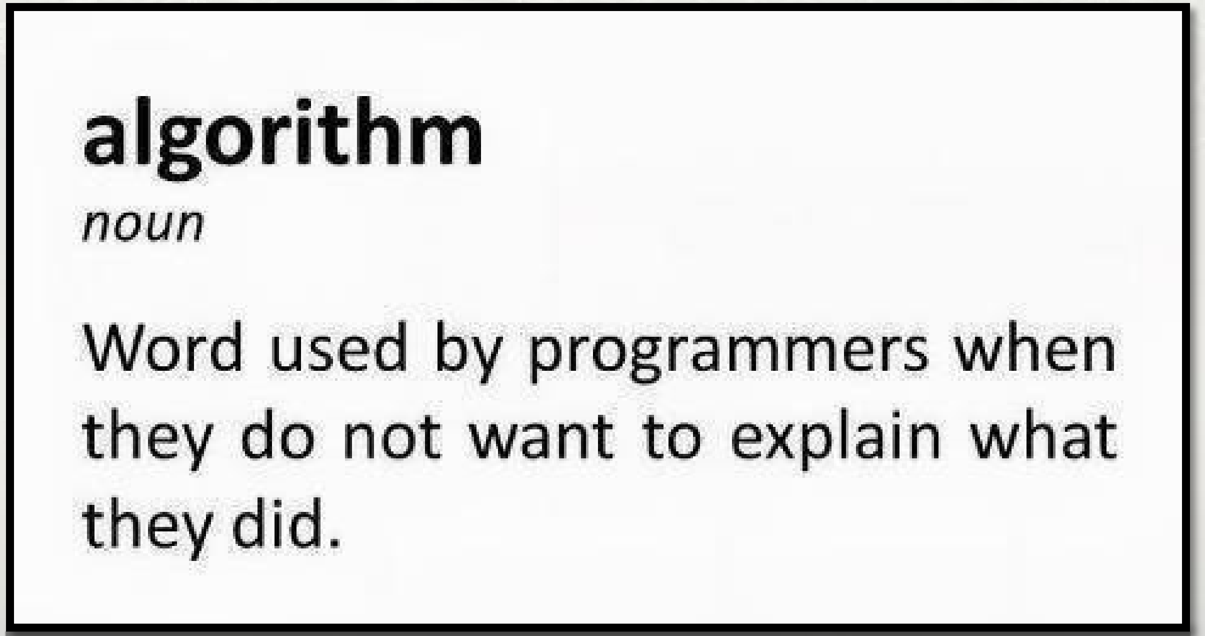
Algorithms 1 – An Introduction

by Hans Wichman



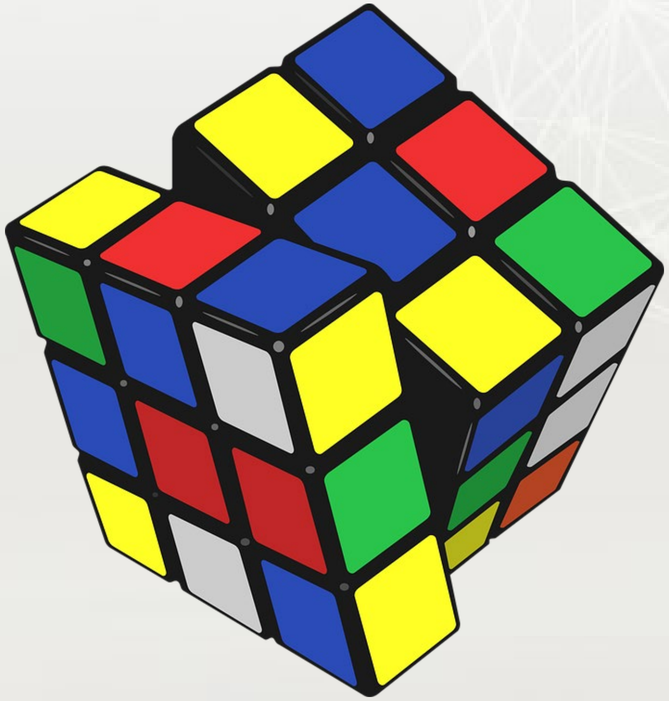
Today's topics

- What is an algorithm?
- What is a 'good' algorithm?
- Why should we learn about algorithms?
- What will we learn ? (AKA Learning objectives)
- Course approach and grading
- Getting started with algorithms & the assignments

What is an algorithm?

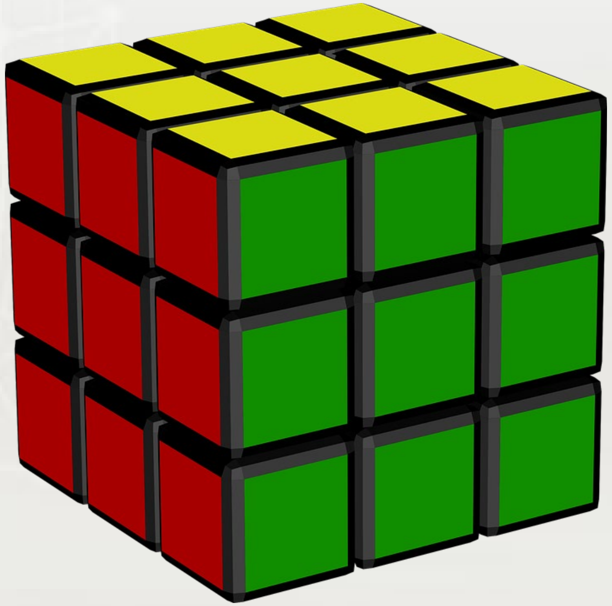
What is an algorithm?

PROBLEM



Algorithm:
the list of steps leading from the problem to the solution

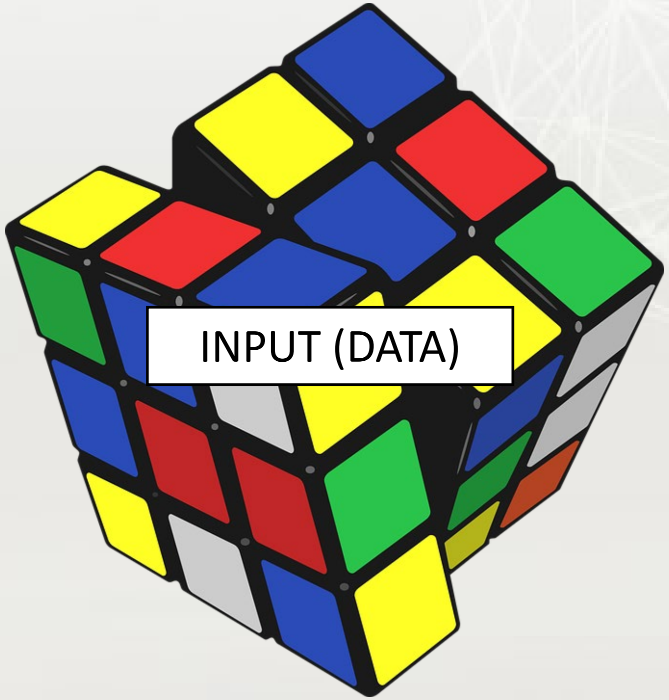
SOLUTION



For example:
If .... then rotate front, 2 times counterclockwise
If .... then rotate down, once clockwise

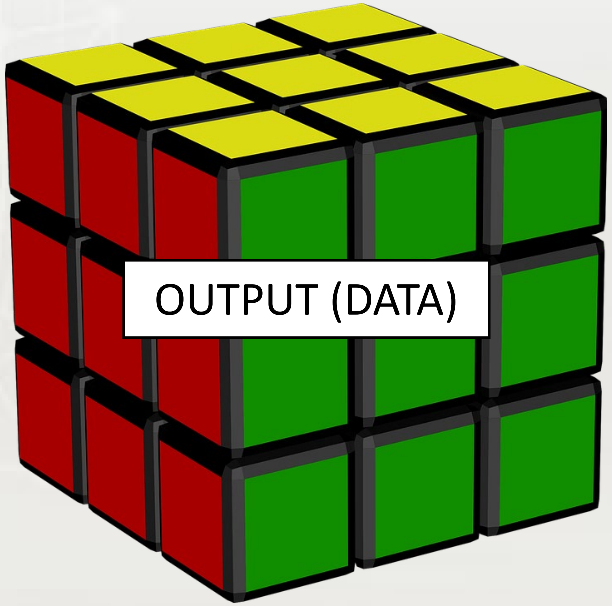
What is an algorithm?

