Sorting

EECS 2011

www.eecs.yorku.ca/course_archive/2017-18/W/2011MF/

Overview

- Review
- Lower bound
- Linear sorting

Question

http://www.eecs.yorku.ca/course_archive/2017-18/W/2011MF/sorting.html

Sorting algorithms

Answer

1. Merge sort
2. Heap sort
3. Selection sort
4. Quick sort
5. Bubble sort
6. Insertion sort

Big O, Big Omega and Big Theta

To capture the running time of algorithms, we use the following notation. Let $f,\ g \in \mathbb{N} \to \mathbb{N}$.

$f \in O(g)$ if

$$\exists k > 0 : \exists m \in \mathbb{N} : \forall n \geq m : f(n) \leq k\ g(n).$$

f is bounded from above by g

$f \in \Omega(g)$ if

$$\exists k > 0 : \exists m \in \mathbb{N} : \forall n \geq m : k\ g(n) \leq f(n).$$

f is bounded from below by g

$f \in \Theta(g)$ if

$$\exists k_\ell > 0 : \exists k_u > 0 : \exists m \in \mathbb{N} : \forall n \geq m : k_\ell\ g(n) \leq f(n) \leq k_u\ g(n).$$

f is bounded from above and below by g

Sorting algorithms

selection sort	$\Theta(n^2)$
insertion sort	$\Theta(n^2)$
bubble sort	$\Theta(n^2)$
merge sort	$\Theta(n \log n)$
quick sort	$\Theta(n^2)$
heap sort	$\Theta(n \log n)$

Overview

- Review
- Lower bound
- Linear sorting

How fast can we sort?

Question

How fast can we sort?

How fast can we sort?

Question

How fast can we sort?

Answer

We can sort n elements in $\Theta(n \log n)$.

How fast can we sort?

Question

How fast can we sort?

Answer

We can sort n elements in $\Theta(n \log n)$.

Question

Can we sort any faster?

How fast can we sort?

Question

How fast can we sort?

Answer

We can sort n elements in $\Theta(n \log n)$.

Question

Can we sort any faster?

Question

Does there exist an algorithm that can sort n elements any faster?

Comparison sorts

Definition

A sorting algorithm is a comparison sort if the sorting is based on comparisons between the elements (and not on the values of the elements).

Comparison sorts

```
insertionSort(a) {
  for (i = 1; i < a.length; i = i + 1) {
    key = a[i];
    j = i;
    while (j > 0 && a[j - 1] > key) {
      a[j] = a[j - 1];
      j = j - 1;
    }
    a[j] = key;
  }
}
```

Question

Is insertion sort a comparison sort?

Comparison sorts

```
insertionSort(a) {
  for (i = 1; i < a.length; i = i + 1) {
    key = a[i];
    j = i;
    while (j > 0 && a[j - 1] > key) {
      a[j] = a[j - 1];
      j = j - 1;
    }
    a[j] = key;
  }
}
```

Question

Is insertion sort a comparison sort?

Question

Yes.

```
mergeSort(a, l, u) {
  if (l + 1 < u) {
    m = (l + u) / 2;
    mergeSort(a, l, m);
    mergeSort(a, m, u);
    merge(a, l, m, u);
  }
}
merge(a, l, m, u) {
  i = l; j = m;
  for (k = l; k < u; k = k + 1) {
    if (i < m && (j >= u || a[i] <= a[j])) {
      b[k] = a[i]; i = i + 1;
    } else {
      b[k] = a[j]; j = j + 1;
    }
  }
  for (k = l, k < u; k = k + 1) {
    a[k] = b[k];
  }
```

Comparison sorts

Question

Is merge sort a comparison sort?

Comparison sorts

Question

Is merge sort a comparison sort?

Question

Yes.

Comparison sorts

Question

Is merge sort a comparison sort?

Question

Yes.

We will see examples of sorting algorithms that are not comparison sorts later in this lecture.

Lower bound

Theorem

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

Lower bound

Theorem

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

Corollary

The worst case running time of any comparison sort is $\Omega(n \log n)$.

Lower bound

Theorem

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

Corollary

The worst case running time of any comparison sort is $\Omega(n \log n)$.

Corollary

Merge sort and heap sort are asymptotically optimal comparison sorts.

