Searching and sorting

Kent Quanrud

January 25, 2021

1 You already know a lot about theoretical computer science

My father has many lame jokes, as fathers do. But one day he came home with a story from work that I thought was pretty funny although I am not sure if it is true. My father 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 – at most, 10. My dad shoots back:

> "You mean, 1024?!"

I can just see my father's grin. My dad might have been off by 1, but the point is clear: your ten fingers make ten *binary digits*, i.e., *bits*. We usually count to ten by starting from two closed fists – 0 – and straightening one finger at a time until our hands our open and we've counted to 10. But there are many more *combinations* of opened and closed fingers – 2^{10} combinations, to be exact. You could probably train yourself to use your fingers to count to 1023 (and make my father proud).

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). 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 of combinations, as a function of the number of digits. We see such an exponential growth for any alphabet size (e.g., bits $\{0, 1\}$ has size 2) – except, of course, for alphabets of size 1.

Over/under. Consider the following simple game from my elementary school days 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 guess by looping through the numbers from 1 to 100; eventually you will guess the number. But a classroom of rowdy children has no patience for that. 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 and 50. Next round, you should guess 25 or 75, and so forth.

How many rounds are required? If you keep 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 $b > 1$, $\log_b(x)$ 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. [1] 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.

We have now introduced two mathematical functions that we have all implicitly understood 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} - I won't both writing it 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 even before this discussion.

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]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?

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. 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 even before this discussion.

So the logarithm rapidly takes literally astronomical numbers to very modest one. 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.

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).

2.1 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 in the list to remind ourselves that we've already placed it in our list. Below we start our 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 our 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

$\cancel{47}, \cancel{7}, \cancel{82}, \cancel{31}, \cancel{87}, \cancel{63}, \cancel{19}, 94, \cancel{86}, \cancel{43}$

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

$\cancel{47}, \cancel{7}, \cancel{82}, \cancel{31}, \cancel{87}, \cancel{63}, \cancel{19}, \cancel{94}, \cancel{86}, \cancel{43}$

So that's how we naturally tend to sort. Let's call this algorithm "`human-sort`"[2]. 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 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 (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. Thus 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. 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.

2.2 Checking a sort

Can we do better? That is, can we sort numbers faster than $O(n^2)$ time? This 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.

[2]This algorithm is more commonly `insertion-sort`.

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 numbers of n numbers $x_1, \dots, x_n$ is sorted in increasing order if and only if $x_i \leq x_{i+1}$ for every index $i = 1, \dots, n-1$. 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 $x_i < x_{i+1}$. So the answer is linear time: $O(n)$.

So now we have a gap. We can sort with `human-sort` in $O(n^2)$ time. We can verify if 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 $x_1, \dots, x_n$, 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 $i \leq n/2 < j$. 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.

2.3 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 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 compare two numbers) to produce each number in the output, so the overall running time on n numbers is $O(n)$.

The above process is commonly called **merging**, and 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 lists. How do we sort the two halves? *By recursion:* we call `merge-sort` on each of the two sublists.

Pseudocode for `merge-sort` is given in Figure 1. 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

```
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 and half and recursively sort each half. */

2. Let $m = \left\lfloor \frac{n}{2} \right\rfloor$.

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. Allocate a new array $C[1..n]$, let $i = 1$, and let $j = 1$.

6. While $i \leq m$ and $j \leq n - m$

   A. if $B_1[i] \leq B_2[j]$

      1. $C[i+j-1] \leftarrow B_1[i]$ and $i \leftarrow i+1$

   B. else

      1. $C[i+j-1] \leftarrow B_2[j]$ and $j \leftarrow j+1$

7. while $i \leq m$

   A. $C[i+j-1] \leftarrow B_1[i]$ and $i \leftarrow i+1$

8. while $j \leq m$

   A. $C[i+j-1] \leftarrow B_2[j]$ and $j \leftarrow j+1$

9. return $C$
```

Figure 1: The merge-sort algorithm.

automatically 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. Note that $\lfloor n/2 \rfloor$ and $\lceil n/2 \rceil$ are both strictly less than n (because $n \geq 2$), so by induction, both of these recursive calls will correctly sort half 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.

