Linear Time Sorting

Lower Bound for Sorting

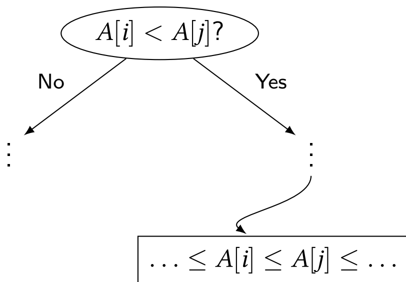
Lower Bound

Theorem

Comparison based sorting of n elements requires $\Omega(n \log n)$ time.

Lower Bound

Recall decision trees.



Decision tree for sorting

- Node decides if $A[i] < A[j]$.
- Leaf represents permutation of A.
  - $n!$ leafs
  - Thus, height is at least $\log n!$

Lower Bound

What is $\log n!$?

$$
\begin{aligned}
\log n! &= \log\big(1 \cdot 2 \cdot \ldots \cdot n\big) \\
&= \log 1 + \log 2 + \ldots + \log(n/2) + \ldots + \log n \qquad (\leq n \log n) \\
&\geq \frac{n}{2} \log(n/2) \\
&= \frac{n}{2} \log n - \frac{n}{2} \\
&\in \Omega(n \log n)
\end{aligned}
$$

Integer Sorting

Comparison based sorting requires $\Omega(n \log n)$ time.

- What if we do something different?

Integer Sorting

- Given n integers in $\{0, \ldots, k-1\}$
- Each integer fits in a word.
- All operations for words are allowed, e. g., $+$, $-$, $\ldots$
- Allows linear time sorting if k is limited.

Counting Sort

Counting Sort

Idea

- Count for each $v \in \{0, \ldots, k-1\}$ how often it is in A.
- Use this to compute index of $A[i]$ in sorted array.

A	5	2_a	4	6_a	3	1	6_b	2_b

C	0	1	2	1	1	1	2
	0	1	2	3	4	5	6

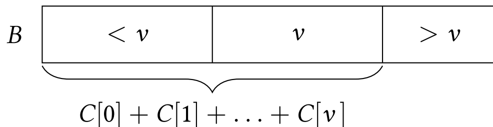
How do we compute the index?

Counting Sort

Last position of some value v in B:

- $\mathrm{Ind}_B(v) = C[0] + C[1] + \ldots + C[v] - 1$

  (-1 because array is 0-based)



Counting Sort

Observation

- $\text{Ind}_B(v) = \text{Ind}_B(v-1) + C[v]$
- Therefore, update C such that $C'[i] = C[0] + C[1] + \ldots + C[i] - 1$.

C	0	1	2	1	1	1	2
C'	-1	0	2	3	4	5	7

Counting Sort

Algorithm

	0	1	2	3	4	5	6	7
A	5	2_a	4	6_a	3	1	6_b	2_b
C								
B								

Counting Sort

Algorithm

- Count (in array C) how often each v is in A.

	0	1	2	3	4	5	6	7
A	5	2_a	4	6_a	3	1	6_b	2_b
C	0	1	2	1	1	1	2	
B								

Counting Sort

Algorithm

- Count (in array C) how often each v is in A.
- Update C such that $\mathrm{Ind}_B(v) = C[v]$.

