Describing algorithms

by Hans Wichman

What do we mean with describing algorithms?

- Pre-code design activity to make it clear what we want to code
- Post-code documentation activity to make it clear what we coded

- Audience: 'others', but first and foremost to ourselves

- Different methods available (different tools in your toolbox)
  - Non visual: Pseudo code
  - Visual: Flowcharts / 'Storyboards' / Animations

Pseudocode

Pseudocode

- Pseudocode is an **informal high-level description** of the operating principle of an algorithm. It uses the structural conventions of a normal programming language, but is **intended for human reading** rather than machine reading.
- Pseudocode is a **step-by-step written outline** of your code that you can **gradually transcribe** into the programming language. Many programmers use it to **plan out** the function of an algorithm before setting themselves to the more technical task of coding.
- Pseudocode serves as an **informal guide**, a tool for **thinking** through program problems, and a communication option that can help you explain your ideas to other people.

In short

- Informal: might look like code, might contain (English) sentences
- No strict rules → whatever works best
- Tool to make things clear / communicate / organize / think
- Ignores a lot of the 'hard' core details:
  - makes it clear what the principle of the algorithm is
  - but it still requires (hard) work to translate it into code
- In theory: independent of implementation language

Some rules of thumb (examples will follow)

- First write down the purpose of the algorithm
- Use one statement per line
- Use white space and indentation
  effectively for breaking up and ordering things
- Choose right abstraction level
- Decide on your audience
- Keep the goal in mind, every tool needs to be applied for the correct reason, in the correct context

Pseudocode in practice

Given a die with 10 sides,
write an algorithm in pseudocode to detect which side is up.

First try:

```
//This algorithm detects which die side is up in a DX (X = #sides,eg D10)
return the die side that is up
```

- Although you can't argue with this, this description is so generic, so close to the problem itself, that the usefulness is extremely limited
- You are still left asking: "Yeah, but how??"

Second iteration

```
//This algorithm detects which die side is up in a die
for each dieSide in the die
        check if dieSide is the closest match if so return dieSide
end for
```

- True, but still circumvents the core problem of 'closest match'
- Again, you are still left asking: "Yeah, but how??"

Third iteration

```
//This algorithm detects which die side is up in a die

matchingDieSide = null
for each dieSide in the die
        if matchingDieSide is null or dieSide.y position > matchingDieSide.y position
                matchingDieSide = dieSide
        end if
end for
```

- Much better, still simple, but clear, to the point etc
- Tells you that you need to be able to deduct a side's y, so not trivial/complete

Is it also correct?

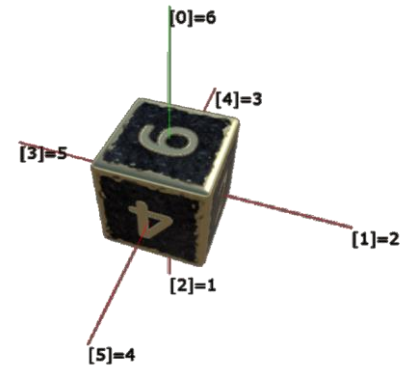
Works perfectly, the center of the side with the number 2 on it, is clearly the side with highest y position.

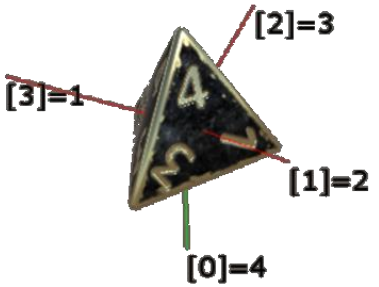
Whoops, here we need to search for the side with the lowest y position !

Alternative 1

```
//This algorithm detects which die side matches in a die
matchingDieSide = null
maxAlignment = 0

for each dieSide in the die
        currentAlignment = dot (dieSide.normal, matchVector)
        if matchingDieSide = null or currentAlignment > maxAlignment
                matchingDieSide = dieSide
                maxAlignment = currentAlignment
        end if*
end for
```



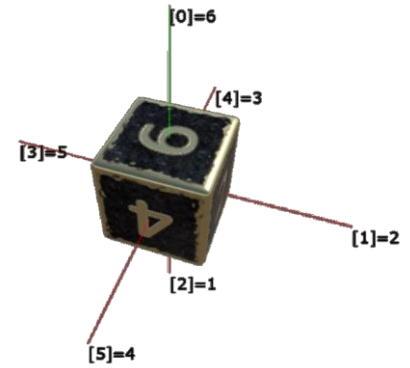


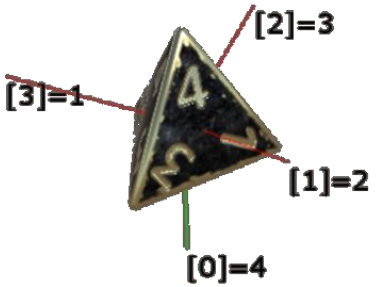
* whether you only use indents, or brackets, or for end for/if end if is completely up to you

