Great Ideas of Theoretical Computer Science

Instructor: Sheng Zhong

June 27, 2023

1 Turing Machine and Halting

What is a computer? This fundamental question was answered by British mathematician Alan Turing, the father of computer science. He created a model called Turing Machine to represent *any possible* computer in the world. This model still works today, with a handful exceptions in laboratories, like quantum computers.

A Turing Machine is an automaton working with a tape. The tape is infinitely long, but filled with a bit string of finite length as input, and the read-and-write head points at the leftmost bit of input at the very beginning. The automaton has a single register that stores its current state. In each step of computation, the automaton reads the content of the current cell from the tape, and then uses it together with the current state to decide the action to take. The action to take includes three parts:

- Decides the new state.
- Decides what to write (in the current cell).
- Decides how to move the read-and-write head. There are three options: moving to the left cell, moving to the right cell, and staying in the current location.

There is one state called *Termination*. Once this state is reached, the Turing Machine terminates its computation and whatever left on the tape is the output.

Formally, we define Turing Machine below.

Definition 1.1 *A Turing Machine is a tuple* (Γ, S, W, M)*, where*

- Γ *is a finite set such that Start*, *Termination* $\in \Sigma$.
- $S : \Gamma \times \{0, 1, \star\} \to \Gamma$ *is a mapping that decides the new state, where* $\star$ *is a symbol representing an empty cell.*
- $W : \Gamma \times \{0, 1, \star\} \to \{0, 1, \star\}$ *is a mapping that decides what to write.*
- $M : \Gamma \times \{0, 1, \star\} \to \{-1, 0, 1\}$ *is a mapping that decides where to move the read-and-write head. Here* -1 *means moving to the left,* 1 *to the right, and* 0 *staying at the current location.*

Bear in mind that a Turing Machine is nothing but a formal representation of an algorithm. Hence, we can construct a Turing Machine for whatever algorithm in mind.

Example 1.1 *Construct a Turing Machine that can reverse a* 9*-bit input.*

This is a very easy exercise and thus we skip the solution. Just encode in the state what has been read/written. For example, use a state R_{010} to represent that 010 has just been read as input; similarly, use a state W_{010} to represent that 010 has just been written as output. Then we can easily define the three mapping needed. For example, $S(\text{Start}, 0) = R_0$, $M(R_0, 1) = 1, \ldots$ You just need to practice writing things formally.

Example 1.2 *Construct a Turing Machine that can reverse any finitely long bitstring.*

You might get seriously wrong if you are thinking along the lines of Example 1.1. Here the key point is that you can't encode in the state what has been read because you have no idea how long the input is—the set of states is finite and thus you can do the encoding only for input with a fixed upper bound of length.

To solve this problem, the main idea is to use $\star$ as the landmark. Assuming, *e.g.*, the input is 01, the Turing Machine takes the following steps:

1. Read the rightmost bit of the input, 1, and notice that there is something (0) to its left. In other words, this bit is not the last bit to copy.
2. Memorize 1, erase it from the tape (*i.e.*, replace it with $\star$), and move to the right.
3. When a $\star$ is seen, write 1, the bit we just memorized.
4. Move back to the left, read the remaining bit 0, and notice there is nothing to its left. Now we realize that it is the last bit to copy. Once we finish copying it, we will never need to come back again.
5. Memorize 0, erase it from the tape, and move to the right.
6. Once we have passed the 1 we wrote earlier, and see a $\star$, write down 0. Since this is the last bit to copy, we are done.

Do you see the above constructed Turing Machine is essentially an algorithm? Of course, nobody is interested in writing algorithms in such an awkward way. We are interested in Turing Machines only because they are an abstract model of computers that enable theoretical studies. We never use them to solve algorithmic problems in practice.

In many cases, you see Turing Machines with more than one tape. By convention, we assume these machines always have both their input and their output on the first tape.

Definition 1.2 *For* $k > 1$*, a* k*-tape Turing Machine is a tuple* (Γ, S, W, M)*, where*

- Γ *is a finite set such that Start*, *Termination* $\in \Sigma$.

- $S : \Gamma \times \{0, 1, \star\}^k \to \Gamma$ *is a mapping that decides the new state, where* $\star$ *is a symbol representing an empty cell.*
- $W : \Gamma \times \{0, 1, \star\}^k \to \{0, 1, \star\}^k$ *is a mapping that decides what to write on each tape.*
- $M : \Gamma \times \{0, 1, \star\}^k \to \{-1, 0, 1\}^k$ *is a mapping that decides where to move the read-and-write heads. Here* -1 *means moving to the left,* 1 *to the right, and* 0 *staying at the current location.*

It turns out that having more tapes would not enable the Turing Machine to solve more problems.

Theorem 1.1 *For any* k*-tape Turing Machine* T*, there is a (one-tape) Turing Machine* T' *such that* T' *halts on an input* x *if and only if* T *halts on* x*, and that in case of halting,* T' *and* T *generates exactly the same output on the same input.*

Why is this true? Because you can essentially simulate k tapes with a single tape. For example, you might want to use $2k$ consecutive cells to simulate a cell from each of the k tapes, and to indicate whether each of the k heads is at this location. Therefore, a simulated reading of all the k heads would involve the actual head moving along the real tape, finding the locations of all simulated heads, reading the contents of the simulated cells ... A proof of (a variant of) Theorem 1.1 can be found in [9]. Interested readers are encouraged to read it.

Similarly, restricting tapes to be infinite in one direction only, or having some read-only tapes in addition to normal tapes, or using a larger set as the alphabet, usually would not affect the computability of problems. Therefore, while you can see many different definitions of Turing Machines across different books, these definitions are usually equivalent to each other.

Example 1.3 *We say* T *is a ternary Turing Machine if the alphabet is* $\{0, 1, 2, \star\}$ *in stead of* $\{0, 1, \star\}$ *. Show that for any ternary Turing Machine* T*, there is a Turing Machine* S*, such that* T *halts on an input* x *if and only if* S *halts on input* x' *(where* x' *is obtained by writing each ternary digit in the binary form, e.g.,* $x = 0121 \to x' = 00011110$*), and that in case of halting,* T *generate exactly the same output on input* x *as* S *generates on input* x'*.*

The key idea is to use a pair of cells of S's tape to represent each cell of T's tape.

Suppose $T = (\Gamma_T, S_T, W_T, M_T)$. We construct $S = (\Gamma_S, S_S, W_S, M_S)$. For each $s \in \Gamma_T$, there are a lot of corresponding states in Γ_S. For example, there should be a state representing that T is in state s and the reading of the current pair on S just starts. There should also be a state representing that T is in state s, the first cell of the current pair is 0, and we are reading the second cell. Yet another state represents that T is in state s, the current pair is $(0, 1)$–which means the current cell on T is 1, and we just starts writing ...

Essentially, S remembers T's state, as well as the incomplete read/write T is doing, so that the work of S on a pair of cells is equivalent to that of T on a single cell.

The full formal specification of S is left for homework.

While the above discussions of (variants of) Turing Machines might be interesting, the Turing Machines we studied so far differ from the computers we use today in an important aspect: Each of them solves a fixed algorithmic problem only. In other words, they do not support *programming*. Now we fix this issue.

Definition 1.3 *We say a Turing Machine T simulates the work of another Turing Machine S with program P if both of the following two requirements are met:*

- *For every finite-length bitstring x, T halts on input $x \star P$ if and only if S halts on x;*
- *In case of halting, the output of T on input $x \star P$ is identical to that of S on input x.*

Definition 1.4 *A Turing Machine T is said to be* Universal *if for every Turing Machine S, there is a program P such that T simulates S with program P.*

Apparently, today's computers are closer to Universal Turing Machines, rather than the single-task Turing Machines we studied earlier. But do Universal Turing Machines exist? The good news is that they do.

Theorem 1.2 *There exists a Universal Turing Machine.*

A proof of Theorem 1.2 can be found in standard textbooks like [1]. The main idea is that we can construct a Universal Turing Machine by adding two tapes to a regular Turing Machine, one for storing the mappings S, W, M of the simulated Turing Machine, the other for storing the current state of the simulated Turing Machine. Since adding extra tapes do not enhance Turing Machines, we can argue that without these extra tapes there is also a Universal Turing Machine.

Given the existence of Universal Turing Machines, we need to study what problems these machines can solve. In reality, computational problems vary greatly. You can hardly find a general framework for all of them. However, in the theory of computation, every computational problem is modeled as the *decision* of a certain *language*.

Suppose Σ is an alphabet (a finite set with $\star \in \Sigma$). A finite sequence of symbols from $\Sigma_0 = \Sigma - \{\star\}$ is called a word. We define $\Sigma_0^\star$ to be the set of all words (including the empty word) over Σ. A subset of $\Sigma_0^\star$ is called a *language*. We can attempt to use a Turing Machine T to *decide* a language L: Put a word x on T's tape as input, run T, and expect T's output to be 1 when $x \in L$, to be 0 otherwise. Ideally, T does exactly what we expect it to do. In this case, the language L is said to be recursive.

Definition 1.5 *A language $L \in \Sigma_0^\star$ is said to be* recursive *if there is a Turing Machine that halts on every input x, outputing 1 when $x \in L$, outputing 0 otherwise. A language $L \in \Sigma_0^\star$ is said to be* recursively enumerable *if there is a Turing Machine that halts on every input $x \in L$; in case of halting, it outputs 1 when $x \in L$, and outputs 0 otherwise.*

Clearly, a recursive language is recursively enumerable, but the converse is not true.

When you compare recursive languages with recursively enumerable languages, you see the difference is in whether the Turing Machine halts on all inputs.[1] So we are interested in the following fundamental question (called *Turing Halting Problem*): Given a Turing Machine T, and a word x, can we decide whether T halts on input x? In other words, is the language of Turing Halting Problem $\{(T, x) | T \text{ halts on input } x\}$ recursive? Unfortunately, the answer is negative.

[1] But given this difference, why is a recursively enumerable language is called "recursively enumerable"? Maybe we have a way to "enumerate" all its words? The answer is yes. Please think of how we can enumerate all its words using a Turing Machine.

Theorem 1.3 *The language of Turing Halting Problem is not recursive.*

A somewhat stronger result is Rice Theorem. We say a property of language is trivial if either all languages have the property, or none of them has it. Now consider a nontrivial property P of (the language accepted by) Turing Machine. Given the description of a Turing Machine, can we decide whether it [2] has the property P? The answer is again negative.

Theorem 1.4 *(Rice) The language of any nontrivial property of (the language accepted by a) Turing Machine is not recursive.*

You might wonder what these theorems mean in reality. Do they really matter? Yes, they do. They tell us that, even if we did not mind the time, space, and communications resources consumed by computation, our computers would still have very limited computational power. There are some problems (in fact, too many problems) that can never be solved by a computer, despite the widely existent over-optimistic belief in computers.

Problems

Problem 1.1 *Construct a Turing Machine whose output is the length of its input. For instance, if the input is* 0110100111*, then the output is* 1010*.*

Problem 1.2 *Write down the full formal specification of S for solution of Example 1.3.*

Problem 1.3 * *We say T is a Turing Machine with one-way tape if its tape is one-way infinite to the right, i.e., its tape has a left end. Show that for any Turing Machine S, there is a Turing Machine T with one-way tape, such that T halts on an input x if and only if S halts on x, and that in case of halting, T and S generate exactly the same output on the same input.*

Problem 1.4 *Show that a language $L \in \Sigma_0^\star$ is recursively enumerable if and only if there is a Turing Machine that halts on every input $x \in L$ and does not halt otherwise.*

Problem 1.5 * *(Final Exam 2020-2) Please decide whether the following proposition is true or false. Prove your answer.*

Proposition 1.1 *For any (one-tape) Turing machine T, there exists another Turing Machine T' such that T' halts on input x if and only if T does not halt on this input.*

Problem 1.6 *(Final Exam 2021-2) For any Turing Machine T, we define a function f_T: If T halts and outputs* 1 *on input x, then $f_T(x) = 1$; otherwise, $f_T(x) = 0$. Is the language $\{(T, x, y) | f_T(x) = f_T(y)\}$ recursive? Prove your answer.*

[2] In fact, the language it accepts. Note that we do not consider nontrivial properties of the Turing Machine that does not belong to the language it accepts. For example, we do not consider which input leads to the shortest running time of the Turing Machine. While this is a non-trivial property of the Turing Machine, it is not a property of the language accepted.

