Sorting

Or: some of what Koz spent about two years of his life on

Koz Ross

5th October, 2017

[AUT CSC]

AUT Computer Science Club

Outline

What is sorting and why we care

Sorting is putting things in some kind of order.

What is sorting and why we care

Sorting is putting things in some kind of order. This is very useful, for a range of reasons:

What is sorting and why we care

Sorting is putting things in some kind of order. This is very useful, for a range of reasons:

- Some tasks are impossible without it (e.g. finding the median)

What is sorting and why we care

Sorting is putting things in some kind of order. This is very useful, for a range of reasons:

- Some tasks are impossible without it (e.g. finding the median)
- Some tasks are much easier on sorted data (e.g. searching for a specific item)

What is sorting and why we care

Sorting is putting things in some kind of order. This is very useful, for a range of reasons:

- Some tasks are impossible without it (e.g. finding the median)
- Some tasks are much easier on sorted data (e.g. searching for a specific item)
- Generalizes many other tasks (e.g. top-k queries)

What is sorting and why we care

Sorting is putting things in some kind of order. This is very useful, for a range of reasons:

- Some tasks are impossible without it (e.g. finding the median)
- Some tasks are much easier on sorted data (e.g. searching for a specific item)
- Generalizes many other tasks (e.g. top-k queries)
- Sorting is used *everywhere*

What is sorting and why we care

Sorting is putting things in some kind of order. This is very useful, for a range of reasons:

- Some tasks are impossible without it (e.g. finding the median)
- Some tasks are much easier on sorted data (e.g. searching for a specific item)
- Generalizes many other tasks (e.g. top-k queries)
- Sorting is used *everywhere*

As a result, sorting is one of *the* oldest computer science problems we have, and has been studied *to death*.

A little tour

A little tour

- A precise definition of sorting

A little tour

- A precise definition of sorting
- Some known sorting algorithms

A little tour

- A precise definition of sorting
- Some known sorting algorithms
- Inherent limit on the performance of *any* algorithm

A little tour

- A precise definition of sorting
- Some known sorting algorithms
- Inherent limit on the performance of *any* algorithm
- Limitations of this result

A little tour

- A precise definition of sorting
- Some known sorting algorithms
- Inherent limit on the performance of *any* algorithm
- Limitations of this result

Without further ado, let's commence!

Some basics

Let $\mathbb{N} = \{0, 1, 2, \ldots\}$ be the set of *natural numbers*.

Some basics

Let $\mathbb{N} = \{0, 1, 2, \ldots\}$ be the set of *natural numbers*. A n*-tuple* $t = (a_0, a_1, \ldots, a_{n-1})$ is a collection of elements with fixed positions. We use $t[i]$ to denote a_i for $i \in 0, 1, \ldots, n - 1$.

Some basics

Let $\mathbb{N} = \{0, 1, 2, \ldots\}$ be the set of *natural numbers*. A n*-tuple* $t = (a_0, a_1, \ldots, a_{n-1})$ is a collection of elements with fixed positions. We use $t[i]$ to denote a_i for $i \in 0, 1, \ldots, n-1$.

Definition

Let A, B be sets. The *Cartesian product* of A and B is the set

$$A \times B = \{(x, y) \mid x \in A, y \in B\}$$

Some basics

Let $\mathbb{N} = \{0, 1, 2, \ldots\}$ be the set of *natural numbers*. A n*-tuple* $t = (a_0, a_1, \ldots, a_{n-1})$ is a collection of elements with fixed positions. We use $t[i]$ to denote a_i for $i \in 0, 1, \ldots, n-1$.

Definition

Let A, B be sets. The *Cartesian product* of A and B is the set

$$A \times B = \{(x, y) \mid x \in A, y \in B\}$$

Example

Let $A = \{1, 2, 3\}, B = \{a, b\}$. $A \times B$ is

$$\{(1, a), (1, b), (2, a), (2, b), (3, a), (3, b)\}$$

Some basics

Let $\mathbb{N} = \{0, 1, 2, \ldots\}$ be the set of *natural numbers*. A n*-tuple* $t = (a_0, a_1, \ldots, a_{n-1})$ is a collection of elements with fixed positions. We use $t[i]$ to denote a_i for $i \in 0, 1, \ldots, n-1$.

