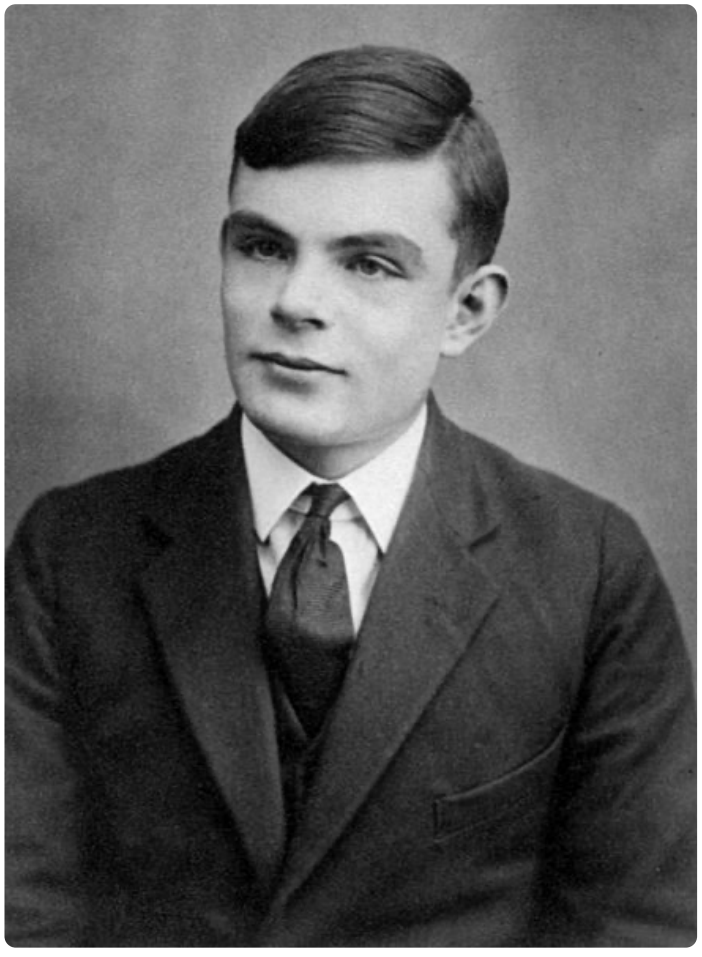
Ada Lovelace



Alan Turing

welcome to Graduate Algorithms

fundamentalalgorithms.com

one big .pdf w/ everything

Last compiled January 9, 2023 at 11:30am.

Graduate Algorithms, CS580, Spring 2023

Lectures: 5:30 to 6:45 PM, on Monday and Wednesday, in ME1130.

Staff:	Email (@purdue.edu)	Office hours
Kent Quanrud	krq	Wed. 3PM at LWSN 1211
Richard Li (TA)	richzli	TBD
Xinyu Luo (TA)	luo446	TBD

Please see the syllabus (page 502) for more information.

Homework

- Homework 0 due January 19.

Class schedule[1]

1. *January 9.* Searching and sorting, including binary search, merge sort, and lower bounds for sorting (chapter 1).

Please read the syllabus - will ask for questions next class

welcome to Randomized Algorithms

fundamentalalgorithms.com

Last compiled January 9, 2023 at 11:30am.

Graduate Algorithms, CS580, Spring 2023

Lectures: 5:30 to 6:45 PM, on Monday and Wednesday, in ME1130.

Staff:	Email (@purdue.edu)	Office hours
Kent Quanrud	krq	Wed. 3PM at LWSN 1211
Richard Li (TA)	richzli	TBD
Xinyu Luo (TA)	luo446	TBD

Please see the syllabus (page 502) for more information.

Homework

- Homework 0 due January 19.

Class schedule[1]

1. *January 9.* Searching and sorting, including binary search, merge sort, and lower bounds for sorting (chapter 1).

Please read the syllabus

- will ask for questions next class

Highlights

Awesome TA's

2 midterms, final

weekly homework

drop lowest 20% of HW scores

late/resubmit HW policy

Part I.

You already know TCS

- counting
- over/under

Counting

Counting

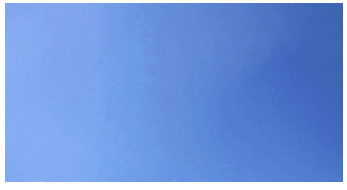
 = 23

RHS much more efficient than LHS

"unary"

Decimal notation: 10^k #'s w/ k digits

Combinatorial explosion!



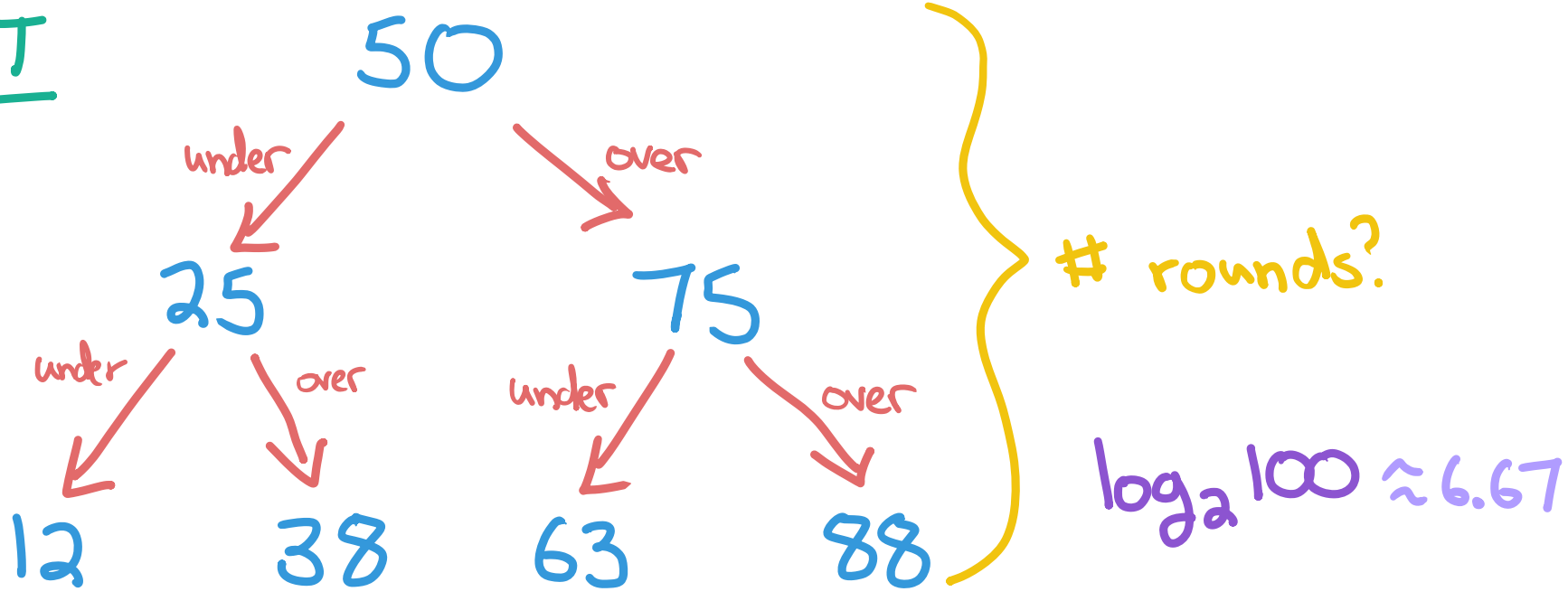
Over/under

guess # between 1 and 100

Slow algo:

for $i = 1, 2, \ldots, 100$, guess i

FAST



$\log_2 100 \approx 6.67$

$\Rightarrow$ 6 rounds

exponentials

$$2^n = \underbrace{2 \cdot 2 \cdot 2 \cdot \cdots \cdot 2}_{n \text{ times}}$$

logarithms

"logarithm (base 2) of n"

$\log_2 n$ s.t. $2^{\log_2 n} = n$

also: $\log_e n$ (aka $\ln$)

$\log_{10} n$

$\log_b n$ for some $b > 1$

Google

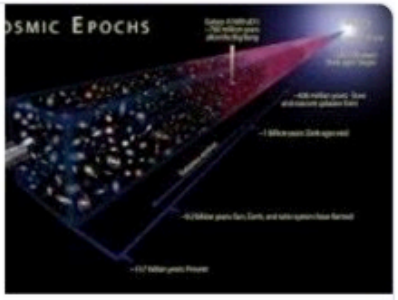
how many atoms in the universe

Sign in

All Images News Shopping Videos More Settings Tools

About 26,300,000 results (0.49 seconds)

10^{82} atoms



At this level, it is estimated that the there are between 10^{78} to 10^{82} **atoms** in the known, observable **universe**. In layman's terms, that works out to between ten quadrillion vigintillion and one-hundred thousand quadrillion vigintillion **atoms**. Jul 30, 2009

How Many Atoms Are There in the Universe? - Universe Today

https://www.universetoday.com › atoms-in-the-universe

$$\log_2(\#\text{atoms}) = \log_2(10^{82}) = 82\log_2(10)$$

$$\approx 3.32$$

$$\approx 272.398$$

$$\log_2(\log_2(\#\text{atoms}))$$

$$\approx \log_2(272.4)$$

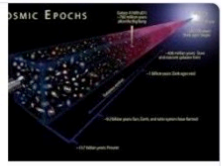
$$\approx 8.\sim$$

$$(2^8 = 256)$$

how many atoms in the universe

All | Images | News | Shopping | Videos | More | Settings | Tools

About 26,300,000 results (0.49 seconds)



10^{82} atoms

At this level, it is estimated that the there are between 10^{78} to 10^{82} **atoms** in the known, observable **universe**. In layman's terms, that works out to between ten quadrillion vigintillion and one-hundred thousand quadrillion vigintillion **atoms**. Jul 30, 2009

