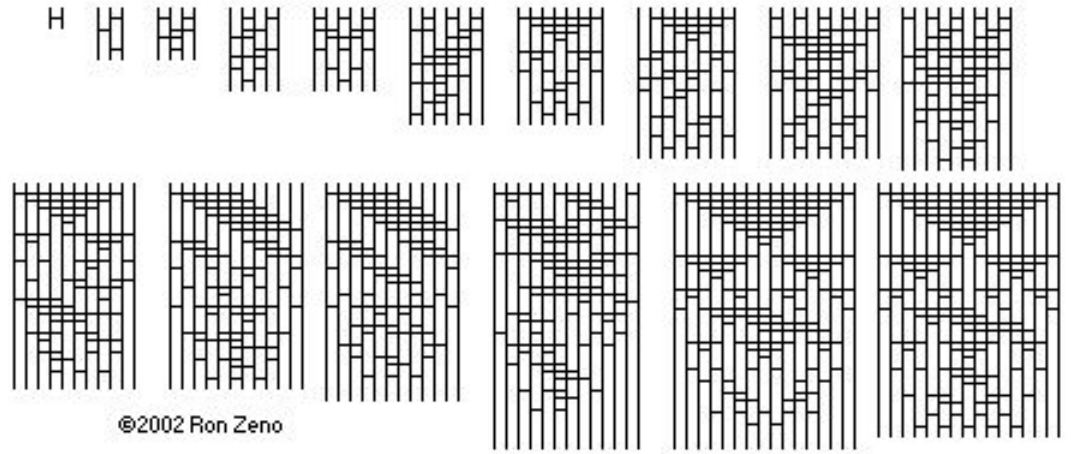


CS61B

Lecture 34: Sorting IV

- Sorting Summary
- Math Problems out of Nowhere
- Theoretical Bounds on Sorting



Other Desirable Sorting Properties: Stability

A sort is said to be stable if order of equivalent items is preserved.

`sort(studentRecords, BY_NAME);`

Bas	3
Fikriyya	4
Jana	3
Jouni	3
Lara	1
Nikolaj	4
Rosella	3
Sigurd	2

`sort(studentRecords, BY_SECTION);`

Lara	1
Sigurd	2
Bas	3
Jana	3
Jouni	3
Rosella	3
Fikriyya	4
Nikolaj	4

Equivalent items don't 'cross over' when being stably sorted.

Other Desirable Sorting Properties: Stability

A sort is said to be stable if order of equivalent items is preserved.

`sort(studentRecords, BY_NAME);`

Bas	3
Fikriyya	4
Jana	3
Jouni	3
Lara	1
Nikolaj	4
Rosella	3
Sigurd	2

`sort(studentRecords, BY_SECTION);`

Lara	1
Sigurd	2
Jouni	3
Rosella	3
Bas	3
Jana	3
Fikriyya	4
Nikolaj	4

Sorting instability can be really annoying! Wanted students listed alphabetically by section.

Arrays.sort

In Java, `Arrays.sort(someArray)` uses:

- Mergesort (specifically the TimSort variant) if `someArray` consists of Objects.
- Quicksort if `someArray` consists of primitives.

Why? See A level problems.

```
static void
```

```
sort(Object[] a)
```

Sorts the specified array of objects into ascending order, according to the **natural ordering** of its elements.

```
static void
```

```
sort(int[] a)
```

Sorts the specified array into ascending numerical order.

Arrays.sort

In Java, `Arrays.sort(someArray)` uses:

- Mergesort (specifically the TimSort variant) if `someArray` consists of Objects.
- Quicksort if `someArray` consists of primitives.

Why?

- When you are using a primitive value, they are the 'same'. A 4 is a 4. Unstable sort has no observable effect.
- By contrast, objects can have many properties, e.g. section and name, so equivalent items CAN be differentiated.

Sorting

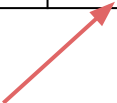
Sorting is a foundational problem.

- Obviously useful for putting things in order.
- But can also be used to solve other tasks, sometimes in non-trivial ways.
 - Sorting improves duplicate finding from a naive N^2 to $N \log N$.
 - Sorting improves 3SUM from a naive N^3 to N^2 .
- There are many ways to sort an array, each with its own interesting tradeoffs and algorithmic features.

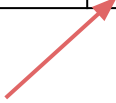
Today we'll discuss the fundamental nature of the sorting problem itself: How hard is it to sort?

Sorts Summary

	Memory	# Compares	Notes	Stable?
Heapsort	$\Theta(1)$	$\Theta(N \log N)$	Bad caching (61C)	No
Insertion	$\Theta(1)$	$\Theta(N^2)$	Best for almost sorted and $N < 15$	Yes
Mergesort	$\Theta(N)$	$\Theta(N \log N)$	Fastest stable sort	Yes
Quicksort LTHS	$\Theta(\log N)$	$\Theta(N \log N)$ expected	Fastest sort	No



This is due to the cost of tracking recursive calls by the computer, and is also an “expected” amount. The difference between $\log N$ and constant memory is trivial.