Proof of theorem

Without loss of generality, we may assume that all elements to be sorted are different.

Question

Why can we assume that?

Proof of theorem

Without loss of generality, we may assume that all elements to be sorted are different.

Question

Why can we assume that?

Answer

Assume that some elements are the same.

$$1, 6, 1, 6, 6$$

Add fractions to make them all different (in linear time).

$$1, 6, 1\tfrac{1}{2}, 6\tfrac{1}{3}, 6\tfrac{2}{3}$$

Sort them.

$$1, 1\tfrac{1}{2}, 6, 6\tfrac{1}{3}, 6\tfrac{2}{3}$$

Drop fractions (in linear time).

$$1, 1, 6, 6, 6$$

Comparisons

Given two elements a_i and a_j, we can compare then using

- $a_i < a_j$
- $a_i \leq a_j$
- $a_i = a_j$
- $a_i \geq a_j$
- $a_i > a_j$

Comparisons

Question

If all elements are different, then it suffices to compare elements using the comparator $\leq$. Why?

Comparisons

Question

If all elements are different, then it suffices to compare elements using the comparator $\leq$. Why?

Answer

The other comparators $<$, $\geq$ and $>$ can all be expressed in terms of $\leq$ since

$$\begin{aligned} a_i < a_j \quad & \text{iff} \quad a_i \leq a_j \\ & \text{iff} \quad a_j \geq a_i \\ & \text{iff} \quad a_j > a_i \end{aligned}$$

Comparisons

Question

If all elements are different, then it suffices to compare elements using the comparator $\leq$. Why?

Answer

The other comparators $<$, $\geq$ and $>$ can all be expressed in terms of $\leq$ since

$$\begin{aligned} a_i < a_j \quad & \text{iff} \quad a_i \leq a_j \\ & \text{iff} \quad a_j \geq a_i \\ & \text{iff} \quad a_j > a_i \end{aligned}$$

Corollary

If all elements are different, then each comparison sort algorithm can be rewritten so it only uses $\leq$.

Decision tree

Definition

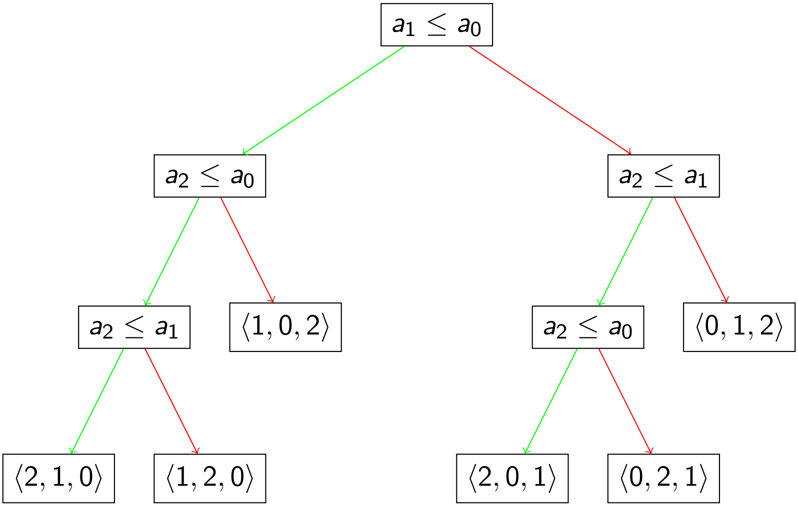
Given the number of elements to be sorted, the decision tree for a sorting algorithm is a binary tree containing the comparisons performed by the sorting algorithm.

Decision tree

```
insertionSort(a) {
  for (i = 1; i < a.length; i = i + 1) {
    key = a[i];
    j = i;
    while (j > 0 && key <= a[j - 1]) { // use <=
      a[j] = a[j - 1];
      j = j - 1;
    }
    a[j] = key;
  }
}
```

Decision tree

The decision tree for insertion sort for three elements can be depicted as follows.



