# Computational solution

International Baccalaureate · IB Diploma · Computer Science · SL

## Scope and prerequisites

This handout teaches the work behind the internal assessment, the **computational solution**, for the course first assessed in 2027. All of it is SL and HL content.

Your teacher will give you the current assessment criteria, word limit and deadlines; this handout does not repeat them, because they belong to the IB guide. It teaches how to do each stage well, with one fictional project as a running example.

Prerequisites: Theme B, especially B2 programming. The example project is invented for practice.

## Choosing a problem

### ■ A real problem with a real user

A good project starts with a **real problem** for a real person or group, the **end user**, whom you can talk to while you work. "A program that does maths" is not a problem; "the chess club's coach spends two hours each week pairing players by hand" is.

Check three things before you commit:

| Question | Why it matters |
|---|---|
| Can I talk to the user several times? | you need their requirements and their feedback |
| Does it need more than a few lines of code? | you must show a range of techniques |
| Can I finish it in the time given? | an unfinished product cannot be tested |

*Running example.* A school's chess club coach wants a program that records results, keeps a rating for each player and suggests fair pairings for each week's games.

### ■ Success criteria

**Success criteria** are the specific, testable things the finished product must do. You agree them with the user, and you **test against every one** at the end, so each must be clear enough to test.

| Weak criterion | Strong criterion |
|---|---|
| The program is easy to use. | The coach can enter a game result in under 30 seconds, with no more than three clicks. |
| It stores the players. | The program stores each player's name, year group and rating, and keeps them after it is closed. |
| Pairings are fair. | Each week, players are paired so that no two players meet twice in a row and rating gaps are as small as possible. |
| It is secure. | Only the coach, with a password, can change results. |

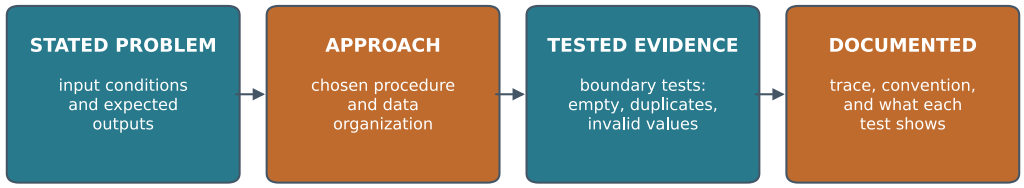
A strong criterion names **who**, **what** and **how it will be checked**. Six to ten criteria are usual for a project of this size.

## Planning and design

### ■ Decompose before you code

Break the problem into parts, then design each part: the data, the algorithms and the screens.

The computational solution pipeline




Running is not the requirement: the delivered contract includes the boundary behaviour.

*A computational solution runs from a stated problem through an approach and tested evidence to documentation*

*Running example.* The chess program splits into: storing players and results; updating ratings after each game; making the week's pairings; the coach's screens.

| Part | Design product |
|---|---|
| data | a table of the fields, their data types and an example; a file or database layout |
| algorithms | a flowchart or pseudocode for each non-trivial process, such as pairing |
| screens | a sketch of each screen, checked with the user |
| testing | a test plan, written **before** coding |

### A test plan

A **test plan** lists each test, the data used and the expected result. Use three kinds of test data:

- **normal data** : typical values that should be accepted;
- **boundary data** : values at the edge of what is allowed;
- **invalid data** : values that should be rejected.

| Test | Criterion | Data | Type | Expected result |
|---|---|---|---|---|
| 1 | result entry | win for player A | normal | A's rating rises; B's falls |
| 2 | stores year group | year group 7 (lowest allowed) | boundary | accepted |
| 3 | stores year group | year group 14 | invalid | rejected with a message |
| 4 | pairing | 9 players | boundary (odd number) | one player gets a bye; no pair repeats |
| 5 | security | wrong password | invalid | results cannot be changed |

Each test points to a success criterion, so at the end you can show which criteria are met.

## Development

### Explaining your techniques

The product must work, but the write-up must also show **how** it works and **why** you chose each technique. For each important technique:

1. name it;
2. show a short, relevant extract of your code;
3. explain what it does in **this** product;
4. say why you chose it over an alternative.