Having proven the algorithm is correct, it remains to analyze the running time. The first step is to model it, mathematically. Let $T(n)$ be the amount of time taken by `merge-sort` on an input of size n. We can define $T(n)$ recursively as follows.

$$\begin{aligned} T(0) &= T(1) \leq C \\ T(n) &\leq T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + Cn. \end{aligned}$$

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 will also lead to a formally justified analysis for all n.

2.4 Recursion trees

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

$$\begin{aligned} T(0) &= T(1) \leq C \\ T(n) &= 2T(n/2) + Cn \end{aligned}$$

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 example, the root of our tree corresponds to our initial call to `merge-sort(`$A[1..n]$`)`. The root will have 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 will 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 will calculate the overall running as follows. For $i \in \mathbb{Z}_{\geq 0}$, let the ***i*th 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 recursively calls

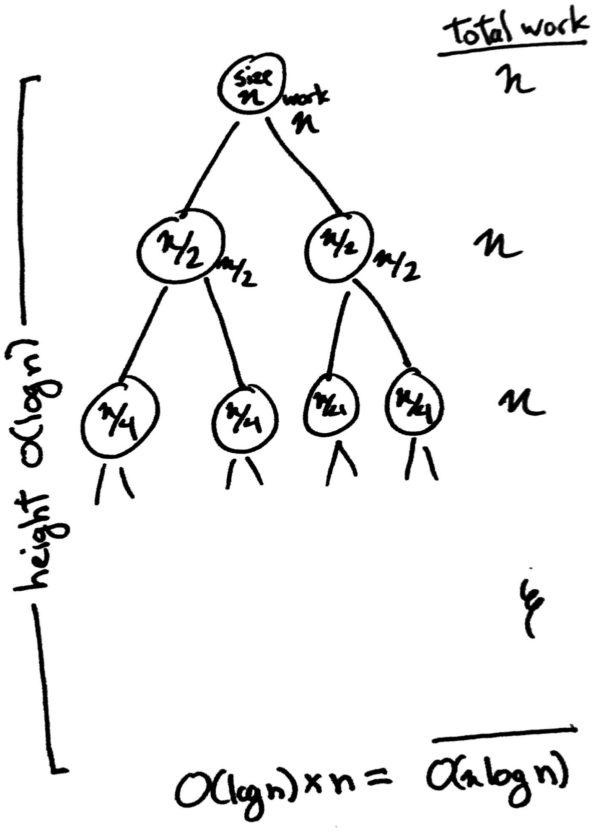


Figure 2: A quick sketch of a recursion tree analysis for `merge-sort`.

(which are accounted for in higher levels). 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

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

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

2.5 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. 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[3]. 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.

Alternatively, we can 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 $T(n)$.

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

2.6 Guess and check.

The recursion tree makes it easy to (literally) see the $O(n \log n)$ running time. Alternatively, if we could have guess that the running time was $O(n \log n)$, and then verified that it satisfies the recursion by induction.

3 A different kind of search problem

So much for sorting. Let us now introduce another problem that is also extremely simple and fundamental to computers: Boolean satisfiability.

A Boolean variable x is one where one can take only one of two values, *true* and `false`. In computers, the Boolean values are represnted by a single bit, with `true` = 1 and `false` = 0. We adopt the convention here as well. In Boolean algebra, we have three basic operations.

1. **Negation**, a unary operator denoted $\neg x$. $\neg$ maps `true` to `false` and `false` to `true`. We sometimes abbreviate $\neg x$.
2. A **disjunction**, also called an **or**, is a binary operator denoted $x \vee y$ for two variables x and y. We have $x \vee y =$ `true` if either x or y is `true`. If both x and y are `false`, then $x \vee y =$ `false`.
3. A **conjunction**, or also called an **and**, is a binary operator denoted $x \wedge y$. We have $x \wedge y =$ `true` if and only if both x and y are `true`; otherwise, $x \wedge y =$ `false`.

These operations can be combined, using parenthesize to notate the order of operations. For example, the formula

$$f(x_1, x_2) = (x_1 \wedge \bar{x}_2) \vee (\bar{x}_1 \wedge x_2)$$

implements what is commonly called the "exclusive-or" of x_1 and x_2.