Problem 1.7 *(Final Exam 2022-2) We say a Turing Machine T* runs faster than *another Turing Machine S on input x, if one of the following requirements is satisfied:*

- *T halts on input x in t steps, S halts on input x in s steps, and $t < s$;*
- *T halts on input x, but S does not halt on input x.*

We say a Turing Machine T runs as fast as *another Turing Machine S on input x, if one of the following requirements is satisfied:*

- *T halts on input x in t steps, S halts on input x in s steps, and $t = s$;*
- *Neither T nor S halts on input x.*

(a) Is the language $\{(T, S, x)|T$ runs faster than S on input $x\}$ recursively enumerable? Prove your answer.
(b) Is the language $\{(T, S, x)|T$ runs faster than, or as fast as S on input $x\}$ recursively enumerable? Prove your answer.

2 Karatsuba Algorithm and the Master Theorem

We learned multiplication in primary school. When we do 12×34, we essentially calculate $1 \times 2 \times 100 + (2 \times 3 + 1 \times 4) \times 10 + 2 \times 4$. In general, we use the formula

$$(\sum_{i=0}^{n-1} x_i \cdot b^i) \cdot (\sum_{i=0}^{n-1} y_i \cdot b^i) = \sum_{i=0}^{n-1} (\sum_{j=0}^{i} x_j y_{i-j}) \cdot b^i + \sum_{i=n}^{2(n-1)} (\sum_{j=i-(n-1)}^{n-1} x_j y_{i-j}) \cdot b^i,$$

where $b = 10$ for decimal calculations, while b is often equal to some power of 2 in computers. This naive algorithm needs to do n^2 "basic"multiplications (*i.e.*, multiplications over $[0, b-1]$) in order to multiply two number from $[0, b^{n-1} - 1]$. Naturally, one might suspect this is the best we could do, because there seems no obvious way to do it faster.

Surprisingly, Russian mathematician Anatoly Karatsuba proposed a counter-intuitive algorithm that easily beats the naive algorithm for multiplication. The idea is quite simple: when you do 12×34, you should not waste your time doing two "basic" multiplications for $(2 \times 3 + 1 \times 4)$. In stead, you should just calculate $(1 + 2) \times (3 + 4) - 1 \times 4 - 2 \times 3$, which is identical to $(2 \times 3 + 1 \times 4)$. Noticing that you have to calculate 1×4 and 2×3 for other parts of the expression anyway, for this specific part you just need to do one "basic" multiplication! Therefore, we have reduced the total amount of work.

Below we present a straightforward recursive algorithm based on Karatsuba's idea.

Algorithm karatsuba(x, y)
if ($x < 2^b$) or ($y < 2^b$)
    return xy;
split x in the middle and get (h_x, l_x);

split y in the middle and get (h_y, l_y);
$r_0 =$ **karatsuba**(l_x, l_y);
$r_1 =$ **karatsuba**$((l_x + h_x), (l_y + h_y))$;
$r_2 =$ **karatsuba**(h_x, h_y);
return the number $r_2 \cdot 2^{2b} + (r_1 - r_2 - r_0) \cdot 2^b + r_0$.

Now, assume that each "basic" multiplication is on a couple of b-bit integers, and that the Karatsuba algorithm needs $T(n)$ "basic" multiplications on a couple of n-bit numbers.Assuming the multiplication of $l_x + h_x$ and $l_y + h_y$ requires no more "basic" multiplication than the multiplication of l_x and l_y (or that of h_x and h_y), we get the following equation:[3]

$$T(n) = 3T(\frac{n}{2}). \tag{1}$$

It is trivial to derive from the above equation that $T(n) = 3^{\log_2 \frac{n}{b}} T(b) = (\frac{n}{b})^{\log_2 3} \approx n^{\log_2 3}$, assuming b is small and n is big. In contrast, the naive algorithm needs n^2 "basic" multiplications. For large n, the difference can be very significant.

The Karatsuba algorithm is the beginning of a search for formulae that helps reduce the computational cost of multiplication. Many more formulae have been found since then. For example, in 2009, Fan and his collaborators developed a number of Karatsuba-like formulae, including some asymmtric ones. Interested readers can refer to [4].

On the other hand, solving recurrences like Eq. (1) is also very interesting. While there are some powerful techniques for solving recurrences, it is not always possible to figure out the precise solution(s) of a particular recurrences. Hence, one possibility is to look at so called asymptotic solutions, rather than precise solutions. To explain this idea, below we review some notations.[4]

Definition 2.1 *Let $T(n)$ and $f(n)$ be positive valued functions defined on positive integers. We say $T(n) = O(f(n))$ if there exist $N > 0$ and $M > 0$ such that for every $n > N$, $T(n) \leq Mf(n)$. We say $T(n) = \Theta(f(n))$ if $T(n) = O(f(n))$ and $f(n) = O(T(n))$.*

Intuitively, the big-O notation for functions defined above is analogous to $\leq$ for real numbers, meaning that $T(n)$ grows no faster than $f(n)$, while the big-Θ for functions is similar to $=$ for real numbers, meaning that $T(n)$ grows *asymptotically* as fast as $f(n)$.

These notations are somewhat unusual in that they do not mean what they appear to mean. Specifically, do not take $T(n) = O(f(n))$ as an equation and try to manipulate it like an equation. For example, you can't derive $O(f(n)) = T(n)$ from $T(n) = O(f(n))$. In fact, $O(f(n)) = T(n)$ is meaningless and we should never write anything like that.

Example 2.1 *Prove the following statements.*

[3]Hereafter, we are sloppy in writing things like $T(\frac{n}{2})$ because T is defined on positive integers but $\frac{n}{2}$ may not be an integer. However, all our discussions hold when we replace $\frac{n}{2}$ with $\lfloor \frac{n}{2} \rfloor$ and do some minor modifications. Hence, to keep it simple, we use expressions like $T(\frac{n}{2})$ as if they were legal.

[4]These notations are used both in algorithm analysis and in mathematical disciplines like number theory. There are minor differences between the interpretations of these notations in different fields. We follow the tradition of algorithm analysis.

1. *If $d \leq d'$, any degree-d polynomial is $O(n^{d'})$.*
2. *Any degree-d polynomial is $\Theta(n^d)$.*
3. *If $f(n) = O(g(n))$ and $g(n) = O(h(n))$, then $f(n) = O(h(n))$.*
4. *Any polynomial $f(n)$ grows no faster than the exp function, i.e., $f(n) = O(e^n)$.*
5. *Any logarithm function $f(n) = a \log_b n$ ($a, b > 0$) grows no faster than a polynomial, i.e., $f(n) = O(n^\epsilon)$ for all $\epsilon > 0$.*

Solution: We skip the first three and show the last two. For all $d > 0$,

$$e^n = \sum_{i=0}^{\infty} \frac{n^i}{i!} \Rightarrow n^d < d! e^n.$$

Hence, $n^d = O(e^n)$, and thus by the second and third statements in this example, we get that any degree-d polynomial is $O(e^n)$.

In $n = O(e^n)$, we replace n with $\epsilon \ln n$, and get that $\epsilon \ln n = O(e^{\epsilon \ln n}) = O(n^\epsilon)$. Since $a \log_b n = \frac{a}{\epsilon \ln b} \cdot \epsilon \ln n = O(\epsilon \ln n)$, by the third statement above, we get that $a \log_b n = O(n^\epsilon)$. □

Example 2.2 *Karatsuba algorithm is a divide-and-conquer algorithm. It divides the original problem into smaller ones, which are easier to solve. Please design a divide-and conquer algorithm for finding the kth smallest element in an n-array. Note that a naive algorithm sorts the elements before finding the kth smallest one and thus requires $O(n \log n)$ comparisons. Can you do better?*

Solution: We examine the array from the beginning, and always keep the k smallest elements we have seen so far, and make sure these k elements are always *sorted*. When we are done, we have found the k smallest elements, not just the kth smallest one, of the entire array.

Easy to see we need only $O(n \log k)$ comparisons. □

To solve recurrence relations that look like Eq. (1), in general, we have the following Master Theorem:

Theorem 2.1 *(Master Theorem for Divide-and-Conquer Recurrence) A recurrence relation $T(n) = aT(\frac{n}{b}) + f(n)$ has the following solution:*

- *If $f(n) = \Theta(n^c)$ where $c < \log_b a$, then $T(n) = \Theta(n^{\log_b a})$.*
- *If $f(n) = \Theta(n^{\log_b a} \log^k n)$ ($k \geq 0$), then $T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$.*

- *If* $f(n) = \Theta(n^c)$ *where* $c > \log_b a$*, and if* $f(n)$ *meets the* regularity condition,[5] *i.e.,* $af(\frac{n}{b}) \leq df(n)$ *for some* $d < 1$ *and sufficiently large* n*, then* $T(n) = \Theta(n^c)$.

At a first look, Theorem 2.1 might be confusing. What do the conditions after those "if"s mean? How do you get the result? Actually, the intuition behind is quite simple. Notice that the right hand side of the recurrence has only two terms, the first being $aT(\frac{n}{b})$ and the second being $f(n)$. Which of them grows faster? This faster growing term dominates the sum.

The three cases considered in Theorem 2.1 correspond to three possibilities: that the first term grows significantly faster, that the two term grows at very roughly "similar" rates, and that the second term grows significantly faster, respectively. Imagine we are in the first case, and we are to do an induction on n for proof. Applying the induction hypothesis to $T(\frac{n}{b})$, we get that $T(n) \leq aM_1(\frac{n}{b})^{\log_b a} + M_2 n^c = M_1 n^{\log_b a} + M_2 n^c$. Since $c < \log_b a$, for large n clearly we ignore the second term[6] and get that $T(n) \leq M_1 n^{\log_b a}$, or, equivalently, $T(n) = O(n^{\log_b a})$. On the other hand, since $f(n)$ is positive valued, the solution to $T(n) = aT(\frac{n}{b}) + f(n)$ grows no slower than the solution to $T(n) = aT(\frac{n}{b})$. Given that the latter is $\Theta(n^{\log_b a})$, the former has to be $\Theta(n^{\log_b a})$, too.

In the second case, also imagine that we are doing an induction. What we get is that

$$\begin{aligned} T(n) &\leq aM_1(\frac{n}{b})^{\log_b a}\log^{k+1}\frac{n}{b} + M_2 n^{\log_b a}\log^k n \\ &= M_1 n^{\log_b a}(\log n - \log b)^{k+1} + M_2 n^{\log_b a}\log^k n \\ &\leq M_1 n^{\log_b a}\log^{k+1} n + M_2 n^{\log_b a}\log^k n \end{aligned}$$

Again, we are ignoring the second term because it is much smaller than the first term for large n. So we get that $T(n) \leq M_1 n^{\log_b a}\log^{k+1} n$, or, equivalently, $T(n) = O(n^{\log_b a}\log^{k+1} n)$. Note that the same approach would *not* produce $T(n) = O(n^{\log_b a}\log^k n)$ for us, because we would end up deriving $T(n) \leq (M_1 + M_2)n^{\log_b a}\log^k n$ from the induction hypothesis $T(\frac{n}{b}) \leq M_1(\frac{n}{b})^{\log_b a}\log^k(\frac{n}{b})$, which does not meet the requirement of an induction.

For a similar reason, if we do an induction in the third case, we get that

$$T(n) \leq aM_1(\frac{n}{b})^c + M_2 n^c = (\frac{aM_1}{b^c} + M_2)n^c,$$

which would not allow us to complete the induction.Therefore, we need an additional condition, the *regularity condition*. In this case, roughly speaking,

$$\begin{aligned} T(n) &= aT(\frac{n}{b}) + f(n) = a^2 T(\frac{n}{b^2}) + af(\frac{n}{b}) + f(n) = \ldots \\ &\approx f(n) + af(\frac{n}{b}) + a^2 f(\frac{n}{b^2}) + \ldots \leq f(n) + df(n) + d^2 f(n) + \ldots = \frac{f(n)}{1-d}. \end{aligned}$$

[5] In fact, the regularity condition alone implies the third case condition $c > \log_b a$. Specifically, suppose "sufficiently large" n means $n \geq n_0$. Then we have

$$f(n) \geq \left(\frac{a}{d}\right)^{\log_b(n/n_0)} \min_{1\leq x\leq n_0} f(x) = \left(\frac{n}{n_0}\right)^{\log_b(a/d)} \min_{1\leq x\leq n_0} f(x) = \Theta(n^{\log_b a + \log_b \frac{1}{d}}).$$

Therefore, writing the third case condition $c > \log_b a$ here is actually redundant. We still write it, just to make the statement of Master Theorem more readable.