How Many Atoms Are There in the Universe? - Universe Today

https://www.universetoday.com › atoms-in-the-universe

$$\log_2(\#\text{atoms})$$
$$= \log_2(10^{82})$$
$$= 82\log_2(10)$$
$$\approx 272.398$$

TCS is about distinguishing

2^n,
combinatorial
explosion

vs

$\log_2 n$,
binary search

Part II: Sorting

INEFFECTIVE SORTS

```
DEFINE HALFHEARTEDMERGESORT(LIST):
    IF LENGTH(LIST) < 2:
        RETURN LIST
    PIVOT = INT(LENGTH(LIST) / 2)
    A = HALFHEARTEDMERGESORT(LIST[:PIVOT])
    B = HALFHEARTEDMERGESORT(LIST[PIVOT:])
    // UMMMMM
    RETURN [A, B] // HERE. SORRY.
```

```
DEFINE FASTBOGOSORT(LIST):
    // AN OPTIMIZED BOGOSORT
    // RUNS IN O(N LOG N)
    FOR N FROM 1 TO LOG(LENGTH(LIST)):
        SHUFFLE(LIST):
        IF ISSORTED(LIST):
            RETURN LIST
    RETURN "KERNEL PAGE FAULT (ERROR CODE: 2)"
```

```
DEFINE JOBINTERVIEWQUICKSORT(LIST):
    OK SO YOU CHOOSE A PIVOT
    THEN DIVIDE THE LIST IN HALF
    FOR EACH HALF:
        CHECK TO SEE IF IT'S SORTED
            NO, WAIT, IT DOESN'T MATTER
        COMPARE EACH ELEMENT TO THE PIVOT
            THE BIGGER ONES GO IN A NEW LIST
            THE EQUAL ONES GO INTO, UH
            THE SECOND LIST FROM BEFORE
        HANG ON, LET ME NAME THE LISTS
            THIS IS LIST A
            THE NEW ONE IS LIST B
        PUT THE BIG ONES INTO LIST B
        NOW TAKE THE SECOND LIST
            CALL IT LIST, UH, A2
        WHICH ONE WAS THE PIVOT IN?
        SCRATCH ALL THAT
        IT JUST RECURSIVELY CALLS ITSELF
        UNTIL BOTH LISTS ARE EMPTY
            RIGHT?
        NOT EMPTY, BUT YOU KNOW WHAT I MEAN
    AM I ALLOWED TO USE THE STANDARD LIBRARIES?
```

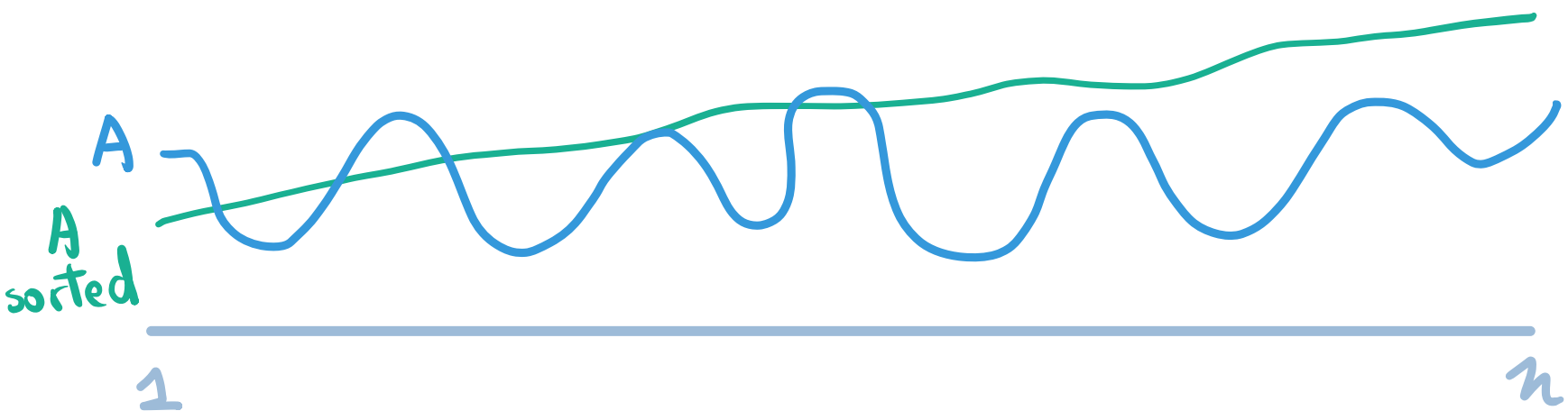
```
DEFINE PANICSORT(LIST):
    IF ISSORTED(LIST):
        RETURN LIST
    FOR N FROM 1 TO 10000:
        PIVOT = RANDOM(0, LENGTH(LIST))
        LIST = LIST[PIVOT:] + LIST[:PIVOT]
        IF ISSORTED(LIST):
            RETURN LIST
    IF ISSORTED(LIST):
        RETURN LIST:
    IF ISSORTED(LIST):  //THIS CAN'T BE HAPPENING
        RETURN LIST
    IF ISSORTED(LIST): // COME ON COME ON
        RETURN LIST
    // OH JEEZ
    // I'M GONNA BE IN SO MUCH TROUBLE
    LIST = [ ]
    SYSTEM("SHUTDOWN -H +5")
    SYSTEM("RM -RF ./")
    SYSTEM("RM -RF ~/*")
    SYSTEM("RM -RF /")
    SYSTEM("RD /S /Q C:\*")  //PORTABILITY
    RETURN [1, 2, 3, 4, 5]
```

Input: n numbers

5, 1, 8, 9, 2, 6, 7, 4, 11, 7, 9

Goal: sorted list

1, 2, 4, 5, 7, 7, 8, 9, 9, 11



Human sort

5, 1, 8, 9, 2, 6, 7, 4, 11, 7, 9

- what comes first?
- what comes second?

1 2

5, 1, 8, 9, 2, 6, 7, 4, 11, 7, 9

- what comes first?
- what comes second?

builds sorted list from beginning to end

each time, we scan the list of n inputs

n iterations

$\times$ n items scanned per iteration

$\approx n^2$ time

Big-O

n	iterations
$\times n$	items scanned per iteration
n^2	time

"$O(n^2)$": there exists constants $N, c > 0$ s.t.

for all $n > N$, the algorithm terminates in

at most cn^2 steps

WTF "step"?! varies by computer, language, ~

low-level details only effect the constant

$O(n)$ hides the constant so we can develop a

general THEORY OF ALGORITHMS

Human sort: $O(n^2)$ time

(aka selection sort)

Better?

n^2 = each # is compared to every other #

can we sort all #'s without comparing every pair?

do we really need all comparisons?

Slightly different perspective

given array $A[1 \sim n]$ that is supposedly sorted

how long does it take to verify A is sorted?

$a_1 \overset{\leq ?}{\wedge} a_2 \overset{\leq ?}{\wedge} a_3 \quad a_4 \quad a_5 \quad a_6 \quad \sim$

$O(n)$

sorted? (A[1~n])
① for i=1,...,n-1
  ⓐ if A[i]>A[i+1]
    ① return False
② return True

Human sort: $O(n^2)$ time

(aka selection sort)

Verifying a sort: $O(n)$

Better?

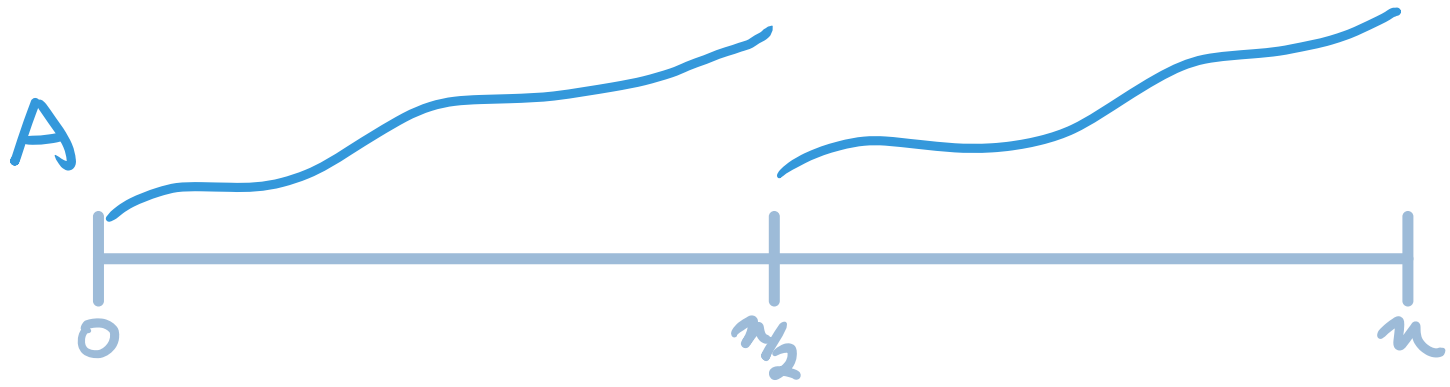
n^2 = each # is compared to every other #

can we sort all #'s without comparing every pair?

do we really need all comparisons?