Definition

Let A, B be sets. The *Cartesian product* of A and B is the set

$$A \times B = \{(x, y) \mid x \in A, y \in B\}$$

Example

Let $A = \{1, 2, 3\}, B = \{a, b\}$. $A \times B$ is

$$\{(1, a), (1, b), (2, a), (2, b), (3, a), (3, b)\}$$

We denote the special case of $A \times A$ as A^2.

Relations

Definition

Let A, B be sets. A *(binary) relation between A and B* is $R \subseteq A \times B$. For $x \in A, y \in B$, we write xRy to mean $(x, y) \in R$.

Relations

Definition

Let A, B be sets. A *(binary) relation between A and B* is $R \subseteq A \times B$. For $x \in A, y \in B$, we write xRy to mean $(x, y) \in R$.

We can think of a relation as explicitly spelling out what things in A and B are related.

Ordering

Definition

Let S be a set. A *(non-strict) ordering on* S is a binary relation $\leqq \subseteq S^2$, such that $\leqq$ has the following properties:

Ordering

Definition

Let S be a set. A *(non-strict) ordering on* S is a binary relation $\leqq \subseteq S^2$, such that $\leqq$ has the following properties:

Antisymmetry: For any $a, b \in S$, if $a \leqq b$ and $b \leqq a$, then $a = b$;

Ordering

Definition

Let S be a set. A *(non-strict) ordering on* S is a binary relation $\leqq \subseteq S^2$, such that $\leqq$ has the following properties:

Antisymmetry: For any $a, b \in S$, if $a \leqq b$ and $b \leqq a$, then $a = b$;

Transitivity: For any $a, b, c \in S$, if $a \leqq b$ and $b \leqq c$, then $a \leqq c$; and

Ordering

Definition

Let S be a set. A *(non-strict) ordering on* S is a binary relation $\leqq \subseteq S^2$, such that $\leqq$ has the following properties:

Antisymmetry: For any $a, b \in S$, if $a \leqq b$ and $b \leqq a$, then $a = b$;

Transitivity: For any $a, b, c \in S$, if $a \leqq b$ and $b \leqq c$, then $a \leqq c$; and

Totality: For any $a, b \in S$, $a \leqq b$ or $b \leqq a$.

Ordering

Definition

Let S be a set. A *(non-strict) ordering on S* is a binary relation $\leqq \subseteq S^2$, such that $\leqq$ has the following properties:

Antisymmetry: For any $a, b \in S$, if $a \leqq b$ and $b \leqq a$, then $a = b$;

Transitivity: For any $a, b, c \in S$, if $a \leqq b$ and $b \leqq c$, then $a \leqq c$; and

Totality: For any $a, b \in S$, $a \leqq b$ or $b \leqq a$.

We can read $a \leqq b$ as 'a does not come after b according to the ordering $\leqq$'.

Ordering

Definition

Let S be a set. A *(non-strict) ordering on S* is a binary relation $\leqq \subseteq S^2$, such that $\leqq$ has the following properties:

Antisymmetry: For any $a, b \in S$, if $a \leqq b$ and $b \leqq a$, then $a = b$;

Transitivity: For any $a, b, c \in S$, if $a \leqq b$ and $b \leqq c$, then $a \leqq c$; and

Totality: For any $a, b \in S$, $a \leqq b$ or $b \leqq a$.

We can read $a \leqq b$ as 'a does not come after b according to the ordering $\leqq$'. We can have multiple orderings on various sets.

Ordering examples

Consider $\mathbb{N}$. We can see that $\leq$ is an ordering on $\mathbb{N}$:

Ordering examples

Consider $\mathbb{N}$. We can see that $\leq$ is an ordering on $\mathbb{N}$:

Antisymmetry: If we have two natural numbers x, y such that $x \leq y$ and $y \leq x$, they must be equal;

Ordering examples

Consider $\mathbb{N}$. We can see that $\leq$ is an ordering on $\mathbb{N}$:

Antisymmetry: If we have two natural numbers x, y such that $x \leq y$ and $y \leq x$, they must be equal;

Transitivity: We can 'chain together' $\leq$ in exactly this way;

Ordering examples

Consider $\mathbb{N}$. We can see that $\leq$ is an ordering on $\mathbb{N}$:

Antisymmetry: If we have two natural numbers x, y such that $x \leq y$ and $y \leq x$, they must be equal;

Transitivity: We can 'chain together' $\leq$ in exactly this way;

Totality: Any two natural numbers can be compared with $\leq$.