Observe that for any $x, y, z \in \{0, 1\}$,

$$(x \vee y) \vee z = x \vee (y \vee z);$$

that is, the placement of parentheses does not matter. This is called the **associative property**, and for this reason we simply denote the above as

$$x \vee y \vee z.$$

Conjunction $\wedge$ is also associative. However, mixing $\wedge$ and $\vee$ is not associative: in general,

$$(x \vee y) \wedge z \neq x \vee (y \wedge z).$$

Instead we have the following rules, called the **distributive property** of $\vee$ and $\wedge$.

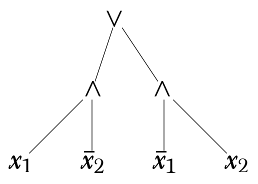
$$x \vee (y \wedge z) = (x \vee y) \wedge (x \vee z),$$
$$(x \vee y) \wedge z = (x \wedge z) \vee (y \wedge z).$$

$\neg$ interacts with $\wedge$ and $\vee$ by the following identity, called **De Morgan's laws**.

$$\neg(x \wedge y) = \bar{x} \vee \bar{y},$$
$$\neg(x \vee y) = \bar{x} \wedge \bar{y}.$$

We leave it to the reader to verify these identities.

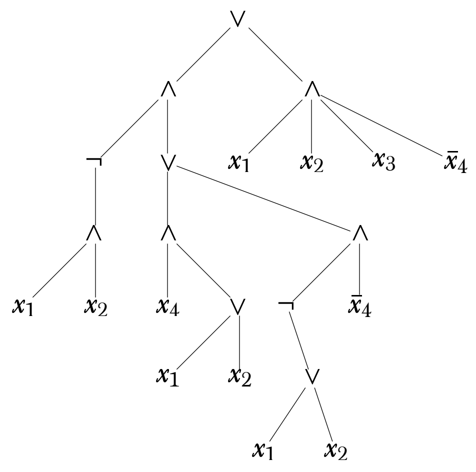
Boolean formulas can have nested parantheses. The part of a formula corresponding to one pair of matching parentheses is called a clause. We can visualize Boolean formulas (and the nested parentheses) in a tree. The following is a tree for exclusive-or.



As a perhaps more interesting example, the formula

$$\begin{aligned} f(x_1,x_2,x_3,x_4) = {} & (x_1 \wedge x_2 \wedge x_3 \wedge \bar{x}_4) \\ & \vee (\neg(x_1 \wedge x_2) \wedge ((x_1 \vee x_2) \wedge x_4) \vee (\neg(x_1 \vee x_2) \wedge \bar{x}_4)) \end{aligned}$$

effectively implements addition of two bits: it is satisfied iff $x_1 + x_2 = x_3x_4$, where x_3x_4 is interpreted as a two-digit binary number. The corresponding tree is as follows.



Boolean logic might look dry, but with a little creativity, it can be very expressive. For example, suppose we want to force two variables x and y to be equal. This can be modeled as

$$(y \wedge x) \vee (\bar{y} \wedge \bar{x}).$$

Adding the above clauses to Boolean formula is like programming y to be the value of x. As another example, suppose we wanted to model the conditional statement, "if x then y else z". This is given by

$$(x \wedge y) \vee (\bar{x} \wedge z).$$

Through these examples, Boolean algebra starts to resemble a programming language.

Both the algebraic and tree-like representations of boolean formulas emphasize a digital quality of Boolean formulas. Boolean logic is exactly the kind of thing that computers are good at. One might naturally try to program an algorithm that can *constructively solve* Boolean formulas, in the following sense:

> Given a Boolean formula $f(x_1, \ldots, x_n)$, find an assignment $x_1, \ldots, x_n \in \{0, 1\}$ for which $f(x_1, \ldots, x_n) = 1$, if one exists.

One can also consider the *decision* version of the problem, as follows.

> Given a Boolean formula $f(x_1, \ldots, x_n)$, output `true` (or 1) if there exists an assignment $x_1, \ldots, x_n \in \{0, 1\}$ for which $f(x_1, \ldots, x_n) = 1$, and otherwise output `false`.