Simple case

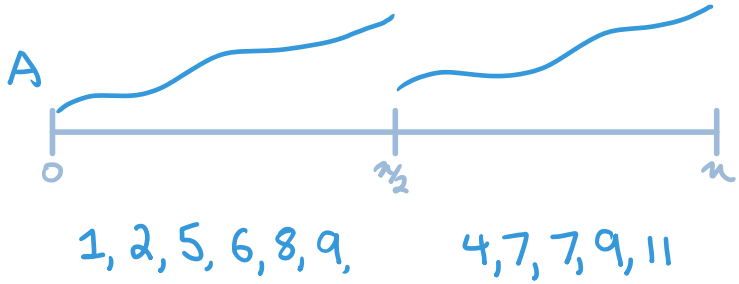
Suppose first & second halves are already sorted



1, 2, 5, 6, 8, 9, 4, 7, 7, 9, 11

Simple case

Suppose first & second halves are already sorted



merge($A[1..m], B[1..n]$)

*/ Given two sorted lists A and B, produces the combined lists in sorted order. */

1. Allocate a new array $C[1..m+n]$, let $i = 1$, and let $j = 1$.
2. While $i \leq m$ and $j \leq n$:
   A. If $A[i] \leq B[j]$:
      1. $C[i+j-1] \leftarrow A[i]$ and $i \leftarrow i+1$.
   B. Else ($B[j] < A[i]$):
      1. $C[i+j-1] \leftarrow B[j]$ and $j \leftarrow j+1$.
3. While $i \leq m$:
   A. $C[i+j-1] \leftarrow A[i]$ and $i \leftarrow i+1$.
4. While $j \leq n$:
   A. $C[i+j-1] \leftarrow B_2[j]$ and $j \leftarrow j+1$.
5. Return C.

merge($A[1..m], B[1..n]$)

/ Given two sorted lists A and B, produces the combined lists in sorted order.*

1. Allocate a new array $C[1..m+n]$, let $i = 1$, and let $j = 1$.
2. While $i \le m$ and $j \le n$:
   A. If $A[i] \le B[j]$:
      1. $C[i+j-1] \leftarrow A[i]$ and $i \leftarrow i+1$.
   B. Else ($B[j] < A[i]$):
      1. $C[i+j-1] \leftarrow B[j]$ and $j \leftarrow j+1$.
3. While $i \le m$:
   A. $C[i+j-1] \leftarrow A[i]$ and $i \leftarrow i+1$.
4. While $j \le n$:
   A. $C[i+j-1] \leftarrow B_2[j]$ and $j \leftarrow j+1$.
5. Return C.

Divide-and-conquer

1. Sort first half recursively
2. Sort second half recursively
3. merge sorted halves together

Designing a recursive algorithm

Divide-and-conquer

1. Sort first half recursively
2. Sort second half recursively
3. merge sorted halves together

① Recursive specification

`merge-sort`$(A[1..n])$: Given an array $A[1..n]$ of comparable elements, returns an array consisting of the elements of A sorted in increasing order.

- declares input/output of algo
- "contract" algo must fulfill
- induction hypothesis for proof of correctness

Designing a recursive algorithm

Divide-and-conquer
1. Sort first half recursively
2. Sort second half recursively
3. merge sorted halves together

① Recursive specification

`merge-sort`$(A[1..n])$: Given an array $A[1..n]$ of comparable elements, returns an array consisting of the elements of A sorted in increasing order.

② Implement spec

- declares input/output of algo
- "contract" algo must fulfill
- induction hypothesis for correctness

`merge-sort`$(A[1..n])$

/ In the base case, the list is so short there is nothing to do. */*

1. If $n \leq 1$ then return A.

Base case

/ In the general case, we divide the input in half and recursively sort each half. */*

2. Let $m = \lfloor \frac{n}{2} \rfloor$.
3. $B_1[1..m] \leftarrow$ merge-sort$(A[1..m])$.
4. $B_2[1..n-m] \leftarrow$ merge-sort$(A[m+1..n])$.

satisfies recursive spec by induction on n

/ Now we merge the two sorted halves in linear time. */*

5. return `merge`(B_1, B_2).

Designing a recursive algorithm

Divide-and-conquer

1. Sort first half recursively
2. Sort second half recursively
3. merge sorted halves together

① Recursive specification

merge-sort($A[1..n]$): Given an array $A[1..n]$ of comparable elements, returns an array consisting of the elements of A sorted in increasing order.

- declares input/output of algo
- "contract" algo must fulfill
- induction hypothesis for correctness

② Implement spec

merge-sort($A[1..n]$)

/* *In the base case, the list is so short there is nothing to do.* */

1. If $n \leq 1$ then return A. ← Base case

/* *In the general case, we divide the input in half and recursively sort each half.* */

2. Let $m = \lfloor \frac{n}{2} \rfloor$.
3. $B_1[1..m] \leftarrow$ merge-sort($A[1..m]$). ← satisfies recursive spec by induction on n
4. $B_2[1..n-m] \leftarrow$ merge-sort($A[m+1..n]$). ← satisfies recursive spec by induction on n

/* *Now we merge the two sorted halves in linear time.* */

5. return merge(B_1, B_2).

③ Prove correctness

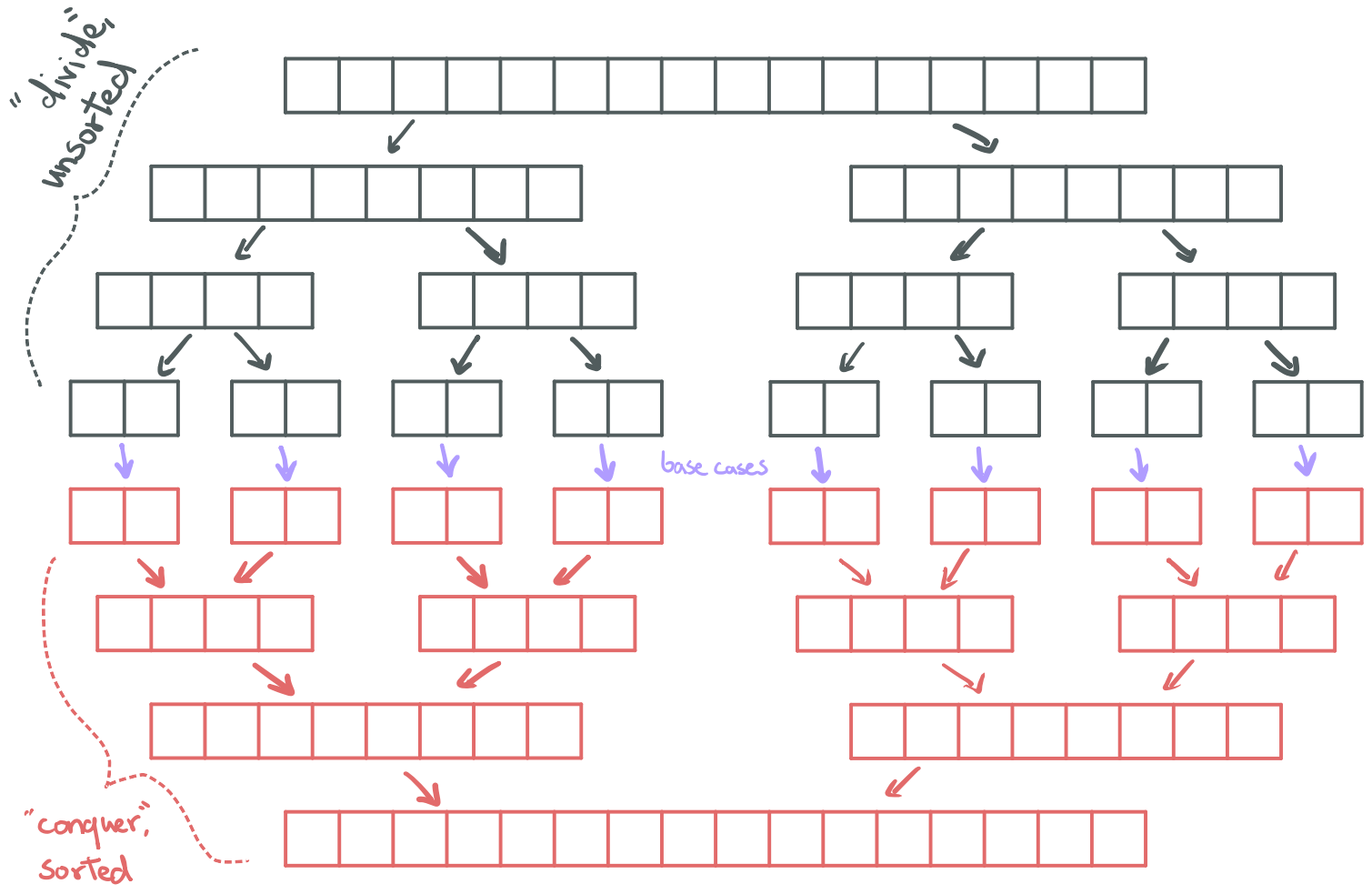
argue algo satisfies recursive spec for all n by induction on n

Divide-and-conquer

1. Sort first half recursively
2. Sort second half recursively
3. merge sorted halves together

`merge-sort`$(A[1..n])$: Given an array $A[1..n]$ of comparable elements, returns an array consisting of the elements of A sorted in increasing order.

```
merge-sort(A[1..n])
/* In the base case, the list is so short there is nothing to do. */
1. If n ≤ 1 then return A.                                   ← Base case
/* In the general case, we divide the input in half and recursively sort each half. */
2. Let m = ⌊n/2⌋.
3. B1[1..m] ← merge-sort(A[1..m]).                           ← satisfies recursive spec by induction on n
4. B2[1..n − m] ← merge-sort(A[m + 1..n]).                   ← satisfies recursive spec by induction on n
/* Now we merge the two sorted halves in linear time. */
5. return merge(B1, B2).
```



Running time analysis

$T(n)$ = time for input of size n

$T(1) = T(0) = 1$

$n > 1$:

$$T(n) = T(\lceil \tfrac{n}{2} \rceil) + T(n - \lceil \tfrac{n}{2} \rceil) + O(n)$$

$$\overset{*}{\approx} 2T(\tfrac{n}{2}) + O(n)$$

* see notes for justification

Divide-and-conquer

1. Sort first half recursively
2. Sort second half recursively
3. merge sorted halves together

`merge-sort`$(A[1..n])$: Given an array $A[1..n]$ of comparable elements, returns an array consisting of the elements of A sorted in increasing order.

