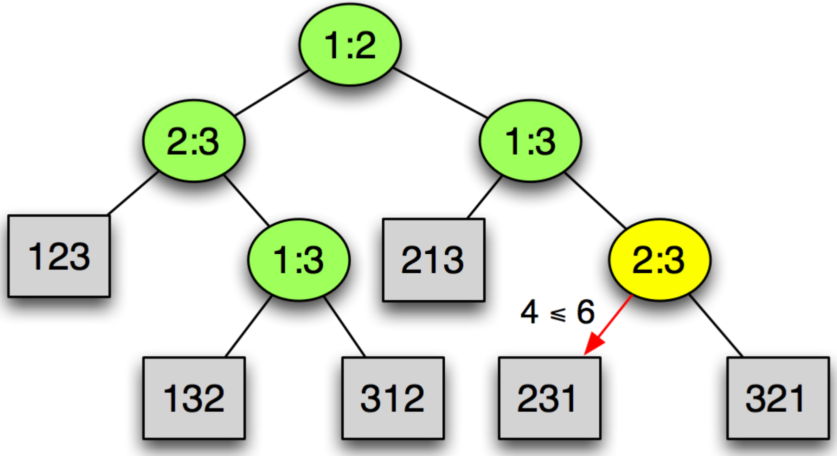
Decision tree for comparison based sorting

Sequence $A = \langle 9, 4, 6 \rangle$



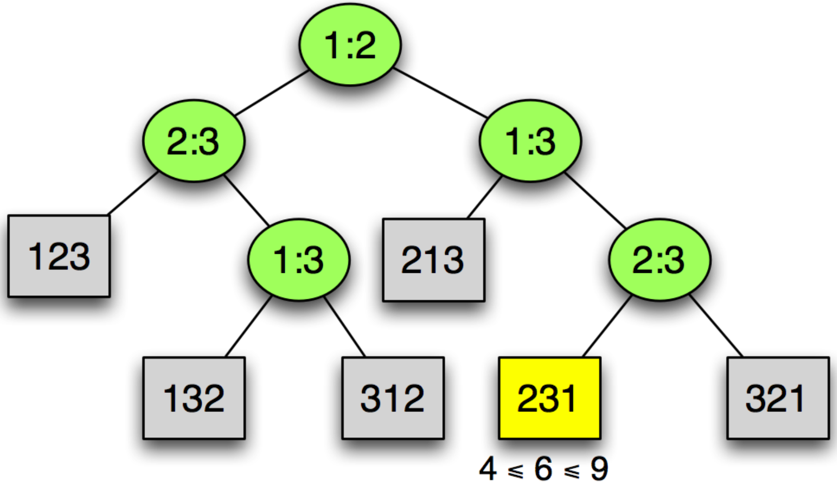
Decision tree for comparison based sorting

Sequence $A = \langle 9, 4, 6 \rangle$



Decision tree for comparison based sorting

Sequence $A = \langle 9, 4, 6 \rangle$ $\Longrightarrow$ Sequence $B = \langle 4, 6, 9 \rangle$



Lower bound on comparison based sorting

Question

What is the best that we can do with comparison-based sorting?

- $n!$ possible permutations,that means $n!$ leaves and one of which is the sorted sequence.
- Minimum number of comparisons is the path from root of the decision tree to one of the $n!$ leaves.
- Each path is not of the same depth
- Path having maximum depth represents worst case
- Path having smallest depth represents best case
- We will prove that worst case cannot be less than $O(n \lg n)$

Lower bound on comparison based sorting

Question

What is the best that we can do with comparison-based sorting?

- $n!$ possible permutations,that means $n!$ leaves and one of which is the sorted sequence.
- Minimum number of comparisons is the path from root of the decision tree to one of the $n!$ leaves.
- Each path is not of the same depth
- Path having maximum depth represents worst case
- Path having smallest depth represents best case
- We will prove that worst case cannot be less than $O(n \lg n)$

Lower bound on comparison based sorting

Question

What is the best that we can do with comparison-based sorting?

- $n!$ possible permutations,that means $n!$ leaves and one of which is the sorted sequence.
- Minimum number of comparisons is the path from root of the decision tree to one of the $n!$ leaves.
- Each path is not of the same depth
- Path having maximum depth represents worst case
- Path having smallest depth represents best case
- We will prove that worst case cannot be less than $O(n \lg n)$

Theorem

The worst-case asymptotic time complexity for any comparison-based sorting algorithm is $\boxed{\Omega(n \lg n)}$.

Theorem

The worst-case asymptotic time complexity for any comparison-based sorting algorithm is $\boxed{\Omega(n \lg n)}$.