[6] This is just an intuitive explanation, not a rigorous proof. When writing a proof, you can't do it this way.

A detailed proof of Theorem 2.1 can be found, *e.g.*, in [3].

Example 2.3 *Solve the following recurrences.*

1. $T(n) = 6T(\frac{n}{2}) + n^4$.
2. $T(n) = 10T(\frac{n}{2}) + n^2$.
3. $T(n) = 8T(\frac{n}{2}) + n^3$.
4. $T(n) = 10T(\frac{n}{2}) + \frac{n^2}{\log n}$.

Solution: We skip the first three and solve only the last recurrence. It is easy to see that $10T(\frac{n}{2}) + n < T(n) < 10T(\frac{n}{2}) + n^2$ for sufficiently large n. Using Theorem 2.1, we get that the solutions to both $T(n) = 10T(\frac{n}{2}) + n$ and $T(n) = 10T(\frac{n}{2}) + n^2$ are $\Theta(n^{\log_2 10})$ asymptotically, and thus our solution is also $\Theta(n^{\log_2 10})$. □

Problems

Problem 2.1 *To calculate the product of two polynomials $f(x) = ax + b$ and $g(x) = ux^2 + vx + w$, a naive algorithm needs 6 multiplications of coefficients. How can you do better?*

Problem 2.2 *The first case of Theorem 2.1 uses the following condition: $f(n) = \Theta(n^c)$ where $c < \log_b a$. In the literature, you may find an alternative condition for this case: $f(n) = O(n^{\log_b a - \epsilon})$ for some $\epsilon > 0$. Prove using these two conditions in the theorem are equivalent.*

Problem 2.3 * *Find a recurrence $T(n) = aT(\frac{n}{b}) + f(n)$ such that $f(n) = \Theta(n^c)$ where $c > \log_b a$, but $f(n)$ does not meet the regularity condition.*

Problem 2.4 *(Final Exam 2020-3) Find the solution of recurrence*

$$T(n) = 9T(\frac{n}{3}) + 4T(\frac{n}{2}) + 36T(\frac{n}{6}).$$

Problem 2.5 *(Final Exam 2021-3) Solve*

$$S(n) = 12S(n-1) - 32S(n-2),$$

where

$$S(2) = 4(S(1) + 16).$$

While you might be able to find a precise solution, you are expected to provide an asymptotic solution only.

3 Examples of Randomized Algorithms

You must have learned a good number of algorithms. They are all *deterministic* in the sense that for any given input, the execution of the algorithm is always the same. There is never any change in any step of the algorithm, from one execution to another. Traditional algorithms, *e.g.*, traditional sorting algorithms, all fall into this category.

Recall the quicksort algorithm you have learned.

Algorithm quicksort$(x_1, x_2, \ldots, x_n)$
pick a pivot x_i arbitrarily; let y, z be empty arrays;
if $n > 1$
    for $j = 1$ to n $(j \neq i)$
        if $x_j < x_i$
            append x_j to array y;
        else
            append x_j to array z;
y'=quicksort(y);
z'=quicksort(z);
return the array $y'|x_i|z'$.

Let $T(n)$ be its running time on an n-array. If we were lucky enough to always pick the middle element of the array as pivot, then we would have $T(n) = \Theta(n) + 2T(\frac{n}{2})$. By Theorem 2.1, we get that $T(n) = \Theta(n \log n)$, which seems good. However, in reality, we could end up picking the smallest or the biggest element as pivot. In such worst cases, we would have $T(n) = \Theta(n) + T(n-1)$, which implies that $T(n) = \Theta(n^2)$.

Note that there is no way to guarantee you pick a "good" pivot. So, what can we do if we worry about the worst cases? Can we somehow avoid them? Well, you should notice that *most* elements are pretty good choices for pivot, and thus a *random* selection of pivot might help us avoid the worst cases (with a good probability). With this minor (but significant) modification, we get the randomized version of quicksort.

Algorithm randomquicksort$(x_1, x_2, \ldots, x_n)$
pick a pivot x_i *uniformly at random*; let y, z be empty arrays;
if $n > 1$
    for $j = 1$ to n $(j \neq i)$
        if $x_j < x_i$
            append x_j to array y;
        else
            append x_j to array z;
y'=quicksort(y);
z'=quicksort(z);
return the array $y'|x_i|z'$.

Assuming each iteration in the for-loop takes one unit of time, then for the above randomized quicksort

algorithm, we have

$$T(n) = n + T(k) + T(n-k-1),$$

where k is the number of elements smaller than the pivot. Considering the uniform distribution of k over $\{0, 1, \ldots, n-1\}$, we easily get that

$$E[T(n)] = n + E[T(k)] + E[T(n-k-1)] = n + 2E[T(k)] = n + \frac{2}{n}\sum_{k=0}^{n-1} E[T(k)],$$

which is a recurrence relation. Although we cannot apply Theorem 2.1 to it, we can use an induction to show $E(T(n)) \leq 2n \ln n$. More precisely, we have the following lemma, which implies that the expected running time of randomized quicksort is $O(n \log n)$.

Lemma 3.1 *Suppose $T(0) = T(1) = 0$ and that for all integer $n > 1$, $T(n) = n + \frac{2}{n}\sum_{k=0}^{n-1} T(k)$. Then for all positive integer n, $T(n) \leq 2n \ln n$.*

Proof: By induction. The inequality clearly holds for $n = 1$.

Now assume it holds for all positive integer $n \leq N$. When $n = N + 1$, we get that

$$\begin{aligned}
T(N+1) &= N + 1 + \frac{2}{N+1}\sum_{k=0}^{N} T(k) \leq N + 1 + \frac{4}{N+1}\sum_{k=2}^{N} k \ln k \\
&\leq N + 1 + \frac{4}{N+1}\int_2^{N+1} k \ln k \, dk = N + 1 + \frac{2}{N+1}\int_2^{N+1} \ln k \, dk^2 \\
&= N + 1 + \frac{2}{N+1}(k^2 \ln k|_2^{N+1} - \int_2^{N+1} k^2 \, d \ln k) \\
&= N + 1 + \frac{2}{N+1}((N+1)^2 \ln(N+1) - 4 \ln 2 - \frac{(N+1)^2 - 4}{2}) \\
&= 2(N+1)\ln(N+1) + \frac{4 - 8\ln 2}{N+1} \\
&< 2(N+1)\ln(N+1),
\end{aligned}$$

which means the inequality holds for $n = N + 1$ as well. □

Since the result of $O(n \log n)$ gives us only an upper bound, naturally one would ask whether the time complexity is $\Theta(n \log n)$, or we might actually get better than that. It turns out that the former is true, because for an increasing and convex[7] function $T()$ as studied in Lemma 3.1, we always have

$$\begin{aligned}
T(n) &= n + \frac{2}{n}\sum_{k=0}^{n-1} T(k) = n + \frac{2}{n}\sum_{k=0}^{n} T(k) - \frac{2T(n)}{n} \\
&\geq n + \frac{2(n+1)}{n} T(\frac{n}{2}) - 4 \ln n \geq 2T(\frac{n}{2}) + n - 4 \ln n.
\end{aligned}$$

[7] Why can we make this assumption? See Problem 3.1.

By the Master Theorem we know that $T(n) = 2T(\frac{n}{2}) + n - 4\ln n \Rightarrow T(n) = \Theta(n \log n)$. So our recurrence has a solution that grows no slower than that. (Besides the approach we discuss above, there are some other interesting ways to analyze the complexity of randomized quicksort algorithm. Interested readers can check out, *e.g.*, [2, 11].)

Now let us look at another algorithm, min-cut algorithm. Suppose we get a simple, connected graph. Recall a cut in this graph is simply a subset of edges that divide the graph into at least two components. A min-cut is a cut that consists of the smallest possible number of edges. Our problem is to use an algorithm to find one.

A standard deterministic algorithm for this problem is to find a maximum flow in the graph, using, *e.g.*, the Ford-Fulkerson Algorithm, but the time complexity would not be very satisfactory (see Problem 3.2). If you need a better algorithm, you may want to take the following observation into consideration: Since a min-cut is of the smallest size among all cuts, when you pick a random edge from the graph, it is quite likely to be outside of the min-cut. More precisely, you can use the following algorithm, designed by Karger:

Algorithm Karger-Min-cut$((V, E))$
Repeat until $|V| \leq 2$
pick an edge $(u, v) \in E$ *uniformly at random*;
$E \leftarrow E - \{(u, v)\}$;
merge u and v into a single vertex in V;
return E.

In each iteration, the probability of an edge in a particular min-cut being picked is no more than $\frac{k}{m}$, where k is the size of the min-cut and m is the total number of edges. Since k is small, we hope to be so lucky that the algorithm never picks an edge from the min-cut before its termination. Will we really be so lucky? Let us proceed to formal analysis of the algorithm.

Lemma 3.2 *The output of Karger algorithm is a cut. It is a min-cut if the min-cut's edges were never picked during the execution of the algorithm.*

Lemma 3.3 *The size of a min-cut is no more than $\frac{2m}{n}$, where n is the number of vertices and m is the number of edges.*

Proof: Note that the set of edges incident with each node is a cut, and thus for each vertex v,

$$k \leq deg(v),$$

where k is the size of a min-cut. On the other hand, we have

$$\sum_{v \in V} deg(v) = 2m.$$

Combining the above, we get that $kn \leq 2m \Rightarrow k \leq \frac{2m}{n}$. □

Theorem 3.1 *During an execution of the Karger algorithm, the probability of a min-cut's edge never being picked is no less than $\frac{2}{n(n-1)}$.*

Proof: Lemma 3.3 tells us that during the first iteration, the probability of a particular min-cut's edge not being picked is $1 - \frac{k}{m} \geq 1 - \frac{2}{n}$. In the second iteration, since the number of vertices is less by 1, this probability lower bound shrinks to $1 - \frac{2}{n-1}$ ... Putting together all iterations, we get that

$$Pr[\text{edges in min-cut never picked}] \geq (1 - \frac{2}{n})(1 - \frac{2}{n-1}) \ldots (1 - \frac{2}{3}) = \frac{2}{n(n-1)}.$$

□

When n is large, the probability lower bound can be quite small, nearly zero. In fact, even when n is small (like 5), we are not happy with it as well (0.1 in the case of $n = 5$). In reality, we definitely want a much better guarantee of success. In order to get such guarantee, we repeat Karger's algorithm for a number of times, and use the output with the smallest cardinality as the min-cut we find. For how many times do we have to repeat the algorithm? We leave it as homework (see Problem 3.3).

So far we have presented two examples of randomized algorithms, namely the random quicksort and Karger's min-cut. If you compare them side by side, you can see a key difference: While the random quicksort algorithm always returns a correct output, Karger's min-cut algorithm has a non-zero probability of failure. Based on this fundamental difference, theoretical computer scientists have defined Las Vegas and Monte Carlo algorithms.

Definition 3.1 *A Las Vegas algorithm is a randomized algorithm that always produces a correct output. A Monte Carlo algorithm is a randomized algorithm whose output can be wrong with a non-zero probability.*

It is tempting to say we want Las Vegas algorithm only, not the error-prone Monte Carlo algorithms. However, Las Vegas algorithms usually come with a cost—their running time usually varies with the input.

Example 3.1 *Find an input on which the randomized quicksort algorithm's running time is $\Theta(n^2)$, where n is the size of input array.*

Solution: An input with all elements being identical always makes array y empty and inserts all remaining elements to array z. Consequently, the number of comparison-insertions done is $(n-1) + (n-2) + \ldots + 1 = \frac{n(n-1)}{2}$. □

If you didn't care too much about the running time, you could turn a Monte Carlo algorithm into a Las Vegas algorithm by repeatedly executing the Monte Carlo algorithm and checking the correctness of the output, as long as the problem to be solved is in the complexity class NP.