`merge-sort`$(A[1..n])$

/ In the base case, the list is so short there is nothing to do. */*

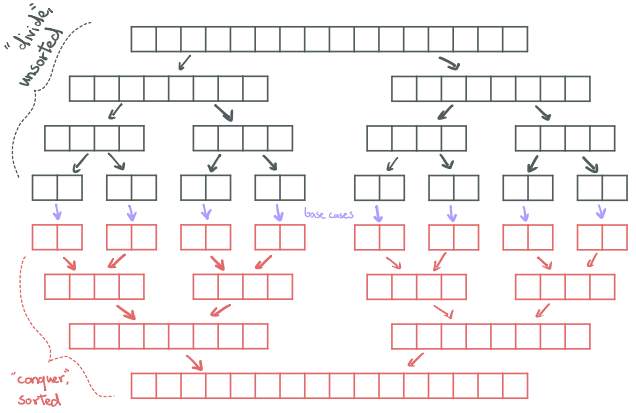
1. If $n \leq 1$ then return A. ← Base case

/ In the general case, we divide the input in half and recursively sort each half. */*

2. Let $m = \lfloor \frac{n}{2} \rfloor$.
3. $B_1[1..m] \leftarrow$ merge-sort$(A[1..m])$. ← satisfies recursive spec by induction on n
4. $B_2[1..n-m] \leftarrow$ merge-sort$(A[m+1..n])$. ← satisfies recursive spec by induction on n

/ Now we merge the two sorted halves in linear time. */*

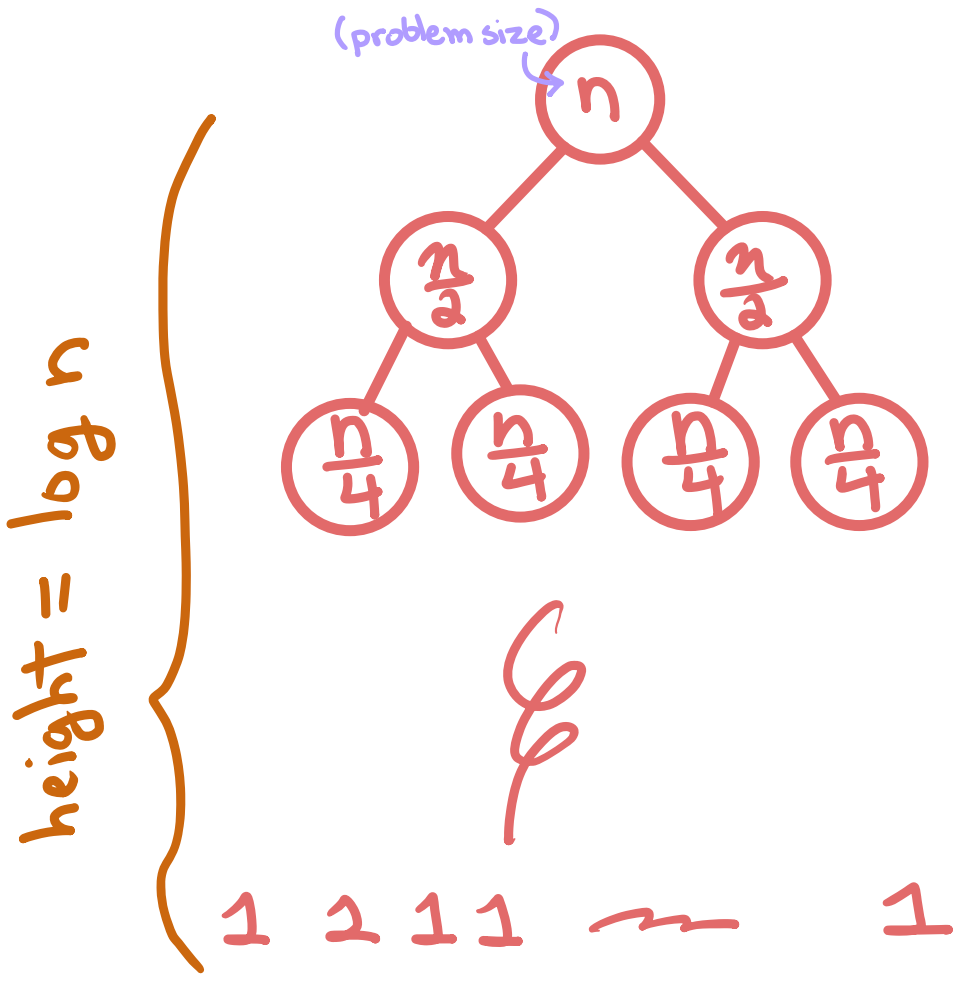
5. return `merge`(B_1, B_2).



Recursion tree

$T(1) = T(0) = 1$

$T(n) = 2T(\frac{n}{2}) + O(n)$



Work

n

$2 \cdot (\frac{n}{2}) = n$

$4 \cdot (\frac{n}{4}) = n$

$\vdots$

$2^k \cdot \frac{n}{2^k} = n$

$+ \; n \cdot 1 = n$

$n \times \log(n)$

$\Rightarrow T(n) = O(n \log n)$

Lower Bounds for Sorting

Lower Bounds for Sorting

merge-sort: $O(n \log n)$. better? optimal?

Goal: prove lower bound for all sorting algorithms

e.g. prove any correct algorithm has to take at least

$\Omega(f(n))$ time

"big-Omega"

(like $O(\cdot)$ but for lower bounds)

Comparison model

- Input: $x_1, \ldots, x_n$ comparable
- only info is via comparison queries

"is $x_i < x_j$?"

Goal lower bound # comparison Q's

Lower Bounds for Sorting

merge-sort: $O(n \log n)$. better? optimal?

Goal: prove lower bound for all sorting algorithms

eg prove any correct algorithm has to take at least $\Omega(f(n))$ time

"big-Omega"

(like $O(\cdot)$ but for lower bounds)

Comparison model

- Input: $x_1, \dots, x_n$ comparable
- only info via comparison queries "is $x_i < x_j$?"

Goal: lower bound # comparisons

Suppose algo makes k queries and outputs list $(x_{i_1}, x_{i_2}, \dots, x_{i_n})$

1. $x_{i_1} < x_{j_1}$?
2. $x_{i_2} < x_{j_2}$?
3. $x_{i_3} < x_{j_3}$?
4. $x_{i_4} < x_{j_4}$?

{

k. $x_{i_k} < x_{j_k}$?

$\Rightarrow (x_{i_1}, x_{i_2}, x_{i_3}, \dots, x_{i_n})$

- output is a function of k queries
- each query has 2 outcomes (YES/NO)

$\Rightarrow$ at most 2^k possible outputs

meanwhile...

$n!$ possible lists

$\Rightarrow 2^k \geq n!$ just to be able to output all lists

$k \geq \log(n!)$

$n! = n \cdot (n-1) \cdot (n-2) \cdots 2 \cdot 1$

$\geq \left(\frac{n}{2}\right)^{n/2}$

$k \geq \log\left(\left(\frac{n}{2}\right)^{n/2}\right) = \frac{n}{2}\log\left(\frac{n}{2}\right) = \Omega(n \log n)$

Comparison model

- Input: $x_1, \cdots, x_n$ comparable
- only info via comparison queries "is $x_i < x_j$?"

Goal lower bound # comparisons

Suppose algo makes k queries and outputs list $(x_{i_1}, x_{i_2}, \cdots, x_{i_n})$

1. $x_{i_1} < x_{j_1}$?
2. $x_{i_2} < x_{j_2}$?
3. $x_{i_3} < x_{j_3}$?
4. $x_{i_4} < x_{j_4}$?

⋮

k. $x_{i_k} < x_{j_k}$?

$\Rightarrow (x_{i_1}, x_{i_2}, x_{i_3}, \cdots, x_{i_n})$

- output is a function of k queries
- each query has 2 outcomes (YES/NO)

$\Rightarrow$ at most 2^k possible outputs

Theorem In the comparison (only) model, any (correct, deterministic) sorting algorithm makes

$$\Omega(n \log n) \text{ comparisons}$$

in the worst case.

Upper bound	Lower bound
$O(n \log n)$ (mergesort)	$\Omega(n \log n)$

in general (any lower bound) $\leq$ (any upper bound)

here we also have

(an upper bound) $\leq$ (some constant) (a lower bound)

[$O(n \log n)$] [$\Omega(n \log n)$]

bounds mutually certify each other to be optimal

"bounds are tight" (up to constants)

Conclusion: $n \log n$ is "right answer" for sorting in comparison model

Chapter 1

Searching and sorting

1.1 You already know a lot about theoretical computer science

I am not sure if the following story is true, but it makes a basic point. Bill is an electrical engineer working in chip verification: that is, he looks for bugs in computer chips. Nevermind the details. At some kind of team meeting at his work, a co-worker proudly boasts that they can count the number of remaining bugs on two hands — that is, at most 10. To which Bill replies:

> "You mean, 1024?!"

He might have been off by 1, but the point is clear: your ten fingers make ten *binary digits*, i.e., *bits*. We usually count starting from two closed fists — 0 — and straightening one finger at a time until our hands are open and we've counted to 10. But there are many more *combinations* of open and closed fingers: 2^{10} combinations, to be exact. You could probably train yourself to use your fingers in binary and count to 1023.