- Let, Maximum height is h
- Complete binary tree having height h has 2^h leaves
- But decision tree might not be a complete binary tree so it can have less than 2^h leaves.
- *Number* of leaves $\leq 2^h$

$2^h \geq n!$ (number of leaves$=n!$)
$\lg 2^h \geq \lg n!$
$h \geq \lg n!$
(Note: by Stirling's approximation $n! = (n/e)^n$, where e is Euler's constant.)
$h \geq \lg((n/e)^n)$
$h \geq (n \lg n - n \lg e)$
$h \geq (n \lg n)$
$h = \Omega(n \lg n)$

Contents

1 Sorting lower bounds
- What's the best we can do?
- Lower bound

2 Sorting in linear time
- Counting sort
- Radix sort
- Conclusion

Sorting in linear time: Counting sort

Counting sort: No comparisons between elements.

- **Input**: $A[1\,..\,n]$, where $A[j] \in \{1, 2, \ldots, k\}$.
- **Output**: $B[1\,..\,n]$, sorted.
- **Auxiliary storage**: $C[1\,..\,k]$.

Sorting in linear time: Counting sort

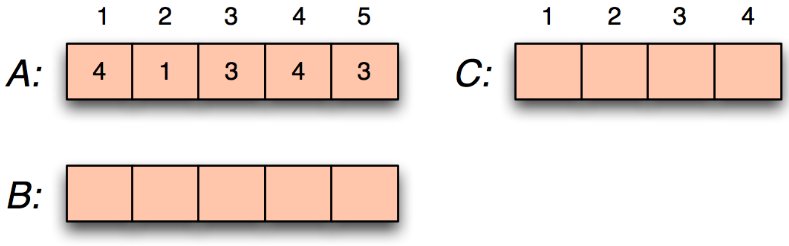
Counting sort: No comparisons between elements.

- **Input**: $A[1\,..\,n]$, where $A[j] \in \{1, 2, \ldots, k\}$.
- **Output**: $B[1\,..\,n]$, sorted.
- **Auxiliary storage**: $C[1\,..\,k]$.

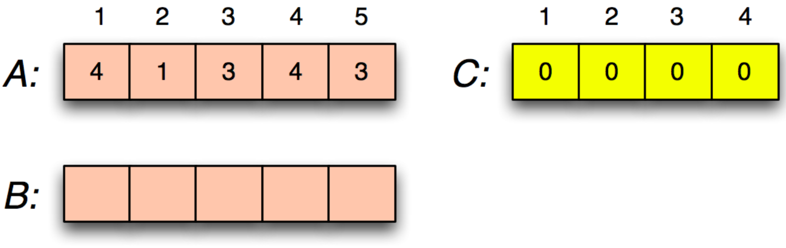
Counting sort algorithm

```
1  for $i \leftarrow 1$ to $k$
2          do $C[i] \leftarrow 0$
3  for $j \leftarrow 1$ to $n$
4          do $C[A[j]] \leftarrow C[A[j]] + 1$            $\triangleright\ C[i] = |\{key = i\}|$
5  for $i \leftarrow 2$ to $k$
6          do $C[i] \leftarrow C[i] + C[i-1]$             $\triangleright\ C[i] = |\{key \leq i\}|$
7  for $j \leftarrow n$ downto $1$
8          do $B[C[A[j]]] \leftarrow A[j]$
9             $C[A[j]] \leftarrow C[A[j]] - 1$
```

Counting sort example

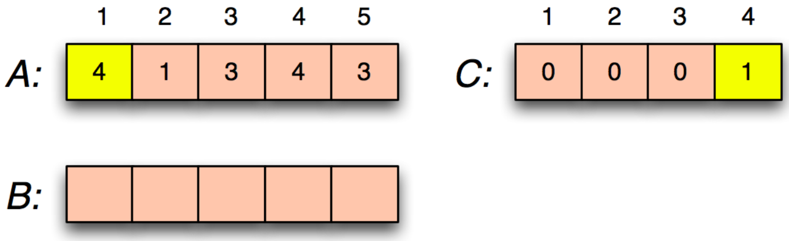


Counting sort example: loop 1



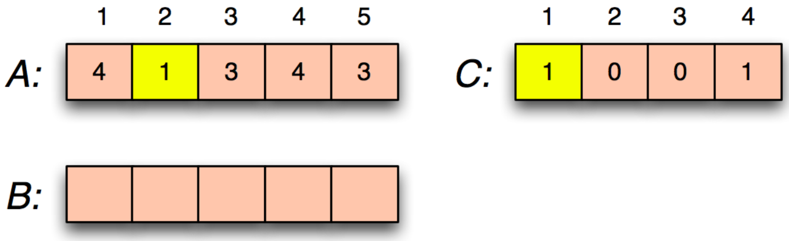
```
for $i \leftarrow 1$ to $k$
        do $C[i] \leftarrow 0$
```