Decision tree

```
selectionSort(a) {
  for (i = 0; i < a.length; i = i + 1) {
    min = i;
    for (j = i + 1; j < a.length; j = j + 1) {
      if (a[j] <= a[min]) { // use <=
        min = j;
      }
    }
    temp = a[i];
    a[i] = a[min];
    a[min] = temp;
  }
}
```

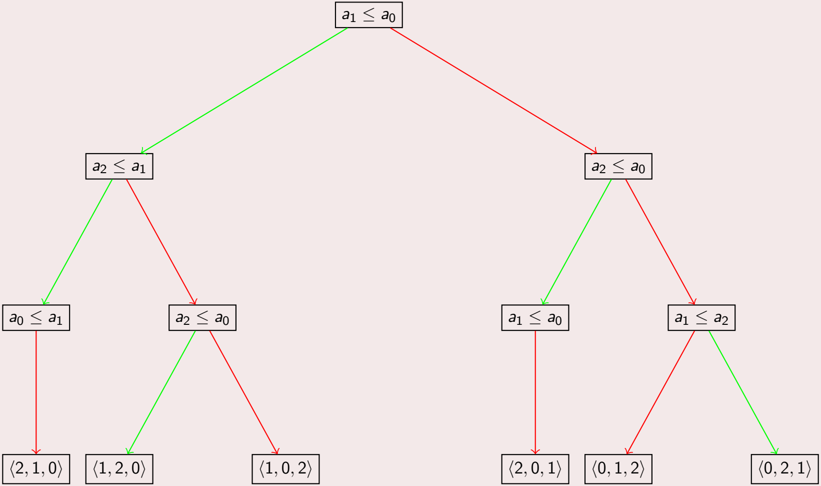
Decision tree

Question

Draw the decision tree for selection sort for three elements.

Decision tree

Answer



Height of decision tree

Observation

The worst case running time of a sorting algorithm of n elements

$$\geq$$

the maximal number of comparisons of a sorting algorithm of n elements

$$=$$

the height of the decision tree for the sorting algorithm of n elements.

Decision tree

Question

Given a decision tree for a sorting algorithm of n elements, what are the leaves?

Decision tree

Question

Given a decision tree for a sorting algorithm of n elements, what are the leaves?

Answer

Permutations of 0, 1, $\ldots$, $n - 1$.

Decision tree

Question

Given a decision tree for a sorting algorithm of n elements, what are the leaves?

Answer

Permutations of $0, 1, \ldots, n-1$.

Question

How many permutations of $0, 1, \ldots, n-1$ are there?

Decision tree

Question

Given a decision tree for a sorting algorithm of n elements, what are the leaves?

Answer

Permutations of 0, 1, . . . , $n - 1$.

Question

How many permutations of 0, 1, . . . , $n - 1$ are there?

Answer

$n \times (n-1) \times \cdots 2 \times 1 = n!$.

Properties of decision trees

Property (Proposition 8.7 of the textbook)

A binary tree of height h has at most 2^h leaves.

Properties of decision trees

Property (Proposition 8.7 of the textbook)

A binary tree of height h has at most 2^h leaves.

Property

A decision tree for a sorting algorithm of n elements has at least $n!$ leaves.

Properties of decision trees

Property (Proposition 8.7 of the textbook)

A binary tree of height h has at most 2^h leaves.

Property

A decision tree for a sorting algorithm of n elements has at least $n!$ leaves.

Question

Why does a decision tree for a sorting algorithm of n elements have at least $n!$ leaves?

Properties of decision trees

Property (Proposition 8.7 of the textbook)

A binary tree of height h has at most 2^h leaves.

Property

A decision tree for a sorting algorithm of n elements has at least $n!$ leaves.

Question

Why does a decision tree for a sorting algorithm of n elements have at least $n!$ leaves?

Answer

Because the n elements can be ordered in any order and, hence, any permutation is a possible outcome.

Properties of decision trees

Property

A binary tree of height h has at most 2^h leaves.

Property

A decision tree for a sorting algorithm of n elements has at least $n!$ leaves.

Conclusion

$2^h \geq n!$ and, hence,

$$h \geq \log(n!) \geq \log\left(\left(\frac{n}{2}\right)^{\frac{n}{2}}\right) = \frac{n}{2}\log\left(\frac{n}{2}\right) \in \Omega(n \log n).$$

Lower bound

Observation

The worst case running time of a sorting algorithm of n elements

$$\geq$$

the maximal number of comparisons of a sorting algorithm of n elements

$$=$$

the height of the decision tree for the sorting algorithm of n elements

$$\geq$$

$$\frac{n}{2}\log\left(\frac{n}{2}\right) \in \Omega(n \log n).$$

Lower bound

Theorem

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

Corollary

The worst case running time of any comparison sort is $\Omega(n \log n)$.