*Running example.* Pairing sorts the players by rating, then pairs neighbours, so that rating gaps stay small:

```python
def make_pairings(players):
    """Pair (name, rating) tuples; return the pairs and the bye."""
    ranked = sorted(players, key=lambda p: p[1], reverse=True)
    bye = None
    if len(ranked) % 2 == 1:
        bye = ranked.pop()      # lowest rating sits out
    pairs = []
    for i in range(0, len(ranked), 2):
        pairs.append((ranked[i][0], ranked[i + 1][0]))
    return pairs, bye
```

*Explanation.* Sorting by rating puts players of similar strength next to each other, so pairing neighbours gives the smallest rating gaps; this meets the fairness criterion. With an odd number of players, the lowest-rated player gets a bye, which test 4 checks. A random pairing would be simpler, but it could match the strongest player with a beginner.

A list of every Python feature you used is not an explanation. Choose the techniques that matter, such as a sort, a search, a file or database, validation, a class, and explain them well.

### Using other people's work

If you use a library, a code example from a website, or an **AI tool**, say where it came from and what you changed. This is part of **academic integrity**: work that is not yours must be **acknowledged**. Your own explanation of how the code works shows that you understand it.

## Testing and evaluation

### Testing against the criteria

Carry out the test plan and record the **evidence**: a screenshot or a short video for each test, with the actual result beside the expected one. A failed test is not a disaster: fix the code, test again, and record both.

| Test | Expected | Actual | Met? |
|---|---|---|---|
| 3 | year group 14 rejected | rejected: "Year group must be 7 to 13" | yes |
| 4 | 9 players: one bye, no repeat pair | bye given; one pair repeated from last week | **no**: fixed by checking last week's pairs, then retested |

### ■ Evaluation

An **evaluation** judges the finished product against **each** success criterion, using the test evidence and the user's feedback.

| Weak | Strong |
|---|---|
| The program works well and the coach likes it. | Criterion 1 is met: in testing the coach entered results in 12 seconds on average (test 1). Criterion 3 was only partly met: the first version repeated a pair (test 4); after the fix, no pair repeated over four trial weeks. |

Then suggest **specific improvements**, linked to the evidence: "add an option to print the pairings, as the coach asked in feedback" is better than "add more features".

## Practice

1. Rewrite this success criterion so that it can be tested: "The program should be fast."
2. For a criterion "The program accepts a player's age from 11 to 18", give one normal, one boundary and one invalid test value.
3. Explain why the test plan is written before the code.
4. A student's write-up lists "variables, loops, if statements, functions". Explain why this earns little, and what should be written instead.
5. The `make_pairings` function above is given 6 players with ratings 1500, 1420, 1610, 1380, 1550, 1490. List the pairs it returns.
6. A student used an AI tool to write a validation function. State what the student must do.

**Answers.**

1. E.g. "The pairings for up to 40 players appear within 2 seconds of the coach pressing *Pair*."
2. Normal: 15. Boundary: 11 or 18. Invalid: 10, 19, or a value such as "twelve".
3. It fixes, from the success criteria, what must be shown to work, so testing is planned rather than chosen after the fact to suit the product; it can also shape the design.
4. It names features without showing why they were used or how they solve the user's problem. Instead: choose a few important techniques, show a short code extract of each, and explain what it does in this product and why it was chosen.
5. Sorted from the highest: 1610, 1550, 1500, 1490, 1420, 1380. Pairs: (1610, 1550), (1500, 1490), (1420, 1380); no bye, as 6 is even.
6. Acknowledge the tool and what it produced, show what was changed, and explain in their own words how the function works.

## Exam tips

- Pick a problem with a real user you can consult, and agree testable success criteria with them early.
- Every criterion → a test in the plan → evidence in testing → a judgement in the evaluation.
- Explain a few techniques well: code extract, what it does here, why it was chosen.
- Acknowledge every library, code source and AI tool.
- Keep the current assessment criteria from your teacher beside you, and check each section of your work against them.