Counting sort example: loop 2



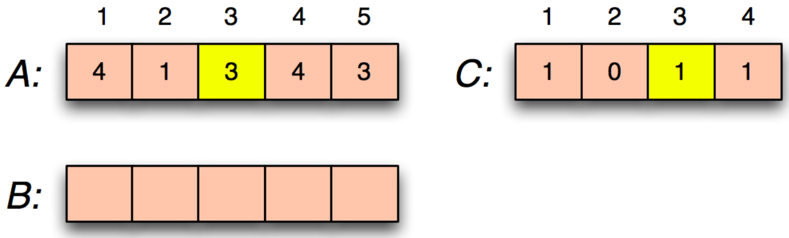
```
for $j \leftarrow 1$ to $n$
        do $C[A[j]] \leftarrow C[A[j]] + 1$        $\triangleright\ C[i] = |\{key = i\}|$
```

Counting sort example: loop 2



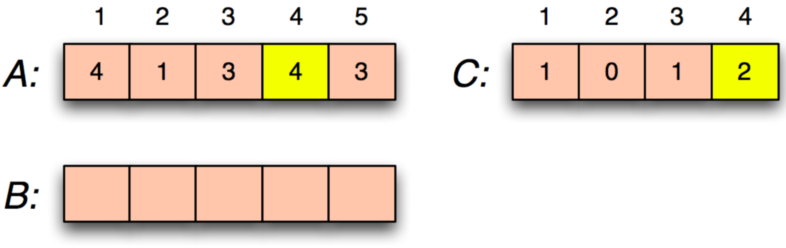
```
for $j \leftarrow 1$ to $n$
    do $C[A[j]] \leftarrow C[A[j]] + 1$        $\triangleright\ C[i] = |\{key = i\}|$
```

Counting sort example: loop 2



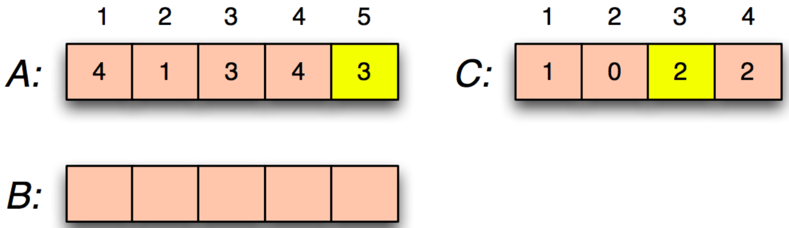
```
for $j \leftarrow 1$ to $n$
        do $C[A[j]] \leftarrow C[A[j]] + 1$        $\triangleright\ C[i] = |\{key = i\}|$
```

Counting sort example: loop 2



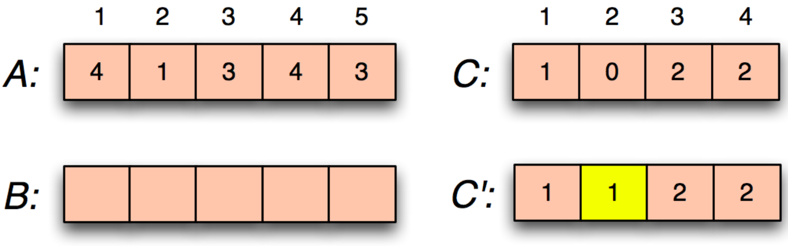
```
for $j \leftarrow 1$ to $n$
    do $C[A[j]] \leftarrow C[A[j]] + 1$        $\triangleright$ $C[i] = |\{key = i\}|$
```

Counting sort example: loop 2



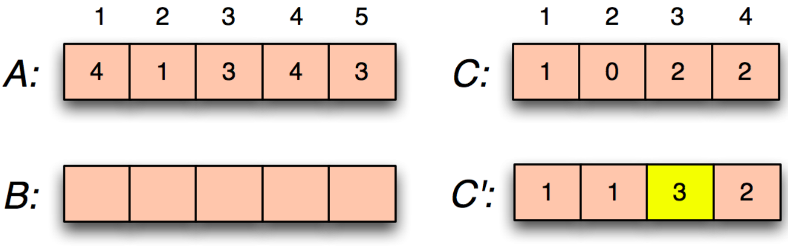
```
for $j \leftarrow 1$ to $n$
    do $C[A[j]] \leftarrow C[A[j]] + 1$          $\triangleright\ C[i] = |\{key = i\}|$
```

Counting sort example: loop 3



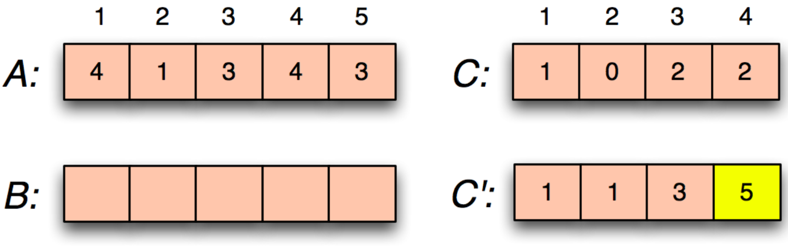
```
for $i \leftarrow 2$ to $k$
        do $C[i] \leftarrow C[i] + C[i-1]$        $\triangleright$ $C[i] = |\{key \leq i\}|$
```