PROBLEM

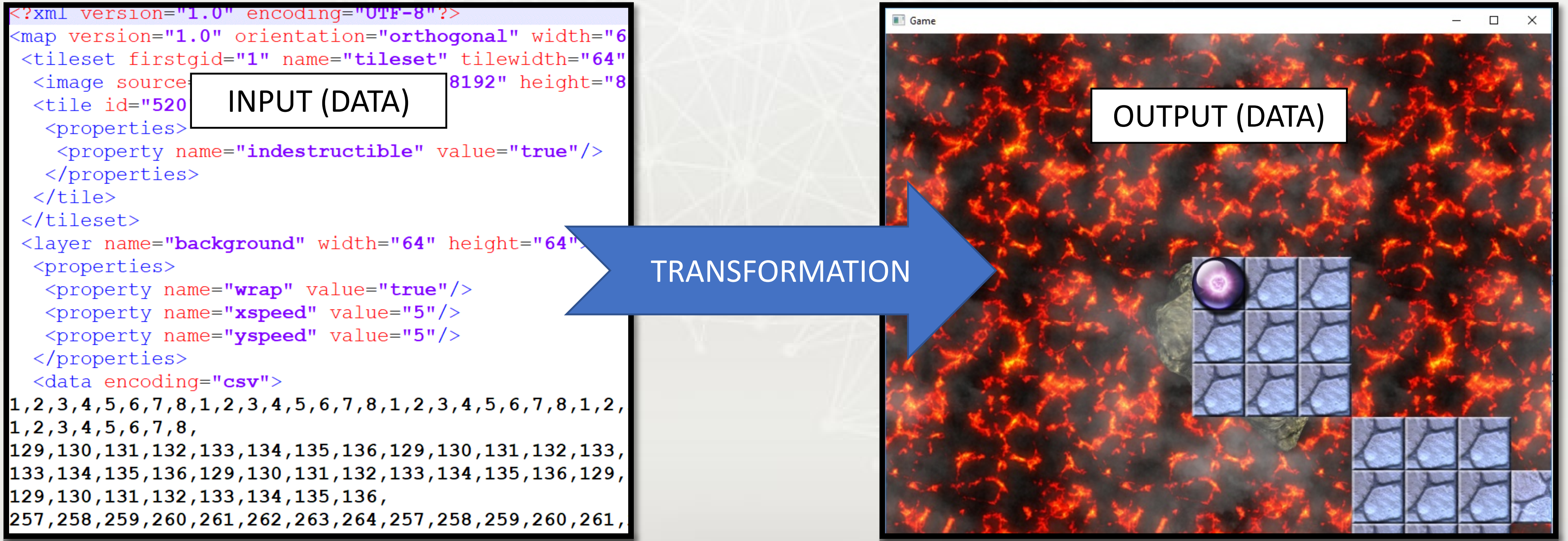


Alternatively we can view an algorithm as a TRANSFORMATION

SOLUTION

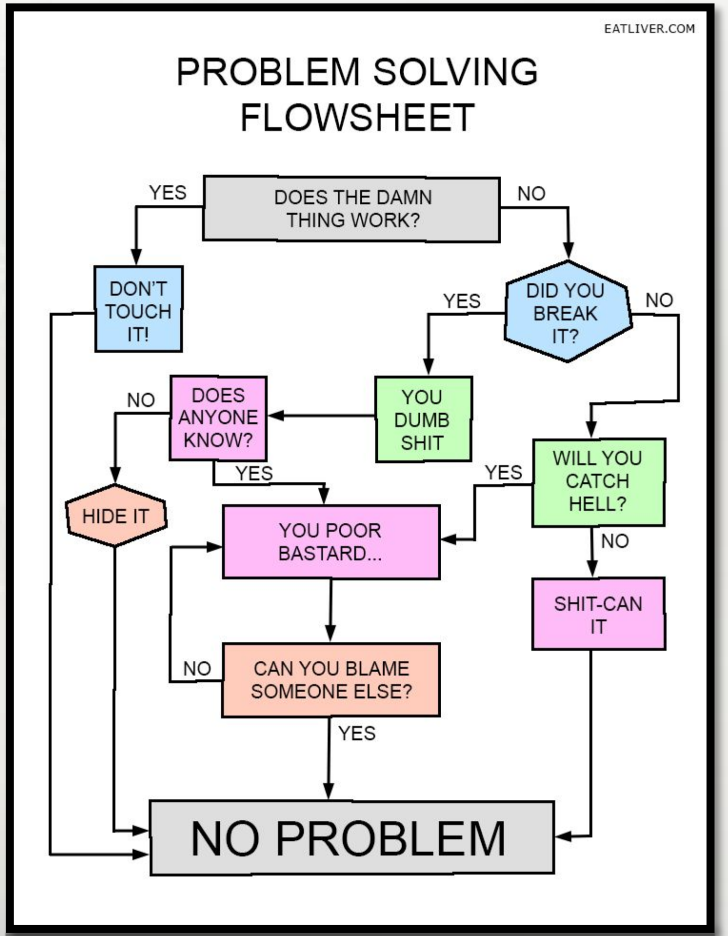


What is an algorithm?



Algorithms <u>exist independently</u> of computers

- Problems & thus algorithms have been around since long before computers ever existed.
- Any stepwise description of a solution to a problem can be called an algorithm, whether you use a computer to solve it or not.
- In fact it is often very helpful to start thinking about algorithms AWAY from the computer.



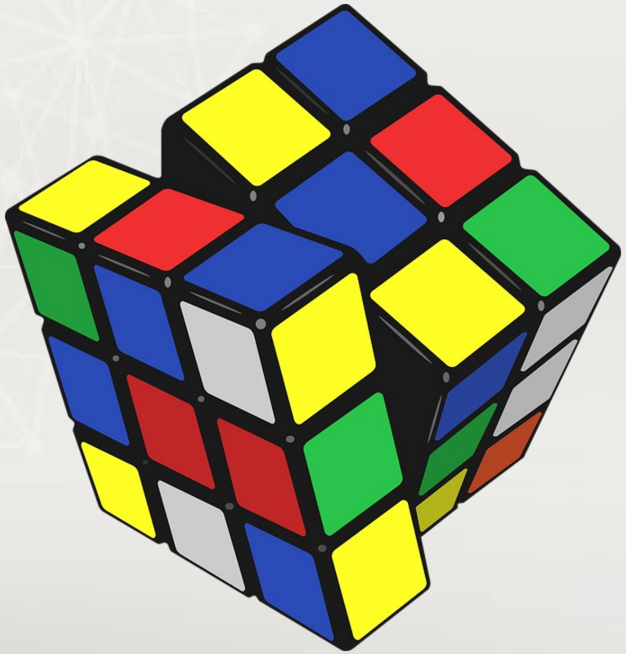
Computers + Algorithms =

- both computer programs and algorithms consist of a bunch of sequential instructions
- computers are very fast at executing (repetitive) instructions → complex algorithms often have a lot of them
- computers are very picky about precise instructions → algorithms need to be very precise

What is a good algorithm?

A ‘good’ algorithm ...

- Solves our specific problem
- Is efficient



Efficiency: Performance & Complexity

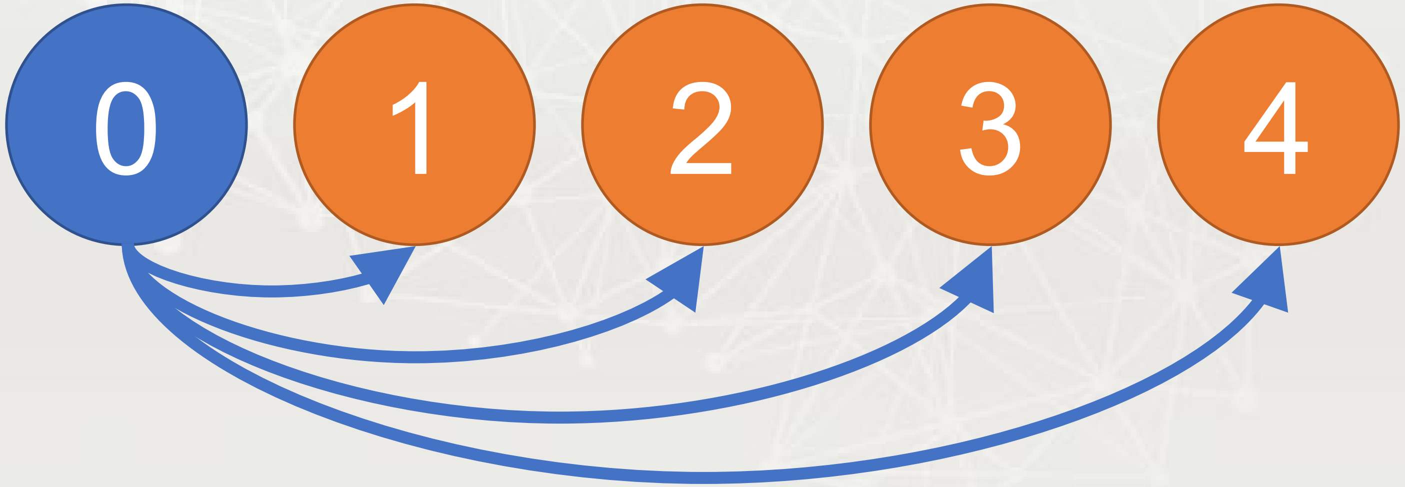
- Performance:
  - Runs within a specific predictable time frame
  - Which time frame is acceptable depends on your context
    - Finding the meaning of life → 7.5 million years is ok!
    - Pathfinding at 60 fps → 7.5 million years is not so ok!
- Complexity, meaning how well does the algorithm scale?
  - What happens to the execution time when we scale the amount of input data?
  - What happens to the memory requirements when we scale the amount of input data?