This is a trick that we all know very well. It is implicit every time we use decimal digits, instead of "caveman" style tallies. Below are two different ways to express the number 23:

$$||||||||||||||||||||||| = 23.$$

Clearly the right-hand side (RHS) is a much more efficient representation than the left-hand side (LHS).[1] The RHS leverages the mathematical fact that the number of strings of k characters from an alphabet of size 10 (namely, $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$) is 10^k. *Duh.* But the k in the top-right corner of "10^k" is extremely important. It indicates an *exponential growth* in the number of possible combinations, as a function of the number of digits. We see such an exponential growth for any alphabet size (e.g., the set of bits $\{0, 1\}$ has size 2) — except, of course, for alphabets of size 1.

[1] In fact, a previous version of this note mistakenly had 25 lines on the left, and nobody noticed.

Over/under. Consider the following simple game we played in elementary school called "over-under". The teacher declares to the class that he or she has secretly picked a number between 1 and 100. The game then proceeds in rounds. Each round, a student from the class guesses a number between 1 and 100. The teacher then replies that the guess is either equal, over, or under the secret number. The game goes back and forth, with the students guessing a number and the teacher giving the over/under, until the students guess the secret number. The goal of the students is to guess the number in as few rounds as possible.

Of course one can use at most 100 guesses by looping through the numbers from 1 to 100; eventually you will guess the number. This approach is slow and boring. There is a better strategy that is pretty obvious, even to children. *Guess* 50. Not because 50 represents an average in the range, but because it is the *median*. The teacher replies either "over" or "under". In the first case, you have restricted the range of possibilities to the numbers 1 through 49. In the second, the remaining range is 51 through 100. In either case, you have reduced the problem from 1 of 100 possibilities, to 1 of 50. Next round, you should guess 25 or 75, and so forth.

How many rounds are required? By picking the median of the remaining range, each round reduces the number of possibilities by half. Thus the number of rounds is the number of "halvings" required to reduce the number of possibilities from 100 to 1. There is a mathematical term for this and it is called the *logarithm base* 2 of 100, denoted $\log_2(x)$. For a base $b > 1$, $\log_b(x)$ is the function defined by

$$b^{\log_b(x)} = x.$$

That is, the number of times we multiply by b (starting from 1) to get x. For us, we have

$$\log_2(100) = 6.643856...$$

This implies that 6 rounds are enough.[2] The real point is that by repeatedly dividing the space in half, we only need $\log(n)$ guesses instead of n, to find a number.

This is our first algorithm; it is called *binary search*. Note that this algorithm isn't "guessing" the secret number; it's deducing it. Formally, `binary-search` finds an element in a sorted array of n elements in $O(\log n)$ time. See fig. 1.1 on the following page for pseudocode.

[2]Why 6 instead of 7? Certainly 7 rounds are enough. But also we can see that 6 rounds are enough to reduce to (strictly) less than 2 choices (because $\log_2(50) = \log_2(100) - 1 < 6$) - but the only integer less than 2 is 1! So 6 actually suffices. We should mention that normally we don't care the difference of one more step when talking about computer algorithms — what's one more step for a computer?

`binary-search(`$A[1..n]$`, `x`)`

/ Given an array of n comparable elements $A[1..n]$ in sorted order, and another comparable element x, returns an index i such that $A[i] = x$, or* `nil` *if no such index exists.* */*

1. If $n = 0$ return `nil`.
2. $m \leftarrow \lceil (n+1)/2 \rceil$.
3. If $A[m] = x$ then return m.
4. If $A[m] > x$ then return `binary-search(`$A[1..m-1], x$`)`.
5. If $A[m] < x$ then return $m +$ `binary-search(`$A[m+1..n], x$`)`.

Figure 1.1: A recursive implementation of binary search.

Exponential and logarithmic. We have now introduced two mathematical functions that we all have understood implicitly since early childhood: the exponential function, 2^x, and the logarithm, $\log_2(x)$. The former represents combinatorial explosion; the latter represents the awesome efficiency of binary search. Algorithm design is basically about appreciating and exploiting the difference between these two functions. As simple as that may sound, it is a concept that we do not totally understand.

There is a world of difference between the exponential function and the logarithm. In fact "a world" may be understating it. Consider the number of atoms in the universe. This is called the *Eddington number*. The current estimate on the Eddington number is about 10^{80}, which would be painful to write out in full. Now, the *logarithm base 2* of the Eddington number is

$$\log_2\left(10^{80}\right) = 80 \log_2(10) \approx 265.75.$$

265.75 is a large number but not astronomical. (You can even count it on two hands!) Now suppose we take the logarithm *again*. We have

$$\log_2 \log_2(\#\text{ atoms in the universe}) \approx \log_2(265.75) \approx 8.$$

8! Just 8! You could have finger-counted to eight the old-fashioned way.

So the logarithm rapidly takes astronomical numbers to very modest ones. 265.75 seems small given the scale of the universe. You could play the over/under game for a secret number between 1 and the Eddington number, and in a medium (if boring) amount of time, win! In many ways, binary search is an ideal algorithm. We will encounter many problems that we wish were as easy as over/under.

1.2 Sorting

This brings us to the topic of sorting. Sorting is maybe the most useful thing a computer can do. Once n items are put in sorted order, we can locate any one of them in $O(\log n)$ time by binary search (i.e., the over-under game).

Human-sort. How do we (as humans) sort? Take for example the following 10 numbers, in arbitrary order.

$$47,\ 7,\ 82,\ 31,\ 87,\ 63,\ 19,\ 94,\ 86,\ 43.$$

Suppose we want to sort these numbers in increasing order. The most natural way to go about this is to build out the sorted list from beginning to end, one at a time. First, we need to find the first number. Scanning the above, we see that it is 7. We write down 7, and mark it off in the input list to remind ourselves that we've already placed 7 in the output list. Below we start our output list on the left and keep track of the marks on the right.

7

47, ~~7~~, 82, 31, 87, 63, 19, 94, 86, 43

The next step, naturally, is to find the second smallest number. We scan the input list looking for the smallest number not yet crossed off: 19.

7, 19

47, ~~7~~, 82, 31, 87, 63, ~~19~~, 94, 86, 43

We continue in this fashion, identifying the third number, then the fourth, then the fifth... Eventually, we will finish the list. The last few lines of our transcript would be as follows.

$\vdots$

7, 19, 31, 43, 47, 63, 82, 86, 87

~~47~~, ~~7~~, ~~82~~, ~~31~~, ~~87~~, ~~63~~, ~~19~~, 94, ~~86~~, ~~43~~

7, 19, 31, 43, 47, 63, 82, 86, 87, 94

~~47~~, ~~7~~, ~~82~~, ~~31~~, ~~87~~, ~~63~~, ~~19~~, ~~94~~, ~~86~~, ~~43~~

So that's how we naturally tend to sort. Let's call this algorithm "`human-sort`".[3] It is intuitive and simple. But is it a *good algorithm*? This question hardly matters for sorting ten numbers. But the point of computers is that they can be automated

[3]This algorithm is more commonly called `selection-sort`.

on huge inputs. When analyzing the running time, we want to understand how the algorithm *scales* with the input size. So we treat the input size as a *variable*. Let n denote the number of numbers in the input. (Above, $n = 10$.) How long does the algorithm take, as a function of n?

Our algorithm `human-sort` has n iterations, where in the ith iteration (out of n), it identifies the ith largest number in increasing order. To identify this number, it scans the entire list of n numbers, having to do a few elementary operations for each (like checking for a mark, or comparing to the smallest number found so far). So the running time is roughly

$$(n \text{ iterations}) \times (\text{scanning } n \text{ items}) \approx n^2 \text{ operations over all.}$$

Is it exactly n^2 operations? The question is not well-posed, because we did not formally define what an "operation" is. But we clearly understand that as we scan the list, we need a constant amount of time to process each item. Maybe it is 1 operation, or maybe it is 10, or maybe it is 100. We won't worry about the exact count, in part because an exact accounting depends on the language and the hardware (which is always improving), and because this level of detail does not really inform high-level algorithm design.

As far we are concerned, there is some constant $C > 0$, *independent of* n, such that `human-sort` takes at most Cn^2 steps / units of time, at least for n sufficiently large. The point is not on C, but rather on the polynomial n^2. This emphasis is codified in a notation called "big-O". For a function $f : \mathbb{N} \to \mathbb{N}$, we say that an algorithm takes $O(f(n))$ time if there are constants $C > 0$ and $N > 0$ such that for all inputs of size $n \geq N$, the algorithm takes at most $Cf(n)$ "steps". Again we won't specify exactly what a step is. You and I can both generally agree on what is and isn't a single step; where there's disagreement, it will only affect the constant C. So we can generally agree on a $O(f(n))$ running time without worrying about what language we're programming in or what computer we're running it on; more broadly, we can begin to develop a *general theory* of algorithms.

In big-O notation, `human-sort` takes $O(n^2)$ time. The hidden constant C may depend on low-level details, but we all agree that *asymptotically* the algorithm grows like n^2. We call this a *quadratic* running time, since n^2 is a quadratic function of n.

Checking a sort. Can we do better? That is, can we sort numbers faster than $O(n^2)$ time? "Better?" is always the question in algorithm design. Let us remind ourselves that many of us have used `human-sort` all our lives without questioning it. $O(n^2)$ is a natural running time, since it is the time required to compare all pairs of numbers. So when we are asking for a faster algorithm, we are really asking if it is possible to sort all numbers without directly comparing every pair of numbers.

Let us ask a related question from a different direction: verification. Given a list of n numbers, how long does it take to *verify* that it is sorted? (We pause to let the reader consider this.)

A list of n numbers $x_1, \ldots, x_n$ is sorted in increasing order if and only if $x_i \leq x_{i+1}$ for every index $i = 1, \ldots, n-1$. To verify a list is sorted, we only need to compare every consecutive pair of numbers, x_i and x_{i+1}, and certify that $x_i \leq x_{i+1}$. So the answer is linear time: $O(n)$.