Counting sort example: loop 3



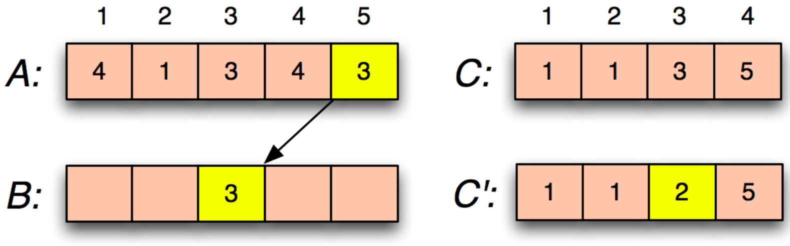
```
for $i \leftarrow 2$ to $k$
    do $C[i] \leftarrow C[i] + C[i-1]$        $\triangleright\ C[i] = |\{key \leq i\}|$
```

Counting sort example: loop 3



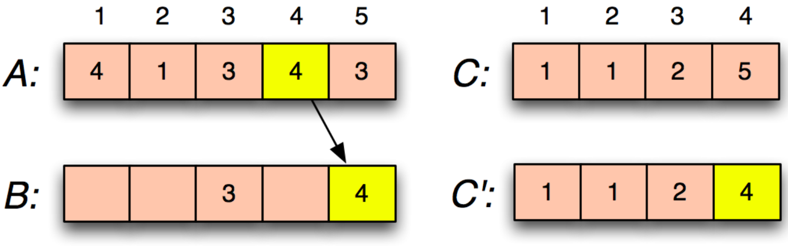
```
for $i \leftarrow 2$ to $k$
    do $C[i] \leftarrow C[i] + C[i-1]$        $\triangleright\ C[i] = |\{key \le i\}|$
```

Counting sort example: loop 4



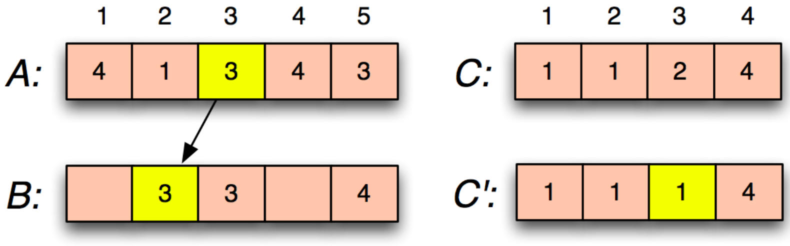
```
for $j \leftarrow n$ downto 1
    do $B[C[A[j]]] \leftarrow A[j]$
       $C[A[j]] \leftarrow C[A[j]] - 1$
```

Counting sort example: loop 4



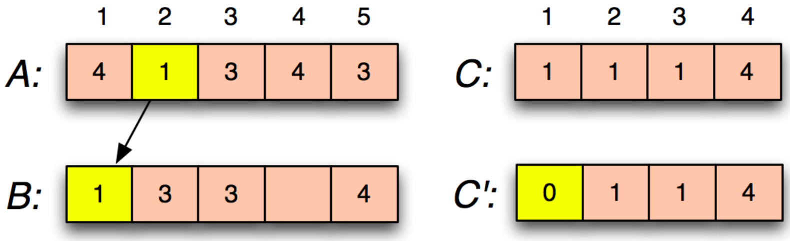
```
for $j \leftarrow n$ downto 1
    do $B[C[A[j]]] \leftarrow A[j]$
       $C[A[j]] \leftarrow C[A[j]] - 1$
```

Counting sort example: loop 4



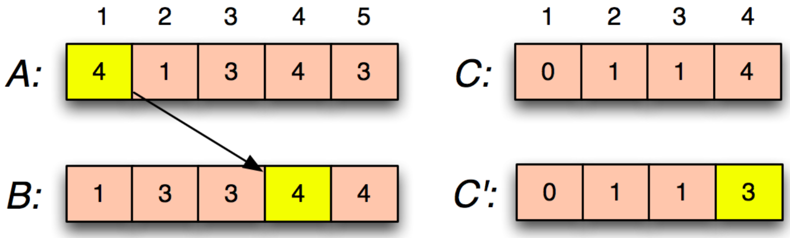
```
for $j \leftarrow n$ downto 1
    do $B[C[A[j]]] \leftarrow A[j]$
       $C[A[j]] \leftarrow C[A[j]] - 1$
```

Counting sort example: loop 4



```
for $j \leftarrow n$ downto 1
    do $B[C[A[j]]] \leftarrow A[j]$
       $C[A[j]] \leftarrow C[A[j]] - 1$
```

Counting sort example: loop 4



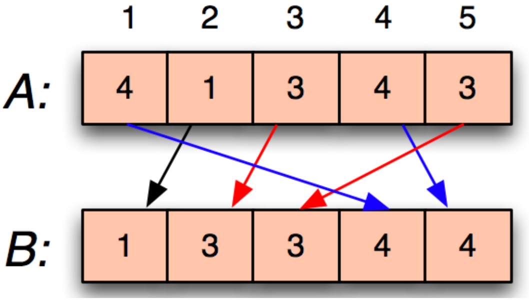
```
for $j \leftarrow n$ downto 1
    do $B[C[A[j]]] \leftarrow A[j]$
       $C[A[j]] \leftarrow C[A[j]] - 1$
```

Sort stability

Counting sort is stable, ie., it preserves the relative order of “equal” elements in the input.