Corollary

Merge sort and heap sort are asymptotically optimal comparison sorts.

Overview

- Review
- Lower bound
- Linear sorting

But first...

… a break.

Overview

- Review
- Lower bound
- Linear sorting
  - Bucket sort
  - Counting sort
  - Radix sort

Linear time sorting

Theorem

The worst case running time of any comparison sort is $\Omega(n \log n)$.

Question

Can we do better if use information other than comparison of elements?

Linear time sorting

Theorem

The worst case running time of any comparison sort is $\Omega(n \log n)$.

Question

Can we do better if use information other than comparison of elements?

Answer

Yes.

Overview

- Review
- Lower bound
- Linear sorting
  - Bucket sort
  - Counting sort
  - Radix sort

Bucket sort

Assumption

All elements come from the interval $[0, N-1]$ for some $N \geq 2$.

Main idea

1. Create N buckets.
2. Place each element in "its" bucket.
3. Concatenate the buckets.

Bucket sort

Elements to be sorted: 2, 4, 4, 1, 0, 2

1. Create five buckets.

0	1	2	3	4

Bucket sort

Elements to be sorted: 2, 4, 4, 1, 0, 2

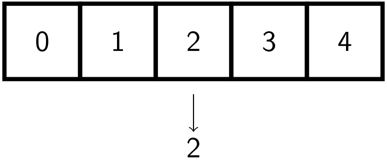
2. Place each element in "its" bucket.

0	1	2	3	4

Bucket sort

Elements to be sorted: 4, 4, 1, 0, 2

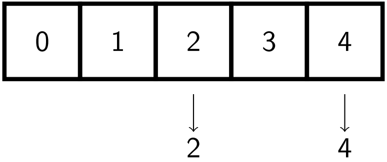
2. Place each element in “its” bucket.



Bucket sort

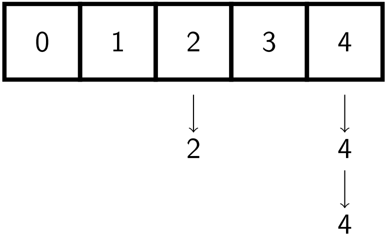
Elements to be sorted: 4, 1, 0, 2

2. Place each element in "its" bucket.



Elements to be sorted: 1, 0, 2

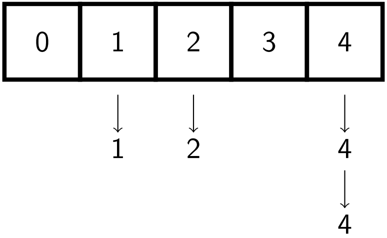
2. Place each element in “its” bucket.



Bucket sort

Elements to be sorted: 0, 2

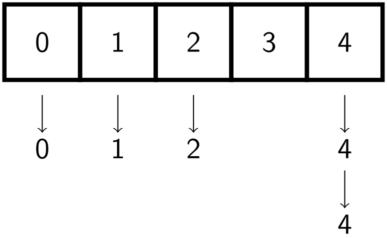
2. Place each element in "its" bucket.



Bucket sort

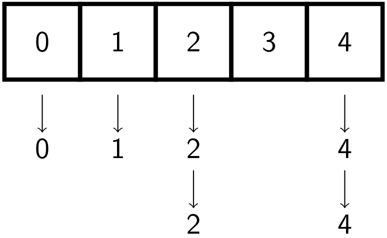
Elements to be sorted: 2

2. Place each element in "its" bucket.

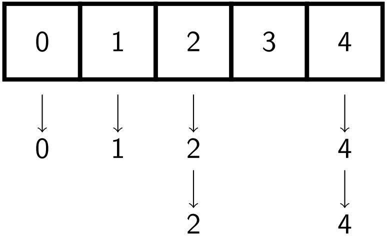


Elements to be sorted:

2. Place each element in "its" bucket.



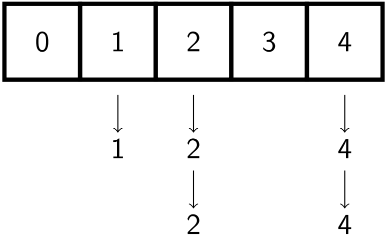
3. Concatenate the buckets.