Problems

Problem 3.1 *Prove the function $T()$ studied in Lemma 3.1 is convex.*

Problem 3.2 *Suppose you are given a simple connected graph of n vertices as input. Design an algorithm that finds a min-cut, based on the famous Ford-Fulkerson algorithm for maximum flow. Assuming there are $\Theta(n^2)$ edges, what kind of worst-case time complexity can you get? You just need to present a time complexity analysis of your own algorithm. We are not asking you to prove your algorithm is optimal.*

Problem 3.3 *Suppose we are given a upper bound $\epsilon(>0)$ of failure probability. In order to guarantee we get a min-cut with probability no less than $1-\epsilon$, for how many times should we repeat Karger's algorithm? Prove your answer is sufficient to guarantee the bound. You don't need to prove your answer is optimal.*

Problem 3.4 *Design a Monte Carlo algorithm for sorting. Analyze its error probability.*

Problem 3.5 *(Final Exam 2022-3) Prove that a simple graph of n vertices has at most $\binom{n}{2}$ min cuts.*

4 Zero-Knowledge Proof, a Cryptographic Primitive

A typical misunderstanding of cryptography is that it is all about encrypting and decrypting data, which is completely wrong. Modern cryptography is actually so powerful that it allows us to do something one can hardly believe, like proving a statement to you without letting you know anything. The latter is called Zero-Knowledge Proof, a very important primitive in modern cryptography.

Since Zero-Knowledge Proofs are part of so called interactive proofs, we first define interactive proof systems.

Definition 4.1 *An interactive proof system (P,V) consists of a pair of probabilistic polynomial-time algorithms*[8]*, prover P and verifier V, that execute in turn. Each time P (resp., V) stops, it sends a message to V (resp., P) before the latter resumes its execution. At the beginning, P starts execution first; at the end, V finishes its execution and generates an output, which is either "accept" or "reject".*

The purpose of estabslishing an interactive proof system is to *prove* a proposition. But does the interactive proof system do its job well? We have the following standard for it.

Definition 4.2 *Let $P(n)$ be the probability of an event depending on a security parameter n. We say $P(n)$ is negligible if for all (increasing and positively valued) polynomial $f(n)$, we have $f(n)=O(\frac{1}{P(n)})$. We say $P(n)$ is a high probability if $1-P(n)$ is negligible.*

In cryptography, the security parameter n is often the length of key. We measure the running time of our algorithms using functions of n; we also measure the running time of adversary algorithms using functions of n. Further, we measure the success probability of our algorithms using functions of n, hoping it to be a high probability, and measure the success probability of adversary algorithms using functions of n, hoping it to be negligible.

Definition 4.3 *An interactive proof system (P,V) for a proposition S is* complete *if whenever S is true, V outputs "accept" with high probability; it is* sound *if whenever S is false, for all (cheating prover) $P^\star$, V outputs "reject" with high probability.*

[8]This definition is semiformal at the best, since "algorithm" is not a well defined mathematical term. A formal definition, which occupies a few pages in [5], is based on Turing machines, rather than on algorithms.

Example 4.1 *Construct an interactive proof system (P, V) for proving input x is a prime number, in each of the cases below.*

1. *(P, V) must be complete and sound.*
2. *(P, V) must be complete, but not sound.*
3. *(P, V) must be sound, but not complete.*
4. *(P, V) must be neither complete nor sound.*

The solution is too easy and thus skipped.

Example 4.2 *Construct a complete and sound interactive proof system (P, V) for proving input (G, G') is a pair of isomorphic graphs.*

Solution: Assume P, the prover, "knows" the isomorphic map θ. Hence, P simply sends θ to V, who checks $\theta(G) = G'$. If the identity holds, V outputs "accept"; otherwise, it outputs "reject." □

In the above example, we constructed an interactive proof that is both complete and sound. However, in order to prove the proposition, the prover has to reveal his knowledge of isomorphic map to the verifier V. In many applications, this is undesirable. Usually we wish that the verifier should learn as little as possible, ideally nothing, from the interactive proof.

Definition 4.4 *In an interactive proof system, the view of a party consists of its input, messages it has received, and all its coin flips.*

Definition 4.5 *An interactive proof system (P, V) is Perfectly Zero-Knowledge (PZK) if for all cheating verifier $V^\star$, there exists a probablistic polynomial-time simulator σ that takes $V^\star$'s input as its own input and completes a simulation successfully with high probability. Under the condition that the simulation is deemed successful, σ's output must have a distribution identical to that of $V^\star$'s view.*

The above definition is quite smart in that we have sidestepped the difficulty of defining an algorithm's knowledge. In stead, we observe that if the verifier's view (everything the verifier sees) can always be simulated by an algorithm that has never seen any secret of the prover, then we can safely say the interactive proof conveys zero knowledge to the verifier. This kind of definition is called the simulation paradigm, and is very frequently used in cryptography.

In Definition 4.5, we had to consider an arbitrary cheating verifier $V^\star$, because the verifier might cheat, and if so, the cheating behavior may affect its view. Hence, we require that not just an honest verifier, but any cheating verifier, should never get any knowledge out of the interactive proof system.

Example 4.3 *Construct a complete, sound, and PZK interactive proof system (P, V) for proving input (G, G') is a pair of isomorphic graphs.*

Solution: Suppose the input is (G, G') such that $\theta(G) = G'$. Repeat the following process for n times (where n is the security parameter): P generates a random isomorphic map θ' and sends $G'' = \theta'(G)$ to V. V challenges P by asking, with probability $\frac{1}{2}$, for the isomorphic map between G and G'', and with the remaining probability $\frac{1}{2}$, for the isomorphic map between G' and G''. P replies with either θ' or $\theta^{-1}\theta'$, depending on which isomorphic map V asked for. V proceeds to the next round if the map is correct, and rejects otherwise. If all rounds have been completed successfully, V accepts. □

A somewhat relaxed version of PZK is Statistically Zero Knowledge (SZK), which is based on the statistic difference of two random variables.

Definition 4.6 *Suppose X and Y are both random variables with range R. The statistic difference between X and Y is*

$$\Delta(X, Y) = \frac{1}{2} \sum_{r \in R} |Pr[X = r] - Pr[Y = r]|.$$

We have to emphasize that the above definition came from the discipline of statistics. If we defined it, we would not include the constant factor of $\frac{1}{2}$.

Definition 4.7 *An interactive proof system (P, V) is Statistically Zero-Knowledge (SZK) if for all cheating verifier $V^\star$, there exists a probablistic polynomial-time simulator σ that takes $V^\star$'s input as its own input, such that the statistic difference between σ's output and $V^\star$'s view is negligible.*

Example 4.4 *Prove every PZK proof is also an SZK proof.*

Solution: There is no doubt that two identical probabilitic distributions have a zero (and thus negligible) statistic difference. However, the definition of PZK allows the simulator to fail with a negligible probability, while the definition of SZK does not. What can we do with this subtle issue?

Assume we have a PZK proof, together with its PZK simulator σ, which fails with a negligible, but not zero, probability. Now we are supposed to construct a simulator σ' that does not fail at all. The key observation is that we can allow σ' to make a random output whenever σ fails. In this way, σ' never needs to fail, but its output can be different with a negligible probability, as long as σ' behaves in exactly the same way as σ in all other cases. This clearly meets our requirement of SZK. □

When SZK is further relaxed, we get Computational Zero Knowledge (CZK), which is based on computational indistinguishability.

Definition 4.8 *Random variables X and Y are computationally indistinguishable from each other if for all probabilistic polynomial-time distinguisher*[9] *D, $|Pr[D(X) = 1] - Pr[D(Y) = 1]|$ is negligible. In this case, we usually write $X \stackrel{C}{=} Y$.*

[9] A distinguisher is an algorithm that outputs only 0 or 1.

Definition 4.9 *An interactive proof system (P, V) is Computationally Zero-Knowledge (CZK) if for all cheating verifier $V^\star$, there exists a probablistic polynomial-time simulator σ that takes $V^\star$'s input as its own input, such that σ's output and $V^\star$'s view is computationally indistinguishable.*

Example 4.5 *Prove every SZK proof is also a CZK proof.*

Solution: For all probablistic polynomial-time distinguisher D, we have

$$\begin{aligned}
|Pr[D(X)=1] - Pr[D(Y)=1]| &= |\sum_r Pr[X=r, D(X)=1] - \sum_r Pr[Y=r, D(Y)=1]| \\
&= |\sum_r (Pr[X=r]Pr[D(X)=1|X=r] - Pr[Y=r]Pr[D(Y)=1|Y=r])| \\
&= |\sum_r (Pr[X=r] - Pr[Y=r])|Pr[D(Z)=1|Z=r] \\
&\leq 2\Delta(X,Y).
\end{aligned}$$

Since the right side is negligible, the left side is also negligible. □

Example 4.6 * *Suppose p, q are prime numbers such that $p = 2q + 1$; g, h are both quadratic residues with respect to modulus p, i.e., there exist α, β such that $g \equiv \alpha^2 \pmod p$, $h \equiv \beta^2 \pmod p$. Assuming $g \neq 1 \pmod p$ and $h \neq 1 \pmod p$, we can show that there exists e ($1 \leq e \leq q-1$) such that $g^e \equiv h \pmod p$. We call e the discrete logarithm of h to base g. (Notice the similarities to, and differences from, the logarithm we learned in middle school!)*

For large p, g, h, computing discrete logarithm is considered infeasible. In other words, given p, g, h, we are not aware of any probablistic polynomial-time algorithm that can compute e.

Now suppose p, g, h are all public and e is a prover's secret. Construct a CZK proof of knowing e.

Solution: Repeat the following process for n times:

The prover picks $f \in \{1, 2, \ldots, q-1\}$ uniformly at random, computes $j = g^f \bmod p$, and sends j to the verifier. The verifier chooses a uniformly random bit as his challenge. If the challenge is 0, then the prover has to send f to the verifier; the verifier accepts after checking $j \equiv g^f \pmod p$, and rejects in case the equation does not hold. If the challenge is 1, then the prover has to send $f' = (e + f) \bmod q$ to the verifier; the verifier accepts after checking $hj \equiv g^{f'} \pmod p$, and rejects in case the equation does not hold. □

In addition to being CZK, the above interactive proof is also a Proof of Knowledge (POK), because its objective is to show the prover has knowledge of the secret discrete logarithm. The formal treatment of POK is well beyond the scope of this lecture, but can be found, *e.g.*, in [5]. Roughly speaking, an interactive proof is POK if there is a so called knowledge extractor that can observe its exection and magically figure out the secret knowledge.

Why isn't POK contradictory to CZK? Because the knowledge extractor is much more powerful than any realistic algorithm. It can even manipulate time to some extent—more precisely, it can arbitrarily stop the interactive proof system at any point, and replay (part of) it from any state. In reality, no algorithm can do that.

Problems

Problem 4.1 *Find a negligible function $p(n)$ such that for all $b > 1$, $\frac{1}{b^n} = O(p(n))$.*

Problem 4.2 *An alternative definition of a complete interactive proof system requires that whenever S is true, V outputs "accept" with probability no less than $\frac{2}{3}$. In what sense is this alternative definition equivalent to ours? Prove your answer.*

Problem 4.3 *Prove the interactive proof we constructed for Example 4.3 is indeed complete, sound, and PZK.*

Problem 4.4 * *An interactive proof system (P, V) is pseudo-zero-knowledge if for all cheating verifier $V^\star$, there exists a probablistic polynomial-time simulator σ that takes $V^\star$'s input as its own input and completes a simulation successfully with high probability. Under the condition that the simulation is deemed successful, σ's output must have a distribution identical to that of $V^\star$'s received messages.*

Construct an interactive proof system that is pseudo-zero-knowledge but not PZK.

Problem 4.5 *Prove the interactive proof we constructed in Example 4.6 is indeed CZK.*

Problem 4.6 *(Final Exam 2020-4) Construct a complete, sound, and SZK proof for graph isomorphism that is not PZK.*

Problem 4.7 *(Final Exam 2021-5) Recall that a distinguisher is an algorithm that outputs only* 0 *or* 1*. If we replace "distinguisher" with "algorithm", we get an alternative definition:*

Definition 4.10 *Random variables X and Y are computationally indistinguishable from each other if for all probabilistic polynomial-time algorithm A, $\sum_v |Pr[A(X) = v] - Pr[A(Y) = v]|$ is negligible.*

Are these two definitions equivalent? If yes, please prove your answer. If no, provide an example that satisfies one of them, but not the other.

Problem 4.8 *(Final Exam 2022-4) Suppose x, y, u, v are random variables and their range is the set of s-bit strings. Is each of the following propositions true? If so, provide a proof. Otherwise, give a counter example.*

(a) If $x \stackrel{c}{=} y, u \stackrel{c}{=} v$, then $(x, u) \stackrel{c}{=} (y, v)$.

(b) If $(x, u) \stackrel{c}{=} (y, v)$, then $x \stackrel{c}{=} y, u \stackrel{c}{=} v$.