You can create a stable Quicksort. However, using unstable partitioning schemes (like Hoare partitioning) and using randomness to avoid bad pivots tend to yield better runtimes.

Math Problems out of Nowhere

A Math Problem out of Nowhere

Consider the functions $N!$ and $(N/2)^{N/2}$

Is $N! \in \Omega((N/2)^{N/2})$? Prove your answer.

- Recall that $\in \Omega$ can be informally be interpreted to mean \geq
- In other words, does factorial grow at least as quickly as $(N/2)^{N/2}$?

A Math Problem out of Nowhere

Consider the functions $N!$ and $(N/2)^{N/2}$

Is $N! \in \Omega((N/2)^{N/2})$? Prove your answer.

$10!$

- $10 * 9 * 8 * 7 * 6 * \dots * 1$

5^5

- $5 * 5 * 5 * 5 * 5$

$N! > (N/2)^{N/2}$, for large N , therefore $N! \in \Omega((N/2)^{N/2})$

Another Math Problem

Given: $N! > (N/2)^{N/2}$, which we used to prove our answer to the previous problem.

Show that $\log(N!) \in \Omega(N \log N)$.

- Recall: \log means an unspecified base.

Another Math Problem


Given that $N! > (N/2)^{N/2}$

Show that $\log(N!) \in \Omega(N \log N)$.

We have that $N! > (N/2)^{N/2}$

- Taking the log of both sides, we have that $\log(N!) > \log((N/2)^{N/2})$.
- Bringing down the exponent we have that $\log(N!) > N/2 \log(N/2)$.
- Discarding the unnecessary constant, we have $\log(N!) \in \Omega(N \log (N/2))$.
- From there, we have that $\log(N!) \in \Omega(N \log N)$.

Since $\log(N/2)$ is the same thing asymptotically as $\log(N)$.



In other words, $\log(N!)$ grows at least as quickly as $N \log N$.

Last Math Problem

In the previous problem, we showed that $\log(N!) \in \Omega(N \log N)$.

Now show that $N \log N \in \Omega(\log(N!))$.

Last Math Problem

Show that $N \log N \in \Omega(\log(N!))$

Proof:

- $\log(N!) = \log(N) + \log(N-1) + \log(N-2) + \dots + \log(1)$
- $N \log N = \log(N) + \log(N) + \log(N) + \dots \log(N)$
- Therefore $N \log N \in \Omega(\log(N!))$

Omega and Theta: yellkey.com/out

Given:

- $N \log N \in \Omega(\log(N!))$
- $\log(N!) \in \Omega(N \log N)$

Which of the following can we say?

- A. $N \log N \in \Theta(\log N!)$
- B. $\log N! \in \Theta(N \log N)$
- C. Both A and B
- D. Neither

Omega and Theta

Given:

- $N \log N \in \Omega(\log(N!))$

Informally: $N \log N \geq \log(N!)$

- $\log(N!) \in \Omega(N \log N)$

Informally: $\log(N!) \geq N \log N$

Which of the following can we say?

A. $N \log N \in \Theta(\log N!)$

B. $\log N! \in \Theta(N \log N)$

Informally: $N \log N = \log(N!)$

C. Both A and B

D. Neither

Summary

We've shown that $\log(N!) \in \Theta(N \log N)$.

- In other words, these two functions grow at the same rate asymptotically.

As for why we did this, we will see in a little while...

Theoretical Bounds on Sorting

Sorting

We have shown several sorts to require $\Theta(N \log N)$ worst case time.

- Can we build a better sorting algorithm?

By comparison sort, I mean that it uses e.g. the `compareTo` method in Java to make decisions.

Let the ultimate comparison sort (TUCS) be the asymptotically fastest possible comparison sorting algorithm, possibly yet to be discovered, and let $R(N)$ be its worst case runtime.

Give the best Ω and O bounds you can for $R(N)$.

It might seem strange to give Ω and O bounds for an algorithm whose details are completely unknown, but you can, I promise!

Sorting

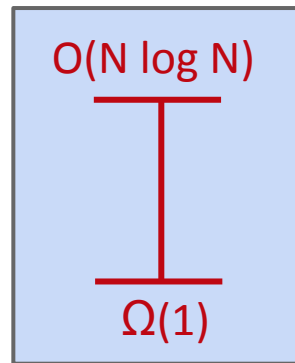
We have shown several sorts to require $\Theta(N \log N)$ worst case time.

- Can we build a better sorting algorithm?

By comparison sort, I mean that it uses e.g. the `compareTo` method in Java to make decisions.

Let the ultimate comparison sort (TUCS) be the asymptotically fastest possible comparison sorting algorithm, possibly yet to be discovered, and let $R(N)$ be its worst case runtime.

- Worst case run-time of TUCS, $R(N)$ is $O(N \log N)$.
 - Obvious: Mergesort is $\Theta(N \log N)$ so $R(N)$ can't be worse!
- Worst case run-time of TUCS, $R(N)$ is $\Omega(1)$.
 - Obvious: Problem doesn't get easier with N .
 - Can we make a stronger statement than $\Omega(1)$?