	0	1	2	3	4	5	6	7
A	5	2_a	4	6_a	3	1	6_b	2_b

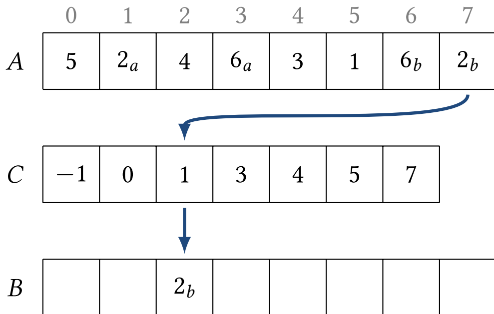
C	-1	0	2	3	4	5	7

B								

Counting Sort

Algorithm

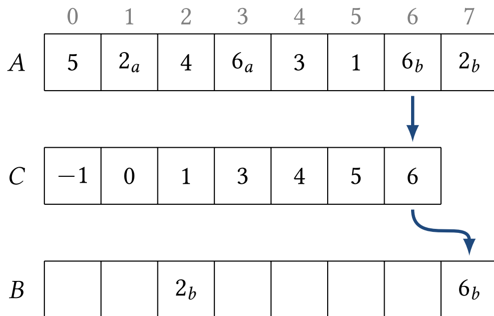
- Count (in array C) how often each v is in A.
- Update C such that $\mathrm{Ind}_B(v) = C[v]$.
- Iterate backwards over A. Copy each element $v = A[i]$ into $B[C[v]]$ and reduce $C[v]$ by 1.



Counting Sort

Algorithm

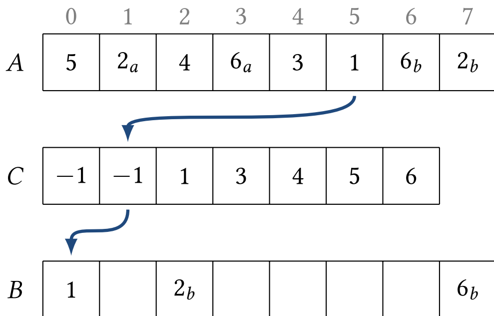
- Count (in array C) how often each v is in A.
- Update C such that $\mathrm{Ind}_B(v) = C[v]$.
- Iterate backwards over A. Copy each element $v = A[i]$ into $B[C[v]]$ and reduce $C[v]$ by 1.



Counting Sort

Algorithm

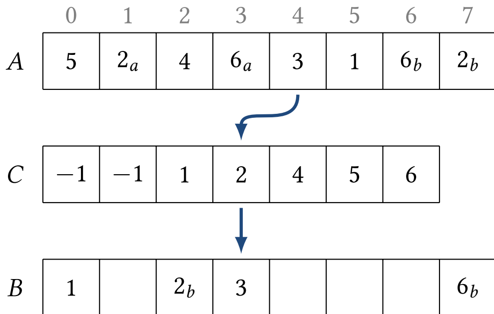
- Count (in array C) how often each v is in A.
- Update C such that $\mathrm{Ind}_B(v) = C[v]$.
- Iterate backwards over A. Copy each element $v = A[i]$ into $B[C[v]]$ and reduce $C[v]$ by 1.



Counting Sort

Algorithm

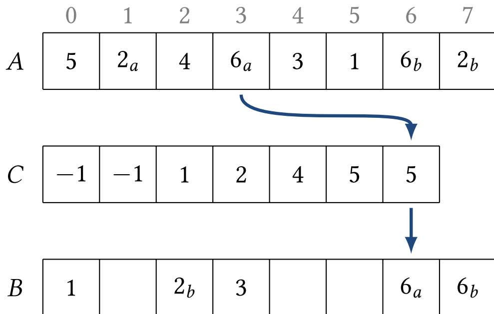
- Count (in array C) how often each v is in A.
- Update C such that $\mathrm{Ind}_B(v) = C[v]$.
- Iterate backwards over A. Copy each element $v = A[i]$ into $B[C[v]]$ and reduce $C[v]$ by 1.



Counting Sort

Algorithm

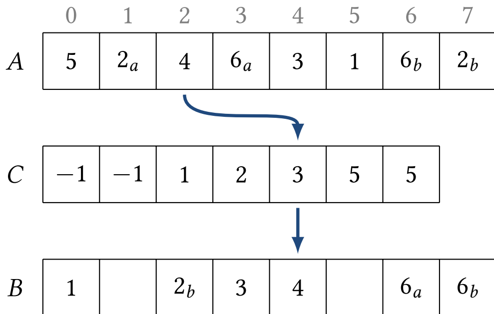
- Count (in array C) how often each v is in A.
- Update C such that $\mathrm{Ind}_B(v) = C[v]$.
- Iterate backwards over A. Copy each element $v = A[i]$ into $B[C[v]]$ and reduce $C[v]$ by 1.