5 Nash Equilibrium, a Solution Concept from Game Theory

Definition 5.1 *A* strategic game *is a tuple*

$$(N, \{A_i\}_{i \in N}, \{\succeq_i\}_{i \in N}),$$

where

- N *is a set called* player *set;*
- *for each* $i \in N$, A_i *is a non-empty set called player* i*'s* action *set;* $A = \prod_{i \in N} A_i$ *is called the set of* action profiles*;*
- *for each* $i \in N$, $\succeq_i$ *is a* preference relation *on* A*.*

A couple of things to note for the term *preference relation* in the above definition:

1. What is a preference relation?
   - Recall a *relation* is simply a set of ordered pairs. Specifically, a relation on A is a set whose elements are ordered pairs of elements of A.
   - A relation on A is called a preference relation if it is complete, reflexive, and transitive.
   - Being complete means for any $a, b \in A$, either (a, b) or (b, a) must appear in the relation.
   - Being reflexive means for any $a \in A$, (a, a) must be in the relation.
   - Being transitive means for any (a, b) and (b, c) in the relation, (a, c) must also be in the relation.
2. Since preference relation is pretty cumbersome, we nearly always use a *utility function* instead.
   - A utility function u is a function from A to $\mathcal{R}$.
   - It must be satisfied that $u(a) \geq u(b) \Leftrightarrow a \succeq b$.

In a strategic game, an action is also called a *pure strategy*, since a player could deterministicly use the action as her strategy. In contrast, a *mixed strategy* is a probability distribution on the action set, which means the player uses a stragegy that randomly chooses an action based on the probability distribution. For convenience, we often treat a mixed strategy as a vector, each of whose components represents the probability of an action being taken. Thus all components of the vector are positive and their sum is equal to 1. The symbol $\Delta(A_i)$ represents the set of probability distributions/vectors on A_i, *i.e.*, the set of mixed strategies on action set A_i. Hereafter we often ambiguously say *strategy* without specifying whether it is pure or mixed, because our discussions can be carried out similarly in both cases.

Definition 5.2 *A Nash Equilibrium is a strategy profile* $s^\star$ *such that* $\forall i \in N$, $\forall s_i \in A_i$,

$$u_i(s_i^\star, s_{-i}^\star) \geq u_i(s_i, s_{-i}^\star),$$

where the subscript $-i$ *denotes "all players except* i*."*

While we have not distinguished pure and mixed strategies for the above definition, it is worth noting that in the mixed strategy version, the utilities considered are always their expected values.

Definition 5.3 *In a strategic game $(N, \{A_i\}_{i\in N}, \{\succeq_i\}_{i\in N})$, for each $i \in N$, there is a* best response function B_i *from the set of strategy profiles of players other than i to the set of strategies of player i, where for each strategy profile s_{-i} of players other than i,*

$$B_i(s_{-i}) = \arg\max_{s_i} u_i(s_i, s_{-i}).$$

Note we are sloppy with notations here, since there may be more than one strategy that maximizes the utility. Therefore, we should have written "$\in$" in stead of "$=$" in the above equation. However, writing "$=$" is a convention and we decide to follow the convention unless there is a high risk of confusion.

For technical reasons, we sometimes consider a best response function for all players: B from the set of strategy profiles of all players to the same set itself, where for each strategy profile s of all players,

$$B(s) = (\arg\max_{s_i^\star} u_i(s_i^\star, s_{-i}))_{i\in N} = (B_i(s_{-i}))_{i\in N}.$$

Again, bear in mind that we have intentionally been ambiguous about where the involved strategies are pure or mixed, since both of them can use the above definition.

Clearly, a Nash Equilibrium is nothing but a fixed point of the best response function B for all players. American mathematician-economist John Nash established the following groundbreaking theorem based on this simple observation:

Theorem 5.1 *(Nash) Any finite strategic game has a mixed strategy Nash Equilibrium.*

In order to prove Nash Theorem, we need to first review a few definitions from real analysis.

Definition 5.4 *In an Euclidean space, a point set S*

- *is* convex *if $\forall a, b \in S$, any point in the line segment $\overline{ab}$ is also in S.*
- *is* closed *if every sequence $\{a_i\}$ ($\forall i$, $a_i \in S$) satisfies that, as long as $\{a_i\}$ converges, $\lim_{i\to\infty} a_i \in S$.*
- *is* bounded *if $\exists K > 0$ such that $\forall a \in S$, $\sum_i |a[i]| \leq K$.*

Definition 5.5 *Suppose X, Y are subsets of an Euclidean space. $F : X \to 2^Y$ is* upper-hemicontinuous *if for any sequences $\{x_i\}, \{y_i\}$ such that $\forall i$, $y_i \in F(x_i)$, as long as $\{x_i\}, \{y_i\}$ both converge,*

$$\lim_{i\to\infty} y_i \in F(\lim_{i\to\infty} x_i).$$

With the above definitions, now we can present the famous fixed point theorem by Japanese-American mathematician Shizuo Kakutani.

Theorem 5.2 *(Kakutani) Let S be a non-empty, convex, closed, bounded point set in a finite dimension Euclidean space. Let $F : S \to 2^S$ be an upper-hemicontinuous set-valued function such that $\forall x \in S$, $F(x)$ is non-empty, convex. Then F has a fixed point, i.e., $\exists x \in S$ such that $x \in F(x)$.*

While the proof of Kakutani Theorem is out of scope, using it to show Nash Theorem is pretty straightforward.

Proof: We are to apply Kakutani Theorem to the best response function B. In order to do this, we note that B is defined on $\Delta(A)$. We just need to check the conditions of Kakutani Theorem one by one.

First, $\Delta(A)$ is non-empty.

Second, $\forall s, s' \in \Delta(A)$, any point in the line segment $\overline{ss'}$ can be written as $s'' = \lambda s + (1-\lambda)s'$ $(0 \leq \lambda \leq 1)$. $\forall i \in N, \forall j$, we have that $s''_i[j] = \lambda s_i[j] + (1-\lambda)s'_i[j] \geq 0$ and that

$$\sum_j s''_i[j] = \lambda \sum_j s_i[j] + (1-\lambda) \sum_j s'_i[j] = \lambda + 1 - \lambda = 1.$$

Hence, $s'' \in \Delta(A)$. This means $\Delta(A)$ is convex.

Third, consider any sequence $\{s^{(k)}\}$ ($\forall k$, $s^{(k)} \in \Delta(A)$) that converges. $\forall i \in N$, $\forall j$, $\forall k$, $(s^{(k)})_i[j] \geq 0 \Rightarrow (\lim_{k\to\infty} s^{(k)})_i[j] \geq 0$. Furthermore, since each player has only a *finite* number of actions in her action space, we have

$$\sum_j \lim_{k\to\infty} s^{(k)}[j] = \lim_{k\to\infty} \sum_j s^{(k)}[j] = 1.$$

Hence, $\lim_{k\to\infty} s^{(k)}[j] \in \Delta(A)$. This means $\Delta(A)$ is closed.

Fourth, $\forall s \in \Delta(A)$, $\sum_{i\in N, j} |s_j[j]| = |N|$. Consequently, $\Delta(A)$ is bounded.

Fifth, $\forall s \in \Delta(A)$, since A_i is finite for all $i \in N$, $\arg\max_{a_i \in A_i} u_i(a_i, s_{-i})$ is definitely non-empty. $\forall a_i^\star \in \arg\max_{a_i \in A_i} u_i(a_i, s_{-i})$, $\forall s'_i \in \Delta(A_i)$,

$$u_i(s'_i, s_{-i}) = \sum_{a_i \in A_i} p(a_i) u_i(a_i, s_{-i}),$$

where $p(a_i)$ is the probability assigned to a_i by s'_i. Hence,

$$u_i(s'_i, s_{-i}) \leq \sum_{a_i \in A_i} p(a_i) u_i(a_i^\star, s_{-i}) = u_i(a_i^\star, s_{-i}).$$

Hence, $a_i^\star \in \arg\max_{s_i \in \Delta(A_i)} u_i(s_i, s_{-i})$. This means $B_i(s_{-i})$ is non-empty. So, $B(s)$ is also non-empty.

Sixth, $\forall s \in \Delta(A)$, $\forall s', s'' \in B(s)$, any point in the line segment $\overline{s's''}$ can be written as $s''' = \lambda s' + (1-\lambda)s''$ $(0 \leq \lambda \leq 1)$. $\forall i \in N$, we have that

$$u_i(s'''_i, s_{-i}) = \sum_{a_i \in A_i} p'''(a_i) u_i(a_i, s_{-i}),$$

where $p'''(a_i)$ is the probability assigned to action a_i by s'''_i. We use $p'(a_i)$, $p''(a_i)$ to represent the probabilities

assigned to action a_i by s_i', s_i'', respectively. Thus, we get that

$$\begin{aligned} u_i(s_i''', s_{-i}) &= \sum_{a_i \in A_i} (\lambda p'(a_i) + (1-\lambda)p''(a_i)) u_i(a_i, s_{-i}) \\ &= \lambda \sum_{a_i \in A_i} p'(a_i) u_i(a_i, s_{-i}) + (1-\lambda) \sum_{a_i \in A_i} p''(a_i) u_i(a_i, s_{-i}) \\ &= \lambda u_i(s_i', s_{-i}) + (1-\lambda) u_i(s_i'', s_{-i}) \\ &= \lambda \max_{s_i^{\triangle} \in \Delta(A_i)} u_i(s_i^{\triangle}, s_{-i}) + (1-\lambda) \max_{s_i^{\triangle} \in \Delta(A_i)} u_i(s_i^{\triangle}, s_{-i}) \\ &= \max_{s_i^{\triangle} \in \Delta(A_i)} u_i(s_i^{\triangle}, s_{-i}). \end{aligned}$$

Hence, $s_i''' \in B_i(s_{-i})$. Therefore, $s''' \in B(s)$. This means $B(s)$ is convex.

Finally, consider sequences $\{s^{(k)}\}$,$\{\bar{s}^{(k)}\}$ ($\forall k$, $s^{(k)}, \bar{s}^{(k)} \in \Delta(A)$) such that $\forall k$, $\bar{s}^{(k)} \in B(s^{(k)})$. As long as the two sequences converge, we have that

$$\forall i \in N, \forall s_i \in \Delta(A), u_i(\bar{s}_i^{(k)}, s_{-i}^{(k)}) \geq u_i(s_i, s_{-i}^{(k)}) \Rightarrow \lim_{k\to\infty} u_i(\bar{s}_i^{(k)}, s_{-i}^{(k)}) \geq \lim_{k\to\infty} u_i(s_i, s_{-i}^{(k)}).$$

Given the continuity of the expected utility function, the above implies that

$$u_i(\lim_{k\to\infty} \bar{s}_i^{(k)}, \lim_{k\to\infty} s_{-i}^{(k)}) \geq u_i(s_i, \lim_{k\to\infty} s_{-i}^{(k)}) \Rightarrow \lim_{k\to\infty} \bar{s}_i^{(k)} \in B_i(\lim_{k\to\infty} s_{-i}^{(k)}).$$

This means $\lim_{k\to\infty} \bar{s}^{(k)} \in B(\lim_{k\to\infty} s^{(k)})$. Therefore, B is upper-hemicontinuous. □

There is another important solution concept called *dominant strategy equilibrium*, which is much stronger than Nash Equilibrium.

Definition 5.6 *In a strategic game $(N, \{A_i\}_{i\in N}, \{\succeq_i\}_{i\in N})$, for a player $i \in N$, a strategy $s_i^\star$ is called a dominant strategy if $\forall s_i, \forall s_{-i}$, $u_i(s_i^\star, s_{-i}) \geq u_i(s_i, s_{-i})$. A strategy profile $s^\star$ is called a* dominant strategy equilibrium *if $\forall i \in N$, $s_i^\star$ is a dominant strategy.*

Clearly, any dominant strategy equilbrium is also a Nash Equilibrium, but the converse is not true. Is there a solution concept strictly weaker than Nash Equilibrium? Yes, there is one, called *rationalizable actions*. We skip its definition since it is quite lengthy. Interested readers can refer to [10].

Yet another important solution concept, *Pareto Optimality*, is neither stronger nor weaker, but parallel to Nash Equilibrium.