Ordering examples

Consider $\mathbb{N}$. We can see that $\leq$ is an ordering on $\mathbb{N}$:

Antisymmetry: If we have two natural numbers x, y such that $x \leq y$ and $y \leq x$, they must be equal;

Transitivity: We can 'chain together' $\leq$ in exactly this way;

Totality: Any two natural numbers can be compared with $\leq$.

We can also do something similar with $\geq$. If we look at other sets, we get many other orderings (e.g. lex and reverse lex for strings).

Defining sorting

Defining sorting

Definition

Let S be a set, t be an n-tuple of elements of S, and $\leqq$ be an ordering on S. Given $t, \leqq$ as input, the *sorting problem* requires us to return an n-tuple t' such that:

Defining sorting

Definition

Let S be a set, t be an n-tuple of elements of S, and $\leqq$ be an ordering on S. Given $t, \leqq$ as input, the *sorting problem* requires us to return an n-tuple t' such that:

- Every element of t is an element of t'; and

Defining sorting

Definition

Let S be a set, t be an n-tuple of elements of S, and $\leqq$ be an ordering on S. Given $t, \leqq$ as input, the *sorting problem* requires us to return an n-tuple t' such that:

- Every element of t is an element of t'; and
- For any $i, j \in 0, 1, \ldots, n-1$, if $i \leq j$ then $t'[i] \leqq t'[j]$.

Defining sorting

Definition

Let S be a set, t be an n-tuple of elements of S, and $\leqq$ be an ordering on S. Given $t, \leqq$ as input, the *sorting problem* requires us to return an n-tuple t' such that:

- Every element of t is an element of t'; and
- For any $i, j \in 0, 1, \ldots, n-1$, if $i \leq j$ then $t'[i] \leqq t'[j]$.

Put simply, the sorting problem requires us to put t 'in order' according to $\leqq$.

Shifting to algorithms

Since we're dealing with algorithms, we have to be precise with our assumptions. We also want to be as general as possible.

Shifting to algorithms

Since we're dealing with algorithms, we have to be precise with our assumptions. We also want to be as general as possible.

Definition

A *comparison sort* is an algorithm for the sorting problem that does not make any additional assumptions about the input tuple, aside from what is required by the sorting problem.

Shifting to algorithms

Since we're dealing with algorithms, we have to be precise with our assumptions. We also want to be as general as possible.

Definition

A *comparison sort* is an algorithm for the sorting problem that does not make any additional assumptions about the input tuple, aside from what is required by the sorting problem.

We can view this in several ways:

Shifting to algorithms

Since we're dealing with algorithms, we have to be precise with our assumptions. We also want to be as general as possible.

Definition

A *comparison sort* is an algorithm for the sorting problem that does not make any additional assumptions about the input tuple, aside from what is required by the sorting problem.

We can view this in several ways:

- Most general kind of sort (assumes only what it must)

Shifting to algorithms

Since we're dealing with algorithms, we have to be precise with our assumptions. We also want to be as general as possible.

Definition

A *comparison sort* is an algorithm for the sorting problem that does not make any additional assumptions about the input tuple, aside from what is required by the sorting problem.

We can view this in several ways:

- Most general kind of sort (assumes only what it must)
- Most 'difficult' sort (no additional 'hacks' we can lean on based on the data)

Shifting to algorithms

Since we're dealing with algorithms, we have to be precise with our assumptions. We also want to be as general as possible.

Definition

A *comparison sort* is an algorithm for the sorting problem that does not make any additional assumptions about the input tuple, aside from what is required by the sorting problem.

We can view this in several ways:

- Most general kind of sort (assumes only what it must)
- Most 'difficult' sort (no additional 'hacks' we can lean on based on the data)
- Closest to a 'pure' view of how hard the sorting problem is (no 'extra baggage' to confuse analysis)

Our approach to analysis

We'll take the 'default' for algorithm analysis:

Our approach to analysis

We'll take the 'default' for algorithm analysis:

RAM model:

- Serial processor (can only do one thing at a time)

Our approach to analysis

We'll take the 'default' for algorithm analysis:

RAM model:
- Serial processor (can only do one thing at a time)
- Non-hierarchical, addressable memory

Our approach to analysis

We'll take the 'default' for algorithm analysis:

RAM model:

- Serial processor (can only do one thing at a time)
- Non-hierarchical, addressable memory
- Primitive operations take constant time

