Bull's eye!

Individual project

Programming Languages and Environments
Department of Computer Science @ NOVA FCT

April 4, 2025
Version α

Versions

α Initial version of the problem statement. (April 4, 2025)

Introduction

Darts is a precision sport where players throw darts at a circular target (dartboard) fixed to a wall. The standard board is divided into numbered sections, with the goal of scoring points by hitting specific areas. Among a plethora of popular games, one can play darts, the games *501* and *301* standout for requiring the players to have fine-grained precision and strategy. In both variants, players start with either *301* or *501* points and aim to reduce their score to exactly zero by throwing the darts at specific segments of the board.

Players take turns throwing three darts per round, subtracting the points scored from their total. The scoring is based on the dartboard's segmentation: single sections award face value points, triple rings multiply the section's value by three, and double sections multiply the section's value by two. The outer bull awards 25 points, while the inner bull (bull's eye) awards 50 points.

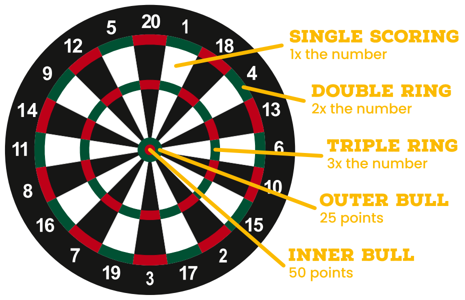

Figure 1: Dart board

A particular challenge lies in the checkout mechanism: to win, a player must precisely reduce their score to zero, **in one round of three darts**, with the final dart landing in a double segment or the inner bull. If a throw reduces the score below zero or leaves one point remaining, the entire turn is invalidated, and the player's score reverts to its previous state. It is important to note that not all scores are possible to checkout within the given constraints. For instance, it is not possible to checkout with a score greater than 170. In a single play, three darts are used, with the sole checkout being the first two darts being triple 20s, followed by a bull's eye.

Problem representation

To represent throws, you can use the following type definition, a sum type `throw`:

```ocaml
type throw =
  | S of int (* singles: S 1, S 2, ..., S 20, S 25 *)
  | D of int (* doubles: D 1, D 2, ..., D 20, D 25 *)
  | T of int (* triples: T 1, T 2, ..., T 20 *)
```
OCaml

Additionally, you can represent a checkout as a list of `throw`, and in the context of this problem, the maximum length of this list should be three, seeing as we only have three darts to throw.

```ocaml
type checkout = throw list
```
OCaml

Finally, all the possible ways to checkout can be given as a list of `checkout`, as such:

```ocaml
type checkouts = checkout list
```
OCaml

For example, the checkout that yields 170 points is represented by the list:

```ocaml
[T 20; T 20; D 25]
```
OCaml

Technical Tasks

This handout is split into two incremental parts. The second part builds upon the first and cannot be completed without successfully implementing the first. The tasks should done using the **inductive style**.

Checking out

Grade bound: 18/20

In this task, you are required to determine the different ways a player can checkout given a target score in a single play. A checkout is a sequence of darts that brings the player's score to exactly zero, with the restriction that the final dart must land in a double score area. In addition, we shall not include misses in considering combinations. For example, the play `D3` is the same as `0 D3` and `0 0 D3`.

Your task is to complete the `compute_checkouts` function in the file `lib/task1.ml`. This function receives a positive integer representing the target score and returns a list of valid checkout sequences:

OCaml
```ocaml
let compute_checkouts (score: int) : checkouts = (* Complete here *)
```

Example

If you execute the following command, which will compute `task1` with an input of 6:

Shell
```
$ dune exec bullseye task1 6
```

You should get the following 14 distinct ways to checkout[1]:

Output
```
D3
D1 D2
D2 D1
S2 D2
S4 D1
D1 D1 D1
D1 S2 D1
S1 S1 D2
S1 S3 D1
S1 T1 D1
S2 D1 D1
S2 S2 D1
S3 S1 D1
T1 S1 D1
```

Checkout equivalence

Grade bound: 20/20

In the previous task, we considered all possible ways to checkout with a given score, including sequences that are simple permutations of each other. However, in this phase, you are tasked with refining your approach to treat checkout sequences as equivalent if they contain the same dart scores in a different order. For example, the checkouts `S1 T1 D1` and `T1 S1 D1` achieve the same final score using the same dart values but in a different order. Since they are functionally identical, we count them as a single unique checkout. Though, note that `D1 D2` is not the same as `D2 D1`, seeing as the final dart is different.

Now, your task is to complete the `compute_checkouts` function, in the file `lib/task2.ml`. This function receives a positive integer representing the target score and returns a list of valid checkout sequences, given this new constraint:

OCaml
```ocaml
let compute_checkouts (score : int) : checkouts = (* Complete here *)
```

[1]The output is sorted by the number of used darts. This is only for presentation purposes, you are not required to sort.

Example

If you execute the following command, computing `task2` with an input of 6:

```
$ dune exec bullseye task2 6                                    Shell
```

You should get the following 11 distinct ways to checkout, given this new constraint:

```
D3                                                              Output
D1 D2
S2 D2
D2 D1
S4 D1
S1 S1 D2
S1 T1 D1
S1 S3 D1
D1 D1 D1
D1 S2 D1
S2 S2 D1
```

Tests and execution

To develop and test your project, you need to accept a github classroom assignment and clone the corresponding git repository. The repository contains a set of tests that validate the programs with various target scores. These tests rely on the `OUnit2` library, which provides a framework for unit testing in OCaml. Before running the tests, ensure that the package is installed by executing the the following command:

```
$ opam install ounit2                                           Shell
```

Once the package is installed, you can run the tests using `dune runtest`. If all tests pass, you should see output similar to this, showcasing the number of tests run for each task and the total execution time:

```
..........                                                      Output
Ran: 15 tests in: XXX seconds.
OK
..........
Ran: 15 tests in: YYY seconds.
OK
```

Conversely, if you want to run tests for a specific task, you can do so by executing the following commands:

- For the first task: `dune build -f @runtest_task1`
- For the second task: `dune build -f @runtest_task2`
- Or both: `dune build -f @runtest_both`

Moreover, to execute your implementation of either task, without running the test suite, you can do so by executing the following command: `dune exec bullseye <task1|task2> <score>`. Executing the first task with an input of `170` should yield:

```
$ dune exec bullseye task1 170                                  Shell
T20 T20 D25
```

Submission of the project

The submission of the project is performed by pushing your changes to the repository associated to the github classroom assignment distributed before the deadline.

Evaluation

The project will be evaluated automatically after submission using tests and then screened to evaluate the correctness and conformance to the requirements.

Remarks

Recent advancements in Large Language Models (LLMs) have significantly changed the way people write code, offering faster development and easier access to solutions. However, LLMs can produce incorrect results or hallucinate information, making it crucial to verify their outputs. As disclosed at the start of the course, you are allowed to use these tools in the project, but **you take full accountability** for your submission. If any part of the solution is generated with the assistance of an LLM (*e.g.*, ChatGPT, Copilot, etc.), the submission must explicitly state where and how LLMs were used (*e.g.*, as a comment).

You are encouraged to discuss general aspects of the project with each other (including in the Discord channel). But, when it comes to find detailed solutions and write concrete code, it has to be an internal effort by each individual. Solving problems and writing code requires intellectual effort, but only with effort can you evolve.

Beware of academic fraud. Each individual is responsible for its project, has to produce original code, and cannot show or offer, directly or indirectly, on purpose or unintentionally, its code to another individual. Note that it is much better to have zero in one of the three projects, than to be immediately excluded from the LAP course.

Have fun!