Now we have a gap. We can sort with `human-sort` in $O(n^2)$ time. We can verify that n numbers are sorted in $O(n)$ time. Can we sort as fast we can verify a list is sorted?

As one more baby step before unveiling our next algorithm, consider the following special case of sorting. Suppose our list of n numbers is *almost* sorted in the following sense. Assume n is even (for simplicity), and suppose the first half and second half of the list are both sorted amongst themselves. That is, out of n numbers $x_1, \ldots, x_n$, we are promised that

$$x_1 < x_2 < \cdots < x_{n/2} \text{ and that } x_{n/2+1} < x_{n/2+2} < \cdots < x_n.$$

We are *not* promised that $x_i < x_j$ when $i \leq n/2 < j$. For example, for the list from our discussion on `human-sort` (page 15), suppose the first and second halves were sorted as

7, 31, 47, 82, 87 and 19, 43 63, 86, 94.

Given a "half"-sorted list in the above sense, how long does it take to produce a fully sorted list? Again we pause and encourage the reader to solve this.

Merge-sort. The reader might have realized the following procedure to combine the two sorted lists into a single sorted list. Let us build the output list one by one, starting from the beginning. What is the smallest number in our two lists? We know automatically that it is the first number in one of the two sublists, so we only have to check these two numbers to identify it. This is in stark contrast to `human-sort`, where we had to scan all of the items to find the first item. Next, we need the second smallest number. Depending on whether the smallest number came from the first or second list, the second smallest number overall will be either the first number in one

`merge`($A[1..m], B[1..n]$)

/ Given two sorted lists A and B, produces the combined lists in sorted order. */*

1. Allocate a new array $C[1..m+n]$, let $i = 1$, and let $j = 1$.
2. While $i \leq m$ and $j \leq n$:
   A. If $A[i] \leq B[j]$:
      1. $C[i+j-1] \leftarrow A[i]$ and $i \leftarrow i+1$.
   B. Else ($B[j] < A[i]$):
      1. $C[i+j-1] \leftarrow B[j]$ and $j \leftarrow j+1$.
3. While $i \leq m$:
   A. $C[i+j-1] \leftarrow A[i]$ and $i \leftarrow i+1$.
4. While $j \leq n$:
   A. $C[i+j-1] \leftarrow B_2[j]$ and $j \leftarrow j+1$.
5. Return C.

Figure 1.2: Merging two sorted lists into a single sorted list in linear time.

list or the second number in the other. To expedite this process, we can keep track of how many numbers we have taken from the top of each list. Then in each step we only need to check the two tops of the remaining parts of the lists for the next smallest number. Continuing in this fashion, we will eventually "zip up" the two sorted lists into a single, globally sorted list. We needed a constant amount of time to produce each number in the output, so the overall running time on n numbers is $O(n)$.

We call this subroutine `merge`. Pseudocode is given in fig. 1.2 above. Formally, we have the following guarantee.

Lemma 1.1. *Given two sorted lists $A[1..m]$ and $B[1..n]$, `merge`(A, B) returns the list containing all the elements of A and B in sorted order in $O(m+n)$ time.*

The `merge` subroutine gives rise to a sorting algorithm called `merge-sort`. At a high-level, `merge-sort` is very simple: given a list of unsorted numbers, divide the list into two halves. Sort the first half and sort the second half separately to produce two sorted lists, each containing half the numbers. Then merge them together to produce one sorted list. How do we sort the two halves? *By recursion:* we call `merge-sort` on each of the two sublists.

When designing a recursive algorithm such as `merge-sort` it is critical – both for ease of implementation, and proving correctness – that we declare what the algorithm

does, as a sort of binding contract. We call this the *recursive specification*. For example, for an array of n comparable elements $A[1..n]$, we define

> `merge-sort`($A[1..n]$): Given an array $A[1..n]$ of comparable elements, returns an array consisting of the elements of A sorted in increasing order.

Here the input and the output (and properties thereof) are made explicitly clear.

A specification makes implementation easier by clarifying our intent. It also solidifies what we can assume from `merge-sort` when making recursive calls to smaller inputs. Indeed, the recursive specification will map directly to the *inductive hypothesis* when proving correctness.

Pseudocode for `merge-sort` is given in fig. 1.3 on the following page. The task is now to analyze the algorithm. A full analysis of `merge-sort` should address two aspects.

1. Correctness: prove that `merge-sort` indeed returns a sorted list.
2. Running time: identify a function $f(n)$, as small as possible, for which we can show that `merge-sort` takes $O(f(n))$ time on an input of n numbers.

We first prove the correctness – that `merge-sort` indeed sorts its input correctly. We prove this by induction on the number of numbers in the input, n.

In the base case, $n \leq 1$, the input is already sorted, and the algorithm simply returns it.

In the general case $n > 1$, we *assume by induction on* n that for all $k < n$, `merge-sort` correctly sorts any input of size k. For $n \geq 2$, the algorithm splits the input into two lists of size $\lfloor n/2 \rfloor$ and $\lceil n/2 \rceil$ and recursively calls `merge-sort` on these inputs. $\lfloor n/2 \rfloor$ and $\lceil n/2 \rceil$ are both strictly less than n because $n \geq 2$. By induction, both of these recursive calls will correctly sort their respective halves of the input. The algorithm then combines the two sorted sublists by merging. We have already discussed the correctness of merging above. Thus `merge-sort` correctly outputs the n numbers in sorted order.

Next we analyze the running time. The first step is to model the running time mathematically. Let $T(n)$ denote the amount of time taken by `merge-sort` on an input of size n. We can define $T(n)$ recursively as follows.

$$T(0) = T(1) \leq C,$$
$$T(n) \leq T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + Cn,$$

`merge-sort(`$A[1..n]$`)`

/ In the base case, the list is so short there is nothing to do. */*

1. If $n \leq 1$ then return A.

/ In the general case, we divide the input in half and recursively sort each half. */*

2. Let $m = \lfloor \frac{n}{2} \rfloor$.
3. $B_1[1..m] \leftarrow$ merge-sort$(A[1..m])$.
4. $B_2[1..n-m] \leftarrow$ merge-sort$(A[m+1..n])$.

/ Now we merge the two sorted halves in linear time. */*

5. return `merge(`B_1, B_2`)`.

Figure 1.3: The `merge-sort` algorithm.

for some constant $C > 0$. Let us point out that when n is even, the second line above simplifies to the cleaner form

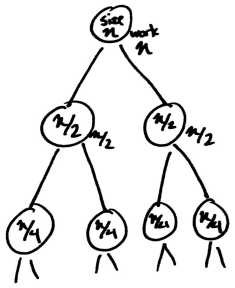
$$T(n) \leq 2T(n/2) + Cn.$$

Morally, when doing an asymptotic analysis in n, we shouldn't fret over ± 1 associated with rounding $n/2$ up and down. Let us assume for the moment that the second recursion (where $T(n) \leq 2T(n/2) + Cn$) is simply a valid recursion for $T(n)$. We will justify this later. Alternatively, the reader can assume that n is a power of two, where the math is automatically cleaner. This assumption can also be justified formally.

Recursion trees. To analyze `merge-sort`, and in particular the recursion

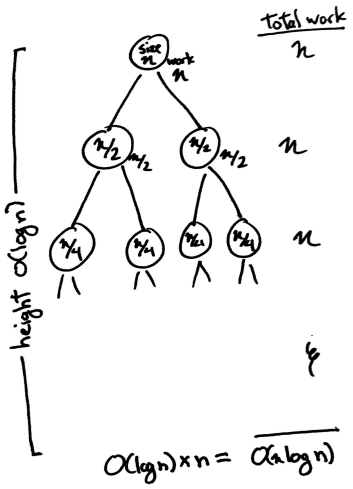
$$T(0) = T(1) \leq C$$
$$T(n) = 2T(n/2) + Cn$$

for some absolute constant $C > 0$, let us briefly describe a useful tool called *recursion trees*. Recursion trees help us visualize the recursive subproblems in a tree-like structure. The idea is to create a tree where each node corresponds to a subproblem. One node is a child of another if the first node is a subproblem of the second.



For `merge-sort`, the root of the tree corresponds to our initial call to `merge-sort`$(A[1..n])$. The root has two children, corresponding to the subproblems for the recursive calls to `merge-sort`$(A[1..\lfloor \frac{n}{2} \rfloor])$ and `merge-sort`$(A[\lfloor \frac{n}{2} \rfloor + 1..n])$. When doing an analysis like this, I literally draw the first few layers of the tree. I annotate each node with the size of the input, as well as the amount of time spent on each subproblem, excluding recursive calls. (E.g., the root will be labeled with size n and work Cn.)

Having visualized the computation in this tree, we calculate the overall running time as follows. For $i \in \mathbb{Z}_{\geq 0}$, let the *ith level* of the tree refer to the subproblems that are i generations away from the root. In our case, the root is the only node in level 0, its two children form level 1, its four grandchildren form level 2, and so forth. To upper bound the total computation, we divide it up into levels. We first upper bound the total amount of work at each level i, excluding recursive calls (which are accounted for in lower levels of the tree). Then we upper bound the number of levels. The overall running time is then bounded above by the product of the work-per-level and the total number of levels.



Fix a level i. Let us label each subproblem with the size of the subproblem, and then the running time spent on that subproblem. In the ith level, we have 2^i subproblems whose total sizes added up to n. The total time spent on each subproblem, excluding recursive calls, is linear in the input size. Thus we spend $O(n)$ time on subproblems in level i, excluding calls for recursive calls.

Next we identify the number of levels. Each level, the problem sizes divide by 2. Consequently, the leaves – corresponding to problems of size 1 – are at level (at most) $O(\log n)$, and the tree has height $O(\log n)$ in all.