Our approach to analysis

We'll take the 'default' for algorithm analysis:

RAM model:
- Serial processor (can only do one thing at a time)
- Non-hierarchical, addressable memory
- Primitive operations take constant time

Asymptotic:
- Based on the size of the input

Our approach to analysis

We'll take the 'default' for algorithm analysis:

RAM model:
- Serial processor (can only do one thing at a time)
- Non-hierarchical, addressable memory
- Primitive operations take constant time

Asymptotic:
- Based on the size of the input
- Care about the *growth rate* of the time required

Our approach to analysis

We'll take the 'default' for algorithm analysis:

RAM model:
- Serial processor (can only do one thing at a time)
- Non-hierarchical, addressable memory
- Primitive operations take constant time

Asymptotic:
- Based on the size of the input
- Care about the *growth rate* of the time required

Worst-case:
- If we'd have different results for same-size inputs, take the worst one

Our approach to analysis

We'll take the 'default' for algorithm analysis:

RAM model:
- Serial processor (can only do one thing at a time)
- Non-hierarchical, addressable memory
- Primitive operations take constant time

Asymptotic:
- Based on the size of the input
- Care about the *growth rate* of the time required

Worst-case:
- If we'd have different results for same-size inputs, take the worst one
- All input tuple elements are unique

Our approach to analysis

We'll take the 'default' for algorithm analysis:

RAM model:
- Serial processor (can only do one thing at a time)
- Non-hierarchical, addressable memory
- Primitive operations take constant time

Asymptotic:
- Based on the size of the input
- Care about the *growth rate* of the time required

Worst-case:
- If we'd have different results for same-size inputs, take the worst one
- All input tuple elements are unique
- Tuple elements are 'random' (i.e. no sorted sub-sequences)

Algorithms we know

Algorithms we know

'Slow' ($O(n^2)$): Bubble sort, insertion sort, selection sort, etc.

Algorithms we know

'Slow' ($O(n^2)$): Bubble sort, insertion sort, selection sort, etc.

'Fast' ($O(n \log(n))$): Mergesort, quicksort, introsort, timesort, etc.

Algorithms we know

'Slow' ($O(n^2)$): Bubble sort, insertion sort, selection sort, etc.

'Fast' ($O(n \log(n))$): Mergesort, quicksort, introsort, timesort, etc.

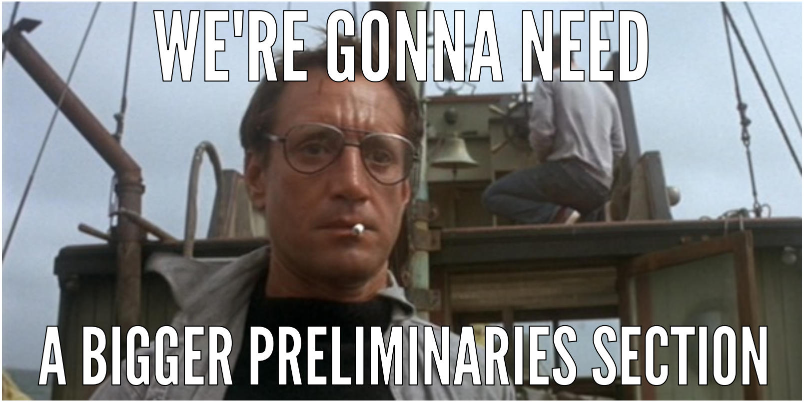
Can we do better than this? Is there *any* algorithm that can beat the 'fast' ones?

Algorithms we know

‘Slow’ ($O(n^2)$): Bubble sort, insertion sort, selection sort, etc.

‘Fast’ ($O(n\log(n))$): Mergesort, quicksort, introsort, timesort, etc.

Can we do better than this? Is there *any* algorithm that can beat the ‘fast’ ones? No.


WE'RE GONNA NEED
A BIGGER PRELIMINARIES SECTION

Factorials and permutations

Definition

Let $n \in \mathbb{N}$. The *factorial of* n is

$$n! = \begin{cases} 1 & \text{if } n = 0\text{; and} \\ n \cdot n - 1 & \text{otherwise} \end{cases}$$

Alternatively,

$$n! = \prod_{i=1}^{n} i = 1 \times 2 \times \cdots \times n$$

Factorials and permutations

Definition

Let $n \in \mathbb{N}$. The *factorial of* n is

$$n! = \begin{cases} 1 & \text{if } n = 0\text{; and} \\ n \cdot n - 1 & \text{otherwise} \end{cases}$$

Alternatively,

$$n! = \prod_{i=1}^{n} i = 1 \times 2 \times \cdots \times n$$

Definition

Let S be a set of n elements. A *permutation of* S is an n-tuple of unique elements of S.