Time scaling example: collision detection

Given 5 balls:

How many checks do we have to do to see if any collide?

1^{st} iteration → 0 vs 1,2,3,4

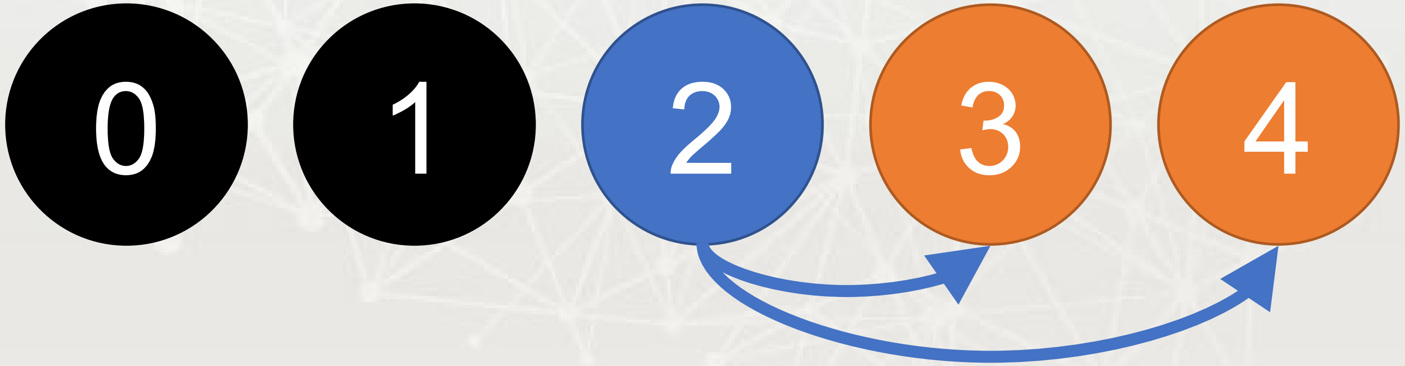


Running total = 4

2nd iteration → 1 vs 2,3,4

Running total = 4 + 3

3rd iteration → 2 vs 3,4



Running total = 4 + 3 + 2

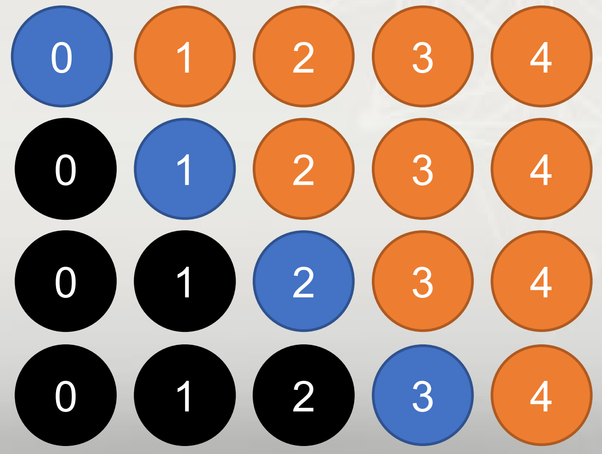
Last iteration → 3 vs 4

Running total = 4 + 3 + 2 + 1 = 10

Basically, in code (Our 1st algorithm!)

```
for (int i = 0; i < objects.Count-1; ++i) {
    for (int j = i+1; j < object.Count; ++j) {
        if (objects[i].hitTest (objects[j])) {
            /* do something */
        }
    }
}
```

This is the interesting part. The core loop that will be executed on every iteration!



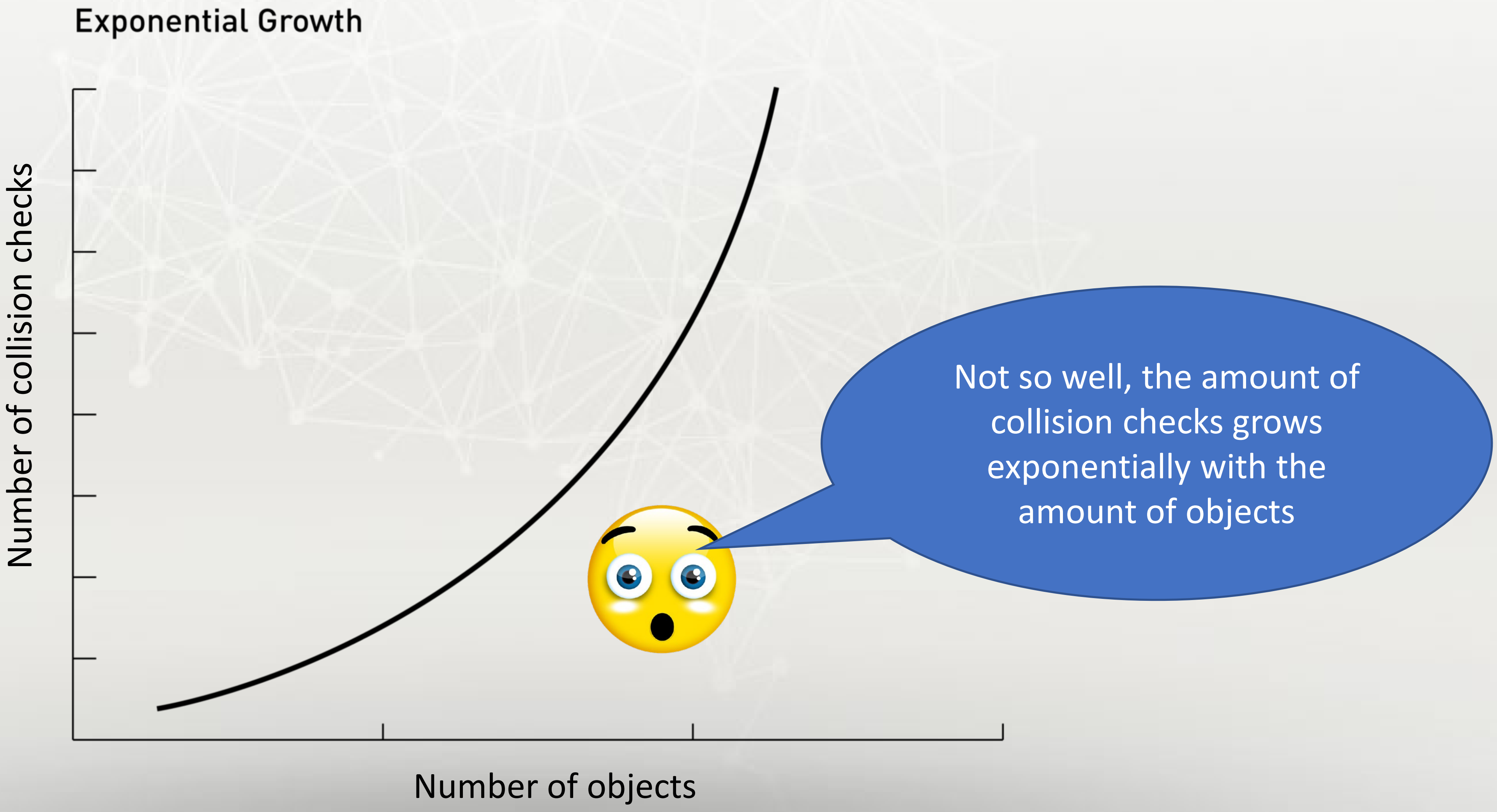
Number of collision check for n objects

Number of objects	Collision checks
5 = 4 + 3 + 2 + 1	10

Number of collision checks for n objects

Number of objects	Collision checks
5 = 4 + 3 + 2 + 1	10
10 = 9 + 8 + 7 + 6 + … + 1	45
100 = 99 + 98 + 97 + … + 1	4.950
1.000 = 999 + 998 + 997 + … + 1	499.500
10.000 = 9999 + 9998 + 9997 + … + 1	49.995.000
….	
1.000.000 = 999.999 + 999.998 + … + 1	499.998.500.001

How well does this algorithm scale?



