1 Competitive programming

Competitive programming is the art of **algorithmic problem solving**. Supported by Computer Science, you devise and implement elegant solutions to well-defined problems checked on a test suite.

1.1 Would you like a problem?

Here's a light-hearted one for a good start:

Tennis Since we're in the UK, you're likely aware of lawn tennis. A tennis match consists of sets, which consist of games, which consist of points. You're given a description of a full match where some $n \leq 10^6$ points were played. The description is a sequence of who won each point. Who won the match?

The obvious solution is to just implement the rules of tennis in your language of choice. If you're persistent enough, this should execute some constant number of operations per each point played. However, this is a largely unnecessary undertaking: it's sufficient to notice that the *winner of a tennis match is the winner of the last point.*

With that out of the way, what does this problem teach us?

- It's good to be lazy (but right!) as a programmer. The less work you do, the faster it'll be done and the more likely it is that you'll be correct.
- Subjectively, some solutions are worse and some are just better (*built different*, if you will). Even in Tennis, the difference lies not just in simplicity, but **performance**. A competitive programmer's way of reasoning about performance is **computational complexity**.

Computational complexity is a way of reasoning about performance metrics of a program. Importantly, it describes their scaling as we increase the problem size. It is usually[1] reasoned about with the well-known (in Computer Science) Big $\mathcal{O}$ notation.

For instance, in Tennis a solution that simulates the game will need to iterate through each of the n points and perform some fixed operations. We would call this a **linear time complexity** – $\mathcal{O}(n)$ – as with *a game that's twice as long the solution takes (approx.) twice as much time.* On the other hand, printing the last point is always just one operation: it's a **constant time** – $\mathcal{O}(1)$ solution.[2]

As another example, consider:

```
x = 0
for i in range(n):
    for j in range(i):
        x += i * j
```

This program execute its innermost loop body $n(n-1)/2 \leq n^2$ times (and that loop body has some constant number of operations in it), thus we say it is $\mathcal{O}(n^2)$.

If a solution is $\mathcal{O}(f(n, ...))$, its execution time grows at most proportionally to $f(n, ...)$.

constant	$\mathcal{O}(1)$
linear	$\mathcal{O}(n)$
quadratic	$\mathcal{O}(n^2)$
logarithmic	$\mathcal{O}(\log n)$
exponential	$\mathcal{O}(2^n)$

Similarly, besides **time complexity** (execution time) we can reason about **space complexity** (memory use). *Usually,* the former upper bounds the latter and is more important.

A common rule of thumb is that 10^7 unit operations take a second, which is on the order of the usual time limit. **Remember:** complexity hides the constant factor – an addition won't take much as a division, which will take less than a (usually constant-time: $\mathcal{O}(1)$) hash table access! Similarly, inherently slower languages (like Python) might not pass the time limit and require *constant-factor optimisations.*

[1] Usually, *usually* means *almost always, except when not.*

[2] Technically, you require linear time to read the input first. This usually doesn't come up.

1.2 Contests

The most common way to practice competitive programming is through **contests**. During a contest:

- You're given some number of problems, each with a textual **problem statement**. The statement gives specific constraints on possible **test cases** and a couple examples to check your understanding.
- For each of the problems you come up with a solution and implement it in a language of your choice that's supported by the online platform. Usually, you interact with platforms through *standard input/output* (e.g. `input` and `print`).
- You then **submit** the solution to the platform and (usually) eventually obtain a verdict (AC – all clear, WA – wrong answer, TLE – time limit exceeded, MLE – memory limit exceeded).
- There is a time limit (2 hours, 5 hours, or even 24 hours and longer), and often a submission limit (which *might* get feedback – what test case failed and with what verdict, how many tests passed, etc.). There are also limits for memory and code size, disabled filesystem/network use, ...

Based on your verdicts and a scoring system a final ranking is obtained.

At competitive programming **workshops**, usually there's a talk about some problem solving techniques, followed by a problem solving session (which is like a contest, but there's no time limit and there's more feedback available on your submissions).