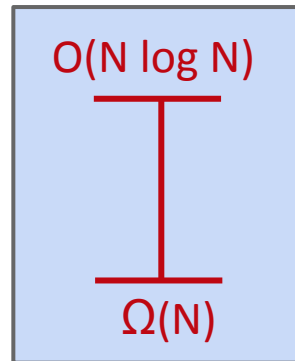


TUCS Worst
Case Θ Runtime

Sorting

Let TUCS be the asymptotically fastest possible comparison sorting algorithm, possibly yet to be discovered.

- Worst case run-time of TUCS, $R(N)$ is $O(N \log N)$. Why?
- Worst case run-time of TUCS, $R(N)$ is also $\Omega(N)$.
 - Have to at least look at every item.

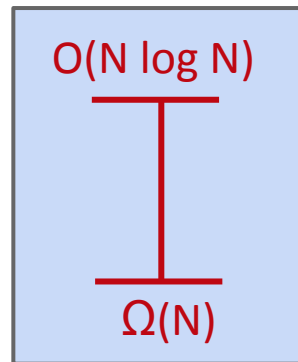


TUCS Worst
Case Θ Runtime

Sorting

We know that TUCS “lives” between N and $N \log N$.

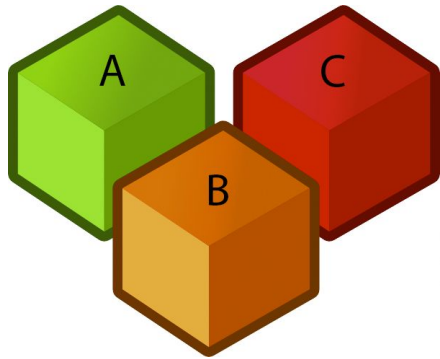
- Worst case asymptotic runtime of TUCS is between $\Theta(N)$ and $\Theta(N \log N)$.
- Can we make an even stronger statement on the lower bound?
 - With a clever argument, yes (as we'll see soon see).
 - Spoiler alert: It will turn out to be $\Omega(N \log N)$
 - This lower bound means that across the infinite space of all possible ideas that any human might ever have for sorting using sequential comparisons, NONE has a worst case runtime that is better than $\Theta(N \log N)$.



TUCS Worst
Case Θ Runtime

The Game of Puppy, Cat, Dog

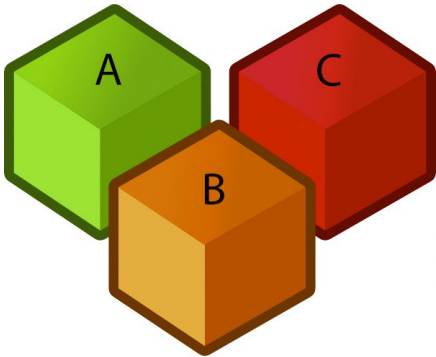
Suppose we have a puppy, a cat, and a dog, each in an opaque soundproof box labeled A, B, and C. We want to figure out which is which using a scale.



The Game of Puppy, Cat, Dog

Suppose we have a puppy, a cat, and a dog, each in an opaque soundproof box labeled A, B, and C. We want to figure out which is which using a scale.

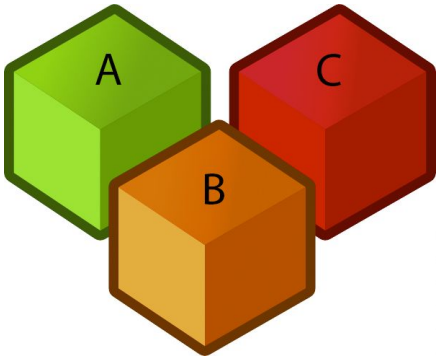
$a < b$	$b < c$		Which is which?
Yes	Yes		



The Game of Puppy, Cat, Dog

Suppose we have a puppy, a cat, and a dog, each in an opaque soundproof box labeled A, B, and C. We want to figure out which is which using a scale.

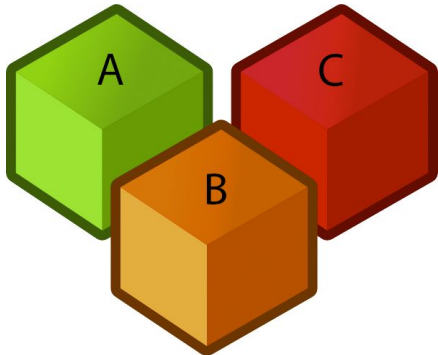
$a < b$	$b < c$		Which is which?
Yes	Yes		a: puppy, b: cat, c: dog (sorted order: abc)
No	No		



The Game of Puppy, Cat, Dog

Suppose we have a puppy, a cat, and a dog, each in an opaque soundproof box labeled A, B, and C. We want to figure out which is which using a scale.

a < b	b < c		Which is which?
Yes	Yes		a: puppy, b: cat, c: dog (sorted order: abc)
No	No		c: puppy, b: cat, a: dog (sorted order: cba)



The Game of Puppy, Cat, Dog: <http://yellkey.com/return>

Suppose we have a puppy, a cat, and a dog, each in an opaque soundproof box labeled A, B, and C. We want to figure out which is which using a scale.

a < b	b < c		Which is which?
Yes	Yes		a: puppy, b: cat, c: dog (sorted order: abc)
No	No		c: puppy, b: cat, a: dog (sorted order: cba)
Yes	No		

Which is which?