Clearly the first problem is at least as hard as the second since being able to find a solution implies you can decide if there exists a solution. But the converse also holds; an algorithm that can decide if a Boolean formula is satisfiable can be used to give an algorithm that also constructs the satisfying assignment (see Exercise 9). We refer to the above problems as that of **Boolean satisfiability**.

To put an algorithmic discussion about satisfiability on firm ground, it is worth establishing the following parameters. Each Boolean formula $f(x_1, \ldots, x_n)$ has a well-defined number of variables, n. We also define the **size** as the number of symbols (parentheses, variables, or operators) required to write it out. Up to constant factors, this is the same as the total number of times that a variable appears in the formula, counted with repetition. (Why?) We generally don't worry about constants so these two quantities are essentially equivalent. Note that the size as defined above corresponds to the input size from an algorithmic point of view.

A very basic aspect of these problems is that, given a potentially satisfying assignment $x \in \{0, 1\}^n$, it is very easy for us to verify that the answer is correct. We simply plug in the values and work through the Boolean algebra to see if the answer comes out as 0 or 1. (Here we would evaluate the innermost parentheses first, and work our way out. Or, in terms of the tree formulation, we work from the leaves up to the roots.) This can be done in linear time in the size of the formula. Because we can efficiently verify a solution, we have at least one algorithm at our disposal: brute force. We can enumerate all 2^n possible assignments to $x \in \{0, 1\}^n$, and for each x, check if it satisfies the formula. This is not an efficient, but a correct algorithm nonetheless.

At a high level, Boolean satisfiability is looking for something that we can recognize (efficiently) when we see it. Any problem of this format is called a **search problem**. The over/under game is also a search problem.

Whereas the over/under game is easy, it is challenging to come up with a (provably) polynomial time algorithm for Boolean satisfiability – even though both problems are of the same "search" metatype. Why is one so much harder than the other? Perhaps one reason is that Boolean satisfiability is so much more expressive. We have already seen various logical statements encoded as Boolean formulas, and it is clear that they can be combined in myriads of ways to form complex thoughts. If we had a very good algorithm for solving Boolean formulas, then we can effectively resolve any question that is logically posed! The consequences – practical, philosophical, and mathematical – would be profound.

3.1 CNF

If Boolean formulas are too general to expect a good algorithm, then perhaps we can restrict ourselves to a special class of Boolean formulas that might be easier. An important class of SAT formulas are those in **conjunctive normal form (CNF)**. These are the formulas that are a conjunction (i.e, an "and") of disjunctions ("or's"), possibly with negations on the variables inside the disjunctions. For example, the following formula is in CNF.

$$\begin{aligned} f(x_1, x_2, x_3, x_4, x_5) = {} & (\bar{x}_1 \vee x_2 \vee \bar{x}_3) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee x_3) \wedge \\ & (x_1 \vee x_4 \vee \bar{x}_5) \wedge (x_1 \vee \bar{x}_4 \vee x_5) \wedge \\ & (x_2 \vee \bar{x}_3 \vee x_4 \vee \bar{x}_5) \wedge (x_2 \vee \bar{x}_3 \vee \bar{x}_4 \vee x_5) \wedge \\ & (\bar{x}_2 \vee x_3 \vee x_4 \vee \bar{x}_5) \wedge (\bar{x}_2 \vee x_3 \vee \bar{x}_4 \vee x_5). \end{aligned}$$

One example of a satisfying solution is the bit string 01011 satisfying the above. (Can you find another?)

The conjunctive normal form appears to clean up Boolean formulas quite a bit. As a tree, a formula in CNF would be almost flat, with one node representing the overarching conjunction, and a child for each disjunctive clause. But this is only a superficial observation. What we really want to know is whether conjunctive normal forms are meaningfully less expressive than Boolean formulas.

Above we showed how to implement assignment and conditional branching with Boolean formulas. As a test case, let us see if we can derive equivalent formulas in CNF.

