# Problem Complexity

- problem complexity is the inherent worst case complexity of a problem
- better algorithms bring the upper bound down
- each better proof brings the lower bound up
- if the upper and lower bound match we know the inherent complexity of the problem

## Trivial Lower Bound

- All problems have a lower bound of the size of the output.

## Sorting $\Omega(n)$ lower bound

- assume some algorithm solves sorting in $o(n)$
- that algorithm must have failed to output at least one element and is therefore incorrect

# Sorting $\Omega(n \log n)$ Lower Bound

**Theorem:** Every comparison-based sorting algorithm has a worst case running time of $\Omega(n \log n)$

**Proof:**

- Assume we are sorting the distinct numbers 1 through $n$
- Each input is a permutation of $\{1, 2, \ldots, n\}$ $n!$ possible inputs
- We want to find a value $k$ such that for any input it takes no more than $k$ comparisons to correctly sort.

- Each comparison results in either $>$ or $<$

  which gives us $2^k$ possible computation paths

- If $2^k < n!$ there are at least two inputs

  with the same computation path

  - this would mean that for one of

    those inputs the algorithm is incorrect

- So by correctness

$$2^k \geq n!$$

- Solve for $k$

$$2^k \geq n!$$

$c\, n \log n$

$$\begin{aligned}
k &\geq \lg(n!) \\
&\geq \lg(n) + \lg(n-1) + \lg(n-2) + \ldots + \lg(1) \\
&\geq \lg(n) + \lg(n-1) + \ldots + \lg\left(\tfrac{n}{2}\right) \\
&\geq \lg\left(\tfrac{n}{2}\right) + \lg\left(\tfrac{n}{2}\right) + \ldots + \lg\left(\tfrac{n}{2}\right) \\
&\geq \tfrac{n}{2} \lg\left(\tfrac{n}{2}\right) \\
&\geq \tfrac{n}{2}\left[\lg(n) - \lg(2)\right] \\
&\geq \tfrac{n}{2}\left[\lg(n) - 1\right] \\
&\geq \tfrac{n}{2}\lg(n) - \tfrac{n}{2} \in \Omega(n \log n)
\end{aligned}$$

## Linear Time Sorting

- Counting Sort
- Radix Sort
- Bucket Sort

## Counting Sort

- works when each of the n elements is an integer in the range 0...K

```
COUNTINGSORT(A, B, k)
1  let C[0, , k] be a new array
2  for i = 0 to k
3      C[i] = 0
4  for j = 1 to A.length
5      C[A[j]] = C[A[j]] + 1
6  for i = 0 to k
7      C[i] = C[i] + C[i − 1]
8  for j = A.length to 1
9      B[C[A[j]]] = A[j]
10     C[A[j]] = C[A[j]] − 1
```

Input size: $n = A.length$

$\Theta(k)$

$\Theta(n)$

$\Theta(k)$

$\Theta(n)$

Complexity

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

when $k \in O(n)$ counting sort runs in $\Theta(n)$ time

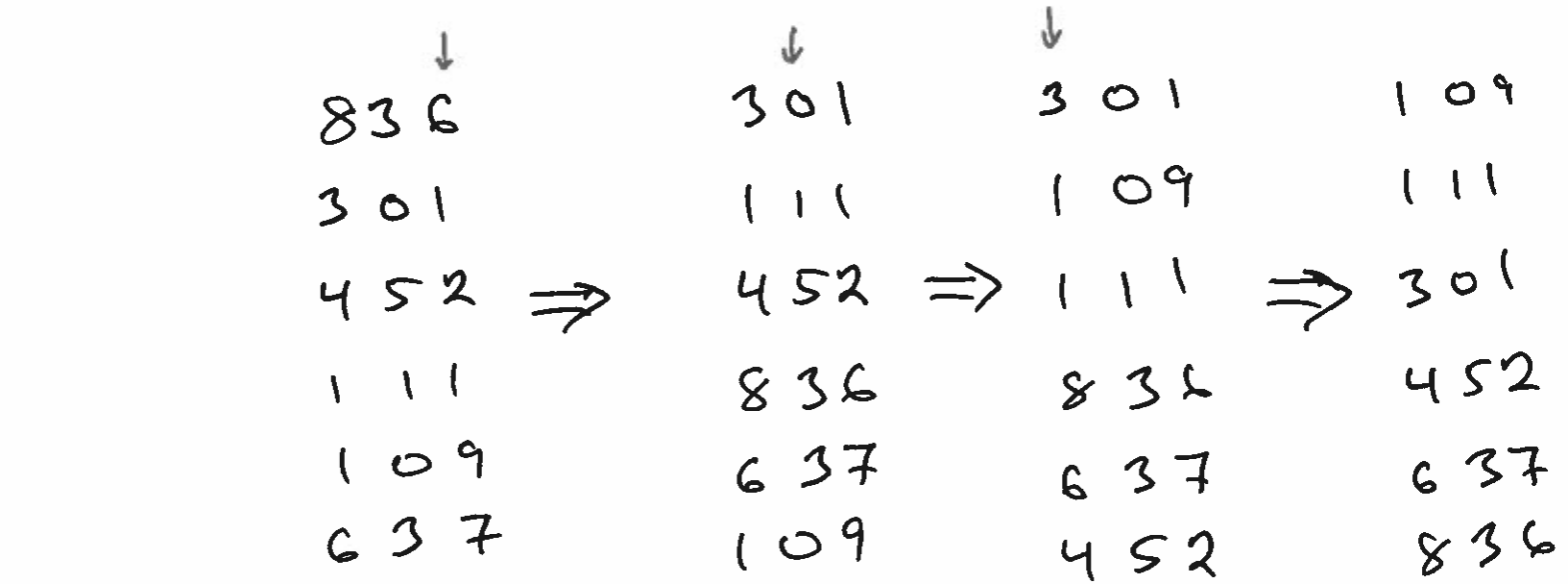
If the range is extremely large this will run slowly