Alternative 1 – Slightly more expressive

```
//This algorithm detects which die side matches in a die
matchingDieSide = null
maxAlignment = currentAlignment = 0.0 // <- this indicates the type

for each dieSide in the die
        //use dot to calculate alignment of two vectors; bigger means more aligned
        currentAlignment = dot (dieSide.normal, matchVector)
        if matchingDieSide = null or currentAlignment > maxAlignment
                matchingDieSide = dieSide
                maxAlignment = currentAlignment
        end if
end for
```





Match with code (from the actual asset)

```
DieSide closestMatch = null;
float closestDot = 0;
bool exactMatch = false;

for (int i = 0; i < sideCount; i++)
{
    DieSide side = _dieSides[i];
    float dot = Vector3.Dot(side.normal, localVectorToMatch);

    if (closestMatch == null || dot > closestDot)
    {
        closestMatch = side;
        closestDot = dot;
    }
}

return closestMatch;
```

Advantages / disadvantages of pseudocode

+

- Quick to evaluate, iterate, discuss solutions
- Easily modified

-

- Not visual, can be complicated to wrap your head around
- Harder to read ***because*** it doesn't match a specific language
- No accepted standard

Pseudocode: is it obligated?

- Is it absolutely required to write pseudo code?
  - It's a tool, which you can use, depending on the job
  - Can be used as a documentation tool either before or after programming
    - Major steps in pseudo code turn into methods in final code
- At minimum, as a programmer, you need to know what it is

More resources

- https://en.wikipedia.org/wiki/Pseudocode
- https://www.geeksforgeeks.org/how-to-write-a-pseudo-code/
- https://www.wikihow.com/Write-Pseudocode

- Etc just google for pseudo-code, but also:
  - notice the difference between the different descriptions
  - ***that*** pseudocode works/is a valid approach is a given,
    but ***how***/**what** works best for you might be different for everyone

The importance of visuals

Visuals

- Pseudocode is nice, but as the old adage says:

"A picture says more than a 1000 words"

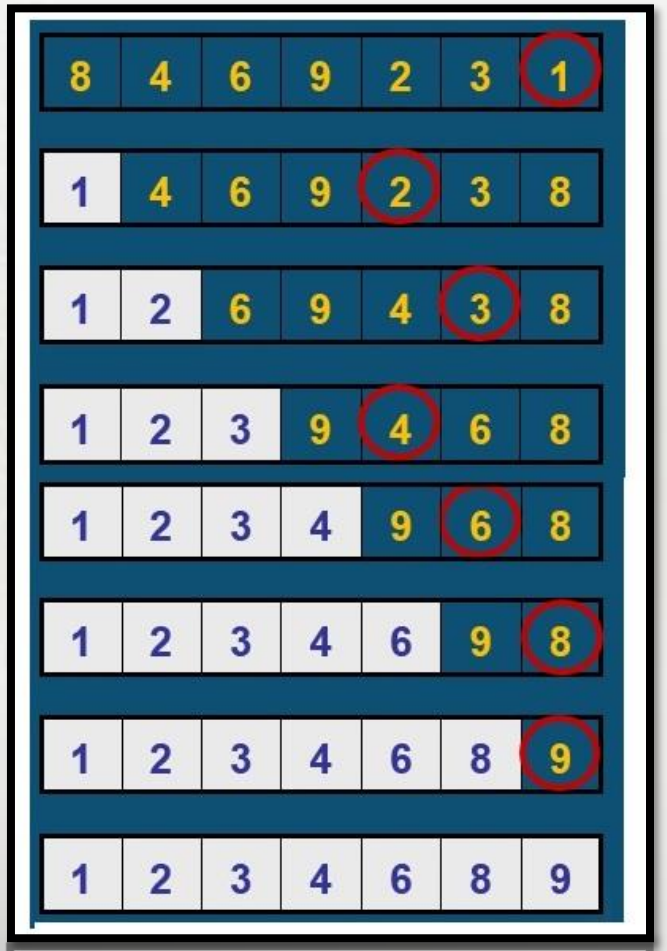
- (but I couldn't find a pretty picture about that ☺)
- Visuals can help organize your thoughts/gain clarity sometimes better than with pseudocode
- It is not one **or** the other, often it is one **and** the other:
  when you *get* the *picture*, you *get* the *algorithm/pseudocode*