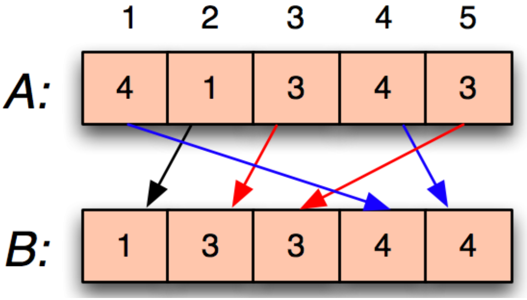
Sort stability

Counting sort is stable, ie., it preserves the relative order of "equal" elements in the input.



Sort stability

Counting sort is stable, ie., it preserves the relative order of "equal" elements in the input.

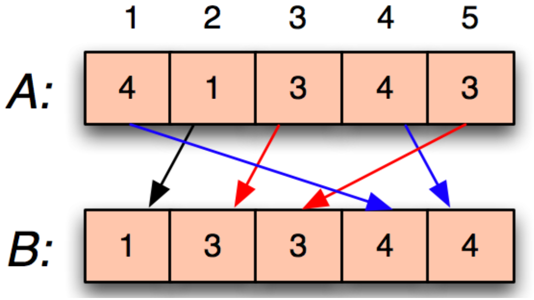


Questions

- Would it still be stable if we had used **for** $j \leftarrow 1$ **to** n instead of **for** $j \leftarrow n$ **downto** 1?

Sort stability

Counting sort is stable, ie., it preserves the relative order of “equal” elements in the input.



Questions

- Would it still be stable if we had used **for** $j \leftarrow 1$ **to** n instead of **for** $j \leftarrow n$ **downto** 1?
- What other sort algorithms that you’ve seen so far are stable?

Counting sort complexity

```
for $i \leftarrow 1$ to $k$
        do $C[i] \leftarrow 0$
for $j \leftarrow 1$ to $n$
        do $C[A[j]] \leftarrow C[A[j]] + 1$
for $i \leftarrow 2$ to $k$
        do $C[i] \leftarrow C[i] + C[i-1]$
for $j \leftarrow n$ downto $1$
        do $B[C[A[j]]] \leftarrow A[j]$
           $C[A[j]] \leftarrow C[A[j]] - 1$
```

Counting sort complexity

$\Theta(k)$ $\left\{ \right.$

```
for $i \leftarrow 1$ to $k$
        do $C[i] \leftarrow 0$
for $j \leftarrow 1$ to $n$
        do $C[A[j]] \leftarrow C[A[j]] + 1$
for $i \leftarrow 2$ to $k$
        do $C[i] \leftarrow C[i] + C[i-1]$
for $j \leftarrow n$ downto $1$
        do $B[C[A[j]]] \leftarrow A[j]$
           $C[A[j]] \leftarrow C[A[j]] - 1$
```

Counting sort complexity

```
$\Theta(k)$  { for $i \leftarrow 1$ to $k$
                  do $C[i] \leftarrow 0$
$\Theta(n)$  { for $j \leftarrow 1$ to $n$
                  do $C[A[j]] \leftarrow C[A[j]] + 1$
               for $i \leftarrow 2$ to $k$
                  do $C[i] \leftarrow C[i] + C[i-1]$
               for $j \leftarrow n$ downto 1
                  do $B[C[A[j]]] \leftarrow A[j]$
                     $C[A[j]] \leftarrow C[A[j]] - 1$
```

Counting sort complexity

```
$\Theta(k)$  {  for $i \leftarrow 1$ to $k$
                    do $C[i] \leftarrow 0$
$\Theta(n)$  {  for $j \leftarrow 1$ to $n$
                    do $C[A[j]] \leftarrow C[A[j]] + 1$
$\Theta(k)$  {  for $i \leftarrow 2$ to $k$
                    do $C[i] \leftarrow C[i] + C[i-1]$
                for $j \leftarrow n$ downto 1
                    do $B[C[A[j]]] \leftarrow A[j]$
                       $C[A[j]] \leftarrow C[A[j]] - 1$
```

Counting sort complexity

```
$\Theta(k)$  { for $i \leftarrow 1$ to $k$
                   do $C[i] \leftarrow 0$
$\Theta(n)$  { for $j \leftarrow 1$ to $n$
                   do $C[A[j]] \leftarrow C[A[j]] + 1$
$\Theta(k)$  { for $i \leftarrow 2$ to $k$
                   do $C[i] \leftarrow C[i] + C[i-1]$
$\Theta(n)$  { for $j \leftarrow n$ downto $1$
                   do $B[C[A[j]]] \leftarrow A[j]$
                      $C[A[j]] \leftarrow C[A[j]] - 1$
```