1.3 Cultural context

Competitive programmers form a neat little "subculture" in the programming community, loathed by everyone that's preparing for programming interviews. I thought it'd be useful to give some context – *feel free to skip this section and move on to today's algorithms*!

Competitive programmers are known for their terrible code quality that derives from the time limits present in contests. Many optimise for low character counts, fast typing, and use of macros. However, they also have to be good at **debugging**, testing, correctness, and performance.

A good way to appease a competitive programmer is to share a quality problem with them (in Polish, the term is *zadanka* – unfortunately, it cannot be defined in English). They often end up as software engineers .

Platforms The most popular platform the runs open online contests is Codeforces (CF), across a range of divisions (Div. 3 for beginners, Div. 2 for intermediate, Div. 1 for advanced). Other common platforms include atcoder, Codechef, Kattis, OJ.

Resources Past problems, besides platforms specific to contests/olympiads, can be found on the above platforms (especially Codeforces/Kattis) and other ones like CSES. I recommend CSES for a robust suite of problems to solve, and the Competitive Programmer's Handbook! Also see `cp-algorithms` for a selection of well-written guides.[3].

Contests The most famous contest is the International Collegiate Programming Contest (ICPC, formerly ACM-ICPC). This is a competition for 3-person teams of university students up to a certain age, divided in various stages. For instance, in Cambridge our teams participate in:

- Subregional stage: **UKIEPC** – this one is somewhat unofficial, usually used by universities to qualify the best teams for regionals.
- Regional stage: **NWERC** – every university can send at least t teams (usually $t \in \{1, 2\}$). NWERCs 2022–2024 have taken place in Delft, Netherlands.
- **ICPC World Finals** – the Real One. Only a couple top teams from regionals are admitted, and a student can only participate a limited number of times.

Teams that do well in regionals/finals are awarded medals (usually 4 of each) and bragging rights.

In some countries, high school Informatics Olympiads (akin to Mathematical Olympiads) are particularly popular. They usually feature competitive programming problems. Of many international regional contests, the most prestigious one is the **International Olympiad in Informatics** (IOI). A good way to tell what countries have good traditional Informatics Olympiads is here.

[3] If you're not convinced why you should learn foreign languages, it's worth noting it's mostly translations of `e-maxx.ru`.

2 Binary search

Although the basic idea of binary search is comparatively straightforward,
the details can be surprisingly tricky.
–Donald Knuth

Binary search is a somewhat popular algorithm. Here's a reminder of how it's usually presented.

Suppose we are given a **sorted** array a of $n = |a|$ integers and we are looking to determine whether x is among ~~us~~ them. For example, say we are looking for 42 in:

$$0 \mid 1 \mid 7 \mid 11 \mid 24 \mid \mathbf{42} \mid 111$$

In $\mathcal{O}(n)$[4] time we can simply iterate the entire list and determine whether x is present. But we are given a strong assumption: the list is sorted, and thus knowing the element at any index i informs us on where x *can* lie in the array. Indeed, supposing we require p such that $a_p = x$ we have the rule:[5]

$$\begin{cases} p \leq i, & \text{if } a_i \geq x \\ p > i, & \text{if } a_i < x \end{cases}$$

Indeed, by checking that in $[0, 1, 7, 11, 24, 42, 111]$ index 4 contains $24 < 42$, I can infer 42 must lie in the suffix of the list – $[42, 111]$ (i.e. $p > 4$). In the end, whichever i I query, I can end up with some sublist of a that must contain x if occurs in a. In relatively few queries I can determine where x might be by checking the midpoint of the possible range.

Abstracting Note that in this problem we don't *really* care about exact numeric values of a_i nor x – we merely need to know whether $x \geq a_i$ for any i. Let us then define $\mathcal{P}(i) = [a_i \geq x]$ and note that now the inference rule becomes:

$$\begin{cases} p \leq i, & \text{if } \mathcal{P}(i) \\ p > i, & \text{otherwise} \end{cases}$$

With that view, let's look at values of $\mathcal{P}(i)$ (with $\bot$ meaning false, and $\top$ meaning true):

$$\bot \mid \bot \mid \bot \mid \bot \mid \bot \mid \top \mid \top$$

Why does the rule work at all? Due to *monotonicity*. If $\mathcal{P}(i)$, then necessarily $\mathcal{P}(i+1)$, since $x \leq a_i$ ($\mathcal{P}(i)$) and $a_i \leq a_{i+1}$ (a sorted) implies $x \leq a_{i+1}$ (transitivity).