Sorted elements:

Bucket sort

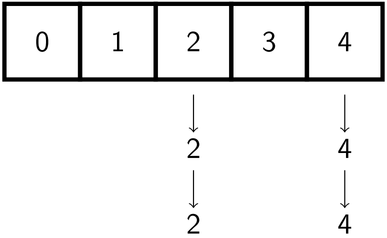
3. Concatenate the buckets.



Sorted elements: 0

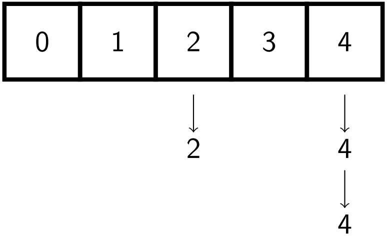
Bucket sort

3. Concatenate the buckets.



Sorted elements: 0, 1

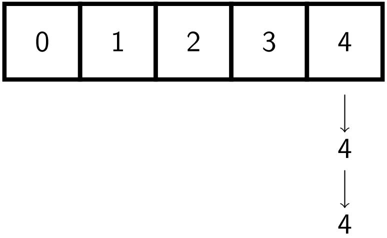
3. Concatenate the buckets.



Sorted elements: 0, 1, 2

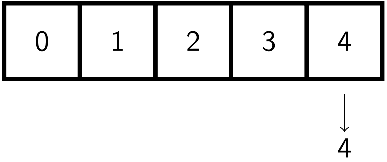
Bucket sort

3. Concatenate the buckets.



Sorted elements: 0, 1, 2, 2

3. Concatenate the buckets.



Sorted elements: 0, 1, 2, 2, 4

Bucket sort

3. Concatenate the buckets.

0	1	2	3	4

Sorted elements: 0, 1, 2, 2, 4, 4

Bucket sort

Question

How can we represent the buckets?

Question

How can we represent the buckets?

Answer

As an array of lists.

Bucket sort

```
bucketSort(a, N) {
  for (i = 0; i < N; i = i + 1) {
    b[i] = empty list;
  }
  for (i = 0; i < a.length; i = i + 1) {
    b[a[i]].add(a[i]);
  }
  j = 0;
  for (i = 0; i < N; i = i + 1) {
    while (!b[i].isEmpty()) {
      a[j] = b[i].remove();
      j++;
    }
  }
}
```

Bucket sort

Question

Express the worst case running time of bucket sort in terms of n and N.

Bucket sort

Question

Express the worst case running time of bucket sort in terms of n and N.

Answer

$O(n + N)$.

Bucket sort

Question

Express the worst case running time of bucket sort in terms of n and N.

Answer

$O(n + N)$.

Note

If $N \in O(n)$ then the worst case running time of bucket sort is $O(n)$.

Overview

- Review
- Lower bound
- Linear sorting
  - Bucket sort
  - Counting sort
  - Radix sort

Counting sort

Assumption

All elements come from the interval $[0, N-1]$ for some $N \geq 2$.

Main idea

1. Create a frequency table with N entries.
2. Keep track of the frequency of each element.
3. Compute the number of elements smaller than or equal to a given element.
4. Place each element in “right” place.

Counting sort

Elements to be sorted: 2, 4, 4, 1, 0, 2

1. Create a frequency table with five entries.

0	1	2	3	4
0	0	0	0	0

Counting sort

Elements to be sorted: 2, 4, 4, 1, 0, 2

2. Keep track of the frequency of each element.

0	1	2	3	4
0	0	0	0	0

Elements to be sorted: 4, 4, 1, 0, 2

2. Keep track of the frequency of each element.

0	1	2	3	4
0	0	1	0	0

Counting sort

Elements to be sorted: 4, 1, 0, 2

2. Keep track of the frequency of each element.

0	1	2	3	4
0	0	1	0	1

Counting sort

Elements to be sorted: 1, 0, 2

2. Keep track of the frequency of each element.

0	1	2	3	4
0	0	1	0	2

Elements to be sorted: 0, 2

2. Keep track of the frequency of each element.

0	1	2	3	4
0	1	1	0	2

Counting sort

Elements to be sorted: 2

2. Keep track of the frequency of each element.

0	1	2	3	4
1	1	1	0	2

Counting sort

Elements to be sorted:

2. Keep track of the frequency of each element.

0	1	2	3	4
1	1	2	0	2

3. Compute the number of elements smaller than or equal to a given element.

0	1	2	3	4
1	1	2	0	2
0	0	0	0	0

Counting sort

3. Compute the number of elements smaller than or equal to a given element.

0	1	2	3	4
1	1	2	0	2
1	0	0	0	0

3. Compute the number of elements smaller than or equal to a given element.

0	1	2	3	4
1	1	2	0	2
1	2	0	0	0

Counting sort

3. Compute the number of elements smaller than or equal to a given element.

0	1	2	3	4
1	1	2	0	2
1	2	4	0	0

3. Compute the number of elements smaller than or equal to a given element.

0	1	2	3	4
1	1	2	0	2
1	2	4	4	0

Counting sort

3. Compute the number of elements smaller than or equal to a given element.

0	1	2	3	4
1	1	2	0	2
1	2	4	4	6

Counting sort

Elements to be sorted: 2, 4, 4, 1, 0, 2

4. Place each element in “right” place.

0	1	2	3	4
1	1	2	0	2
1	2	4	4	6

Sorted elements:

Counting sort

Elements to be sorted: 2, 4, 4, 1, 0, 2

4. Place each element in "right" place.

0	1	2	3	4
1	1	2	0	2
1	2	4	4	6

Sorted elements:

Counting sort

Elements to be sorted: 2, 4, 4, 1, 0, 2

4. Place each element in "right" place.

0	1	2	3	4
1	1	2	0	2
1	2	4	4	6

Sorted elements:

Counting sort

Elements to be sorted: 2, 4, 4, 1, 0, 2

4. Place each element in “right” place.

0	1	2	3	4
1	1	2	0	2
1	2	4	4	6

Sorted elements:

			2		

Counting sort

Elements to be sorted: 2, 4, 4, 1, 0, 2

4. Place each element in "right" place.

0	1	2	3	4
1	1	2	0	2
1	2	3	4	6

Sorted elements:

			2		

Counting sort

Elements to be sorted: 2, 4, 4, 1, 0, 2

4. Place each element in "right" place.

0	1	2	3	4
1	1	2	0	2
1	2	3	4	6

Sorted elements:

			2		

Counting sort

Elements to be sorted: 2, 4, 4, 1, 0, 2

4. Place each element in "right" place.

0	1	2	3	4
1	1	2	0	2
1	2	3	4	6

Sorted elements:

			2		

Counting sort

Elements to be sorted: 2, 4, 4, 1, 0, 2

4. Place each element in “right” place.

0	1	2	3	4
1	1	2	0	2
1	2	3	4	6

Sorted elements:

0			2		

Counting sort

Elements to be sorted: 2, 4, 4, 1, 0, 2

4. Place each element in "right" place.

0	1	2	3	4
1	1	2	0	2
0	2	3	4	6

Sorted elements:

0			2		

Counting sort

Elements to be sorted: 2, 4, 4, 1, 0, 2

4. Place each element in "right" place.

0	1	2	3	4
1	1	2	0	2
0	2	3	4	6

Sorted elements:

0			2		

Counting sort

Elements to be sorted: 2, 4, 4, 1, 0, 2

4. Place each element in “right” place.

0	1	2	3	4
1	1	2	0	2
0	2	3	4	6

Sorted elements:

0			2		

Counting sort

Elements to be sorted: 2, 4, 4, 1, 0, 2

4. Place each element in "right" place.

0	1	2	3	4
1	1	2	0	2
0	2	3	4	6

Sorted elements:

0	1		2		

Counting sort

Elements to be sorted: 2, 4, 4, 1, 0, 2

4. Place each element in "right" place.

0	1	2	3	4
1	1	2	0	2
0	1	3	4	6

Sorted elements:

0	1		2		

Counting sort

Elements to be sorted: 2, 4, 4, 1, 0, 2

4. Place each element in “right” place.

0	1	2	3	4
1	1	2	0	2
0	1	3	4	6

Sorted elements:

0	1		2		

Counting sort

Elements to be sorted: 2, 4, 4, 1, 0, 2

4. Place each element in “right” place.

0	1	2	3	4
1	1	2	0	2
0	1	3	4	6

Sorted elements:

0	1		2		

Elements to be sorted: 2, 4, 4, 1, 0, 2

4. Place each element in "right" place.

0	1	2	3	4
1	1	2	0	2
0	1	3	4	6

Sorted elements:

0	1		2		4

Counting sort

Elements to be sorted: 2, 4, 4, 1, 0, 2

4. Place each element in “right” place.

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

Sorted elements:

0	1		2		4

Counting sort

Elements to be sorted: 2, 4, 4, 1, 0, 2

4. Place each element in "right" place.

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

Sorted elements:

0	1		2		4

Counting sort

Elements to be sorted: 2, 4, 4, 1, 0, 2

4. Place each element in “right” place.

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

Sorted elements:

0	1		2		4

Counting sort

Elements to be sorted: 2, 4, 4, 1, 0, 2

4. Place each element in "right" place.

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

Sorted elements:

0	1		2	4	4

Counting sort

Elements to be sorted: 2, 4, 4, 1, 0, 2

4. Place each element in "right" place.

0	1	2	3	4
1	1	2	0	2
0	1	3	4	4

Sorted elements:

0	1		2	4	4

Counting sort

Elements to be sorted: 2, 4, 4, 1, 0, 2

4. Place each element in "right" place.

0	1	2	3	4
1	1	2	0	2
0	1	3	4	4

Sorted elements:

0	1		2	4	4

Counting sort

Elements to be sorted: 2, 4, 4, 1, 0, 2

4. Place each element in "right" place.

0	1	2	3	4
1	1	2	0	2
0	1	3	4	4

Sorted elements:

0	1		2	4	4

Counting sort

Elements to be sorted: 2, 4, 4, 1, 0, 2

4. Place each element in “right” place.

0	1	2	3	4
1	1	2	0	2
0	1	3	4	4

Sorted elements:

0	1	2	2	4	4

Counting sort

Elements to be sorted: 2, 4, 4, 1, 0, 2

4. Place each element in "right" place.

0	1	2	3	4
1	1	2	0	2
0	1	2	4	4

Sorted elements:

0	1	2	2	4	4

Counting sort

```
countingSort(a, N) {
  for (i = 0; i < N; i = i + 1) {
    f[i] = 0;;
  }
  for (i = 0; i < a.length; i = i + 1) {
    f[a[i]] = f[a[i]] + 1;
  }
  for (i = 1; i < N; i = i + 1) {
    f[i] = f[i-1] + f[i];
  }
  for (i = a.length - 1; i >= 0; i = i - 1) {
    b[f[a[i]] = a[i];
    f[a[i]] = f[a[i]] - 1;
  }
  for (i = 0; i < a.length; i++) {
    a[i] = b[i];
  }
}
```

Question

Express the worst case running time of bucket sort in terms of n and N.

Question

Express the worst case running time of bucket sort in terms of n and N.

Answer

$O(n + N)$.

Question

Express the worst case running time of bucket sort in terms of n and N.

Answer

$O(n+N)$.

Note

If $N \in O(n)$ then the worst case running time of bucket sort is $O(n)$.

Radix Sort

Input:

- An array of *N* numbers.
- Each number contains *d* digits.
- Each digit between *[0…k-1]*

Output:

- Sorted numbers.

Each digit (column) can be sorted (e.g., using Counting Sort).

Which digit to start from?

RadixSort

344	125	125
125	134	224
333	143	225
134	224	325
224	225	134
334	243	333
143	344	334
225	333	143
325	334	243
243	325	344

All meaning in first sort lost.

Radix Sort

1. Start from the least significant digit, sort
2. Sort by the next least significant digit
3. Are the last 2 columns sorted?
4. Generalize: after j iterations, the last j columns are sorted
5. Loop invariant: Before iteration i, the keys have been correctly stable-sorted with respect to the *i-1* least-significant digits.

Radix sort

Radix-Sort(A,d)

- **for** i$\leftarrow$1 **to** d
- **do** use a stable sort to sort A on digit i

Analysis:

Given n d-digit numbers where each digit takes on up to k values, Radix-Sort sorts these numbers correctly in $\Theta(d(n+k))$ time.

Radix sort – example (binary)

Sorting a sequence of 4-bit integers