Counting sort complexity

```
$\Theta(k)$      for $i \leftarrow 1$ to $k$
                     do $C[i] \leftarrow 0$
$\Theta(n)$      for $j \leftarrow 1$ to $n$
                     do $C[A[j]] \leftarrow C[A[j]] + 1$
$\Theta(k)$      for $i \leftarrow 2$ to $k$
                     do $C[i] \leftarrow C[i] + C[i-1]$
$\Theta(n)$      for $j \leftarrow n$ downto 1
                     do $B[C[A[j]]] \leftarrow A[j]$
                        $C[A[j]] \leftarrow C[A[j]] - 1$
```

$\Theta(n + k)$

Running time of Counting sort

The worst-case running time of Counting sort is $O(n + k)$.

Running time of Counting sort

The worst-case running time of Counting sort is $O(n+k)$.

Observations

- If $k = O(n)$, then the worst case running time is $\Theta(n)$.

Running time of Counting sort

The worst-case running time of Counting sort is $O(n+k)$.

Observations

- If $k = O(n)$, then the worst case running time is $\Theta(n)$.
- But didn't we just prove that sorting takes $\Omega(n \lg n)$ time?

Running time of Counting sort

The worst-case running time of Counting sort is $O(n+k)$.

Observations

- If $k = O(n)$, then the worst case running time is $\Theta(n)$.
- But didn't we just prove that sorting takes $\Omega(n \lg n)$ time?
- So what's wrong with this picture?

Running time of Counting sort

The worst-case running time of Counting sort is $O(n+k)$.

Observations

- If $k = O(n)$, then the worst case running time is $\Theta(n)$.
- But didn't we just prove that sorting takes $\Omega(n \lg n)$ time?
- So what's wrong with this picture?

And the answer is ...

- The $\Omega(n \lg n)$ is for comparison sorting.

Running time of Counting sort

The worst-case running time of Counting sort is $O(n + k)$.

Observations

- If $k = O(n)$, then the worst case running time is $\Theta(n)$.
- But didn't we just prove that sorting takes $\Omega(n \lg n)$ time?
- So what's wrong with this picture?

And the answer is ...

- The $\Omega(n \lg n)$ is for comparison sorting.
- Counting sort is **not** a comparison sort.

Running time of Counting sort

The worst-case running time of Counting sort is $O(n + k)$.

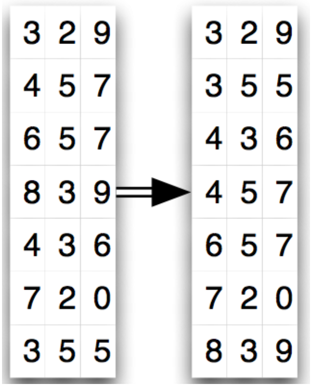
Observations

- If $k = O(n)$, then the worst case running time is $\Theta(n)$.
- But didn't we just prove that sorting takes $\Omega(n \lg n)$ time?
- So what's wrong with this picture?

And the answer is ...

- The $\Omega(n \lg n)$ is for comparison sorting.
- Counting sort is **not** a comparison sort.
- In fact, counting sort does not use a single comparison.

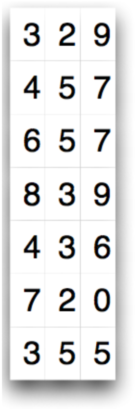
Radix sort



Radix sort basics

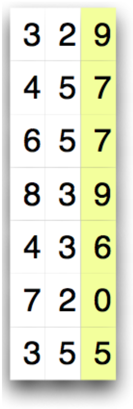
- Digit by digit sort.
- Can be either *most-significant* digit first, or *least-significant* digit first.
- A good way is to stably sort *least-significant* digit first.

Radix sort in action



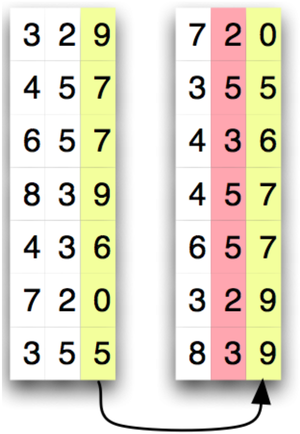
3	2	9
4	5	7
6	5	7
8	3	9
4	3	6
7	2	0
3	5	5