RADIXSORT$(A, d)$

1 **for** $i = 1$ **to** $d$

2 use a stable sort to sort $A$ on digit $i$

- works on a list of $n$ $d$-digit numbers in which each digit can take on one of $k$ values
- start by sorting the least significant digit



# Radix Sort Complexity

- Assume Counting sort as the intermediate sort.
- $\Theta(n+k)$ per iteration of the loop
- $d$ iterations
- $\Theta(d(n+k))$ total
- if $k \in \overset{O}{\Theta}(n)$

  time $\in \Theta(dn)$

- How do we break a key into digits?
  - $n$ - keys
  - $b$ bits/word
  - Break each word into $r$-bit digits

$$d = \left\lceil \frac{b}{r} \right\rceil$$

  - using counting sort $k = 2^r - 1$

Example

32-bit words

8-bit digits

$b = 32$ $\quad d = \left\lceil \frac{32}{8} \right\rceil = 4$

$r = 8$

$k = 2^8 - 1 = 255$

Time $\in \Theta\left(\frac{b}{r}(n + 2^r)\right)$

$\Theta(4(n + 2^8)) \in \Theta(n)$

A contains real numbers between 0 and 1

BUCKETSORT($A$)

```
1  n = A.length
2  let B[0...n − 1] be a new array
3  for i = 0 to n − 1
4       make B[i] an empty list
5  for i = 1 to n
6       insert A[i] into list B[⌊nA[i]⌋]
7  for i = 0 to n − 1
8       sort list B[i] with insertion sort
9  concatenate the list B[0], B[1], ..., B[n − 1] together in order
```

Best Case:

The input is uniformly distributed resulting in 1 item per bucket $\Theta(n)$

Worst Case:

All items map to the same bucket $\Theta(n^2)$

0.20

0.37

0.14

0.25

0.87

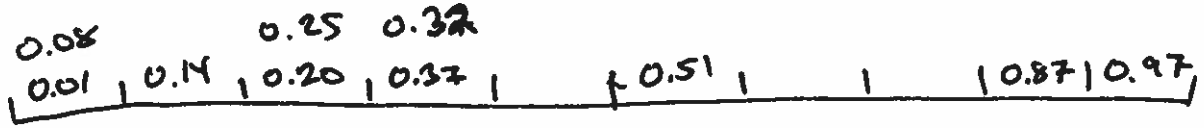
0.01

0.08

0.97

0.32

0.51




0.01 0.08 0.14 0.20 0.25 0.32 0.37

0.51 0.87 0.97

# Problem of the Day

**(a)** Prove that summing all elements of an unstructured array of $n$ integers requires $\Omega(n)$ additions.

**(b)** Give an example of an array of $n$ integers where the sum can be determined in $O(1)$ additions. (Hint: The integers can be anything you want)

**(a)** Prove that summing all elements of an unstructured array of $n$ integers requires $\Omega(n)$ additions.

**Solution:** Assume for the sake of contradiction that there exists some algorithm that can sum any array in $o(n)$ time. That is to say, with a strictly faster than linear time. Given an array containing non-zero integers $x_1, x_2, ..., x_n$ it produces some sum $X$. Since the algorithm is strictly faster than linear time there must be at least one $x_i$ which is never examined during the running time of the algorithm. Changing the value of $x_i$ to any other non-zero value will change the sum of the array to some new value $Y$, but because $x_i$ is never examined by the algorithm this change will have no impact on the sum produced by the algorithm. Since the algorithm will still output $X$, but the correct answer is $Y$ and $X \neq Y$ this algorithm has not correctly summed the array. Therefore no such algorithm can exist.

**(b)** Give an example of an array of $n$ integers where the sum can be determined in $O(1)$ additions.

**Solution:** One example would be an array containing just the integers $1...n$ in any order. We already know that the sum of this array is $\frac{n(n+1)}{2}$, so a single addition is enough. Note that this is not a contradiction of the proof from part (a), since in order to sum the array in less than linear time we had to impose restrictions on the input. Problem complexity is always a lower bound on the worst case input.