Definition 5.7 *In a strategic game $(N, \{A_i\}_{i\in N}, \{\succeq_i\}_{i\in N})$, a strategy profile $s^\star$ of all players is* Pareto Optimal *if for any other strategy profile s of all players, either $\forall i \in N$, $u_i(s^\star) = u_i(s)$; or $\exists i \in N$, $u_i(s^\star) > u_i(s)$.*

Clearly, a Nash Equilibrium may not be Pareto Optimal, and a Pareto Optimal strategy profile may not be a Nash Equilibrium.

Problems

Problem 5.1 *Construct a strategic game that has exactly one mixed strategy Nash Equilibrium but no pure strategy Nash Equilibrium.*

Problem 5.2 *Construct a strategic game that has no mixed strategy Nash Equilibrium.*

Problem 5.3 *Construct a strategic game that has exactly one dominant (pure) strategy equilibrium, but exactly two (pure strategy) Nash equilibria.*

Problem 5.4 *(Final Exam 2020-5) Consider a strategic game (N, A, U) where the set of players $N = \{1, 2\}$. The sets of actions are $A_1 = A_2 = \mathcal{R}^{2020}$. In other words, each player's action is a 2020-dimensional vector of real numbers. The utility of each player i ($i \in N$) is defined as*

$$U_i(a_1, a_2) = \sum_{1 \le j_1, j_2 \le 2020} a_1[j_1] a_2[j_2] V[j_1, j_2],$$

where V is a 2020×2020 matrix of real numbers. Does this game always have a pure strategy Nash Equilibrium for all possible values of V? If yes, provide a proof. If no, provide a counter example.

Problem 5.5 *(Final Exam 2021-6) The preference of a rational player can be represented by a preference relation, or a utility function. Prove that for each preference relation R, if all action sets are finite, there exists a utility function u such that for any action profiles a and b,*

$$a \succeq b \Leftrightarrow u(a) \ge u(b).$$

Problem 5.6 *(Final Exam 2022-5) If a mixed strategy is not a pure strategy (i.e., assigns non-zero probabilities to two or more actions), then we call it a randomized strategy. If in a Nash Equilibrium all players use randomized strategies, then we call it a Randomized Strategy Nash Equilibrium. Now consider a finite game between two players, in which each player has exactly two actions to choose from. Is it possible that there are exactly two Randomized Strategy Nash Equilibria in this game? Prove your answer.*

6 Continued Fractions, a Topic from Number Theory

We begin this section with a problem from elementary school:

Example 6.1 *Let A, B, C be positive integers. Suppose*

$$\frac{24}{5} = A + \frac{1}{B + \frac{1}{C}}.$$

Find A, B, C.

There is no doubt that $A = 4, B = 1, C = 4$. A fraction written this way is called a *continued fraction*. More formally, we have the following definition.

Definition 6.1 *A finite continued fraction is a function defined on a sequence of real-valued variables $a_0, a_1, \ldots, a_N$:*

$$[a_0, a_1, \ldots, a_N] = a_0 + \cfrac{1}{a_1 + \ldots \; + \frac{1}{a_N}}.$$

Intuitively, we can see the sequence $[a_0]$, $[a_0, a_1], \ldots [a_0, a_1, \ldots, a_N]$ as a gradual approach to the value of the continued fraction. Hence, each term of the sequence (the nth of which is $[a_0, a_1, \ldots, a_n]$) is called a *convergent*.

But is the sequence $[a_0]$, $[a_0, a_1], \ldots [a_0, a_1, \ldots, a_N]$ really a gradual approach to the value of the continued fraction? In other words, does $[a_0, a_1, \ldots, a_n]$ get "closer" to $[a_0, a_1, \ldots, a_N]$ when n grows? In general, this may NOT be true. Consider, for example, the continued fraction $[1, \frac{1}{2}, -\frac{3}{2}, -1]$.[10] It is easy to calculate:

$$[1] = 1; [1, \frac{1}{2}] = 3; [1, \frac{1}{2}, -\frac{3}{2}] = -5; [1, \frac{1}{2}, -\frac{3}{2}, -1] = 11.$$

Therefore, the third convergent $[1, \frac{1}{2}, -\frac{3}{2}]$ is farther from the value of the continued fraction than the first two convergents.

Nevertheless, if all a_is are positive, we can actually guarantee convergents are getting "closer" to the value of the continued fraction, in the following sense.

Theorem 6.1 *Suppose for all i ($0 \leq i \leq N$), $a_i > 0$. Let c_i be the ith convergent of $[a_0, a_1, \ldots, a_N]$, i.e., $c_i = [a_0, a_1, \ldots, a_i]$. Then we have $c_0 < c_2 < \ldots < c_N$ and $c_1 > c_3 > \ldots > c_N$. In other words, the even numbered convergents approach the value of the continued fraction from the left and the odd numbered convergents approach it from the right.*

Proof: By induction on i, we can easily show that for all odd i and all $a_i < a_i'$, $[a_0, a_1, \ldots, a_i] > [a_0, a_1, \ldots, a_i']$; for all even i and all $a_i < a_i'$, $[a_0, a_1, \ldots, a_i] < [a_0, a_1, \ldots, a_i']$. (See Problem 6.2)

Hence, for any odd $k \geq 1$,

$$c_{k+2} = [a_0, a_1, \ldots, a_{k+2}] = [a_0, a_1, \ldots, a_{k-1}, a_k + \frac{1}{a_{k+1} + \frac{1}{a_{k+2}}}] < [a_0, a_1, \ldots, a_{k-1}, a_k] = c_k.$$

Similarly, for any even $k \geq 0$,

$$c_{k+2} = [a_0, a_1, \ldots, a_{k+2}] = [a_0, a_1, \ldots, a_{k-1}, a_k + \frac{1}{a_{k+1} + \frac{1}{a_{k+2}}}] > [a_0, a_1, \ldots, a_{k-1}, a_k] = c_k.$$

Now we see that odd numbered convergents are decreasing and even numbered convergents are increasing. The only thing remaining is to show the relationship between convergents and c_N, the value of the continued fraction.

[10] How do we come up with this example? See Problem 6.1.

If N is odd, then we have shown $c_1 > c_3 > \ldots > c_N$. For any even $k \geq 0$,

$$c_k = [a_0, a_1, \ldots, a_k] < [a_0, a_1, \ldots, a_{k-1}, a_k + \frac{1}{[a_{k+1}, \ldots, a_N]}] = c_N.$$

If N is even, the proof is similar. □

If all a_is are positive integers, then we say the continued fraction $[a_0, a_1, \ldots, a_N]$ is a *simple* continued fraction. When we study continued fractions, in most cases we focus on simple continued fractions only.

Obviously each simple continued fraction is equal to a positive rational number. Next, we show the reverse is true as well.

Theorem 6.2 *For all positive rational number $\frac{p}{q}$ where p and q are positive integers relative prime to each other, there exists a simple continued fraction*

$$[a_0, a_1, \ldots, a_N] = \frac{p}{q}.$$

Proof: We apply Euclidean algorithm to p and q: Let $a_0 = \lfloor \frac{p}{q} \rfloor$, and $a'_1 = p - a_0 q$. (Note that $\frac{p}{q} = a_0 + \frac{a'_1}{q}$.) For $i = 1$, if $a'_i = 0$, then we are done; otherwise, let $a_i = \lfloor \frac{q}{a'_i} \rfloor$, and $a'_{i+1} = q - a_i a'_i$. (Note that $\frac{p}{q} = a_0 + \frac{a'_1}{q} = a_0 + \frac{1}{\frac{q}{a'_1}} = a_0 + \frac{1}{a_1 + \frac{a'_2}{a'_1}} = [a_0, a_1 + \frac{a'_2}{a'_1}]$.)

For each $i \geq 2$, if $a'_i = 0$, then we are done; otherwise, let $a_i = \lfloor \frac{a'_{i-1}}{a'_i} \rfloor$, and $a'_{i+1} = a'_{i-1} - a_i a'_i$. (Note that $\frac{p}{q} = [a_0, a_1, \ldots, a_{i-2}, a_{i-1} + \frac{a'_i}{a'_{i-1}}] = [a_0, a_1, \ldots, a_{i-1}, \frac{a'_{i-1}}{a'_i}] = [a_0, a_1, \ldots, a_{i-1}, a_i + \frac{a'_{i+1}}{a'_i}]$.)

This process must terminate, because $a'_{i+1} < a'_{i-1}$. When it terminates, we get the simple continued fraction we need. □

Given that each positive rational number can be written as a simple continued fraction, naturally we ask whether there is only one such continued fraction. The answer is no (and "yes"—see below).

Theorem 6.3 *If $x = [a_0, a_1, \ldots, a_N]$ is a simple continued fraction where $N \geq 1$ is odd, then there exist an even number M and a simple continued fraction $[b_0, b_1, \ldots, b_M]$ such that $x = [b_0, b_1, \ldots, b_M]$. If $x = [a_0, a_1, \ldots, a_N] > 1$ is a simple continued fraction where $N \geq 0$ is even, then there exist an odd number M and a simple continued fraction $[b_0, b_1, \ldots, b_M]$ such that $x = [b_0, b_1, \ldots, b_M]$.*

Proof: We provide a proof for the first part only, because the second part can be proved similarly.

If $a_N = 1$, then we have $x = [a_0, a_1, \ldots, a_N] = [a_0, a_1, \ldots, a_{N-1}, 1] = [a_0, a_1, \ldots, a_{N-1} + 1]$.

If $a_N > 1$, then, defining $b_N = a_N - 1$, we have $x = [a_0, a_1, \ldots, a_N] = [a_0, a_1, \ldots, a_{N-1}, b_N + 1] = [a_0, a_1, \ldots, a_{N-1}, b_N, 1]$. □

If you take a closer look at the above proof, you see that we are converting a continued fraction with $a_N = 1$ to another one with $a_N > 1$, or the other way around. In fact, if we restrict our attention to those simple continued fractions with $a_N > 1$, we can guarantee every one is unique.

Theorem 6.4 *Suppose* $[a_0, a_1, \ldots, a_N]$ *and* $[b_0, b_1, \ldots, b_M]$ *are simple continued fractions with* $a_N > 1$, $b_M > 1$. *If* $[a_0, a_1, \ldots, a_N] = [b_0, b_1, \ldots, b_M]$, *then* $N = M$ *and for each* i $(0 \leq i \leq N)$, $a_i = b_i$.

The proof is not hard and can be found, *e.g.*, in [7].

If we consider only those with $a_N = 1$, we can get a similar result.

Example 6.2 *Suppose* $[a_0, a_1, \ldots, a_N]$ *and* $[b_0, b_1, \ldots, b_M]$ *are simple continued fractions with* $a_N = b_M = 1$. *If* $[a_0, a_1, \ldots, a_N] = [b_0, b_1, \ldots, b_M]$, *prove that* $N = M$ *and that for each* i $(0 \leq i \leq N)$, $a_i = b_i$.

Solution: Since $a_N = b_M = 1$, it is easy to see that $[a_0, a_1, \ldots, a_{N-2}, a_{N-1} + 1] = [a_0, a_1, \ldots, a_N] = [b_0, b_1, \ldots, b_M] = [b_0, b_1, \ldots, b_{M-2}, b_{M-1} + 1]$. Applying Theorem 6.4 to this identity, we get exactly what we want. □

Definition 6.2 *An infinite continued fraction is the limit of the corresponding finite continued fractions, if it exists:*

$$[a_0, a_1, \ldots] = \lim_{N \to \infty} [a_0, a_1, \ldots, a_N].$$

It is called simple if all a_i*s are positive integers.*

Theorem 6.5 *Every infinite simple continued fraction* $[a_0, a_1, \ldots]$ *converges, i.e., the limit* $\lim_{N\to\infty}[a_0, a_1, \ldots, a_N]$ *exists.*

Proof: Since the odd numbered convergents decrease[11] and they have a_0 as their lower bound, the limit $\lim_{N\to\infty}[a_0, a_1, \ldots, a_{2N+1}]$ exists. Similarly, we know that the limit $\lim_{N\to\infty}[a_0, a_1, \ldots, a_{2N}]$ also exists. Applying the result of Problem 6.1, we get that

$$\lim_{N\to\infty}[a_0, a_1, \ldots, a_{2N+1}] - \lim_{N\to\infty}[a_0, a_1, \ldots, a_{2N}] = \lim_{N\to\infty}([a_0, a_1, \ldots, a_{2N+1}] - [a_0, a_1, \ldots, a_{2N}])$$

$$= \lim_{N\to\infty}\left(\frac{p_{2N+1}}{q_{2N+1}} - \frac{p_{2N}}{q_{2N}}\right) = \lim_{N\to\infty}\left(\frac{p_{2N+1}q_{2N} - p_{2N}q_{2N+1}}{q_{2N+1}q_{2N}}\right).$$

Applying an induction on n to the result of Problem 6.1, we can easily prove that for all $N \geq 0$, $|p_{2N+1}q_{2N} - p_{2N}q_{2N+1}| = |p_1q_0 - p_0q_1| = 1$. Consequently, $\left|\frac{p_{2N+1}q_{2N} - p_{2N}q_{2N+1}}{q_{2N+1}q_{2N}}\right| \leq \frac{1}{2N(2N+1)}$, which implies its limit must be 0. Therefore, $\lim_{N\to\infty}[a_0, a_1, \ldots, a_{2N+1}] = \lim_{N\to\infty}[a_0, a_1, \ldots, a_{2N}]$, which means $\lim_{N\to\infty}[a_0, a_1, \ldots, a_N]$ exists and is equal to them. □

Just like rational numbers being written as finite simple continued fractions, irrational numbers can be written as infinite simple continued fractions. See, *e.g.*, [7], for a proof of the following theorem.