1. **Assignment in CNF.** Suppose we wanted to model the constraint "$z = x$" in CNF. We have already observe that this is encoded by $(z \wedge x) \vee (\bar{z} \wedge \bar{x})$, but this is not a CNF. To this end,

$$\begin{aligned} (z \wedge x) \vee (\bar{z} \wedge \bar{x}) & \overset{(a)}{=} (z \vee (\bar{z} \wedge \bar{x})) \wedge (x \vee (\bar{z} \wedge \bar{x})) \\ & \overset{(b)}{=} (z \vee \bar{z}) \wedge (z \wedge \bar{x}) \wedge (x \vee \bar{z}) \wedge (x \vee \bar{x}) \\ & \overset{(c)}{=} (z \vee \bar{x}) \wedge (x \vee \bar{z}); \end{aligned}$$

   a CNF, as desired. Here (a) and (b) are by the distributive property. (c) drops the tautologies $(x \vee \bar{x})$ and $(z \vee \bar{z})$.

2. **Conditional branching in CNF.** Suppose we want to model the boolean statement "if x then y else z". We have

$$\begin{aligned} (x \wedge y) \vee (\bar{x} \wedge z) & \overset{(d)}{=} (x \vee \bar{x}) \wedge (y \vee \bar{x}) \wedge (x \vee z) \wedge (y \vee z) \\ & \overset{(e)}{=} (y \vee \bar{x}) \wedge (x \vee z) \wedge (y \vee z); \end{aligned}$$

   a CNF, as desired. Here (d) is by the distributive property and (e) drops the tautology $(x \vee \bar{x})$.

For at least the above two cases, CNF is just as expressive as general Boolean formulas. But these are just examples and do not constitute a complete proof. A much stronger theorem that truly establishes their (algorithmic) equivalence is as follows.

Theorem 1. *Given any Boolean formula $f(x_1, \ldots, x_n)$ of n variables and size m, in polynomial time, one can construct a Boolean formula $g(x_1, \ldots, x_n, x_{n+1}, \ldots, x_p)$ with $p = O(m + n)$ variables and size $O(m + n)$ such that f is satisfiable iff g is satisfiable. Therefore, a polynomial time algorithm deciding if a Boolean formula in CNF is satisfiable implies a polynomial time algorithm for deciding if a (general) Boolean formula is satisfiable.*

The best way to see this is to prove it yourself. See Exercise 5. (Yes, I believe you can do it.)

For (a wild and crazy) example, in the following formula, we convert the Boolean formula $f(x_1, x_2, x_3, x_4)$ for bitwise addition into a CNF formula g with 15 variables.

$$
\begin{gathered}
g(x_1, \dots, x_{15}) = \\
(\bar{x}_{14} \vee x_1) \wedge (\bar{x}_{14} \vee x_2) \wedge (x_{14} \vee \bar{x}_1 \vee \bar{x}_2) \wedge (\bar{x}_{15} \vee x_3) \wedge (\bar{x}_{15} \vee \bar{x}_4) \wedge (x_{15} \vee \bar{x}_3 \vee x_4) \wedge \\
(\bar{x}_6 \vee x_1) \wedge (\bar{x}_6 \vee x_2) \wedge (x_6 \vee \bar{x}_1 \vee \bar{x}_2) \wedge (x_7 \vee \bar{x}_1) \wedge (x_7 \vee \bar{x}_2) \wedge (\bar{x}_7 \vee x_1 \vee x_2) \wedge \\
(\bar{x}_8 \vee x_4) \wedge (\bar{x}_8 \vee x_7) \wedge (x_8 \vee \bar{x}_4 \vee \bar{x}_7) \wedge (x_9 \vee \bar{x}_1) \wedge (x_9 \vee \bar{x}_2) \wedge (\bar{x}_9 \vee x_1 \vee x_2) \wedge \\
(\bar{x}_{10} \vee \bar{x}_9) \wedge (\bar{x}_{10} \vee \bar{x}_4) \wedge (x_{10} \vee x_9 \vee x_4) \wedge (x_{11} \vee \bar{x}_8) \wedge (x_{11} \vee \bar{x}_{10}) \wedge (\bar{x}_{11} \vee x_8 \vee x_{10}) \wedge \\
(\bar{x}_{12} \vee \bar{x}_6) \wedge (\bar{x}_{12} \vee x_{11}) \wedge (x_{12} \vee x_6 \vee \bar{x}_{11}) \wedge (x_{13} \vee \bar{x}_5) \wedge (x_{13} \vee \bar{x}_{12}) \wedge (\bar{x}_{13} \vee x_5 \vee x_{12}).
\end{gathered}
$$

The formula g above has the following more precise guarantee. For $x_1, x_2, x_3, x_4 \in \{0, 1\}$, we have $f(x_1, x_2, x_3, x_4) = 1$ iff there exists $x_5, \dots, x_{15} \in \{0, 1\}$ such that $g(x_1, \dots, x_{15}) = 1$.