1. a: puppy, b: cat, c: dog (sorted order: abc)
2. a: puppy, c: cat, b: dog (sorted order: acb)
3. c: puppy, a: cat, b: dog (sorted order: cab)
4. c: puppy, b: cat, a: dog (sorted order: cba)

The Game of Puppy, Cat, Dog

Suppose we have a puppy, a cat, and a dog, each in an opaque soundproof box labeled A, B, and C. We want to figure out which is which using a scale.

a < b	b < c		Which is which?
Yes	Yes		a: puppy, b: cat, c: dog (sorted order: abc)
No	No		c: puppy, b: cat, a: dog (sorted order: cba)
Yes	No		

Which is which? How do we resolve the ambiguity?

1.

a: puppy, b: cat, c: dog (sorted order: abc)

2.

a: puppy, c: cat, b: dog (sorted order: acb)

3.

c: puppy, a: cat, b: dog (sorted order: cab)

4.

c: puppy, b: cat, a: dog (sorted order: cba)
- a?

c?

b
- c?

a?

b

The Game of Puppy, Cat, Dog

Suppose we have a puppy, a cat, and a dog, each in an opaque soundproof box labeled A, B, and C. We want to figure out which is which using a scale.

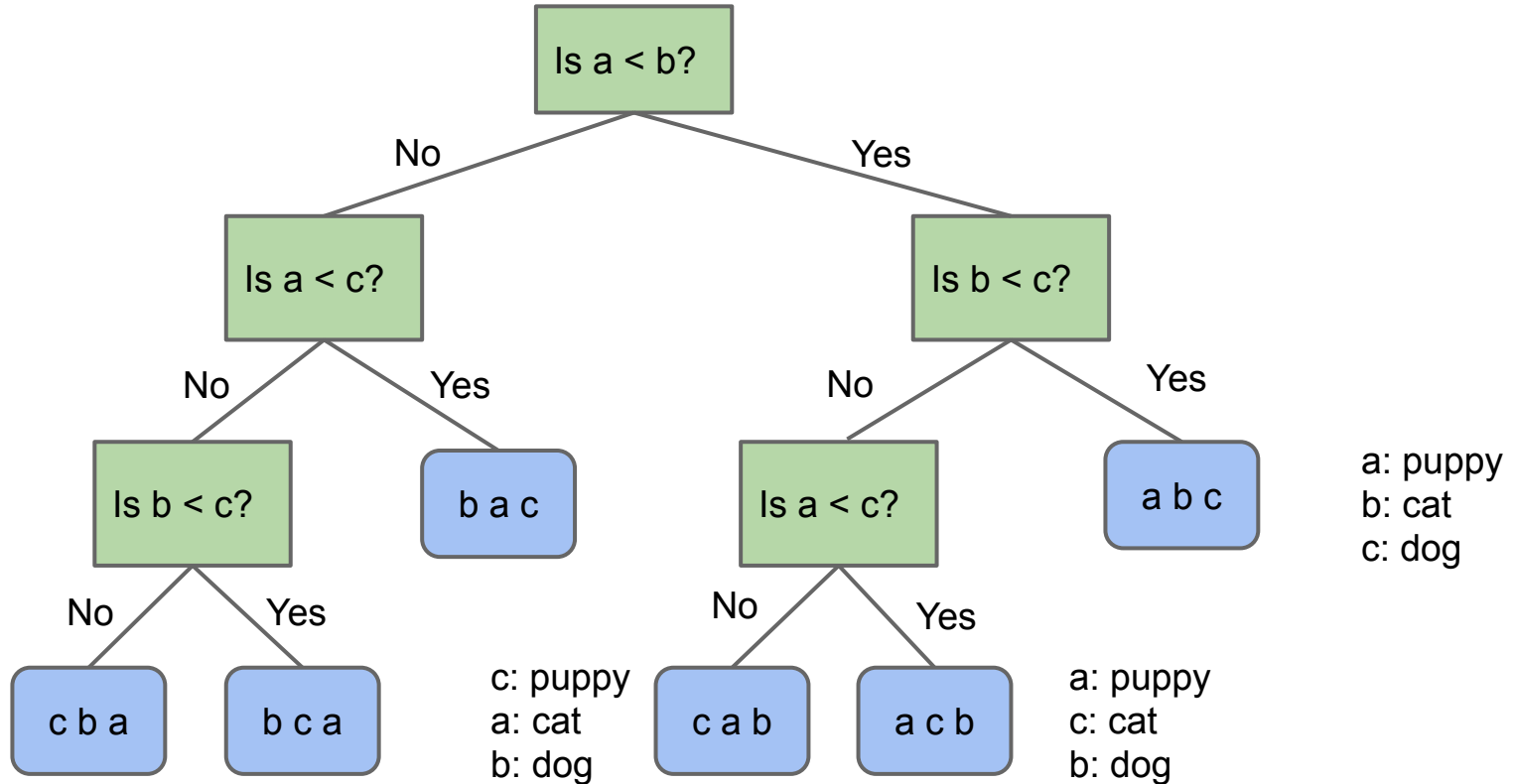
$a < b$	$b < c$	$a < c?$	Which is which?
Yes	Yes	N/A	a: puppy, b: cat, c: dog (sorted order: abc)
No	No	N/A	c: puppy, b: cat, a: dog (sorted order: cba)
Yes	No	Yes	a: puppy, c: cat, b: dog (sorted order: acb)