Theorem 6.6 *The value of each infinite simple continued fraction is irrational. Each positive irrational number can be written as one and only one infinite simple continued fraction.*

[11] Bear in mind that, although we are now studying infinite continued fractions, these convergents can be considered convergents of a *finite* simple continued fraction.

There is an algorithm called *continued fraction algorithm*, which finds the infinite simple continued fraction for any given positive irrational number. (In fact, it can also be applied to positive rational numbers and so it works for any positive real number.) Specifically, for any input x, the algorithm takes the following steps:

1. $x' \leftarrow x$; $a_0 \leftarrow \lfloor x' \rfloor$;
2. for each $i \geq 1$ do:
   if $x' = a_{i-1}$ then terminate; otherwise $x' \leftarrow \frac{1}{x'-a_{i-1}}$, $a_i \leftarrow \lfloor x' \rfloor$;

The above algorithm terminates at a certain point if x is rational. It keeps going forever if x is irrational. In the latter case, if we force the algorithm to terminate at some point, the result is a finite simple continued fraction, which is an approximation of x. With a bit more analysis we can show that the error of this approximation is less than $\frac{1}{q^2}$, where q is the denominator of the value of the finite simple continued fraction. This further implies the following theorem.

Theorem 6.7 *For any positive irrational number ζ, there are infinitely many rational numbers $\frac{p}{q}$ such that*

$$|\zeta - \frac{p}{q}| < \frac{1}{q^2}.$$

Proof: (Dirichlet Argument) Consider an arbitrary $Q > 1$. The $Q+1$ numbers 0, $\zeta - \lfloor \zeta \rfloor$, $2\zeta - \lfloor 2\zeta \rfloor$, $\ldots$, $Q\zeta - \lfloor Q\zeta \rfloor$ are all distributed in the interval $[0, 1)$. If we divide $[0, 1)$ into Q smaller intervals $[0, \frac{1}{Q})$, $[\frac{1}{Q}, \frac{2}{Q})$, $\ldots$, $[\frac{Q-1}{Q}, 1)$, then two of them must fall into the same small interval, *i.e.*, there exist p_1, p_2 ($0 \leq p_1 < p_2 \leq Q$) such that $|p_1\zeta - \lfloor p_1\zeta \rfloor - (p_2\zeta - \lfloor p_2\zeta \rfloor)| < \frac{1}{Q}$. The above inequality is equivalent to $|\zeta - \frac{\lfloor p_2\zeta \rfloor - \lfloor p_1\zeta \rfloor}{p_2 - p_1}| < \frac{1}{Q(p_2-p_1)} < \frac{1}{(p_2-p_1)^2}$. Defining $q = p_2 - p_1$ and $p = \lfloor p_2\zeta \rfloor - \lfloor p_1\zeta \rfloor$, we get that

$$|\zeta - \frac{p}{q}| < \frac{1}{Qq} < \frac{1}{q^2}. \tag{2}$$

This result looks like what we need, except that we need infinitely many such rational numbers $\frac{p}{q}$, not just one. See Example 6.3 below for why there are infinitely many of them. □

Example 6.3 *Show that Dirichlet Argument provides infinitely many rational number approximations $\frac{p}{q}$ to ζ.*

Solution: Suppose that there are only finitely many such rational numbers $\frac{p}{q}$. Then there exists a sufficiently large $Q_0 > 1$ such that all these $\frac{p}{q}$ satisfy that $|\frac{p}{q} - \zeta| > \frac{1}{Q_0 q}$. For any $Q > Q_0$, we must have

$$|\frac{p}{q} - \zeta| > \frac{1}{Q_0 q} > \frac{1}{Qq}. \tag{3}$$

However, recall that the Dirichlet argument began with choosing an arbitrary $Q > 1$. We can definitely choose a $Q > Q_0$, and expect the argument to remain valid. The rational number $\frac{p}{q}$ constructed based on such a Q must satisfy Equation (2), which contradicts Equation (3). □

Given that we can approximate any irrational number with error bound $\frac{1}{q^2}$, it is natural to ask whether we can improve the bound to $\frac{1}{q^3}$, $\frac{1}{q^4}$, ... It turns out that we may not be able to improve the error bound. Our main difficulty comes from *algebraic numbers*, as defined below.

Definition 6.3 *A number is called an algebraic number, if it is a root of a polynomial with integral coefficients. Otherwise, it is called a transcendental number.*

Theorem 6.8 *(Liouville) If an irrational number ζ is a root of a degree-n polynomial with integral coefficients, then there is a real number $c(\zeta) > 0$ such that for all rational numbers $\frac{p}{q}$, $|\zeta - \frac{p}{q}| > \frac{c(\zeta)}{q^n}$.*

Problems

Problem 6.1 *For a continued fraction $[a_0, a_1, \ldots, a_N]$, define*

$$p_0 = a_0; q_0 = 1; \quad p_1 = a_0 a_1 + 1; q_1 = a_1;$$
$$p_{n+2} = a_{n+2} p_{n+1} + p_n; \quad q_{n+2} = a_{n+2} q_{n+1} + q_n.$$

Show that for all $n \geq 0$, $\frac{p_n}{q_n} = [a_0, a_1, \ldots, a_N]$.

Problem 6.2 *Suppose for all i ($0 \leq i \leq N$), $a_i > 0$. Show that, if N is odd and $a_N < a'_N$, then $[a_0, a_1, \ldots, a_N] > [a_0, a_1, \ldots, a'_N]$; if N is even and $a_N < a'_N$, then $[a_0, a_1, \ldots, a_N] < [a_0, a_1, \ldots, a'_N]$.*

Problem 6.3 *A continued fraction $[a_0, a_1, \ldots, a_N]$ is called* pseudo-simple *if one of a_is is a negative integer and all the others are positive integers. Is it true that for all positive rational number $\frac{p}{q}$ where p and q are positive integers relative prime to each other, there exists a pseudo-simple continued fraction*

$$[a_0, a_1, \ldots, a_N] = \frac{p}{q}?$$

If so, provide a proof. Otherwise, provide a counter example.

Problem 6.4 *Suppose a continued fraction $[a_0, a_1, \ldots]$ is* cyclic. *That is, there exist $k > 0, \ell > 0$ such that for all $i \geq k$, $a_{i+\ell} = a_i$. Prove there exist integers u, v, w, x such that $[a_0, a_1, \ldots] = \frac{v + w\sqrt{x}}{u}$.*

Problem 6.5 $\star$ *Prove that there is no injection from the set of transcendental numbers to the set of algebraic numbers.*

Problem 6.6 $\star$ *(USA TST Jan 2016-1) Let $\sqrt{3} = 1.b_1 b_2 b_3 \ldots_{(2)}$ be the binary representation of $\sqrt{3}$. Prove that for any positive integer n, at least one of the digits b_n, b_{n+1}, ..., b_{2n} equals 1.*

Problem 6.7 *(Final Exam 2021-7) Prove the following lemma: If we have real numbers $\xi > 0$ and $\alpha > 1$, and there exist infinitily many positive integers a, b such that*

$$0 < \left|\xi - \frac{a}{b}\right| < \frac{1}{b^\alpha},$$

then ξ is irrational.

Problem 6.8 *(IberoAmerica MO 2009-5) Let $a_1 = 1$. For $n \geq 1$, define*

$$a_{2n} = a_n + 1; \quad a_{2n+1} = \frac{1}{a_{2n}}.$$

Prove that each rational number appears in the sequence exactly once.

7 Condition Number and Ill-Conditioned Matrix

In linear algebra, we have learned that any linear system can be written in the matrix form. Suppose we have an equation system

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n &= b_1 \\ a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n &= b_2 \\ \qquad\qquad \ldots\ldots & \\ a_{n1}x_1 + a_{n2}x_2 + \ldots + a_{nn}x_n &= b_n. \end{cases}$$

We can always write it as $A\vec{x} = \vec{b}$ where

$$A = \begin{pmatrix} a_{11} & a_{12} & \ldots & a_{1n} \\ a_{21} & a_{22} & \ldots & a_{2n} \\ \ldots & \ldots & \ldots & \ldots \\ a_{n1} & a_{n2} & \ldots & a_{nn} \end{pmatrix}, \vec{b} = \begin{pmatrix} b_1 \\ b_2 \\ \ldots \\ b_n \end{pmatrix}.$$

How do we solve the equation system? Assuming A is nonsingular, theoretically we just write the unique solution $\vec{x} = A^{-1}\vec{b}$ and are thus done. Nevertheless, in reality, we need to do more than that—we need to calculate the value of $\vec{x}$ in stead of just writing a formula for it. In order to do this, we need to first calculate the value of $A^{-1} = \frac{\mathbf{adj}(A)}{\det A}$ and then multiply it by $\vec{b}$. But how can we calculate $\mathbf{adj}(A)$, the adjugate of matrix A? By definition, it is $\mathbf{adj}(A) = (((-1)^{i+j} \det M_{ij})_{1 \leq i,j \leq n})^T$, where M_{ij} is the matrix obtained by removing the ith row and jth column from A. If there were a nice algorithm of $\Theta(n^{2+\epsilon})$ time complexity[12] for calculating a determinant, solving the linear system could take $\Theta(n^{4+\epsilon})$ time.[13] When we were facing a big matrix with $n = 10^8$, the fastest computer in the world might not be fast enough for solving the system. Therefore, in

[12] In this section, for simplicity, we always assume each arithmetic operation takes a constant amount of time.

[13] We might be able to lower the overall time complexity a bit by using some optimization tricks, but that would not help too much.

reality, we always use approaches like Gaussian Elimination to solve linear systems. A standard implementation of Gaussian Elimination has time complexity of $\Theta(n^3)$, which is somewhat better than the solution based on adjugates.

However, regardless of the algorithm you choose for solving the system, you are confident with the solution $\vec{x}$ you find only because when you calculate $A\vec{x}$, you get back $\vec{b}$. With real numbers in stead of symbols, you might not get back exactly $\vec{b}$, but you would get back something very close to $\vec{b}$. Otherwise the "solution" you find might not be a good solution.

Example 7.1 *Manually solve the equation system*

$$\begin{cases} 9999x_1 + 9998x_2 &= 1 \\ 9996x_1 + 9995x_2 &= 1 \end{cases}$$

Solution: A simple observation of the original linear system gives us $x_1 = 1$ and $x_2 = -1$ immediately. But let us be dumb and use Gaussian Elimination to solve the system. If our calculations are precisely enough, we get that

$$\begin{pmatrix} 9999 & 9998 & | & 1 \\ 9996 & 9995 & | & 1 \end{pmatrix} \Rightarrow \begin{pmatrix} 9999 & 9998 & | & 9997 \\ 0 & -0.00030003 & | & 0.0003 \end{pmatrix},$$

and thus the solution we find is again $x_1 = 1$ and $x_2 = -1$. □

The real problem arises in case our calculations are not so precise. Here we might get something like $x_1 \approx 1.0001$ and $x_2 \approx -0.9999$, which seems not so different from $x_1 = 1$ and $x_2 = -1$. But when we plugged this approximate "solution" to the original system, we would get that

$$\begin{cases} 9999 \times 1.0001 + 9998 \times (-0.9999) &= 2.9997 \\ 9996 \times 1.0001 + 9995 \times (-0.9999) &= 2.9991. \end{cases}$$

Consequently, the vector $(1.0001, -0.9999)$ does not appear to be close to the solution at all. Imagine you are solving a linear system whose precise solution consists of irrational numbers, not whole numbers or rational numbers. You may have great difficulty calculating the solution because vectors close to the solution may not appear to be close.