Why is the rule useful? Because x, if it occurs in a, is the smallest number $\geq x$ in a. What we are thus looking to determine is the smallest p for which $\mathcal{P}(p)$ – and the rule allows exactly that. It might occur that $\neg\mathcal{P}$ everywhere, in which case $a < x$ everywhere and thus x cannot be in a.

Generalising With that done, we can summarise what binary search is really about:

Given a range $[\texttt{lo}, \texttt{hi})$ and a monotonic predicate $\mathcal{P} : [\texttt{lo}, \texttt{hi}) \to \mathbb{B}$, find the minimum $p \in [\texttt{lo}, \texttt{hi})$ such that $\mathcal{P}(p)$ holds, or determine it does not exist.

The algorithm is as follows:

```
def binary_search(P: int -> bool, lo: int, hi: int) -> int | None:
    # let t be the minimum index for which P(t)
    # invariant: t lies in [lo, hi], if it exists
    lo0, hi0 = lo, hi
    while lo < hi:
        mid = (lo+hi) // 2  # midpoint: floor divide by 2
        if P(mid):
            hi = mid        # t <= mid
        else:
            lo = mid + 1    # t > mid
    # if lo = hi0, P holds nowhere in [lo0, hi0]
    return lo if lo < hi0 else None
```

[4] By now, instead of *oh open paren en close paren* the reader should pronounce this in their head as *linear*.

[5] This is technically a bit wasteful, and more information can be inferred. However, for the purpose of generality – and simplicity! – follow this approach for now.

Note this implementation follows the rule above! This is the binary search I always use and reduce all my problems to. Binary search is notorious for unpleasant edge cases, comparisons the wrong way, and other similarly subtle errors.

It thus becomes obvious the time complexity is $\mathcal{O}(\log n)$ ($n = \texttt{hi} - \texttt{lo}$ if $\texttt{lo}, \texttt{hi} \in \mathbb{Z}$) – one can check n is halved with each iteration and the loop terminates with $n = 0$.

Applying To solve the find-in-sorted-list problem, we simply use the binary search routine defined above and handle the possible cases.[6]

```
p = binary_search(lambda i: a[i] >= x, 0, len(a))
if p is None:
    print("no x in a -> all elements in a are smaller!)
# p is the smallest i s.t. a[i] >= x
elif a[p] == x:
    print(f"x is in a -> index {}")
else:
    print(f"no x in a -> the smallest element >= x is > x"
```

The binary search could be inlined here, but this natural abstraction leads to clearer code.[7]

Advice The key to solving problems involving binary search is *monotonicity*. It is a hugely beneficial property that is also used in other techniques. Look for properties (predicates) that are initially false and later true (e.g. as you traverse an array, as time passes, ...). If there is one, you can likely binary search over it to determine what's the earliest point when it's satisfied.

In my early days of competitive programming I struggled with binary search problems. I thought too much of *searching an array* rather than in general – *searching a predicate defined over a range*.

In general, evaluations of the predicate might not take constant time. They might involve accessing a data structure or simulating a process (even running a greedy algorithm). Despite this, binary search scales really well – logarithmic complexity is often as good as you can ask for, though there are sometimes ways to *optimise out the logarithm factor*.

[6] `lambda x: ...` defines an "anonymous" function, effectively doing `def _(x): return ...`

[7] Most competitive programmers would avoid passing a function as an argument and simply implement the binary search directly – my technique might be unorthodox to some. In fact, that might be easier to understand at first to do it without an extra routine, and it might be necessary for Python to pass time limits – abstractions rarely come for free!

3 Greedy algorithms

Greedy algorithms are all about making locally optimal (some would say *obvious*) choices that turn out to be globally optimal. One thinks of the solution as constructed from these local actions/parts that build up the entire result.