Pseudocode vs visualization: example 1

Pseudocode of Selection Sort

```
SELECTION-SORT(A)
1.    for j ← 1 to n-1
2.        smallest ← j
3.        for i ← j + 1 to n
4.            if A[ i ] < A[ smallest ]
5.                smallest ← i
6.        Exchange A[ j ] ↔ A[ smallest ]
```

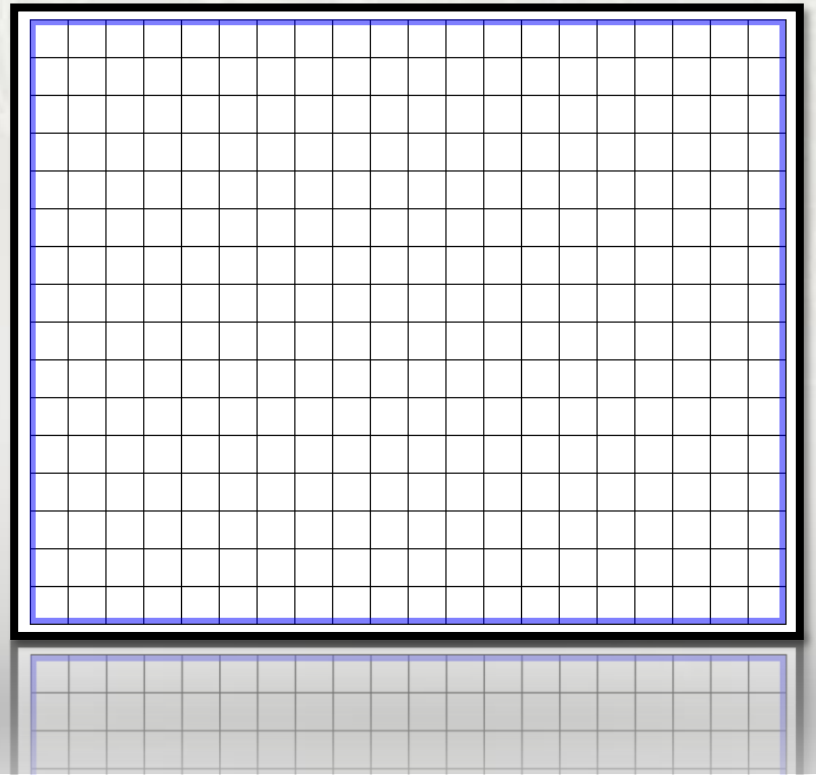
vs



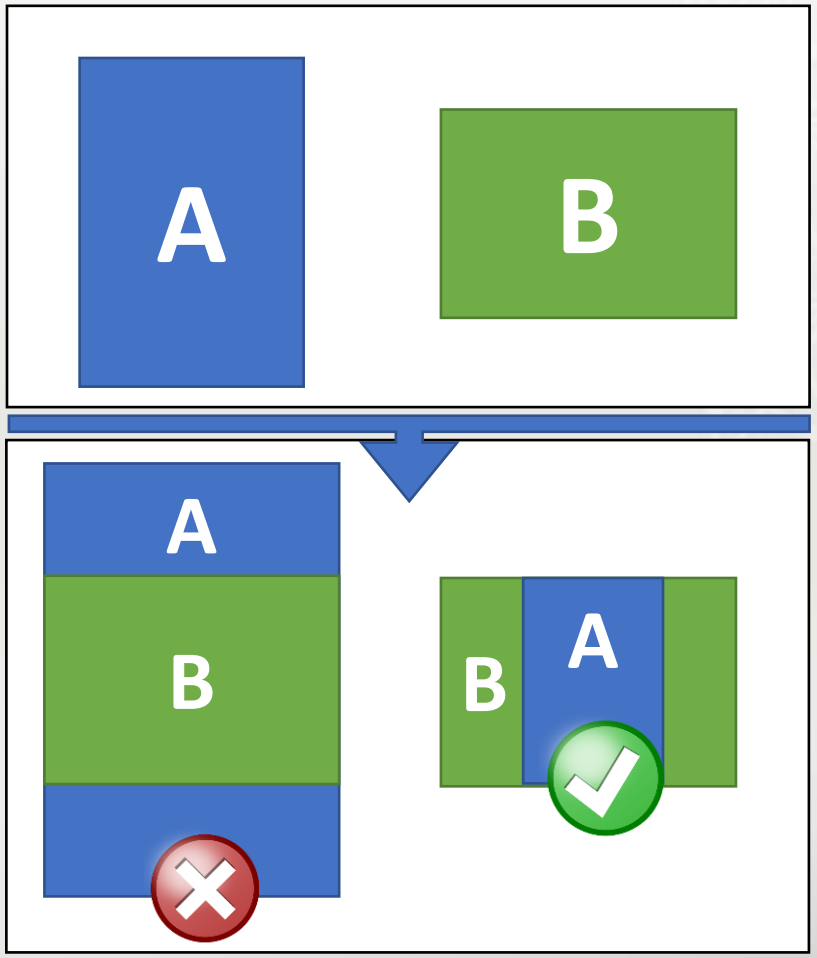
Pseudocode vs visualization: example 2

Divide all dividable rooms, until there are no more dividable rooms left

vs

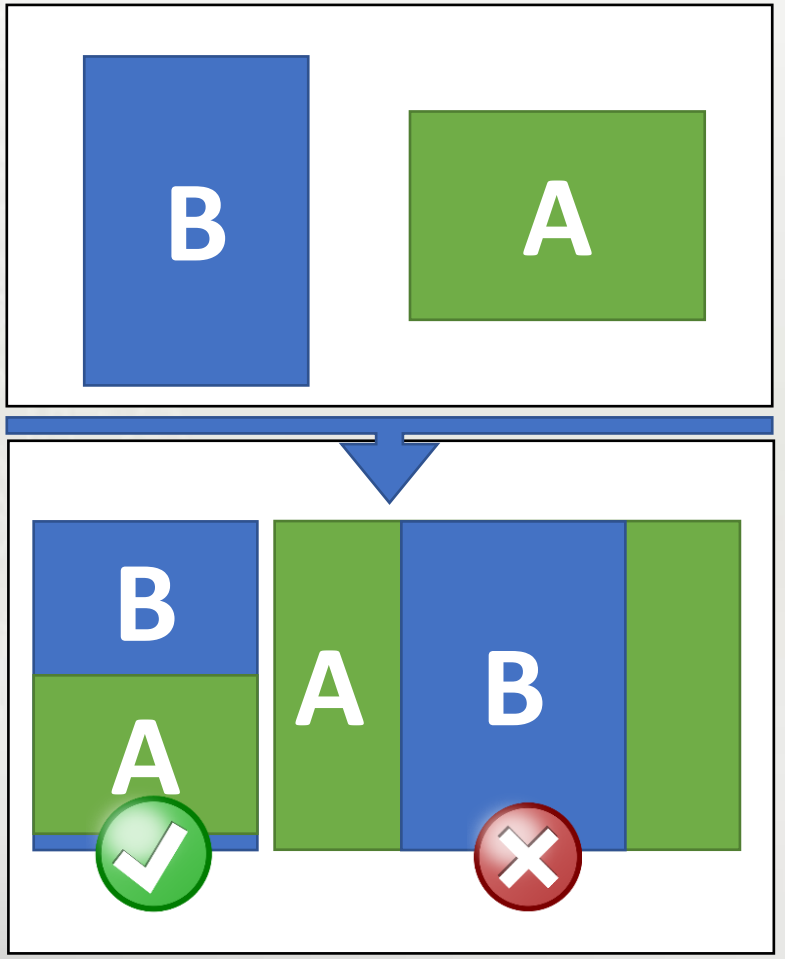


Pseudocode vs visualization: example 3



"Write an algorithm that scales A so that it fits exactly into B, keeping it's aspect ratio intact."

```
//This algoritm scales A to fit into B
set scale1 = B.width/A.width
set scale2 = B.height/A.height
return min (scale1, scale2)
```



Summing up

- There are multiple ways to describe/document algorithms:
  - pseudocode
  - visuals (sketches, animations, storyboards, etc)
  - actual code

- They go hand in hand:
  - one leads to the other
  - one is not better or worse than the other, they complement each other

Moral of the story

- Write things down (on paper) before you actually touch a computer:
  - some algorithms are better drawn,
  - some are better explained through a piece of pseudocode,
- but almost **no** algorithm is best explained through a piece of *undocumented* source code
- Read the rubrics & grading criteria ☺ :

Criterium	Insufficient	Sufficient	Good	Excellent
Student describes algorithms using e.g. flowcharts & pseudocode	There are no demonstratable flowcharts, logbook, pseudocode for any of the implemented algorithms, or they don't explain the actual implementation.	Student has kept a (digital or analog) notebook with drawings (sketches/flowcharts) or pseudocode, showing thought and (upfront) design for at least one of the assignments.	Student has kept a (digital or analog) notebook with drawings and pseudocode, showing thought and (upfront) design for multiple assignments.	Student has clearly invested a lot of time in up front design, documenting algorithms in drawings/pseudocode with a very clear design demonstrating insight for most assignments.