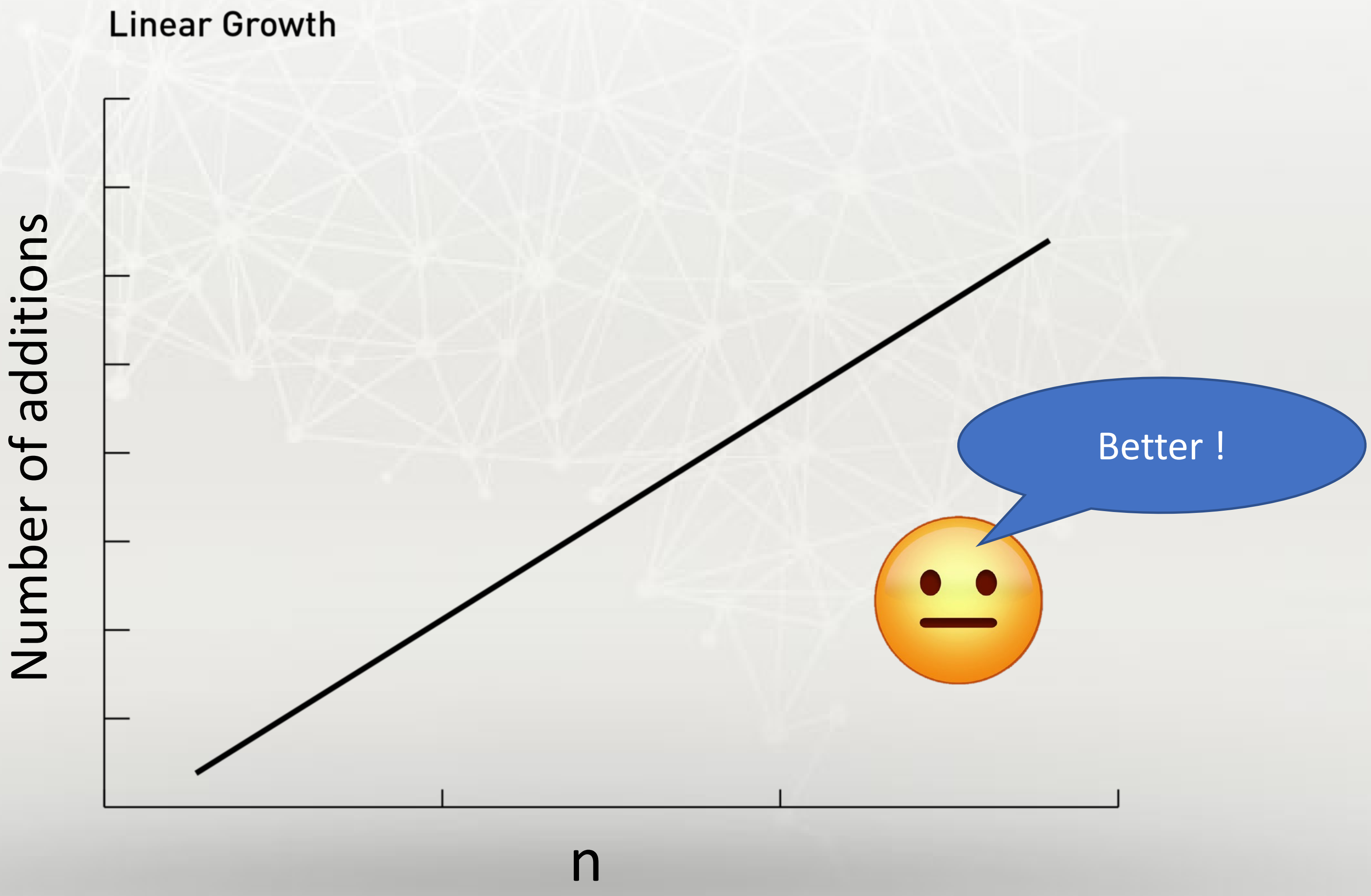
How about the algorithm we used to "Calculate the amount of collisions for n objects" ?

- 5 objects: 4 + 3 + 2 + 1 → 4 additions
- 10 objects: 9 + 8 + 7 + 6 + .. + 1 → 9 additions
- 1000 objects: 999 + 998 + .. + 1 → 999 additions
- etc

```
int amountOfCollisions = 0;
int amountOfObjects = …;
for (int i = 1; i < amountOfObjects; ++i) {
    amountOfCollisions += i;
}
```

This is the interesting part. The core loop that will be executed on every iteration!

How well does this algorithm scale?



But isn’t there a smarter way to calculate something like 4 + 3 + 2 + 1 (given n=5) ??

Given 4 + 3 + 2 + 1

Add 1 + 2 + 3 + 4

Total 5 + 5 + 5 + 5

Equals 5 * 4 → which is n * (n-1)
but it's twice the amount we need, so divide by 2.

We found a smarter way! → n * (n-1) / 2

“Calculate the amount of collisions for n objects” Old vs New comparison for n = 5

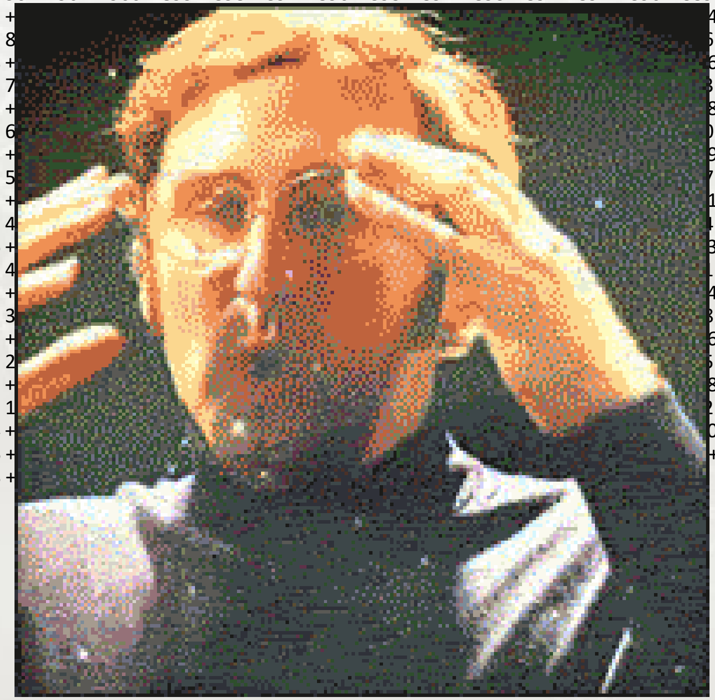
4 + 3 + 2 + 1 = 10

VS

5 * 4 / 2 = 10

“Calculate the amount of collisions for n objects” Old vs New comparison for n = 1000