Now we combine everything together to get the overall running time. We have

$$(O(\log n) \text{ levels}) \times (O(n) \text{ time per level}) = O(n \log n) \text{ total time.}$$

$O(n \log n)$ is a sizeable improvement over the $O(n^2)$ time required by `human-sort`.

On the rounding error. We simplified our discussion by assuming the running time could be modeled by the recursion

$$T(n) \leq 2T(n/2) + Cn,$$

for an absolute constant C.

This can be justified by the following trick. Let us redefine $T(n)$ as the maximum amount of time taken by `merge-sort` on any input of size *less than or equal to* n. Note that by definition $T(n)$ is increasing.[4] Then we analyze $T(n)$ for the special case where n is a power of two, for which the simpler recursion is correct.

Now consider any other value n that is not a power of two. Then the value $n' = 2^{\lceil \log_2 n \rceil}$ is a power of 2, and $n \leq n' < 2n$. We can use $T(n')$ to upper bound $T(n)$. Since n' is within a constant factor of n, $T(n')$ is within a constant factor of $T(n)$, and lazily upperbounding by $T(n')$ is not so bad.

A different way to justify the rounding error is as follows. We first write

$$T(n) \leq 2T(n/2 + 1) + Cn$$

Then we can define $U(n) = T(n + 2)$. Then we have

$$\begin{aligned} U(n) &= T(n+2) \\ &\leq T(\lfloor (n+2)/2 \rfloor) + T(\lceil (n+2)/2 \rceil) + C(n+2) \\ &\leq 2T(n/2+2) + C(n+2) \\ &= 2U(n/2) + C'n \end{aligned}$$

for $C' = (C + 2)$. So we could apply our cleaner analysis directly to $U(n)$ instead of $T(n)$, which in turn bounds $T(n)$.

Guess and check. The recursion tree makes it easy to (literally) see the $O(n \log n)$ running time. Alternatively, we could have guessed that the running time was at most $Dn \log n$ for some constant D, and then verified directly that it satisfies the recursion by induction on n.

1.3 Lower bounds for sorting

We now adopt a different perspective and consider *lower* bounds for sorting. That is, we want to identify some function $f(n)$ for which we can prove that *any* sorting algorithm takes *at least* $\Omega(f(n))$ time, in the worst case, on an input of size n.[5]

Lower bounds might seem less intuitive than upper bounds. It is clear how to prove an upper bound $O(f(n))$ for a given problem (such as sorting): we provide an

[4] Of course this seems true already just based on the `merge-sort` code, but it seems annoying to prove this formally.

[5] "big-Ω", $\Omega(\cdots)$, is the lower bound counterpart of $O(\cdots)$. Some quantity x is $\Omega(f(n))$ if there exists constants $C, N > 0$ such that $x \geq Cf(n)$ for $n \geq N$.

algorithm along with a proof that the algorithm is correct and runs in $O(f(n))$ time in the worst-case. *Any single* correct algorithm and proof would suffice. For lower bounds, we need to argue that *every* correct algorithm – including all those we have never even imagined – must have at least a certain worst-case running time. Our reasoning must generalize beyond any specific algorithm, which is typically a more abstract discussion than when analyzing a single concrete algorithm.

We will obtain a lower bound specifically in the *comparison model*. The comparison model assumes the input is a list of distinct elements of some definite order, where the only information we can gain about these elements is by pairwise comparisons. Formally, we define a *comparison query* as a query where we take two elements x and y and find out if either $x < y$ or $y < x$. Rather than try to lower bound the running time directly, we will lower bound the number of comparison queries required by any sorting algorithm. This translates to a lower bound on the running time under the reasonable assumption that each comparison takes at least constant time.

Now, fix any sorting algorithm. Suppose the sorting algorithm uses only k comparisons in the worst case, and then (allegedly) outputs the sorted list. Consider the *transcript* of outputs of these comparisons, where for each comparison we record whether the first or second element in the query was larger. Observe that the only external information the algorithm takes in is the results of these queries. In particular, the first comparison query is predetermined. The second comparison query can only depend on the outcome of the first comparison. In general the ith comparison depends only on the outcomes of the preceding $i - 1$ comparisons. Ultimately, the algorithm outputs some reordered list as a function of *only* the outputs of the k comparison queries. Each query has 2 possible outcomes. Therefore the algorithm can output at most 2^k different lists.

Meanwhile, there are $n!$ different ways to order n elements (a.k.a. *permutations*). So to be able to produce all $n!$ possible permutations, we must have $2^k \geq n!$. That is,

$$k \geq \log_2(n!).$$

What is $\log_2(n!)$? The exact value of $n!$ is difficult to handle. However, we are only interested in a constant factor estimate. Moreover, a very loose estimate of $n!$ will suffice because we will take the log of that estimate anyway. To this end, observe that

$$n! = n(n-1)(n-2)\cdots 1 \geq (n/2)^{n/2},$$

hence

$$\log(n!) \geq \log\Big((n/2)^{n/2}\Big) = (n/2)(\log(n) - 1) = \Omega(n \log n).$$

Thus, up to constants, any algorithm must make at least $n \log n$ comparisons (just like `merge-sort`!) simply to be able to output all possible permutations.

Theorem 1.2. *Any deterministic comparison based sorting algorithm requires at least* $\log(n!) = \Omega(n \log n)$ *comparisons in the worst case.*

The theorem brings some closure to our discussion: `merge-sort` is essentially best possible. This conclusion – where we have an upper bound that is *tight* (i.e., best possible) with the lower bound – is the ideal for any problem we are interested in. Most problems are not understood this well.

While we often focus on algorithms, lower bounds like theorem 1.2 are important too. We will see some problems that are so tricky we haven't found good algorithms despite many years of research – but instead we have (conditional) lower bounds that suggest these problems *should be hard*.

We sometimes think of upper bounds and lower bounds as competing in an adversarial relationships – and certainly they bound one another mathematically – but in another sense they are two sides of the same coin. Neither the $O(n \log n)$ upper bound nor the $\Omega(n \log n)$ lower bound are as compelling taken alone as they are together, where they *mutually certify* one another to be tight (up to constants).

Now, every theorem has its terms and conditions and so we remind ourselves of exactly what we proved. The theorem states that when the elements can only be compared pairwise, any correct sorting algorithm must make at least $\Omega(n \log n)$ pairwise comparisons. This does not limit related tasks such as finding the median element (which we discuss later) or sorting only the first k elements (exercise 1.8). Another situation the theorem does not cover is where we are sorting integers represented by bit strings of a fixed length. One might try to take advantage of the bits and somehow sort the numbers without actually having to compare the numbers pairwise. (The `radix-sort` algorithm takes this approach.)

1.4 Further sorting algorithms

`merge-sort` is clean and serves as a nice introduction to divide-and-conquer and for solving recurrences. It is also optimal in the comparison model. However it is not the only $O(n \log n)$ time algorithm and here we list a few more approaches.

Sorting with heaps. Recall that the *heap* data structure (cf. appendix B.2) maintains a collection of comparable items with direct access to the minimum one. More precisely, the method `insert` will insert an element, and `remove-min` will remove the minimum element, both in $O(\log n)$ time, where n is the number of items in the heap.

Note that a heap does not necessarily sort all the elements; it merely promises to keep track of the smallest one. Still we can use a heap to sort the items, as follows.

`heap-sort`$(x_1, \ldots, x_n)$

1. Initialize an empty heap H.
2. Insert each element x_i into H.
3. Repeatedly call `remove-min` on H until the heap is empty, and return the elements in the order they are output by H.

Given the running times for the heap operations, it is easy to analyze the sorting algorithm above. Each element takes $O(\log n)$ to insert. Each element removed takes $O(\log n)$ time. This gives the following bound.

Theorem 1.3. `heap-sort` *sorts n items in $O(n \log n)$ time.*

`heap-sort` is our first example of using a data structure to facilitate faster running times.

Quicksort. The algorithm `quick-sort` is very simple and effective in practice, but also *randomized*. `quick-sort` is (like `merge-sort`) a divide and conquer algorithm, but divides the data in a different and randomized way `merge-sort`. Given n unsorted items, it selects one item x *uniformly at random*. This item x is called the *pivot*. It then divides the remaining items into two (unsorted) sets: the set of items smaller than x, and the set of items bigger than x. (It takes $O(n)$ items to do this.) We sort each set separately by recursive calls to `quick-sort`, and combine the lists at the end with x in the middle. See fig. 1.4 on the next page for pseudocode.

Intuitively, when selecting a random pivot x_i, we are hoping that x_i is close to the median. Then A and B will roughly divide the input in half. If indeed x_i always divided the input in half (or close to half), then a $O(n \log n)$ running time follows from essentially the same analysis as `merge-sort`. Of course we may be unlucky and in the worst case `quick-sort` could run in $O(n^2)$ time. But this is the wrong way to look at it.

When analyzing randomized algorithms, we are more interested in bounding the *average running time* for *any* input. Here "average" is meant exclusively to the randomization within the algorithm. (We are not averaging over a distribution of inputs.) Here, one can show that that `quick-sort` runs in $O(n \log n)$ time on average (for any list of n items).

Theorem 1.4. `quick-sort` *sorts n items in $O(n \log n)$ randomized time in expectation.*

`quick-sort`$(x_1, \ldots, x_n)$

/ We assume the elements are distinct for simplicity. */*

1. If $n = 0$ return the empty list and if $n = 1$ return the list consisting of x_1.
2. Otherwise randomly sample x_i uniformly at random.
3. Let $A = \{x_j : x_j < x_i\}$ and let $B = \{x_j : x_j > x_i\}$. *// $n - 1$ comparisons.*
4. Recursively sort A and sort B, and combine the sorted sublists with x_i inserted in between. Return the combined list.

Figure 1.4: The `quick-sort` algorithm.