A good example of algorithmically greedy behaviour in real life is how you might take rides on public transport. Say you perfectly know your route and won't make changes to it (this is realistic if you just chose the first result on, say, Google Maps). Say there's a sequence of bus lines to take and stops where you get on/off. Then whenever you're waiting for a bus, it obviously makes sense to (locally) take the earliest one of the right line – *usually* it will be the first one to arrive at your next transit, and you'll thus be able to take the earliest possible bus of your next line. These local greedy choices of earliest-is-best compose into a globally optimal solution.

Kayaks [4th Polish OI] Suppose we have some n rowers and a number of one- or two-person kayaks. Rower i has weight w_i, and a kayak can only support weight W. What's the minimum number of kayaks needed to accommodate everyone?

There are certain intuitions one can have about this problem. For instance, it makes sense not to waste spare space on a kayak – we should pack it up as much as possible. It's possible to build up a solution around this idea. Another approach is to exploit an inherent *degeneracy* in how the problem is constrained. It can be shown that if a solution exists, then one of a really specific form does as well – and we shall construct it using an *exchange argument*.

Exchange argument Take an **optimal** valid kayak assignment $\{(i_1, j_1), ..., (i_k, j_k), i_{k+1}, \ldots, i_\ell\}$. Consider people of weights $a \le b \le c \le d$ such that they together got assigned to 2 two-person kayaks. Then we must have one of the following:

- The pairings are (a, c) and (b, d). **Notice** that the pairing (a, d) and (b, c) would also work, since $a + c \le b + d \le W$ and $b + c \le b + d \le W$.
- Similarly, if the pairings are (a, b) and (c, d), then (a, d) and (b, c) would also work.
- And in the case of (a, d) and (b, c) we won't exchange anything.

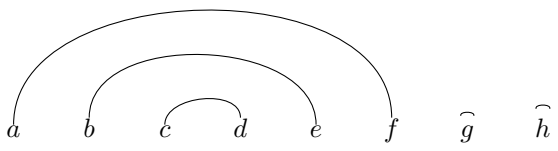
Through these cases, we can construct a solution where for any pair of two-person kayaks with weights (x, y) and (z, w), we must have either $x < z < w < y$ or $z < x < y < w$.

This can be thought of as *simplifying* the relative ordering of people in kayaks. The three cases are seen below (note we reduce the first and second to the third):

[Note that in a different problem we may have instead reduced to the second case, which would lead to neighbours being paired together.]

Analogously, say that we have a 2-person kayak (a, b) and 1-person kayak c. Then, we can enforce $a < b < c$ – if b was heaviest, we could replace it with c, while preserving kayak constraints.

By applying all the above exchanges, it can be shown that if an optimal solution exists, there's also one of this simplified form:



At this point, it might become obvious what the algorithm should be:

- Until there's no one left, take the heaviest rower left.
- If there's anyone else left, consider the lightest rower left – if they can be paired with the heaviest, put them in a kayak together; otherwise put the heaviest by themselves.

The **exchange argument** is an elegant way of designing correct-by-construction greedy algorithms. It relies on showing the solution can be reduced to a particular form that can be easily produced greedily. However, it's just one possible way of discovering greedy algorithms!

Advice Greedy algorithms are one of the widest solution domains and are often considered *ad hoc*, as there are few techniques to follow (the exchange argument is perhaps most common, followed by *just try something that might work*). To find greedies, it often helps to look at small examples and play around with simple ideas. Indeed, *a solution that fits in the time limit exists*.[8] Often, a greedy argument will be part of a bigger solution, be it as part of a dynamic programming algorithm or a data structure.

As a beginner, the biggest trap is following your intuition a bit too much – it's important to try to build up reasons on *why* your ideas might be correct. Often, bad greedy solutions end up as a patchwork of ideas: you start with something simple, and as you find corner cases add up extra heuristics. This is a telltale sign the solution is wrong.[9] Even if you're right, a complicated solution is difficult to debug. It's good to imagine a sketch of an argument on why your approach is correct in the long run, and simpler solutions are usually easier to argue for.

Thank you and hope to see you next time! -Kuba

[8] Usually.
[9] Usually, but disappointingly not as much as I'd like.