An even more special case of CNF is where every clause has exactly three variables. Satisfiability problems of this precise form are called **3SAT**. One might hope for a better algorithm for 3SAT, but again we are thwarted by a polynomial time reduction.

Theorem 2. *Given any Boolean formula $f(x_1, \dots, x_n)$ of n variables and size m, in polynomial time, one can construct a Boolean formula $g(x_1, \dots, x_n, x_{n+1}, \dots, x_p)$ with $p = O(m)$ variables and size $O(m + n)$ such that f is satisfiable iff g is satisfiable. In particular, a polynomial time algorithm deciding 3SAT implies a polynomial time algorithm for SAT.*

The proof is left as Exercise 6.

3.2 Searching for answers

We have danced around but failed to resolve the following question.

Is there a polynomial time algorithm for deciding if a boolean formula is satisfiable?

This is a huge open question, the most important question in theoretical computer science and arguably in all of mathematics.

The general feeling is no, insofar as such an algorithm would be too good to be true. If the answer was yes, then we would have an efficient algorithm for *anything* that can be expressed in boolean logic. But we have already seen that boolean logic is very expressive – maybe too expressive to expect a computer to efficiently process.[4]

As daunting as the question posed above might seem, we have asked and resolved big questions before. Consider the following question posed by David Hilbert and Wilhelm Ackermann in 1928, called the *Entscheidungsproblem*. Paraphrasing, and omitting mathematical details, the question is roughly:

Can a computer compute (i.e., decide) any logical problem?

For example, a computer can compute whether a boolean formula is satisfiable, by brute-forcing through all the possible solutions. (This is inefficient, but the point is that it works.) Is there anything a computer cannot compute, by brute-force or by other means? Remarkably this question

[4]This description somewhat simplifies the debate; there are other, more nuanced opinions.

has been answered. The answer is *yes*, there are indeed *incomputable problems*, and it was proven independently by Church [1] and Turing [3]. We stress that the *Entscheidungsproblem* was posed *before* computers existed, purely in terms of mathematical logic. A major contribution in Turing's approach was to define a simplified model of a computer (sometimes called a Turing machine), and then restate the *Entscheidungsproblem* in terms of this model. Turing's initial model for a computer (inspired by a typewriter) had a huge influence on modern computer design [2] and continues to serve as a useful model of computation.

4 Exercises

Exercise 1. 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, 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 2. Let $A[1..n] \in \mathbb{R}^n$ be an array of n numbers. 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). Design and analyze an algorithm (as fast as possible) that returns the first k smallest numbers in A in sorted order.

Exercise 3. 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 $d \leq c$.

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. Design and analyze an algorithm (as fast as possible) for counting the number of comparable pairs.

Exercise 4. Show that any Boolean formula with n variables and total size m can be converted to a Boolean formula with n variables and total size $O(m)$ such that every clause contains at most two sub-clauses or variables. That is, in the tree representation of the formula, each $\wedge$ and $\vee$ would have two children. For example, the formula for XOR, $f(x_1, x_2) = (\bar{x}_1 \wedge x_2) \vee (x_1 \wedge \bar{x}_2)$, is in this form.

Exercise 5. Show that any boolean formula with n variables and total size m can be converted to a boolean formula in CNF with $O(m + n)$ variables, $O(m)$ clauses, and total size $O(m)$.[5][6]

Exercise 6. Show that any CNF with n variables and total size m can be converted into a 3-SAT problem with $O(m)$ clauses and $O(m + n)$ variables.

Exercise 7. Let $k \geq 3$. Recall that the k-SAT problem is the special case of SAT where $f(x_1, \ldots, x_n)$ is a CNF with exactly k-variables per clause.