Which is which? How do we resolve the ambiguity? Ask if $a < c$.

1. a: puppy, b: cat, c: dog (sorted order: abc)
2. **a: puppy, c: cat, b: dog (sorted order: acb)**
3. **c: puppy, a: cat, b: dog (sorted order: cab)**
4. c: puppy, b: cat, a: dog (sorted order: cba)

Puppy, Cat, Dog - A Graphical Picture for $N = 3$

The full decision tree for puppy, cat, dog:



The Game of Puppy, Cat, Dog, yellkey.com/guess

How many questions would you need to ask to definitely solve the “puppy, cat, dog, walrus” problem?

- A. 3
- B. 4
- C. 5
- D. 6

The Game of Puppy, Cat, Dog

How many questions would you need to ask to definitely solve the “puppy, cat, dog, walrus” problem?

- A. 3
- B. 4
- C. 5**
- D. 6

Proof:

- If $N=4$, how many permutations? $4! = 24$
 - For $N=3$: $3!=6$
- So we need a binary tree with 24 leaves.
 - How many levels minimum? $\lg(24) = 4.58$, so 5 is the minimum.
 - \lg just means \log_2 (log base 2)

Generalized Puppy, Cat, Dog

How many questions would you need to ask to definitely solve the generalized “puppy, cat, dog” problem for N items?

- Give your answer in big Omega notation.

Hint: For $N=4$, we said the answer was 5 based on the following argument:

- Decision tree needs $4! = 24$ leaves.
- So we need $\lg(24)$ rounded up levels or 5.

Generalized Puppy, Cat, Dog

How many questions would you need to ask to definitely solve the generalized “puppy, cat, dog” problem for N items?

Answer: $\Omega(\log(N!))$

Hint: For N , we have the following argument:

- Decision tree needs $N!$ leaves.
- So we need $\lg(N!)$ rounded up levels, which is $\Omega(\log(N!))$

Generalizing Puppy, Cat, Dog

Finding an optimal decision tree for the generalized version of puppy, cat, dog (e.g. $N=6$: puppy, cat, dog, monkey, walrus, elephant) is an open problem in mathematics.

- (To my knowledge) Best known trees known for $N=1$ through 15 and $N=22$:
 - For more, see: <http://oeis.org/A036604>

Deriving a sequence of yes/no questions to identify puppy, cat, dog is hard. An alternate approach to solving the puppy, cat, dog problem:

- Sort the boxes using any generic sorting algorithm.
 - Leftmost box is puppy.
 - Middle box is cat.
 - Right box is dog.

Sorting, Puppies, Cats, and Dogs

Why do we care about these (no doubt adorable) critters?

A solution to the sorting problem also provides a solution to puppy, cat, dog.

- In other words, puppy, cat, dog **reduces** to sorting.
- Thus, any lower bound on difficulty of puppy, cat, dog must ALSO apply to sorting.

Physics analogy: Climbing a hill with your legs (CAHWYL) is one way to solve the problem of getting up a hill (GUAH).

- Any lower bound on energy to GUAH must also apply to CAHWYL.
- Example bound: Takes $m \cdot g \cdot h$ energy to climb hill, so using legs to climb the hill takes at least $m \cdot g \cdot h$ energy.

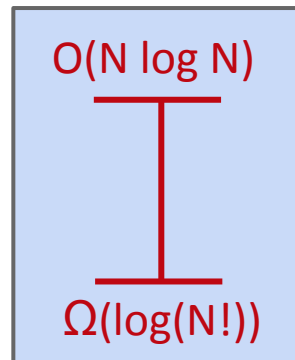
Sorting Lower Bound

We have a lower bound on puppy, cat, dog, namely that it takes $\Omega(\log(N!))$ comparisons to solve such a puzzle in the worst case.

Since sorting with comparisons can be used to solve puppy, cat, dog, then sorting also takes $\Omega(\log(N!))$ comparisons in the worst case.