`quick-sort` is appealing because it is very simple to implement; this simplicity appears to translate well to practice. Indeed it is the default sorting subroutine in `unix`. The only tricky part is in the analysis, because the analysis is *probabilistic*. Actually the analysis itself is not too complicated, were it not for that probability theory is new (and appears strange) for many students.

We briefly sketch the high-level ideas here since it is interesting; we will do the analysis again much more carefully when we study randomized algorithms in detail in chapter 23.

Clearly the running time is proportional to the total number of comparisons. Let y_i refer to the ith largest element. For $i < j$, what are the odds that y_i is compared to y_j? It is the probability that y_i or y_j gets randomly selected as a pivot before any of the in-between elements, $y_{i+1}, \ldots, y_{j-1}$. Among the elements $y_i, \ldots, y_j$, all are equally likely to be selected first. Thus the probability of comparing y_i to y_j is $1/(j - i + 1)$. Now, the key observation is that

> *by* linearity of expectation *(which is introduced in chapter 23), the expected number of comparisons equals the sum, over all $i < j$, of the probability that y_i is compared to y_j.*

Now the sum of $1/(j - i + 1)$, over all $1 \leq i < j \leq n$, comes out to $O(n \log n)$.

One can show further that `quick-sort` runs in $O(n \log n)$ time with exceedingly high probability; in particular, the $O(n^2)$ running time will almost never occur (for large n). This analysis requires additional probabilistic tools called *concentration bounds*.

Etcetera. There are many more possible sorting algorithms as well as heuristics[6] and hybrid approaches that help performance in practice. For example, the Python `sort` subroutine checks the input for large *runs* (consecutive subsequences that are already sorted) that can effectively be skipped; in particular, it takes $O(n)$ time if the list is already sorted.

1.5 Additional notes and references

`merge-sort` was invented by Jon von Neumann in 1945 [Knu73]. The topics discussed here can also be found in [Eri19, §1.4–1.7], [DPV08, §2.3], [KT06, §5.1–5.2], [Blu11b, Chap. 2 and 5], and [Knu73]. Our discussion assumed some comfort with the subtle and important notion of *induction*. We discuss induction at greater length in chapter 2.

Spring 2022 lecture materials. Click on the links below for the following files:

- Handwritten notes prepared before the lecture.
- Handwritten notes annotated during the presentation.
- (Re-)Recorded video lecture.

1.6 Exercises

Exercise 1.1. Use recursion trees to solve the following recurrences.

1. $T(n) = 3T(n/3) + 6n$ and $T(1) = 2$.
2. $T(n) = 2T(n/3) + 4n$ and $T(1) = 7$.
3. $T(n) = 4T(n/3) + n$ and $T(1) = 11$.

Exercise 1.2. Recall that binary search can find one element out of a sorted list of elements in $O(\log n)$ time. (Assuming the elements are arranged in, say, an array, so that we can access the ith element in constant time.) Here we will try to obtain a matching lower bound.

The general problem gives as input a list of n elements $(x_1, \ldots, x_n)$ in sorted order and with constant time access for any index, as well as an additional element k that acts as a key. For simplicity we may assume that k is promised to be one of the elements. The goal is to identify an index i such that $x_i = k$. We restrict ourselves to a comparison-only model where we may ask comparison queries such as "is $k < x_i$?",

[6] Here we use "heuristics" to refer to any techniques that (may be very useful but) do not theoretically improve the worst-case running times.

"is $k \leq x_i$?", "is $k = x_i$?", and so forth. Prove a lower bound of $\Omega(\log n)$ comparison queries (in the worst case) for any search algorithm that correctly finds k in the sorted list $(x_1, \ldots, x_n)$.

Exercise 1.3. Problems 1–3 describe a search problem on a nonempty array $A[1..n]$ of comparable elements.[7] For each of these problems, design a recursive algorithm that solves the search problem as fast as possible.[8] Briefly analyze the running time of each algorithm.[9]

1. Let $A[1..n]$ be in a *rotated* sorted order. That is, there exists an index $i \in [n]$, called the *rotation index*, such that the concatenation of $A[i..n]$ and $A[1..i-1]$ is sorted in increasing order. (One can think of $A[1..n]$ as being sorted initially, and then rotated cyclically to the right by $i - 1$ slots). The goal is to compute the unknown rotation index i. For simplicity, you may assume all the elements are distinct.

2. We say that $A[1..n]$ is a *mountain* if there exists an index $i \in [n]$, called the *peak*, such that $A[1..i]$ is sorted in increasing order and $A[i+1..n]$ is sorted in nondecreasing order. Given a mountain $A[1..n]$, find the peak. For simplicity, you may assume all elements are distinct.

3. We say an index $i \in [n]$ is a *local minimum* if either (a) $i = n = 1$, (b) $i = 1$ and $A[1] \leq A[2]$; (c) $i = n$ and $A[n] \leq A[n-1]$; or (d) $1 < i < n$, $A[i-1] \geq A[i]$ and $A[i] \leq A[i+1]$. The goal is to find a local minimum from an array $A[1..n]$.

4. Prove that each of the problems above have a lower bound of $\Omega(\log n)$ queries in the comparison model.

5. (Extra credit) Invent your own fun and interesting search problem on an array of comparable elements.

Exercise 1.4. Let $A[1..n]$ be an array of integers in the set $\{1..n\}$, sorted in increasing order. (There may be repeats.) A *fixed point* is an index $i \in [n]$ such that $A[i] = i$.

[7] Given two elements x and y, you can query if $x < y$, $x > y$, or $x = y$.

[8] This includes a recursive spec, and pseudocode implementing the recursive spec.

[9] For the sake of brevity we do not ask for an explicit analysis of correctness. We will treat your recursive spec as the induction hypothesis of a proof by induction, and fill in the rest.

1. (2 pt.) Prove the following by induction on $j - i$: *for all pairs of indices i, j such that $1 \leq i \leq j \leq n$, $i \leq A[i]$, and $A[j] \leq j$, there is a fixed point k in the range $i \leq k \leq j$.*

   This claim shows in particular that A has a fixed point.

2. (2 pt.) Moving onto developing algorithms, give a recursive spec for the problem of finding a fixed point in A.[10]

3. (2 pt.) Give a recursive algorithm (the faster the better) implementing your recursive spec.

4. (2 pt.) Prove your algorithm is correct by induction. (There may or may not be some redundancy in your argument with part 1. You may also use the mathematical fact established in part 1.)

5. (2 pt.) Analyze the running time of your algorithm.

Exercise 1.5. Let $A[1..m]$ and $B[1..n]$ be two sorted arrays of comparable elements. Given an integer $k \in \{1, \ldots, m + n\}$, the goal is to find the kth largest element among the combined elements of A and B. For simplicity you make assume that all the elements are distinct.[11]

1. Define a recursive spec for a recursive algorithm to solve this problem.

2. Show that if either m or n is ≤ 5 then we can find the kth element in $O(1)$-time by direct means. (This is to help declutter your implementation and avoid fencepost errors.)

3. Give a recursive algorithm implementing your recursive spec. You may simply refer to the previous part for your base case.

4. Prove your algorithm is correct by induction.

5. Analyze the running time of your algorithm.

[10] There may be more than one fixed point; any fixed point is fine.

[11] By assuming standard data structures keeping track of the offset and length, you can get (a pointer to) the subarray $A[i..j]$ of an array $A[1..m]$ in $O(1)$ time.

Exercise 1.6. Let $A[1..n] \in \mathbb{R}^n$ be an array of n numbers. An *inversion* is a pair of numbers out of increasing order; more precisely, a pair of indices $i, j \in [n]$ such that $i < j$ and $A[i] > A[j]$. Design and analyze an algorithm (as fast as possible) for counting the number of inversions in A.

Exercise 1.7. Consider coordinates in the plane; i.e., pairs (a, b) where $a, b \in \mathbb{R}$. Recall that two coordinate pairs (a, b) and (c, d) are *comparable* if either (a) $a \leq c$ and $b \leq d$, or (b) $c \leq a$ and $d \leq b$.

Given a set of n coordinate pairs $(a_1, b_1), \ldots, (a_n, b_n)$, consider the problem of computing the number of pairs of coordinate-pairs that are comparable. You may assume for simplicity that all pairs are distinct. Design and analyze an algorithm (as fast as possible) for counting the number of comparable pairs.

Exercise 1.8. Let $A[1..n] \in \mathbb{R}^n$ be an array of n comparable elements. Let $k \in \mathbb{N}$ be an input parameter where $k \leq n$. (We are particularly interested in the case where k is much smaller than n). Consider the problem where we want to obtained a sorted list of the k smallest elements.

1. Design and analyze an algorithm (as fast as possible) that returns the k smallest numbers in A in sorted order.

2. Prove a lower bound (large as possible, up to constants) on the number of comparisons required by any correct algorithm.

Food for thought: How does your lower bound compare to your upper bound obtained in exercise 1.8?

Exercise 1.9. Here we develop a sorting algorithm that tries to take advantage of duplicates when there are very few distinct elements in the input.

In the *sorting with multiplicities* problem, the input, as usual, is an array of n comparable elements. The output is a little different from usual sorting. Suppose there are k distinct elements among the n in the input. The output should consist of a list of the k distinct elements, in sorted order, along with the frequencies of each element. (You do not know the value of k *a priori*.)

For example, if we are sorting letters alphabetically, and the input consisted of $(a, b, c, a, b, c, b, c, a, b, c)$, then the output would look like the list $((a, 3), (b, 4), (c, 4))$.

1. Design and analyze a recursive algorithm as possible for sorting with multiplicities. (Your running time may depend on k, and may be faster than $O(n \log n)$ when k is much smaller than n.)

2. Prove a lower bound (large as possible, up to constants) on the number of comparisons required by any correct algorithm for sorting with multiplicities.[12]

[12]Right now, our best lower bound is not tight with our best upper bound.