1. Show that a polynomial time algorithm for k-SAT implies a polynomial time algorithm for $(k + 1)$-SAT.
2. Show that a polynomial time algorithm for $(k + 1)$-SAT implies a polynomial time algorithm for k-sat.

[5]It might simplify things to first apply the preceding exercise.

[6]As a second (appropriately vague) hint, how do you express "$a = b \vee c$" and "$a = b \wedge c$" in CNF?

Exercise 8. Recall that a **palindrome** is a string spelled the same forward or backwards, such as "racecar". Likewise we say that a bit string $x \in \{0,1\}^n$ if $x_i = x_{n+1-i}$ for all indices $i = 1, \ldots, n$. Describe a SAT formula in CNF of size $O(n)$ for a Boolean formula $f : \{0,1\}^n \to \{0,1\}$ that accepts palindromes (i.e., $f(x) = 1$ iff x is a palindrome).

Exercise 9. Suppose one had access to a polynomial time algorithm that solves the decision version of Boolean satisfiability (outputting `true` or `false` depending on whether there *exists* a satisfying assignment.). Show how to use this algorithm to obtain a polynomial time algorithm for the constructive version of Boolean satisfiability, outputting a satisfying assignment if one exists.

Exercise 10. In the **Max 2-SAT** problem, one is given a CNF $f(x_1, \ldots, x_n)$ with *exactly* two variables per clause (i.e., a 2-CNF). The goal is to find an assignment that satisfies *as many* clauses as possible. (Note that we are not interested in knowing if f is exactly satisfiable.) In the decision version of the problem, we are given a 2-CNF formula f and an integer $k \in \mathbb{N}$, and the goal is to decide if there exists an assignment that satisfies at least k clauses in f.

Either (a) show that a polynomial time algorithm for Max 2-SAT implies a polynomial time algorithm for boolean satisfiability (in general), or (b) design and analyze a polynomial time algorithm that, given a 2-CNF formula f, computes an assignment that maximizes the number of satisfied clauses.

Exercise 11. In the **1-in-3-SAT** problem, one is given a 3-CNF $f(x_1, \ldots, x_n)$. The goal is to find a satisfying assignment such that every disjunctive clause (with three variables) is satisfied by *exactly* one of the variances.

Either (a) show that a polynomial time algorithm for 1-in-3-SAT implies a polynomial time algorithm for SAT, or (b) design and analyze a polynomial time algorithm for 1-in-3-SAT.

Exercise 12. A Boolean formula is in **disjunctive normal form (DNF)** if it is is disjunction of conjunctions. For example, for formula

$$f(x, y, z) = (x \land y) \lor (\bar{x} \land z),$$

which models "if x then y else z", is in disjunctive normal form. Note that by DeMorgan's law,

$$(a \lor b) \land (c \lor d) = (a \land c) \lor (a \land d) \lor (b \land c) \lor (b \land d),$$

so any CNF formula can be converted to a DNF formula.

Either (a) show that a polynomial time algorithm for DNF satisfiability implies a polynomial time algorithm for boolean satisfiability (in general), or (b) design and analyze a polynomial time algorithm for deciding if a DNF formula is satisfiable.

Exercise 13. Let $f : \{0,1\}^n \to \{0,1\}$ be a Boolean function on n variables. Prove that there is a CNF $\varphi(x_1, \ldots, x_n)$ of finite size such that $f = \varphi$.

References

[1] Alonzo Church. "An unsolvable problem of elementary number theory". In: *American Journal of Mathematics* 58.2 (1936). Available at `http://phil415.pbworks.com/f/Church.pdf`, pp. 345–363.

[2] John von Neumann. “First Draft of a Report on the EDVAC”. In: *IEEE Ann. Hist. Comput.* 15.4 (Oct. 1993), 27–75. URL: `https://en.wikipedia.org/wiki/First_Draft_of_a_Report_on_the_EDVAC`.

[3] A. M. Turing. “On Computable Numbers, with an Application to the Entscheidungsproblem”. In: *Proceedings of the London Mathematical Society* s2-42.1 (Jan. 1937), pp. 230–265. URL: `https://academic.oup.com/plms/article-pdf/s2-42/1/230/4317544/s2-42-1-230.pdf`.