Or in other words:

- Any sorting algorithm using comparisons, no matter how clever, must use at least $k = \lg(N!)$ compares to find the correct permutation. So even TUCS takes at least $\lg(N!)$ comparisons.
- $\lg(N!)$ is trivially $\Omega(\log(N!))$, so TUCS must take $\Omega(\log(N!))$ time.
- So, how does $\log(N!)$ compare to $N \log N$?



TUCS Worst
Case Θ Runtime


Another Math Problem

Earlier, we showed that $\log(N!) \in \Omega(N \log N)$ using the proof below.

- In other words, $\log(N!)$ grows at least as quickly as $N \log N$.

Proof from earlier that $\log(N!) \in \Omega(N \log N)$:

- We know that $N! \geq (N/2)^{N/2}$.
- Taking the log of both sides, we have that $\log(N!) \geq \log((N/2)^{N/2})$.
- Bringing down the exponent we have that $\log(N!) \geq N/2 \log(N/2)$.
- Discarding unnecessary constants, we have $\log(N!) \in \Omega(N \log N)$

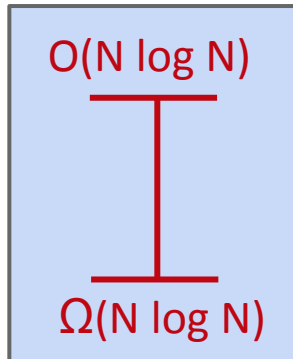


Recall that changing base is just multiplying by a constant.

The Sorting Lower Bound (Finally)

Since TUCS is $\Omega(\lg N!)$ and $\lg N!$ is $\Omega(N \log N)$, we have that **TUCS is $\Omega(N \log N)$** .

Any comparison based sort requires at least order $N \log N$ comparisons in its worst case.



TUCS Worst
Case Θ Runtime

The Sorting Lower Bound (Finally)

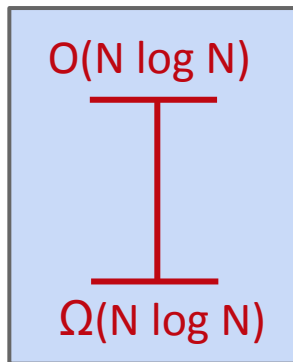
Since TUCS is $\Omega(\lg N!)$ and $\lg N!$ is $\Omega(N \log N)$, we have that **TUCS is $\Omega(N \log N)$** .

Any comparison based sort requires at least order $N \log N$ comparisons in its worst case.

Proof summary:

- Puppy, cat, dog is $\Omega(\lg N!)$, i.e. requires $\lg N!$ comparisons.
- TUCS can solve puppy, cat, dog, and thus takes $\Omega(\lg N!)$ compares.
- $\lg(N!)$ is $\Omega(N \log N)$
 - This was because $N!$ is $\Omega(N/2)^{N/2}$

Informally: $\text{TUCS} \geq \text{puppy, cat, dog} \geq \lg N! \geq N \log N$



TUCS Worst
Case Θ Runtime

Optimality

	Memory	# Compares	Notes	Stable?
Heapsort	$\Theta(1)$	$\Theta(N \log N)$	Bad caching (61C)	No
Insertion	$\Theta(1)$	$\Theta(N^2)$	Best for almost sorted and $N < 15$	Yes
Mergesort	$\Theta(N)$	$\Theta(N \log N)$	Fastest stable sort	Yes
Quicksort LTHS	$\Theta(\log N)$	$\Theta(N \log N)$ expected	Fastest sort	No

The punchline:

- Our best sorts have achieved absolute asymptotic optimality.
 - Mathematically impossible to sort using fewer comparisons.
 - Note: Randomized quicksort is only probabilistically optimal, but the probability is extremely high for even modest N . Are you worried about quantum teleportation? Then don't worry about Quicksort.

Next Time...

Today we proved that any sort that uses comparisons has runtime $\Omega(N \log N)$.

Next time we'll discuss how we can sort in $\Theta(N)$ time.

- Not impossible, just can't compare anything while we sort!

Sounds of Sorting (Fun)

Sounds of Sorting Algorithms (of 125 items)

Starts with selection sort: <https://www.youtube.com/watch?v=kPRA0W1kECg>

Insertion sort: <https://www.youtube.com/watch?v=kPRA0W1kECg&t=0m9s>

Quicksort: <https://www.youtube.com/watch?v=kPRA0W1kECg&t=0m38s>

Mergesort: <https://www.youtube.com/watch?v=kPRA0W1kECg&t=1m05s>

Heapsort: <https://www.youtube.com/watch?v=kPRA0W1kECg&t=1m28s>

LSD sort: <https://www.youtube.com/watch?v=kPRA0W1kECg&t=1m54s> [coming next Wednesday]

MSD sort: <https://www.youtube.com/watch?v=kPRA0W1kECg&t=2m10s> [coming next Wednesday]

Shell's sort: <https://www.youtube.com/watch?v=kPRA0W1kECg&t=3m37s> [bonus from last time]

Questions to ponder (later... after class):

- How many items for selection sort?
- Why does insertion sort take longer / more compares than selection sort?
- At what time stamp does the first partition complete for Quicksort?
- Could the size of the input to mergesort be a power of 2?
- What do the colors mean for heapsort?
- How many characters are in the alphabet used for the LSD sort problem?
- How many digits are in the keys used for the LSD sort problem?