Factorials and permutations

Definition

Let $n \in \mathbb{N}$. The *factorial of n* is

$$n! = \begin{cases} 1 & \text{if } n = 0\text{; and} \\ n \cdot n - 1 & \text{otherwise} \end{cases}$$

Alternatively,

$$n! = \prod_{i=1}^{n} i = 1 \times 2 \times \cdots \times n$$

Definition

Let S be a set of n elements. A *permutation of S* is an n-tuple of unique elements of S.

Example

Two possible permutations of $S = \{1, 2, 3\}$ are $(1, 3, 2), (2, 3, 1)$.

Relating factorials and permutations

Lemma

Let S be a set of n elements. There are $n!$ possible permutations of S.

Relating factorials and permutations

Lemma

Let S be a set of n elements. There are $n!$ possible permutations of S.

Proof.

We use induction on n.

Relating factorials and permutations

Lemma

Let S be a set of n elements. There are $n!$ possible permutations of S.

Proof.

We use induction on n. When $n = 0$, we observe that the only permutation is $()$. As $0! = 1$, the lemma holds for $n = 0$.

Relating factorials and permutations

Lemma

Let S be a set of n elements. There are $n!$ possible permutations of S.

Proof.

We use induction on n. When $n = 0$, we observe that the only permutation is $()$. As $0! = 1$, the lemma holds for $n = 0$.

When $n > 0$, suppose that the lemma holds to some $k \geq 0$. Thus, there are $k!$ permutations of any k-element set S.

Relating factorials and permutations

Lemma

Let S be a set of n elements. There are $n!$ possible permutations of S.

Proof.

We use induction on n. When $n = 0$, we observe that the only permutation is $()$. As $0! = 1$, the lemma holds for $n = 0$.

When $n > 0$, suppose that the lemma holds to some $k \geq 0$. Thus, there are $k!$ permutations of any k-element set S. Without loss of generality, we observe that any $k + 1$-element set S' is equal to some k-element set S with an additional element $u \notin S$.

Relating factorials and permutations

Lemma

Let S be a set of n elements. There are $n!$ possible permutations of S.

Proof.

We use induction on n. When $n = 0$, we observe that the only permutation is $()$. As $0! = 1$, the lemma holds for $n = 0$.

When $n > 0$, suppose that the lemma holds to some $k \geq 0$. Thus, there are $k!$ permutations of any k-element set S. Without loss of generality, we observe that any $k + 1$-element set S' is equal to some k-element set S with an additional element $u \notin S$. Thus, to convert a permutation of S into a permutation p of S', we need to 'insert' u into p.

Relating factorials and permutations

Lemma

Let S be a set of n elements. There are $n!$ possible permutations of S.

Proof.

We use induction on n. When $n = 0$, we observe that the only permutation is $()$. As $0! = 1$, the lemma holds for $n = 0$.

When $n > 0$, suppose that the lemma holds to some $k \geq 0$. Thus, there are $k!$ permutations of any k-element set S. Without loss of generality, we observe that any $k + 1$-element set S' is equal to some k-element set S with an additional element $u \notin S$. Thus, to convert a permutation of S into a permutation p of S', we need to 'insert' u into p. As there are $k + 1$ possible positions to insert into for each permutation of S, the number of possible permutations of S' is thus $k!(k + 1) = (k + 1)!$. Thus, the lemma holds for all n. $\square$

Why this matters

Why this matters

Given our assumptions (worst-case), the sorting problem basically involves finding a specific permutation (the one where everything is in the right place).

Why this matters

Given our assumptions (worst-case), the sorting problem basically involves finding a specific permutation (the one where everything is in the right place). Thus, the amount of work *any* comparison sort will have to do will be based on $n!$ somehow.

Why this matters

Given our assumptions (worst-case), the sorting problem basically involves finding a specific permutation (the one where everything is in the right place). Thus, the amount of work *any* comparison sort will have to do will be based on $n!$ somehow.