Counting Sort

Algorithm

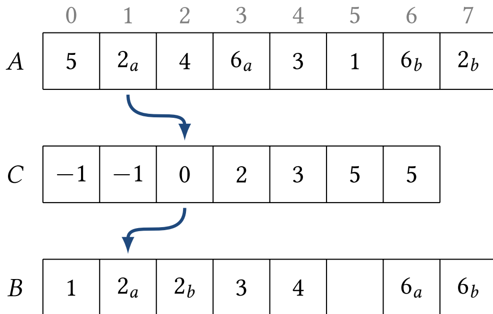
- Count (in array C) how often each v is in A.
- Update C such that $\mathrm{Ind}_B(v) = C[v]$.
- Iterate backwards over A. Copy each element $v = A[i]$ into $B[C[v]]$ and reduce $C[v]$ by 1.



Counting Sort

Algorithm

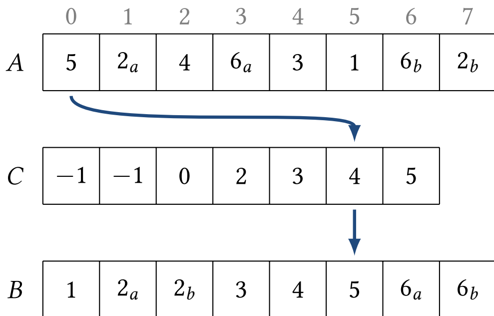
- Count (in array C) how often each v is in A.
- Update C such that $\text{Ind}_B(v) = C[v]$.
- Iterate backwards over A. Copy each element $v = A[i]$ into $B[C[v]]$ and reduce $C[v]$ by 1.



Counting Sort

Algorithm

- Count (in array C) how often each v is in A.
- Update C such that $\text{Ind}_B(v) = C[v]$.
- Iterate backwards over A. Copy each element $v = A[i]$ into $B[C[v]]$ and reduce $C[v]$ by 1.



Counting Sort

Properties

- Runtime: $\mathcal{O}(n+k)$, i. e., $\mathcal{O}(n)$ if $k \in \mathcal{O}(n)$.
- Memory: $\mathcal{O}(n+k)$ (for C and B)
- Stable (if implemented correctly)

Counting Sort

Input: An array A such that $|A| = n$ and $A[i] \in \{0, 1, \ldots, k-1\}$.

Output: An array B containing the elements of A in sorted order.

1 Create two arrays B and C with $|B| = n$, $|C| = k$ and initial value 0.

2 **For** $i := 0$ **To** $n - 1$

3 $\quad C[A[i]] := C[A[i]] + 1$

4 $C[0] := C[0] - 1$

5 **For** $i := 1$ **To** $k - 1$

6 $\quad C[i] := C[i] + C[i-1]$

7 **For** $i := n - 1$ **DownTo** 0

8 $\quad B[C[A[i]]] := A[i]$

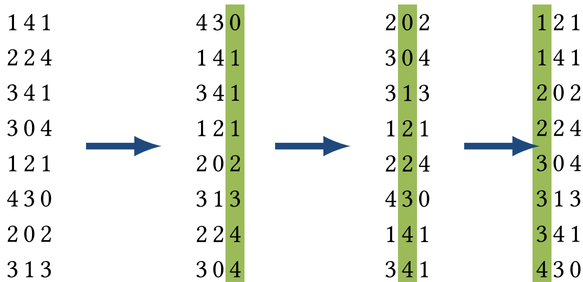
9 $\quad C[A[i]] := C[A[i]] - 1$

Radix Sort

Radix Sort

Idea

- Imagine integers as sequence of d digits with base b.
- Sort integers by digit.
- Start with the least significant digit and use a stable sort (e. g. counting sort).



Radix Sort – Runtime

Input

- n integers in range $[0, k-1]$ and a base b

Runtime

- Sorting by a single digit: $\mathcal{O}(n+b)$
- Sorting by all d digits: $\mathcal{O}((n+b)\cdot d) = \mathcal{O}((n+b)\log_b k)$
- $\mathcal{O}(nc)$ time if $b = n$ and $k \leq n^c$
  linear if c is constant

Other properties

- Stable
- $\mathcal{O}(n+b)$ additional memory is used.

We assume counting sort for sorting by digit.