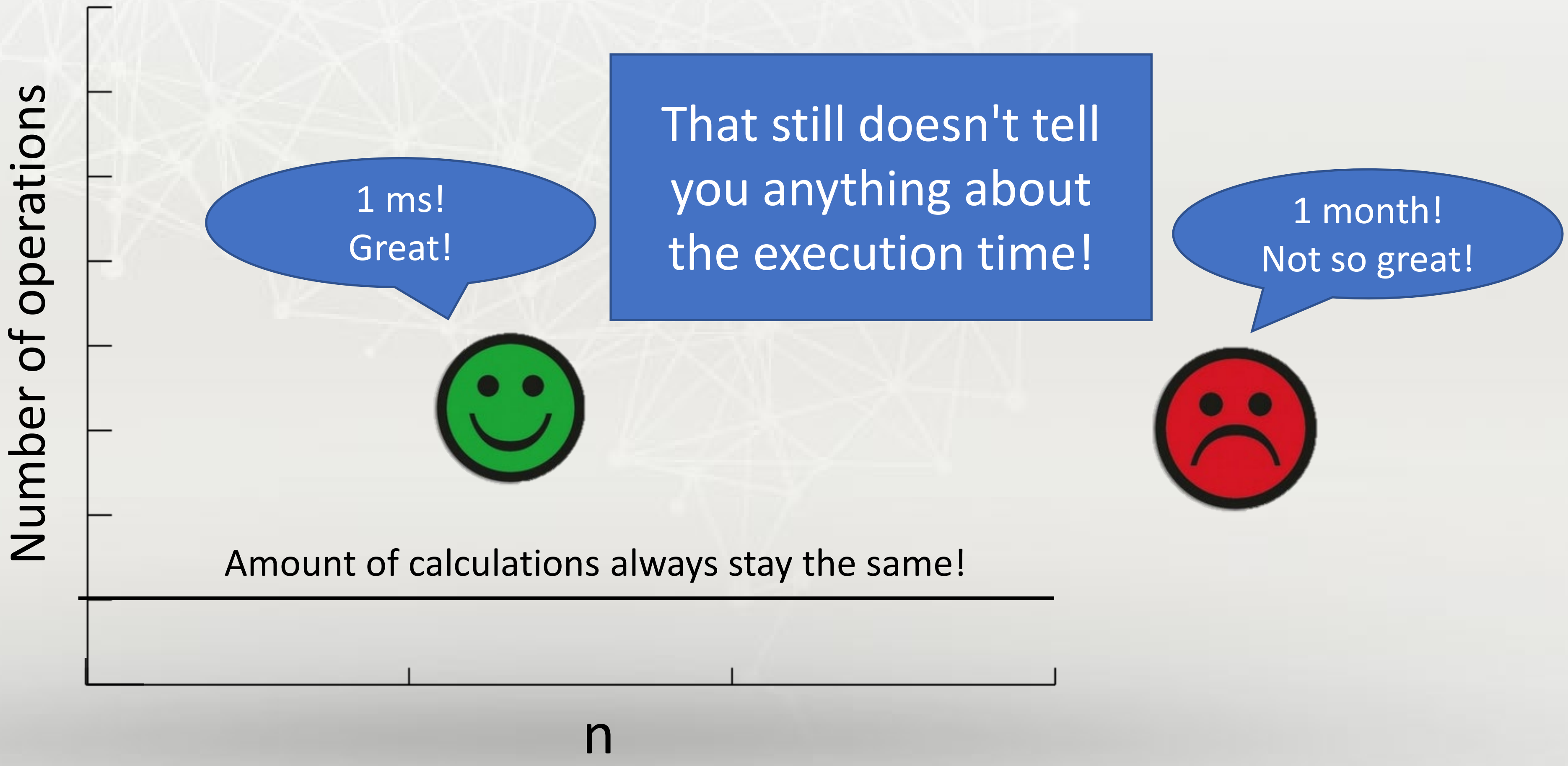
999 + 998 + 997 + 996 + 995 + 994 + 993 + 992 + 991 + 990 + 989 + 988 + 987 + 986 + 985 + 984 + 983 + 982 + 981 + 980 + 979 + 978 + 977 + 976 + 975 + 974 + 973 + 972 + 971 + 970 + 969 + 968 + 967 + 966 + 965 + 964 + 963 + 962 + 961 + 960 + 959 + 958
+ 957 + 956 + 955 + 954 + 953 + 952 + 951 + 950 + 949 + 948 + 947 + 946 + 945 + 944 + 943 + 942 + 941 + 940 + 939 + 938 + 937 + 936 + 935 + 934 + 933 + 932 + 931 + 930 + 929 + 928 + 927 + 926 + 925 + 924 + 923 + 922 + 921 + 920 + 919 + 918 + 917 +
916 + 915 + 914 + 913 + 912 + 911 + 910 + 909 + 908 + 907 + 906 + 905 + 904 + 903 + 902 + 901 + 900 + 899 + 898 + 897 + 896 + 895 + 894 + 893 + 892 + 891 + 890 + 889 + 888 + 887 + 886 + 885 + 884 + 883 + 882 + 881 + 880 + 879 + 878 + 877 + 876 + 875
+ 874 + 873 + 872 + 871 + 870 + 869 + 868 + 867 + 866 + 865 + 864 + 863 + 862 + 861 + … 47 + 846 + 845 + 844 + 843 + 842 + 841 + 840 + 839 + 838 + 837 + 836 + 835 + 834 +
833 + 832 + 831 + 830 + 829 + 828 + 827 + 826 + 825 + 824 + 823 + 822 + 821 + 820 + 8 … 5 + 805 + 804 + 803 + 802 + 801 + 800 + 799 + 798 + 797 + 796 + 795 + 794 + 793 + 792
+ 791 + 790 + 789 + 788 + 787 + 786 + 785 + 784 + 783 + 782 + 781 + 780 + 779 + 778 + … 64 + 763 + 762 + 761 + 760 + 759 + 758 + 757 + 756 + 755 + 754 + 753 + 752 + 751 +
750 + 749 + 748 + 747 + 746 + 745 + 744 + 743 + 742 + 741 + 740 + 739 + 738 + 737 + 7 … 3 + 722 + 721 + 720 + 719 + 718 + 717 + 716 + 715 + 714 + 713 + 712 + 711 + 710 + 709
+ 708 + 707 + 706 + 705 + 704 + 703 + 702 + 701 + 700 + 699 + 698 + 697 + 696 + 695 + … 81 + 680 + 679 + 678 + 677 + 676 + 675 + 674 + 673 + 672 + 671 + 670 + 669 + 668 +
667 + 666 + 665 + 664 + 663 + 662 + 661 + 660 + 659 + 658 + 657 + 656 + 655 + 654 + 6 … 0 + 639 + 638 + 637 + 636 + 635 + 634 + 633 + 632 + 631 + 630 + 629 + 628 + 627 + 626
+ 625 + 624 + 623 + 622 + 621 + 620 + 619 + 618 + 617 + 616 + 615 + 614 + 613 + 612 + … 98 + 597 + 596 + 595 + 594 + 593 + 592 + 591 + 590 + 589 + 588 + 587 + 586 + 585 +
584 + 583 + 582 + 581 + 580 + 579 + 578 + 577 + 576 + 575 + 574 + 573 + 572 + 571 + 5 … 7 + 556 + 555 + 554 + 553 + 552 + 551 + 550 + 549 + 548 + 547 + 546 + 545 + 544 + 543
+ 542 + 541 + 540 + 539 + 538 + 537 + 536 + 535 + 534 + 533 + 532 + 531 + 530 + 529 + … 15 + 514 + 513 + 512 + 511 + 510 + 509 + 508 + 507 + 506 + 505 + 504 + 503 + 502 +
501 + 500 + 499 + 498 + 497 + 496 + 495 + 494 + 493 + 492 + 491 + 490 + 489 + 488 + 4 … + 473 + 472 + 471 + 470 + 469 + 468 + 467 + 466 + 465 + 464 + 463 + 462 + 461 + 460
+ 459 + 458 + 457 + 456 + 455 + 454 + 453 + 452 + 451 + 450 + 449 + 448 + 447 + 446 + … 32 + 431 + 430 + 429 + 428 + 427 + 426 + 425 + 424 + 423 + 422 + 421 + 420 + 419 +
418 + 417 + 416 + 415 + 414 + 413 + 412 + 411 + 410 + 409 + 408 + 407 + 406 + 405 + 4 … + 390 + 389 + 388 + 387 + 386 + 385 + 384 + 383 + 382 + 381 + 380 + 379 + 378 + 377
+ 376 + 375 + 374 + 373 + 372 + 371 + 370 + 369 + 368 + 367 + 366 + 365 + 364 + 363 + … 49 + 348 + 347 + 346 + 345 + 344 + 343 + 342 + 341 + 340 + 339 + 338 + 337 + 336 +
335 + 334 + 333 + 332 + 331 + 330 + 329 + 328 + 327 + 326 + 325 + 324 + 323 + 322 + 3 … 3 + 307 + 306 + 305 + 304 + 303 + 302 + 301 + 300 + 299 + 298 + 297 + 296 + 295 + 294
+ 293 + 292 + 291 + 290 + 289 + 288 + 287 + 286 + 285 + 284 + 283 + 282 + 281 + 280 + … 66 + 265 + 264 + 263 + 262 + 261 + 260 + 259 + 258 + 257 + 256 + 255 + 254 + 253 +
252 + 251 + 250 + 249 + 248 + 247 + 246 + 245 + 244 + 243 + 242 + 241 + 240 + 239 + 2 … 5 + 224 + 223 + 222 + 221 + 220 + 219 + 218 + 217 + 216 + 215 + 214 + 213 + 212 + 211
+ 210 + 209 + 208 + 207 + 206 + 205 + 204 + 203 + 202 + 201 + 200 + 199 + 198 + 197 + … 83 + 182 + 181 + 180 + 179 + 178 + 177 + 176 + 175 + 174 + 173 + 172 + 171 + 170 +
169 + 168 + 167 + 166 + 165 + 164 + 163 + 162 + 161 + 160 + 159 + 158 + 157 + 156 + 1 … 2 + 141 + 140 + 139 + 138 + 137 + 136 + 135 + 134 + 133 + 132 + 131 + 130 + 129 + 128
+ 127 + 126 + 125 + 124 + 123 + 122 + 121 + 120 + 119 + 118 + 117 + 116 + 115 + 114 + … 00 + 99 + 98 + 97 + 96 + 95 + 94 + 93 + 92 + 91 + 90 + 89 + 88 + 87 + 86 + 85 + 84 + 83 +
82 + 81 + 80 + 79 + 78 + 77 + 76 + 75 + 74 + 73 + 72 + 71 + 70 + 69 + 68 + 67 + 66 + 65 + … + 47 + 46 + 45 + 44 + 43 + 42 + 41 + 40 + 39 + 38 + 37 + 36 + 35 + 34 + 33 + 32 + 31 +
30 + 29 + 28 + 27 + 26 + 25 + 24 + 23 + 22 + 21 + 20 + 19 + 18 + 17 + 16 + 15 + 14 + 13 +



1000 * 999 / 2

So how well does this new algorithm scale?

No matter ‘n’, the calculation always **takes the same amount** of steps!



Algorithmic Complexity → Big O notation

- 'Algorithmic complexity' means 'how well does the algorithm scale'
- We saw three examples: constant, linear and exponential complexity
- Expressed using the Big O notation (O for Order of Magnitude).
- If we scale input size n, what happens to #algorithmicsteps?
  - constant amount → O = 1, written as O(1) (constant relation)
  - constant amount * n → O = n, written as O(n) (linear relation)
  - constant amount * n^2 → O = n^2, written as O(n^2) (exponential relation)
  - (... more variants later ...)