The factorial function is a gigantic pain to analyze. Let's make our lives simpler:

Why this matters

Given our assumptions (worst-case), the sorting problem basically involves finding a specific permutation (the one where everything is in the right place). Thus, the amount of work *any* comparison sort will have to do will be based on $n!$ somehow.

The factorial function is a gigantic pain to analyze. Let's make our lives simpler:

Definition (Stirling approximation)

$\log_2(n!)$ is $O(n \log(n))$ and $n \log(n)$ is $O(\log_2(n!))$.

Why this matters

Given our assumptions (worst-case), the sorting problem basically involves finding a specific permutation (the one where everything is in the right place). Thus, the amount of work *any* comparison sort will have to do will be based on $n!$ somehow.

The factorial function is a gigantic pain to analyze. Let's make our lives simpler:

Definition (Stirling approximation)

$\log_2(n!)$ is $O(n \log(n))$ and $n \log(n)$ is $O(\log_2(n!))$.

Essentially, the Stirling approximation tells us that $\log_2(n!)$ and $n \log(n)$ grow at the same rate asymptotically.

Why this matters

Given our assumptions (worst-case), the sorting problem basically involves finding a specific permutation (the one where everything is in the right place). Thus, the amount of work *any* comparison sort will have to do will be based on $n!$ somehow.

The factorial function is a gigantic pain to analyze. Let's make our lives simpler:

Definition (Stirling approximation)

$\log_2(n!)$ is $O(n \log(n))$ and $n \log(n)$ is $O(\log_2(n!))$.

Essentially, the Stirling approximation tells us that $\log_2(n!)$ and $n \log(n)$ grow at the same rate asymptotically. We say $\log_2(n!)$ is $\Theta(n \log(n))$ in such a case.

The proof, at last

Theorem

Any comparison sort must perform at least $\Theta(n \, \textsf{log}(n))$ *comparisons between elements of the input.*

The proof, at last

Theorem

Any comparison sort must perform at least $\Theta(n \log(n))$ *comparisons between elements of the input.*

Proof.

In order to do its work, a comparison sort must find one specific permutation out of $n!$ possibilities.

The proof, at last

Theorem

Any comparison sort must perform at least $\Theta(n \log(n))$ *comparisons between elements of the input.*

Proof.

In order to do its work, a comparison sort must find one specific permutation out of $n!$ possibilities. Each comparison we perform can eliminate at most half of the possible permutations at that step.

The proof, at last

Theorem

Any comparison sort must perform at least $\Theta(n \log(n))$ *comparisons between elements of the input.*

Proof.

In order to do its work, a comparison sort must find one specific permutation out of $n!$ possibilities. Each comparison we perform can eliminate at most half of the possible permutations at that step. Thus, any comparison sort must perform at least $\log_2(n!)$ comparisons to ensure correctness.

The proof, at last

Theorem

Any comparison sort must perform at least $\Theta(n \log(n))$ comparisons between elements of the input.

Proof.

In order to do its work, a comparison sort must find one specific permutation out of $n!$ possibilities. Each comparison we perform can eliminate at most half of the possible permutations at that step. Thus, any comparison sort must perform at least $\log_2(n!)$ comparisons to ensure correctness. By the Stirling approximation, this is $\Theta(n \log(n))$. $\square$

The proof, at last

Theorem

Any comparison sort must perform at least $\Theta(n \log(n))$ comparisons between elements of the input.

Proof.

In order to do its work, a comparison sort must find one specific permutation out of $n!$ possibilities. Each comparison we perform can eliminate at most half of the possible permutations at that step. Thus, any comparison sort must perform at least $\log_2(n!)$ comparisons to ensure correctness. By the Stirling approximation, this is $\Theta(n \log(n))$. $\square$

Corollary

Under our assumptions, no comparison sort with a time complexity better than $\Theta(n \log(n))$ (and thus, $O(n \log(n))$) can exist.

Is there nothing we can do?

Is there nothing we can do?

This is definitely a strong result that transcends any particular algorithm. However:

Is there nothing we can do?

This is definitely a strong result that transcends any particular algorithm. However:

- There are *practical* improvements we can make:

Is there nothing we can do?

This is definitely a strong result that transcends any particular algorithm. However:

- There are *practical* improvements we can make:
  - Not all data will be this bad!

Is there nothing we can do?

This is definitely a strong result that transcends any particular algorithm. However:

- There are *practical* improvements we can make:
  - Not all data will be this bad!
  - Not all $O(n \log(n))$ algorithms are born equal (consider timsort versus mergesort)

Is there nothing we can do?