Sounds of Sorting Algorithms

Starts with selection sort: <https://www.youtube.com/watch?v=kPRA0W1kECg>

Insertion sort: <https://www.youtube.com/watch?v=kPRA0W1kECg&t=0m9s>

Quicksort: <https://www.youtube.com/watch?v=kPRA0W1kECg&t=0m38s>

Mergesort: <https://www.youtube.com/watch?v=kPRA0W1kECg&t=1m05s>

Heapsort: <https://www.youtube.com/watch?v=kPRA0W1kECg&t=1m28s>

LSD sort: <https://www.youtube.com/watch?v=kPRA0W1kECg&t=1m54s>

MSD sort: <https://www.youtube.com/watch?v=kPRA0W1kECg&t=2m10s>

Shell's sort: <https://www.youtube.com/watch?v=kPRA0W1kECg&t=3m37s>

Questions to ponder (later... after class):

- How many items for selection sort?
- Why does insertion sort take longer / more compares than selection sort?
- At what time stamp does the first partition complete for Quicksort?
- Could the size of the input to mergesort be a power of 2?
- What do the colors mean for heapsort?

Sorting Implementations (Extra)

A Note on Implementations

Concrete implementations are nice for solidifying understanding.

- Implementing these yourself provides much deeper understanding than just reading my code.
- You are not responsible for the details of these specific implementations.
- Given enough time, you should be able to implement any of these sorts.

Utility Methods For Sorting

```
/** Returns true if v < w, false otherwise. */  
private static boolean less(Comparable v, Comparable w) {  
    return (v.compareTo(w) < 0);  
}
```

```
/** Swaps a[i] and a[j]. */  
private static void exch(Object[] a, int i, int j) {  
    Object swap = a[i];  
    a[i] = a[j];  
    a[j] = swap;  
}
```


Selection Sort

```
public static void selSort(Comparable[] a) {
    int N = a.length;
    for (int i = 0; i < N; i += 1) {
        int min = i;

        /** Find smallest item among unfixed items. */
        for (int j = i+1; j < N; j += 1) {
            if (less(a[j], a[min])) {
                min = j;
            }
        }

        exch(a, i, min);
    }
}
```

Key ideas: Among unfixed items, find minimum in $\Theta(N)$ time and swap to the front. Subproblem has size $N-1$. Total runtime is $N + N-1 + \dots + 1 = \Theta(N^2)$.

Insertion Sort

```
public static void insSort(Comparable[] a) {  
    int N = a.length;  
    for (int i = 0; i < N; i++) {  
  
        /* Swap item until it is in correct position. */  
        for (int j = i; j > 0; j -= 1) {  
  
            /* If left neighbor is less than me, stop. */  
            if less(a[j-1], a[j]) {  
                break;  
            }  
            exch(a, j, j-1);  
        }  
    }  
}
```

Key ideas: For each item (starting at leftmost), swap leftwards until in place. For item k , takes $\Theta(k)$ worst case time. Runtime is $1 + 2 + \dots + N = \Theta(N^2)$.

Selection and Insertion Sort Runtimes (Code Analysis)

Selection sort: Runtime is independent of input, always $\Theta(N^2)$.

- $\sim N^2/2$ compares and $\sim N^2/2$ exchanges. $\Theta(N^2)$ runtime.

Insertion sort: Runtime is strongly dependent on input. $\Omega(N)$, $O(N^2)$

- Best case (sorted): $\sim N$ compares, 0 exchanges: $\Theta(N)$
- Worst case (reverse sorted): $\sim N^2/2$ compares, $\sim N^2/2$ exchanges: $\Theta(N^2)$

```
for (int i = 0; i < N; i += 1) {
    int min = i;
    for (int j = i+1; j < N; j += 1) {
        if (less(a[j], a[min])) {
            min = j;
        }
    }
    exch(a, i, min);
}
```

```
for (int i = 0; i < N; i++) {
    for (int j = i; j > 0; j -= 1) {
        if (less(a[j-1], a[j])) {
            break;
        }
        exch(a, j, j-1);
    }
}
```

Mergesort (Merge Method)

```
/** Given sorted arrays a and b, return sorted array
 * containing all items from a and b. Can be optimized
 * to avoid creating new arrays for every merge. */
private static Comparable[] merge(Comparable[] a, Comparable[] b) {
    Comparable[] c = new Comparable[a.length + b.length];
    int i = 0, j = 0;
    for (int k = 0; k < c.length; k++) {
        if (i >= a.length) { c[k] = b[j]; j += 1; }
        else if (j >= b.length) { c[k] = a[i]; i += 1; }
        else if (less(b[j], a[i])) { c[k] = a[j]; j += 1; }
        else { c[k] = b[i]; i += 1; }
    }
    return c;
}
```