Radix sort in action

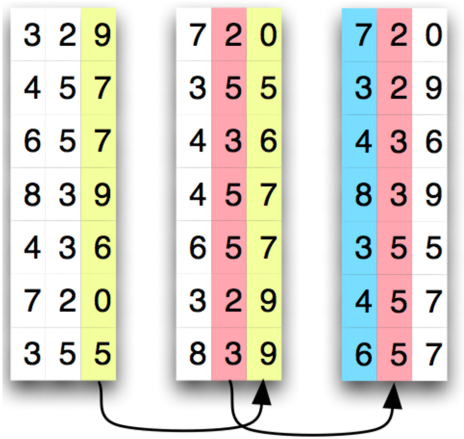


3	2	9
4	5	7
6	5	7
8	3	9
4	3	6
7	2	0
3	5	5

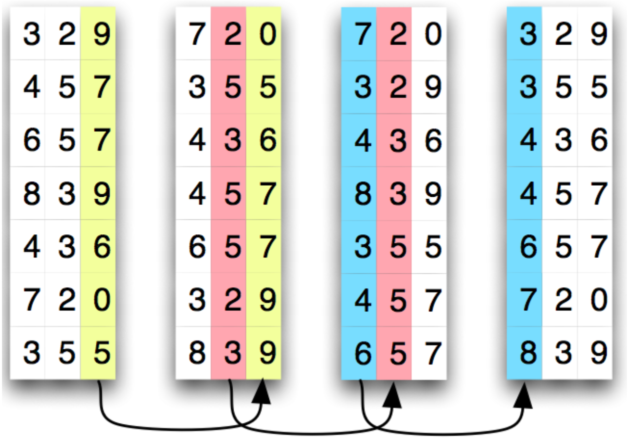
Radix sort in action



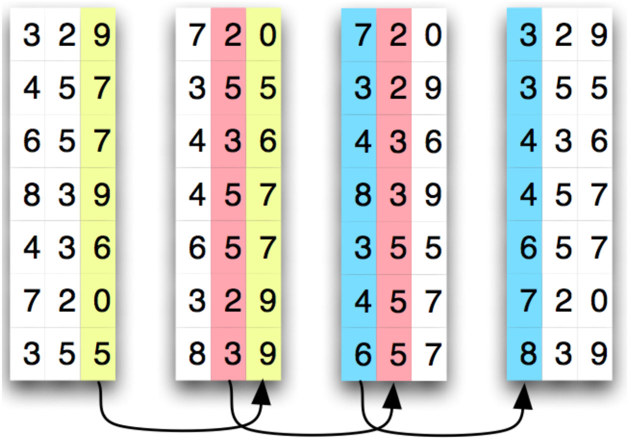
Radix sort in action



Radix sort in action



Radix sort in action



Analysis: For numbers in the range $[0 \mathinner{..} n^d - 1]$, radix sort runs in $\Theta(dn)$ time.

Bucket Sort

Assumption:

- Input elements are ‘distributed uniformly’ over known range, e.g. fractions in [0,1)
- or, integers in the range $[1, \ldots, m]$

Idea:

- Divide the interval to k buckets
- Distribute the inputs into k buckets
- Sort the numbers in each bucket
- Scan sorted buckets and combine them to produce the output array

Bucket Sort

Bucket Sort algorithm

```
1  $n \leftarrow$ length of $A[\dots]$ // input array
2  $k \leftarrow$ number of buckets
3  **for** $i \leftarrow 1$ **to** $n$
4          **do** insert $A[i]$ into list $B[\lfloor (kA[i])/m \rfloor]$
5  **for** $j \leftarrow 1$ **to** $k$
6          **do** sort list $B[j]$ using $Insertion - sort$
7  Concatenate the lists $B[1], B[2], \dots, B[k]$ together in order
```

Bucket Sort

- First 'for-loop' takes $O(n)$ time
- **Average case**: Line:5 takes $O(k)$ time in total. Line:6 takes $O(k * n/k) = O(n)$ in average case based on the assumption that the input elements are uniformly distributed over the range [1 m] or [0 1). So overall complexity becomes $O(n + k + n) = O(n)$
- **Worst case**: Worst case occurs when input elements does not satisfy the assumption that means they are not uniformly distributed over the range and all of them goes into one bucket. So, sorting elements at that bucket takes $O(n^2)$ in worst case. That means second 'for-loop' cause $O(n^2)$. Thus Bucket Sort takes $O(n + n^2) = O(n^2)$ in worst case.

Conclusion

- Linear-time sorting algorithms beat the $\Omega(n \lg n)$ lower bound of comparison-based sorts by *not* doing any element comparison.

Conclusion

- Linear-time sorting algorithms beat the $\Omega(n \lg n)$ lower bound of comparison-based sorts by *not* doing any element comparison.
- Counting sort is a $\Theta(n)$ time algorithm if $k = O(n)$, and it is stable.

Conclusion

- Linear-time sorting algorithms beat the $\Omega(n \lg n)$ lower bound of comparison-based sorts by *not* doing any element comparison.
- Counting sort is a $\Theta(n)$ time algorithm if $k = O(n)$, and it is stable.
- Radix sort is a $\Theta(dn)$ algorithm for numbers in $[0 \,..\, n^d - 1]$ range.

Conclusion

- Linear-time sorting algorithms beat the $\Omega(n \lg n)$ lower bound of comparison-based sorts by *not* doing any element comparison.
- Counting sort is a $\Theta(n)$ time algorithm if $k = O(n)$, and it is stable.
- Radix sort is a $\Theta(dn)$ algorithm for numbers in $[0 \mathinner{.\,.} n^d - 1]$ range.
- Radix sort is an excellent algorithm is trivial to implement, and works well for large inputs.