Algorithmic Complexity → Big O notation

- This usually takes some time to get used to:
  - Input size * 2 → #algorithmicsteps * 2 → Linear → O(n)
  - Input size * 2 → #algorithmicsteps * 4 → Linear → O(n)

Indicators

- A good indicator for algorithmic (time) complexity is the amount of nested loops ☺

- Amount of nested loops:
  - 0 → O(1) (constant time complexity)
  - 1 → O(n) (linear time complexity)
  - 2 → O(n^2) (exponential time complexity)

- Keep in mind: it's just an indicator not absolute truth

What do you need to write a good algorithm?

Algorithmic ingredients 1

- Algorithms are expressed in steps (flowcharts, pseudo code, text, etc)
- Steps are implemented using code structures (variables, loops, etc)
- Code structures are built on top of data structures (lists, maps/dictionaries, graphs, etc)

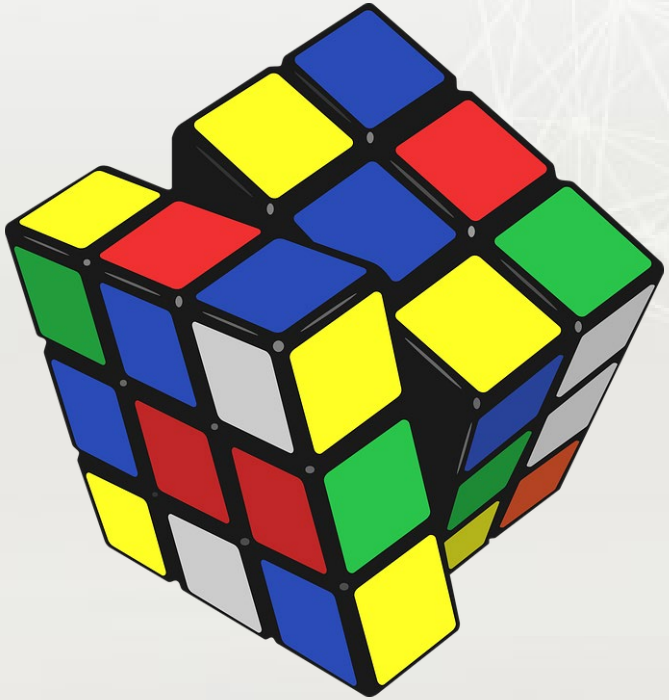
Algorithmic ingredients 2

- Problem solving skills:
  - How do you start?
  - What do you do if things go wrong?

- Time, examples, practice, luck/strokes of insight

Why a course on algorithms?

Algorithms are everywhere, but implementing/creating them is not trivial



Algorithm*:

F3 – U2 – D`1 – B1 – L2 – R'1 - D`1 – B1 – L2 –
R'1 - U2 – D`1 – B1 – L2 – R'1 - D`1 – B1 – L2 –
R'1 - U2 – D`1 – B1 – L2 U2 – D`1 – B1 – L2
U2 – D`1 – B1 – L2 – F1- R'1 - D`1 – B1 – L2 – R'1
-U2 – D`1 – B1 – L2 – R'1 - D`1 – B1 – L2 – R'1 -
U1 – B1 – L2 – R'1 - D`1 – B1 – L2 – R'1 -
U2 – D`1 – B1 – L2 – R'1 - D`1 – B1 – L2 – R'1 -.....

*https://ruwix.com/the-rubiks-cube/how-to-solve-the-rubiks-cube-beginners-method/

Learning more about algorithms allows you to...

- solve more complicated problems than before
- solve problems more efficiently than before
- write more interesting programs than before
- get internships/jobs
- stand on the shoulders of giants
- not get stuck

- basically: become a better programmer/problem solver

Learning Objectives

What should you already know

- Variables (int, float, bool)
- Code structures (if-then-else, for, while)
- Basic list/array manipulation (more on this in lecture 2)
  - new, Add, Remove, IndexOf, Contains, etc
- Basic object orientation:
  - objects & classes
  - inheritance and polymorphism ?
  - abstract classes and interfaces ?

- If any of these are not clear, some extra study/explanation during labs might be required (check additional resources on blackboard).

Official learning objectives for this course...

The student:

1. describes algorithms using
   e.g. flowcharts & pseudocode
2. uses/creates the right data structure
   to implement a given algorithm
3. implements list, dictionary and graph based
   algorithms iteratively and recursively
4. tests and debugs an algorithm
5. explains algorithmic complexity

Official CMGT Competencies

- CMGT 1. Technological research and analysis
- CMGT 2. Designing, prototyping and realizing
- CMGT 3. Testing and rolling out
- CMGT 5. Conceptualising
- CMGT 6. Designing

If, at the end of this course, you feel that you…

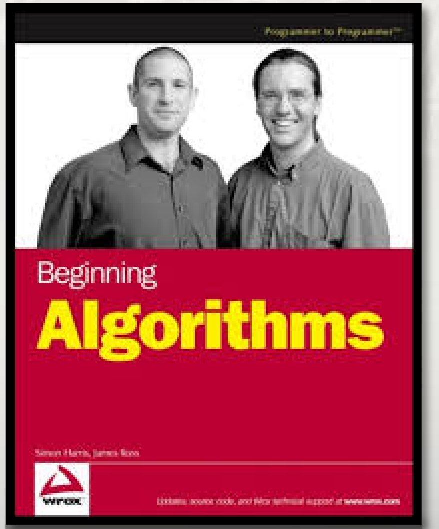
- … are more confident in tackling problems …
- … have more practical skills/tools to solve problems …
- … are able to do things that you were not able to do before …

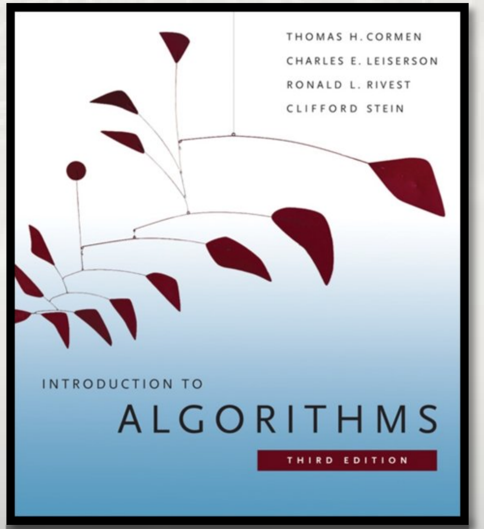
… then we can all be pretty happy and go and enjoy a well deserved holiday.

Course approach & grading

What we will *not* do in this course... 1/2

- We will ***not*** discuss all algorithms or all data structures
- We will ***not*** take the standard ICT approach of writing your own lists, sorting algorithms etc
- However there are some very good resources for that in case you are interested.
- Recommended reading material (during the course or during the holiday):





What we will *not* do in this course... 2/2

- Architecture
- Design patterns
- Unit testing
- Object oriented analysis & design

- **Not** part of this course, subject of next year's Architecture course
  (This also explains why the starting code looks the way it looks ☺)

(wo)

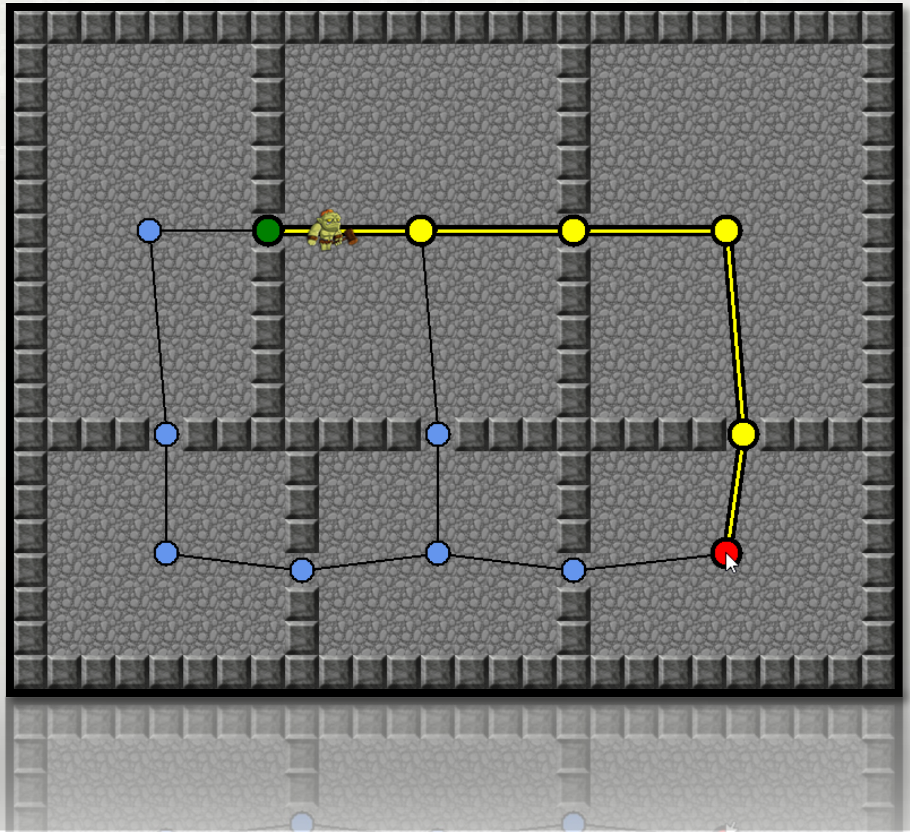
Give a man a fish
and you feed him for a *day*.
Teach a man to fish
and you feed him for a *lifetime*.