Mergesort

```
/** Mergesort. Can be optimized to avoid creation of subarrays. */
public static Comparable[] mergesort(Comparable[] input) {
    int N = input.length;
    if (N <= 1) return input;
    Comparable[] a = new Comparable[N/2];
    Comparable[] b = new Comparable[N - N/2];
    for (int i = 0; i < a.length; i += 1) a[i] = input[i];
    for (int i = 0; i < b.length; i += 1) b[i] = input[i + N/2];
    return merge(mergesort(a), mergesort(b));
}
```

Key ideas: Each merge costs $\Theta(N)$ time and $\Theta(N)$ space, and generates two subproblems of size $N/2$. At level L of the sort, there are 2^L subproblems of size $N/2^L$. Since $L = \Theta(\log N)$, runtime is $\Theta(N \log N)$.

Interview Question

```
/** Mergesort. Can be optimized to avoid creation of subarrays. */  
public static Comparable[] mergesort(Comparable[] input) {  
    int N = input.length;  
    if (N <= 1) return input;  
    Comparable[] a = new Comparable[N/2];  
    Comparable[] b = new Comparable[N - N/2];  
    for (int i = 0; i < a.length; i += 1) a[i] = input[i];  
    for (int i = 0; i < b.length; i += 1) b[i] = input[i + N/2];  
    return merge(mergesort(a), mergesort(b));  
}
```

How can the above mergesort implementation be improved?

- Try and avoid making copies a and b, by adding parameters to the merge routine. `merge(input, 0, 5, 6, 10);`
- Use a different for small N: Like maybe insertion sort. Industrial strength mergesorts, use insertion sort for $N < 15$.

Interview Question

```
/** Mergesort. Can be optimized to avoid creation of subarrays. */
public static Comparable[] mergesort(Comparable[] input) {
    int N = input.length;
    if (N <= 1) return input;
    Comparable[] a = new Comparable[N/2];
    Comparable[] b = new Comparable[N - N/2];
    for (int i = 0; i < a.length; i += 1) a[i] = input[i];
    for (int i = 0; i < b.length; i += 1) b[i] = input[i + N/2];
    return merge(mergesort(a), mergesort(b));
}
```

How can the above mergesort implementation be improved?

Heapsort With Separate PQ

```
/** Uses a MaxPQ to do the sorting. Requires  $\Theta(N)$  space. */
public static void lameHeapsort(Comparable[] items) {
    MaxPQ<Comparable> maxPQ = new MaxPQ<Comparable>();
    for (Comparable c : items) {
        maxPQ.insert(c);
    }
    /** Repeatedly remove largest item and put at end of array.
        Using a MinPQ is more intuitive, but a MaxPQ can be
        adapted to use no extra space (next slide). */
    for (int i = items.length - 1; i >= 0; i -= 1) {
        items[i] = maxPQ.removeLargest();
    }
}
```

Key ideas: Create a max heap of all items [$\Theta(N \log N)$], then delete max N times [$\Theta(\log N)$ per delete]. Requires $\Theta(N)$ space.

In-Place Heapsort (with root in position 0).

```
/** Sorts the given array by first heapifying, then removing
 * each item from the max heap, one-by-one. */
public static void sort(Comparable[] pq) {
    int N = pq.length;
    /* Sink in reverse level order. Can be optimized
     to exclude the bottom level. */
    for (int k = N; k >= 0; k -= 1) {
        sink(pq, k, N);
    }
    while (N > 1) {
        exch(pq, 0, N); // swap root and last item
        N -= 1;         // mark deleted item as off limits
        sink(pq, 0, N); // sink the root
    }
}
```

Key ideas: Max-Heapify [$\Theta(N)$], then delete max N times [$\Theta(\log N)$ per delete]

In-Place Heapsort Sink Operation (with root in position 0).

```
/** Given item in position pq[cur], repeatedly swaps the item
 * with its largest child if necessary for heap property. */
private static void sink(Comparable[] pq, int cur, int N) {
    /* Repeatedly sink until no children are left. */
    while (2 * cur <= N) {
        int left = 2 * cur + 1; // 0-based array
        int right = left + 1;
        int largerChild = left;
        /* If right child exists and is larger. */
        if (right >= N && less(pq[left], pq[right])) {
            largerChild = right;
        }
        if (!less(pq[cur], pq[largerChild])) {
            break;
        }
        exch(pq, cur, largerChild);
        cur = largerChild;
    }
}
```

Citations

Title image: <http://www.angelfire.com/blog/ronz/Articles/999SortingNetworksReferen.html>

Cubes: http://www.clker.com/cliparts/6/9/3/2/1197122947130754155jean_victor_balin_Cubes.svg.hi.png

Scale: <http://www.clipartbest.com/cliparts/94T/bAe/94TbAejig.png>

Puppy: http://assets.nydailynews.com/polopoly_fs/1.1245686!/img/httpImage/image.jpg_gen/derivatives/article_970/afp-cute-puppy.jpg

Cat: <http://animalia-life.com/cat.html>

Dog: <http://animalia-life.com/dogs.html>