Conclusion

- Linear-time sorting algorithms beat the $\Omega(n \lg n)$ lower bound of comparison-based sorts by *not* doing any element comparison.
- Counting sort is a $\Theta(n)$ time algorithm if $k = O(n)$, and it is stable.
- Radix sort is a $\Theta(dn)$ algorithm for numbers in $[0\,..\,n^d - 1]$ range.
- Radix sort is an excellent algorithm is trivial to implement, and works well for large inputs.
- Radix sort often uses counting sort as the stable auxiliary sorting routine.
- If value of d is close to n then Radix sort takes $O(n^2)$, so in that case Bucket sort can be used as ‘d’ term is absent in it’s running time

Conclusion

- Linear-time sorting algorithms beat the $\Omega(n \lg n)$ lower bound of comparison-based sorts by *not* doing any element comparison.
- Counting sort is a $\Theta(n)$ time algorithm if $k = O(n)$, and it is stable.
- Radix sort is a $\Theta(dn)$ algorithm for numbers in $[0 \,..\, n^d - 1]$ range.
- Radix sort is an excellent algorithm is trivial to implement, and works well for large inputs.
- Radix sort often uses counting sort as the stable auxiliary sorting routine.
- If value of d is close to n then Radix sort takes $O(n^2)$, so in that case Bucket sort can be used as 'd' term is absent in it's running time

Questions to ask (and remember)

Conclusion

- Linear-time sorting algorithms beat the $\Omega(n \lg n)$ lower bound of comparison-based sorts by *not* doing any element comparison.
- Counting sort is a $\Theta(n)$ time algorithm if $k = O(n)$, and it is stable.
- Radix sort is a $\Theta(dn)$ algorithm for numbers in $[0 \mathinner{..} n^d - 1]$ range.
- Radix sort is an excellent algorithm is trivial to implement, and works well for large inputs.
- Radix sort often uses counting sort as the stable auxiliary sorting routine.
- If value of d is close to n then Radix sort takes $O(n^2)$, so in that case Bucket sort can be used as ‘d’ term is absent in it’s running time

Questions to ask (and remember)

- What is the lower bound of comparison-based sorting algorithms?

Conclusion

- Linear-time sorting algorithms beat the $\Omega(n \lg n)$ lower bound of comparison-based sorts by *not* doing any element comparison.
- Counting sort is a $\Theta(n)$ time algorithm if $k = O(n)$, and it is stable.
- Radix sort is a $\Theta(dn)$ algorithm for numbers in $[0 \mathinner{..} n^d - 1]$ range.
- Radix sort is an excellent algorithm is trivial to implement, and works well for large inputs.
- Radix sort often uses counting sort as the stable auxiliary sorting routine.
- If value of d is close to n then Radix sort takes $O(n^2)$, so in that case Bucket sort can be used as 'd' term is absent in it's running time

Questions to ask (and remember)

- What is the lower bound of comparison-based sorting algorithms?
- Are there sorting algorithms that beat this lower bound?

Conclusion

- Linear-time sorting algorithms beat the $\Omega(n \lg n)$ lower bound of comparison-based sorts by *not* doing any element comparison.
- Counting sort is a $\Theta(n)$ time algorithm if $k = O(n)$, and it is stable.
- Radix sort is a $\Theta(dn)$ algorithm for numbers in $[0 \mathinner{..} n^d - 1]$ range.
- Radix sort is an excellent algorithm is trivial to implement, and works well for large inputs.
- Radix sort often uses counting sort as the stable auxiliary sorting routine.
- If value of d is close to n then Radix sort takes $O(n^2)$, so in that case Bucket sort can be used as 'd' term is absent in it's running time

Questions to ask (and remember)

- What is the lower bound of comparison-based sorting algorithms?
- Are there sorting algorithms that beat this lower bound?
- How can you beat a proven lower bound?

Conclusion

- Linear-time sorting algorithms beat the $\Omega(n \lg n)$ lower bound of comparison-based sorts by *not* doing any element comparison.
- Counting sort is a $\Theta(n)$ time algorithm if $k = O(n)$, and it is stable.
- Radix sort is a $\Theta(dn)$ algorithm for numbers in $[0 \, .. \, n^d - 1]$ range.
- Radix sort is an excellent algorithm is trivial to implement, and works well for large inputs.
- Radix sort often uses counting sort as the stable auxiliary sorting routine.
- If value of d is close to n then Radix sort takes $O(n^2)$, so in that case Bucket sort can be used as 'd' term is absent in it's running time

Questions to ask (and remember)

- What is the lower bound of comparison-based sorting algorithms?
- Are there sorting algorithms that beat this lower bound?
- How can you beat a proven lower bound?
- Why do we recommend sorting *least-significant* digits first in radix sort?