This is definitely a strong result that transcends any particular algorithm. However:

- There are *practical* improvements we can make:
  - Not all data will be this bad!
  - Not all $O(n \log(n))$ algorithms are born equal (consider timsort versus mergesort)
- We usually know more about our data (numbers, limited number of unique items, strings, etc)

Is there nothing we can do?

This is definitely a strong result that transcends any particular algorithm. However:

- There are *practical* improvements we can make:
  - Not all data will be this bad!
  - Not all $O(n \log(n))$ algorithms are born equal (consider timsort versus mergesort)
- We usually know more about our data (numbers, limited number of unique items, strings, etc)
- RAM is not the most accurate model of modern computers:

Is there nothing we can do?

This is definitely a strong result that transcends any particular algorithm. However:

- There are *practical* improvements we can make:
  - Not all data will be this bad!
  - Not all $O(n \log(n))$ algorithms are born equal (consider timsort versus mergesort)
- We usually know more about our data (numbers, limited number of unique items, strings, etc)
- RAM is not the most accurate model of modern computers:
  - Modern machines are parallel — lots of different optimality points there!

Is there nothing we can do?

This is definitely a strong result that transcends any particular algorithm. However:

- There are *practical* improvements we can make:
  - Not all data will be this bad!
  - Not all $O(n\log(n))$ algorithms are born equal (consider timsort versus mergesort)
- We usually know more about our data (numbers, limited number of unique items, strings, etc)
- RAM is not the most accurate model of modern computers:
  - Modern machines are parallel — lots of different optimality points there!
  - Modern memory is *very* hierarchical — also lots of optimality points to consider

Is there nothing we can do?

This is definitely a strong result that transcends any particular algorithm. However:

- There are *practical* improvements we can make:
  - Not all data will be this bad!
  - Not all $O(n \log(n))$ algorithms are born equal (consider timsort versus mergesort)
- We usually know more about our data (numbers, limited number of unique items, strings, etc)
- RAM is not the most accurate model of modern computers:
  - Modern machines are parallel — lots of different optimality points there!
  - Modern memory is *very* hierarchical — also lots of optimality points to consider
- Data is not static or centralized anymore:

Is there nothing we can do?

This is definitely a strong result that transcends any particular algorithm. However:

- There are *practical* improvements we can make:
  - Not all data will be this bad!
  - Not all $O(n \log(n))$ algorithms are born equal (consider timsort versus mergesort)
- We usually know more about our data (numbers, limited number of unique items, strings, etc)
- RAM is not the most accurate model of modern computers:
  - Modern machines are parallel — lots of different optimality points there!
  - Modern memory is *very* hierarchical — also lots of optimality points to consider
- Data is not static or centralized anymore:
  - Interest in partial or online sorts

Is there nothing we can do?

This is definitely a strong result that transcends any particular algorithm. However:

- There are *practical* improvements we can make:
  - Not all data will be this bad!
  - Not all $O(n \log(n))$ algorithms are born equal (consider timsort versus mergesort)
- We usually know more about our data (numbers, limited number of unique items, strings, etc)
- RAM is not the most accurate model of modern computers:
  - Modern machines are parallel — lots of different optimality points there!
  - Modern memory is *very* hierarchical — also lots of optimality points to consider
- Data is not static or centralized anymore:
  - Interest in partial or online sorts
  - Fully-dynamic sorts (data can change in arbitrary ways)

Is there nothing we can do?

This is definitely a strong result that transcends any particular algorithm. However:

- There are *practical* improvements we can make:
  - Not all data will be this bad!
  - Not all $O(n \log(n))$ algorithms are born equal (consider timsort versus mergesort)
- We usually know more about our data (numbers, limited number of unique items, strings, etc)
- RAM is not the most accurate model of modern computers:
  - Modern machines are parallel — lots of different optimality points there!
  - Modern memory is *very* hierarchical — also lots of optimality points to consider
- Data is not static or centralized anymore:
  - Interest in partial or online sorts
  - Fully-dynamic sorts (data can change in arbitrary ways)
  - Distributed computing

The most important corollary

Corollary

Know your data, your tools and your problem, and you can work (or discover) performance wonders.

The most important corollary

Corollary

Know your data, your tools and your problem, and you can work (or discover) performance wonders.

Still work to be done in this area — for many years to come!

Questions?