Chinese Proverb

Chinese Proverb

1 overall assignment + assessment

"Create a prototype for a roguelike dungeon crawler with procedurally generated levels & pathfinding"



The assignment is (supposed to be) a puzzle

- The goal is clear, but the steps aren't
- Doable, but challenging:
  - there are only 3 assignments (1 assignment per 2 weeks)
  - option to pick your own skill level
  - starting code has been provided
  - plenty of time per assignment → ± 20 hours per assignment (and you might need it)
- But yes, you will get frustrated & you will struggle sometimes

The *assessment* is more than the *assignment*

- It's not only about solving the problem, but also about:
  - up front design (drawings/pseudo code)
  - performance
  - code conventions
  - etc
- *Solving* the problem is priority No 1, but the other things have to be in order as well!
- Detailed assessment criteria can be found on blackboard (!)
- Check the grading form!!

Covered so far ...

- *What is an algorithm?*
- *What is a 'good' algorithm?*
- *Why should we learn about algorithms?*
- *What will we learn ? (AKA Learning objectives)*
- *Course approach and grading*

- ***Next up:* Getting started with algorithms & the assignments**

Getting Started

Divide & Conquer

Algorithmic design principle 1

Divide & Conquer

- An algorithm solves a problem
- Problems can often be split into smaller problems

- An assignment such as "Create a prototype for a roguelike dungeon crawler with procedurally generated levels & pathfinding" is too big to tackle in one go

We need to break it down so we can worry about 1 problem at a time

Divide & conquer in practice

"Create a prototype for a roguelike dungeon crawler with procedurally generated levels & pathfinding"

Big problem, broken down into 3 smaller problems/assignments:

- Assignment 1 → Generate a dungeon
- Assignment 2 → Generate a nodegraph (+ Morc da Orc)
- Assignment 3 → Implement pathfinding

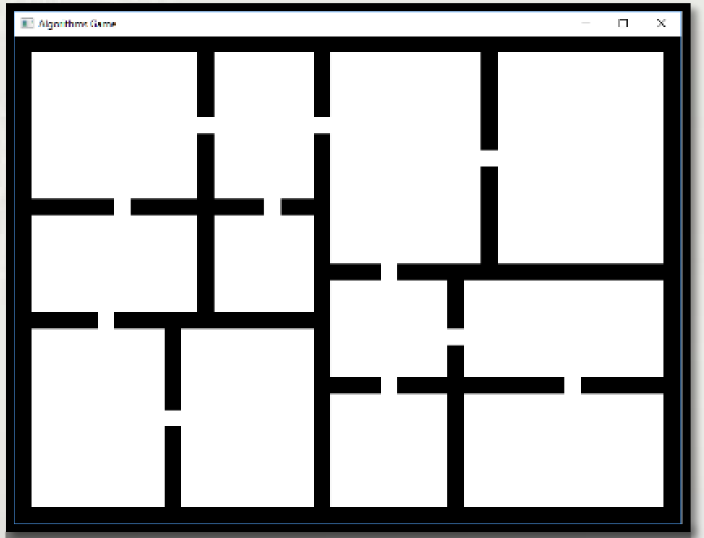
*The sprites for Morc and two of his brothers are available for free on blackboard

Assignment 1

- Generate a Dungeon based on 'binary space partitioning'
  - Binary means 2
  - Space partitioning means subdividing a space in some way

- In other words: generate a dungeon
  by subdiving a space in two until we
  can't subdivide it any more
  (and then find out where to place the doors).

- So part of the solution is already provided in this
  assignment (the approach/general idea) !



Questions to ask yourself

- Is this problem small enough or should we subdivide it any further?
- Are the *sub* problems small enough or should we subdivide *them* any further?
- When is a sub problem small enough? When you ...
  - know **what** to do
  - know **how** to do it
  - ... or at least have some idea of know where to start ...
- Divide & Conquer: Divide *until you can* conquer...

Divide & Conquer applied to ‘Generating a dungeon’

- Is the problem small enough or can we subdivide it any further?
  - Sounds like a pretty big problem (meaning: it still consists of multiple steps)
- Is it clear what we have to do?
  - Not exactly, need more info
- How can we find out what to do exactly?

Acting out the algorithm

Algorithmic design principle 2

Acting out the algorithm

If you cannot write down/visualize the algorithm step by step on paper, chances are you cannot implement it

- Step 1 – Literally step away from the computer
- Step 2 – Act out the algorithm on paper, recording the steps
- Step 3 – Prototype on the computer
- Step 4 – Start over at step 1 if necessary

- Important: keep a notebook/logbook! See grading criteria!

Example result

Generating a dungeon

- Is it more clear what we have to do now? Yes, a little bit!
- Is the problem trivial or can we subdivide it any further?
  - Well it's clearer, but we could subdivide it even further:
    1. Generate the rooms (through binary space partioning)
    2. Generate doors based on the rooms (through ... ?)

- Can these two sub problems be subdivided even further?

Generating the rooms

- Again, a problem we can split into multiple sub problems:
  1. Check whether a room can be subdivided
  2. Determine whether to do a horizontal or vertical split
  3. Subdivide a room (randomly)
  4. Repeat 1 until there are no more rooms to subdivide left

Generating the doors

- After generating the rooms, we have to generate doors!
  - Again: act out the algorithm, maybe even on squared scrap paper
  - Observe yourself and your thought processes as you decide where to place doors: why are you placing the door where you are placing it?
  - How can you break this down again into little steps?

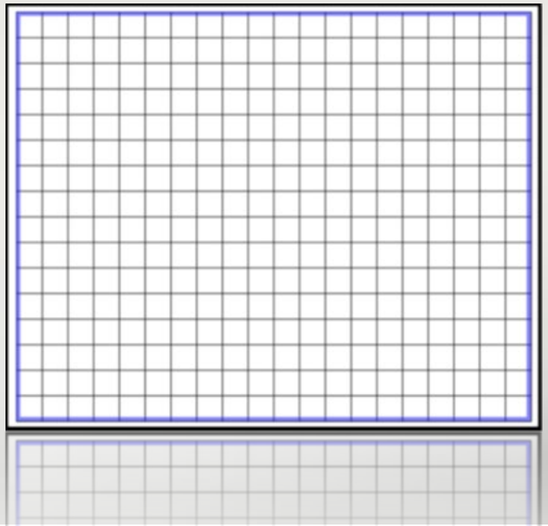
Iteration

Algorithmic design principle 3

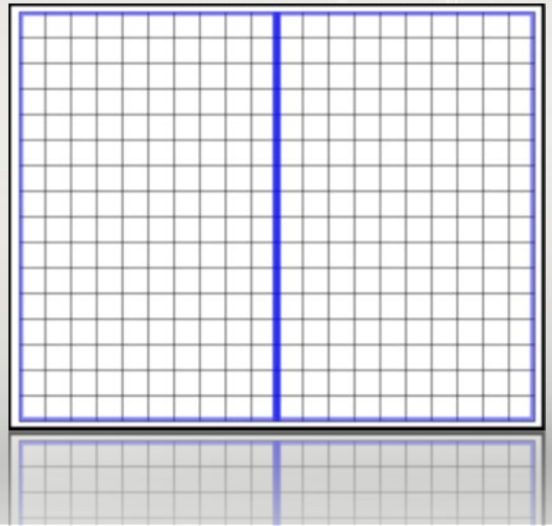
Iteration

- Algorithm execution involves a lot of iteration (eg for loops)
- But Algorithm development ALSO involves a lot of iteration
- We can view this as another example of Divide & Conquer