Generally speaking, we want that a small relative error in the solution leads to a small relative change in $\vec{b}$ when we plug it into the linear system. Formally, we want to keep $\frac{|A\vec{\Delta x}|/|A\vec{x}|}{|\vec{\Delta x}|/|\vec{x}|}$ small. Whether we can succeed or not depends on the matrix A itself.

Definition 7.1 *The* Euclidean norm *of an $n \times n$ real matrix A is* $||A|| = \max_{\vec{x} \in \mathcal{R}^n, \vec{x} \neq \vec{0}} \frac{|A\vec{x}|}{|\vec{x}|}$.

Definition 7.2 *Let A be an $n \times n$ nonsingular real matrix. Its* condition number *is* $\mathbf{cond}(A) = ||A|| \cdot ||A^{-1}||$.

Example 7.2 *For $n \times n$ nonsingular real matrices A and B, prove that* $\mathbf{cond}(AB) \leq \mathbf{cond}(A)\mathbf{cond}(B)$.

Solution: The inequality we need to prove is equivalent to that $||AB|| \cdot ||B^{-1}A^{-1}|| \leq ||A|| \cdot ||A^{-1}|| \cdot ||B|| \cdot ||B^{-1}||$. If we can show $||AB|| \leq ||A|| \cdot ||B||$, then we can do the same thing to their inverse matrices, and finally combining the two inequalities to obtain what we want. To see this, simply observe

$$||AB|| = \max_{\vec{x} \in \mathcal{R}^n, \vec{x} \neq \vec{0}} \frac{|AB\vec{x}|}{|\vec{x}|} \leq \max_{B\vec{x} \in \mathcal{R}^n, B\vec{x} \neq \vec{0}} \frac{|AB\vec{x}|}{|B\vec{x}|} \cdot \max_{\vec{x} \in \mathcal{R}^n, \vec{x} \neq \vec{0}} \frac{|B\vec{x}|}{|\vec{x}|} = ||A|| \cdot ||B||.$$

□

Lemma 7.1 *For an $n \times n$ nonsingular real matrix A, we always have*

$$\frac{|A\vec{\Delta x}|/|A\vec{x}|}{|\vec{\Delta x}|/|\vec{x}|} \leq \mathbf{cond}(A).$$

Proof: Easy to see

$$\frac{|A\vec{\Delta x}|/|A\vec{x}|}{|\vec{\Delta x}|/|\vec{x}|} = \frac{|A\vec{\Delta x}||\vec{x}|}{|\vec{\Delta x}||A\vec{x}|} = \frac{|A\vec{\Delta x}|}{|\vec{\Delta x}|} \frac{|A^{-1}A\vec{x}|}{|A\vec{x}|} \leq ||A|| \cdot ||A^{-1}|| = \mathbf{cond}(A).$$

□

So as long as $||A||$ and $||A^{-1}||$ are reasonably small, $\mathbf{cond}(A)$ is also small and we are happy. Otherwise, $\mathbf{cond}(A)$ is large and we call the matrix A *ill-conditioned*, which implies there could be big trouble in solving the linear system.

To calculate the value of $\mathbf{cond}(A)$, we need to calculate $||A||$ and $||A^{-1}||$. It turns out that they are linked to the *Singular Value Decomposition* (SVD) of A.

Definition 7.3 *For an $n \times n$ real matrix A, there are n* singular values *and corresponding* singular vectors.[14] *The first singular value*

$$\sigma_1 = \max_{|\vec{x}|=1, \vec{x} \in \mathcal{R}^n} \frac{|A\vec{x}|}{|\vec{x}|},$$

and the first singular vector

$$\vec{v}_1 = \arg \max_{|\vec{x}|=1, \vec{x} \in \mathcal{R}^n} \frac{|A\vec{x}|}{|\vec{x}|}.$$

For $k = 2, \ldots, n$, the kth singular value

$$\sigma_k = \max_{\vec{x} \perp \mathbf{span}\{\vec{v}_1, \ldots, \vec{v}_{k-1}\}, |\vec{x}|=1, \vec{x} \in \mathcal{R}^n} \frac{|A\vec{x}|}{|\vec{x}|},$$

and the kth singular vector

$$\vec{v}_k = \arg \max_{\vec{x} \perp \mathbf{span}\{\vec{v}_1, \ldots, \vec{v}_{k-1}\}, |\vec{x}|=1, \vec{x} \in \mathcal{R}^n} \frac{|A\vec{x}|}{|\vec{x}|}.$$

[14] The singular vectors defined here are actually the *right* singular vectors. There are also *left* singular vectors, which are the column vectors of the matrix U in SVD. For simplicity, we only consider right singular vectors in our lecture notes, and call them the singular vectors.

Intuitively, we are examining how matrix A stretches/compresses vectors. Obviously, vectors in different directions are stretched/compressed differently. The first singular vector is the direction in which vectors are stretched the most or compressed the least (and so $||A|| = \sigma_1$). We use a vector of unit length to represent this direction. Correspondingly, the first singular value is ratio vectors in this direction being stretched/compressed. The kth singular vector and singular value are similar to the first, except that we must eliminate the impact of previously found singular vectors on the stretching/compression. Therefore, we limit our study to the subspace orthogonal to all previously found singular vectors.

Now suppose a matrix has r non-zero singular values, and the remaining singular values are all 0. By the definition above, we immediately get that, any of the last $n - r$ singular vectors, and thus any $\vec{x}$ in the span of these $n - r$ singular vectors, makes $A\vec{x}$ equal to 0. Hence, the span of them is part of the null space of A. The span of the first r singular vectors is orthogonal to the span of the last $n - r$ singular vectors, and has all its vectors satisfying $A\vec{x} \neq 0$. This means the null space of A is exactly the span of the last $n - r$ singular vectors. Consequently, the rank of A is equal to r.

Theorem 7.1 *For an $n \times n$ nonsingular real matrix A, suppose $\sigma_1, \ldots, \sigma_n$ are its singular values, and $\vec{v}_1, \ldots, \vec{v}_n$ are its singular vectors. Then there exists an orthogonal matrix U such that*

$$A = U\Sigma V^T, \tag{4}$$

where Σ is a diagonal matrix with σ_k ($k = 1, \ldots, n$) being the kth element on the diagonal, and $V = [\vec{v}_1, \ldots, \vec{v}_n]$. Equation (4) is called the SVD of matrix A.

Proof: We construct $U = [\frac{A\vec{v}_1}{\sigma_1}, \ldots, \frac{A\vec{v}_n}{\sigma_n}]$. For $k = 1, \ldots, n$, $|\frac{A\vec{v}_k}{\sigma_k}| = \frac{|A\vec{v}_k|}{\sigma_k} = |\vec{v}_k| = 1$. So to show U is orthogonal, we just need to show its column vectors are pairwise orthogonal. This can be done using an induction (details of which can be found, *e.g.*, in [8]).

On the other hand, because V is clearly orthogonal, we easily get that

$$U\Sigma V^T V = U\Sigma = [A\vec{v_1}, \ldots, A\vec{v_n}] = AV \Rightarrow U\Sigma V^T = A.$$

□

Theorem 7.1 tells us that whenever we apply a linear transformation A to $\vec{x}$, essentially we do it in three steps:

1. We apply an orthogonal transformation V^T, changing the direction of the vector without touching its length. The effect of this rotation is that the direction of each singular vector is mapped to that of an axis.

2. We stretch/compress the vector in the direction of each axis, by the ratio of the singular value.

3. We apply another orthogonal transformation U, again changing the direction without touch the length. The effect of this rotation is that the direction of each axis is mapped to that of a column vector of U.

Theorem 7.1 does not give us a good algorithm for computing singular values of a matrix. Such algorithms are discussed in texts of numerical analysis such as [12]. For practical purposes, you can safely assume a software package like Matlab is available.

If we have the SVD of a matrix A, we get the SVD of A^{-1} as well: $A^{-1} = V\Sigma^{-1}U^T$. Hence, the singular values of A^{-1} is nothing but $\frac{1}{\sigma_n}, \ldots, \frac{1}{\sigma_1}$. This implies that $||A^{-1}|| = \frac{1}{\sigma_n}$ and $\mathbf{cond}(A) = \frac{\sigma_1}{\sigma_n}$.

Finally, we should realize that there are many similarities and connections between SVD and the eigenvalue decomposition we studied in our linear algebra course. In particular, by Theorem 7.1, any nonsingular real matrix A can be written as $A = \sum_{i=1}^{n} \sigma_i \vec{u}_i \vec{v}_i^T$. If A is symmetric, it can also be written as $A = \sum_{i=1}^{n} \lambda_i \vec{w}_i \vec{w}_i^T$, where λ_i is the ith eigenvalue and $\vec{w}_i$ is the corresponding normal eigenvector such that $\vec{w}_1, \ldots, \vec{w}_n$ are pairwise orthogonal. In this case, we can even show that $\sigma_i = |\lambda_i|$ (see Problem 7.4).

Problems

Problem 7.1 *(Adapted from Kannan and Guruswami's Problem Set 2 [6]) We have studied SVD of nonsingular square matrices only. Now consider, in stead, a $m \times n$ ($m \geq n$) matrix $A = [\vec{a}_1, \ldots, \vec{a}_n]$. Suppose $\vec{a}_1, \ldots, \vec{a}_n$ are pairwise orthogonal but any two of them have different lengths. What are the singular values of A?*

Problem 7.2 *Suppose there is an $n \times n$ nonsingular real symmetric matrix A, and $\lambda_1, \ldots, \lambda_n$ ($|\lambda_1| \geq \ldots \geq |\lambda_n|$) are its eigenvalues. Prove $\frac{|\lambda_1|}{|\lambda_n|} \leq \mathbf{cond}(A)$.*

Problem 7.3 *Suppose there is an $n \times n$ nonsingular real matrix A. Prove that $A^T A$ is positive definite.*

Problem 7.4 *Suppose A is an $n \times n$ nonsingular symmetric real matrix, with $\lambda_1, \ldots, \lambda_n$ being its n distinct eigenvalues. Also suppose A has n distinct singular values. Prove that the singular values are exactly $|\lambda_1|, \ldots, |\lambda_n|$.*

References

[1] Sanjeev Arora and Boaz Barak. Computational Complexity: A Modern Approach. Cambridge University Press, 2009.

[2] James Aspnes. Notes on Randomized Algorithms. Lecture Notes, 2019. Available at: http://cs.yale.edu/homes/aspnes/classes/469/notes.pdf

[3] Thomas H. Cormen. Charles E. Leiserson. Ronald L. Rivest. Clifford Stein. Introduction to Algorithms. Second Edition. The MIT Press, 2001.

[4] Haining Fan, Ming Gu, Jiaguang Sun, and Kwok-Yan Lam. Obtaining More Karatsuba-Like Formulae over the Binary Field. IET Information Security, Vol. 6 No. 1, pp. 14-19, 2012.

[5] Oded Goldreich. Foundations of Cryptography, Vol. 1. Cambridge University Press, 2001.

[6] Venkatesan Guruswami and Ravi Kannan. Problem Sets, 2012. Available at: https://www.cs.cmu.edu/~venkatg/teaching/CStheory-infoage/

[7] G. H. Hardy and E. M. Wright.An Introduction to Theory of Numbers. Sixth Edition. Oxford University Press, 2008.

[8] John Hopcroft and Ravi Kannan. Computer Science Theory for the Information Age. Lecture Notes, 2012. Available at: https://www.cs.cmu.edu/˜venkatg/teaching/CStheory-infoage/hopcroft-kannan-feb2012.pdf

[9] Laszlo Lovasz. Notes on Complexity of Algorithms. Lecture Notes, 2018. Available at: http://web.cs.elte.hu/˜kiraly/complexity.pdf

[10] M. J. Osborne and A. Rubinstein. A Course in Game Theory. MIT Press, 1994.

[11] Debmalya Panigrahi. Notes on Design and Analysis of Algorithms. Lecture Notes, 2016. Available at: https://www2.cs.duke.edu/courses/spring16/compsci330/Notes/LVMC.pdf

[12] J. Stoer and R. Bulirsch. Introduction to Numerical Analysis, 3rd Edition. Springer, 2002.