Teaching mathematically demanding computer science topics to software engineering students:

Is there any reason?

Is there any hope?

Antti Valmari

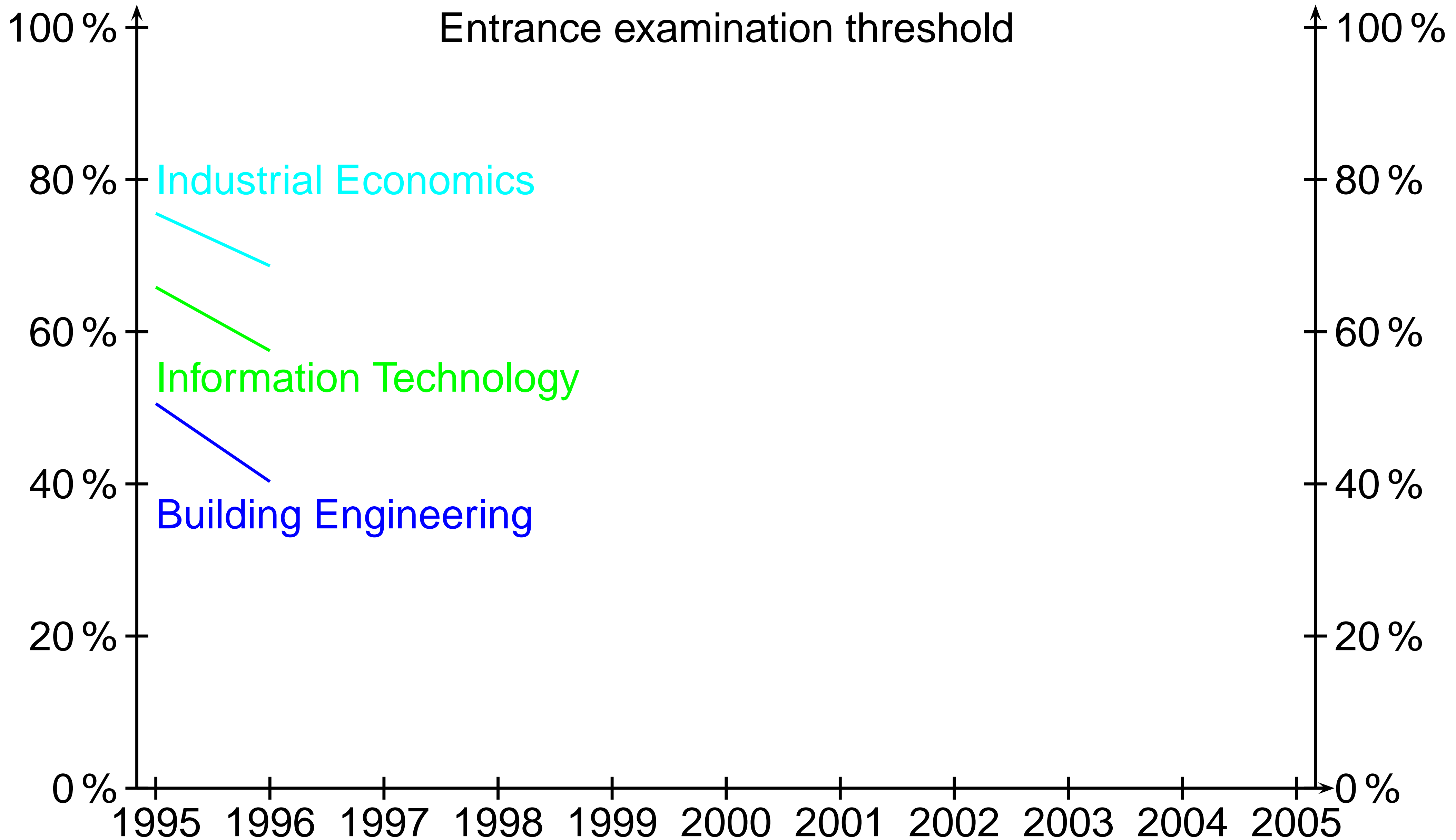
Tampere University of Technology

2006–06–27

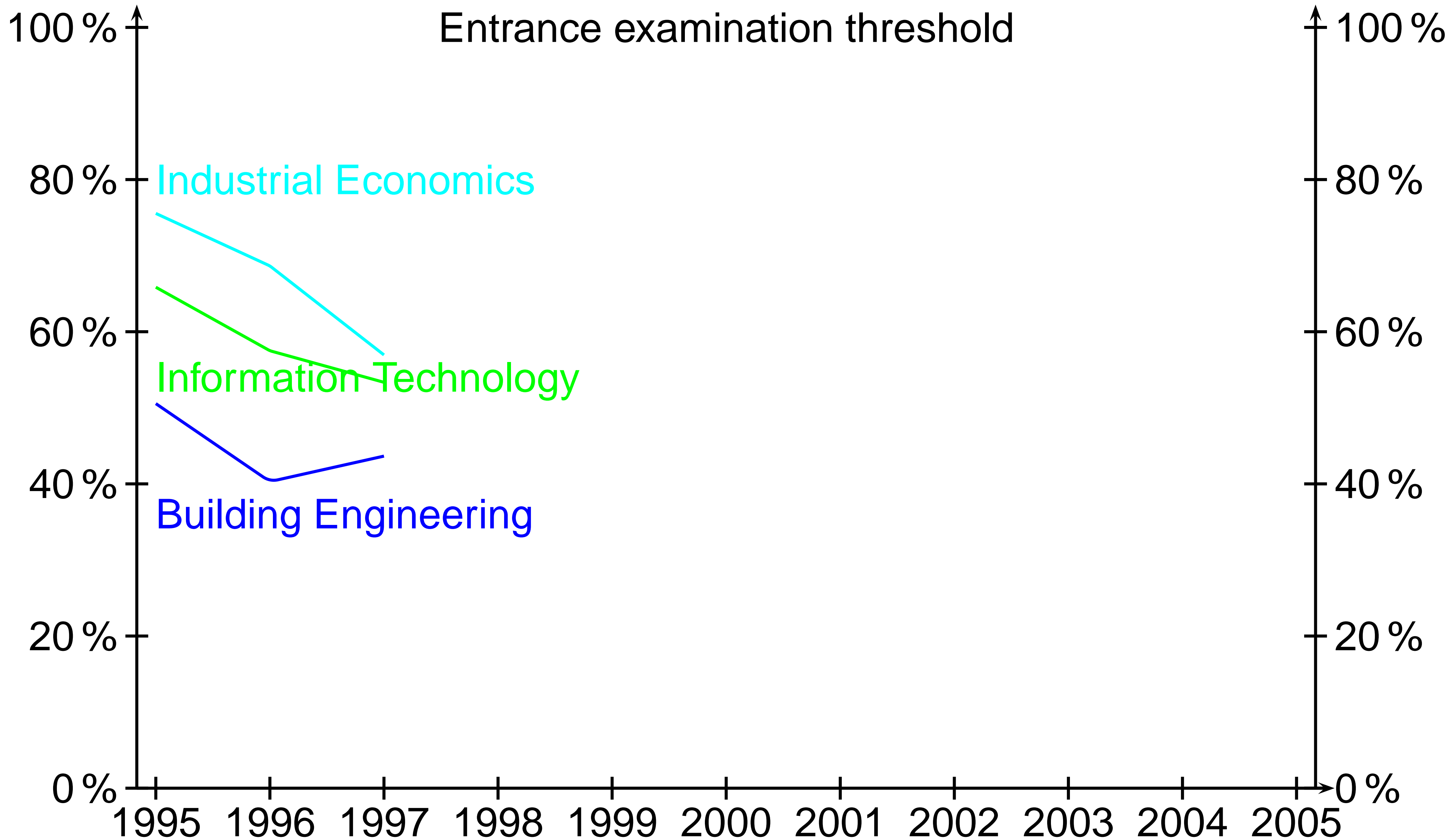
Contents

1	Why does this bother *me*?	3