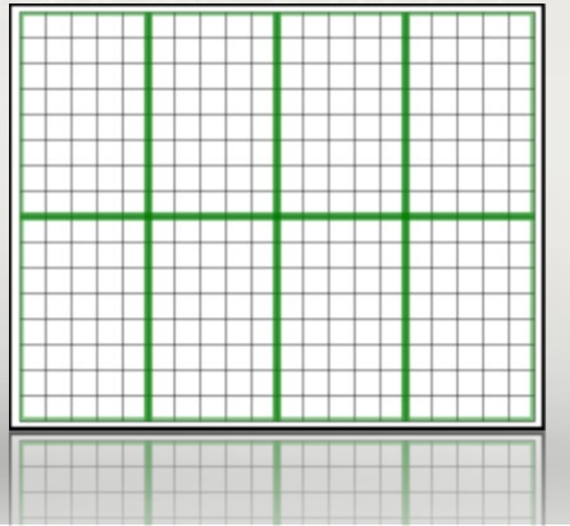
Step 1 – 1 room



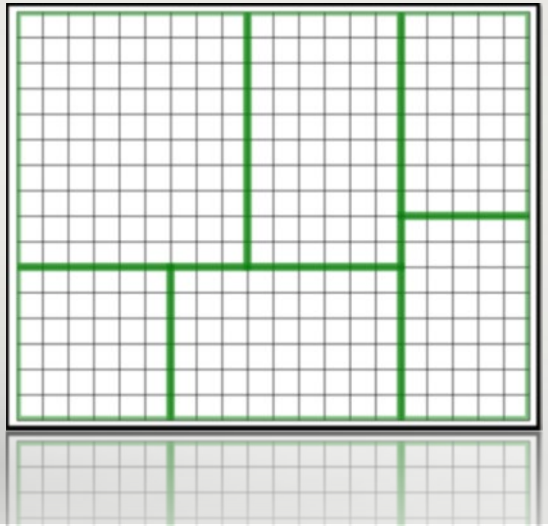
Step 2 – 2 rooms even splits



Step 3 – all rooms even splits



Step 4 – all rooms random splits



Experiment

Algorithmic design principle 4

Experimentation

- Given the input and output requirements of a problem, it is important to play around
- For example, given the dungeon size and minimum room size, can you:
  - generate 1 room with the size of the dungeon?
  - explain how the room coordinates work?
  - generate random rooms?
  - subdivide a single room into two rooms?
  - print whether a room meets the minimum size requirement?
  - print whether any rooms have an overlap?
- Constantly think: what else can I try that I haven't tried yet no matter how small
- The mindset of experimentation helps avoiding analysis paralysis

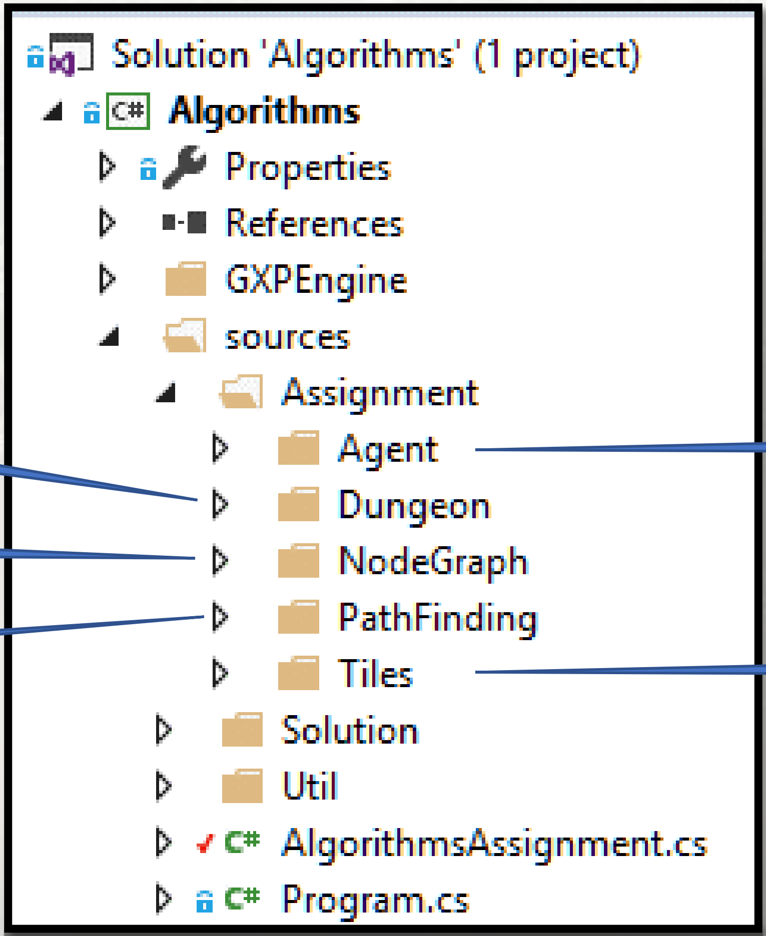
Other tips & principles

- Console spamming
- Code introspection through the debugger
- The binary elimination principle
- Trace tables
- etc etc

- Check the additional resources section on blackboard!

Short code walkthrough

Divide & conquer applied to the code setup



Starting code assignment 1

Starting code assignment 2

Starting code assignment 3

Starting code assignment 2

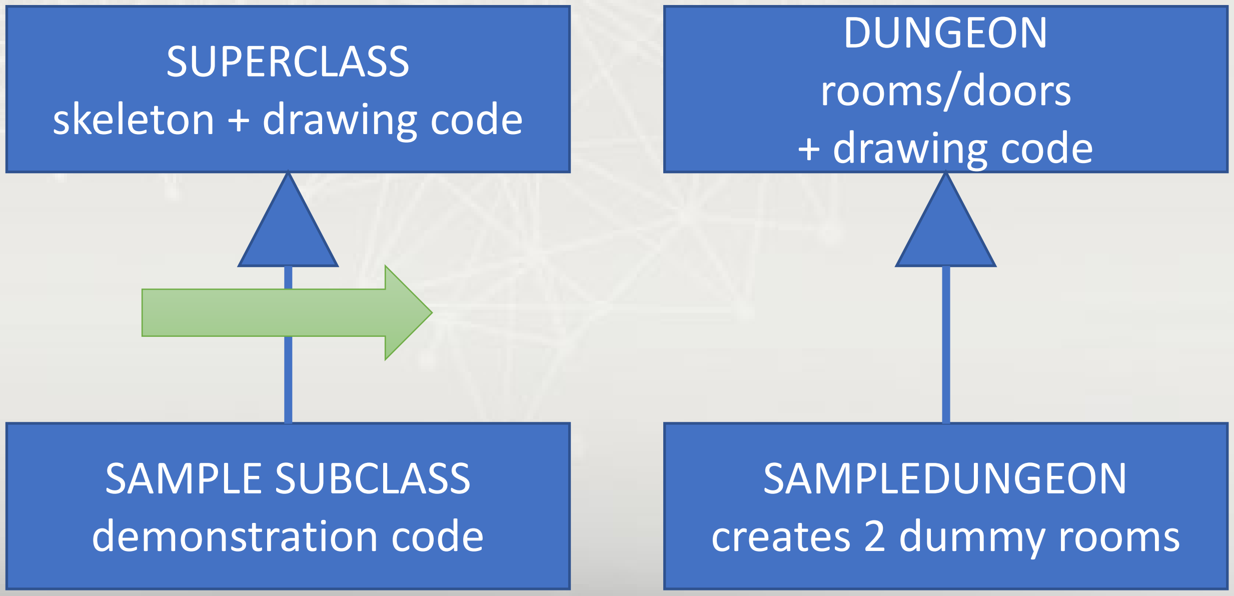
Extra code assignment 2

AlgorithmAssignment class

- Lists the code skeleton for all assignments
- (Un)comment as necessary
- Feel free to:
  - split into methods
  - remove to-do's while you are implementing
  - restructure etc.

What does the starting code provide?

- Minimal data structures (or the skeleton of it) for every assignment
- Drawing code for debugging/visualization for every assignment
- Setup for each package is the same:



The Dungeon class

- List<Room> _rooms;
- List<Door> _doors;

} the data*

- drawRooms() / drawRoom()
- drawDoors() / drawDoor()

} the view*

- No room/door generation logic whatsoever:
  - abstract void generate (...);

*if interested, some background info on the design choices with respect to the starting code architecture can be found on blackboard

The SampleDungeon class

- Subclasses Dungeon
- overrides generate
- adds 2 rooms and 1 door (just to show the general idea)

Your SufficientDungeon class

- Should:
  - subclass Dungeon
  - override generate
  - subdivide rooms, place doors etc until you have a dungeon ☺

Closing words

Summing up, we had a look at:

- What an algorithm is
- What a 'good' algorithm is
- Why it is important to learn about algorithms
- What we will learn during this course
- Course approach and grading
- Introduction to assignment 1
  (1.1 only, for 1.2 and 1.3 you are on your own)

What's next?

- This week lab/homework:
  - check the C# essential slides
  - check the grading criteria and assignments
  - download/start with assignment 1
  - try and solve a rubik cube ? ☺
- Next lecture:
  - all about ***Lists*** (& a bit about HearthStone)

- **Check the schedule on blackboard every week for details on homework and lecture preparation!!**

Your turn! Good luck!