2	Is mathematics necessary any more?	4
3	Is there any hope?	14
4	Concurrency	22

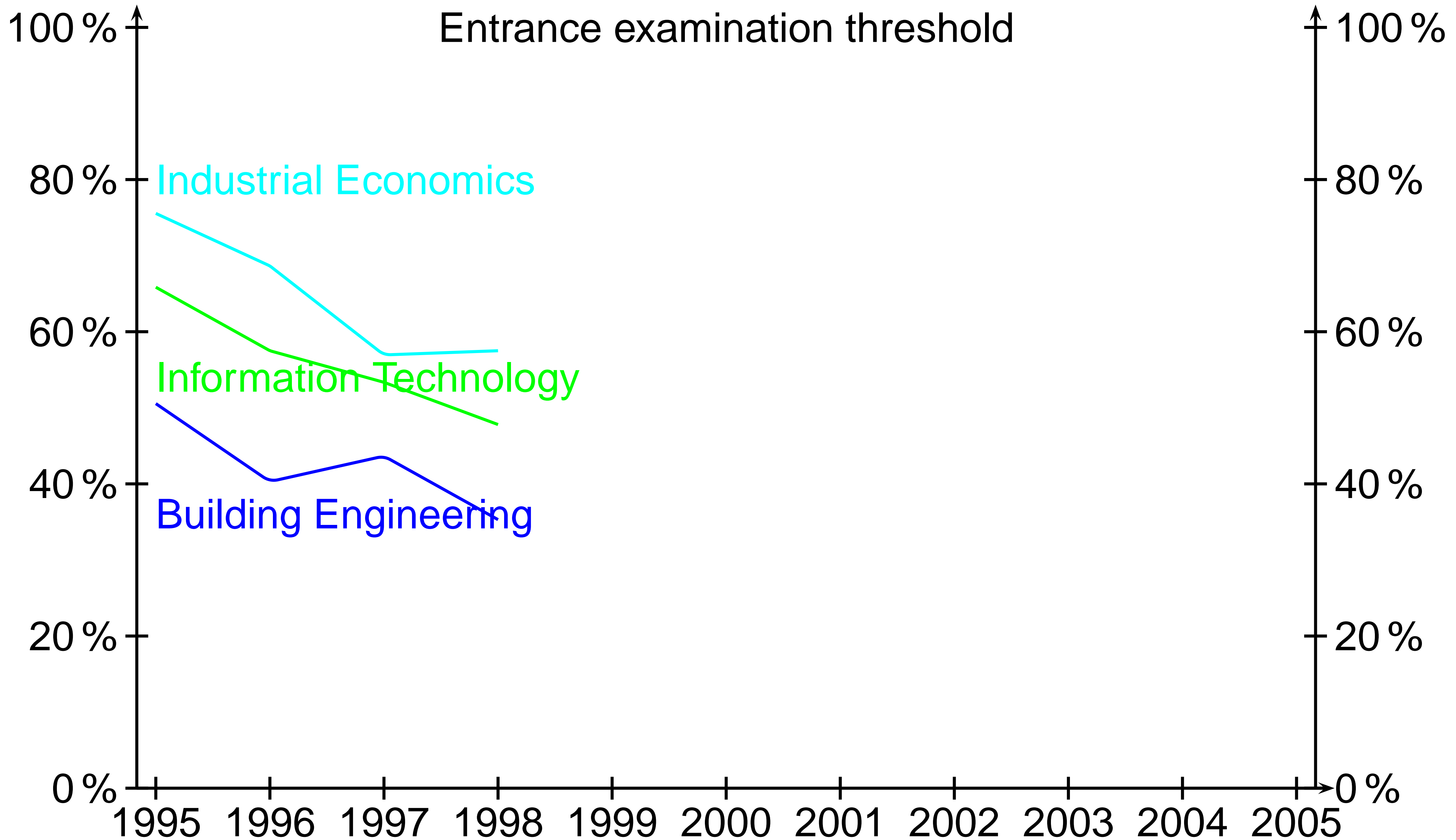
1 Why does this bother *me*?



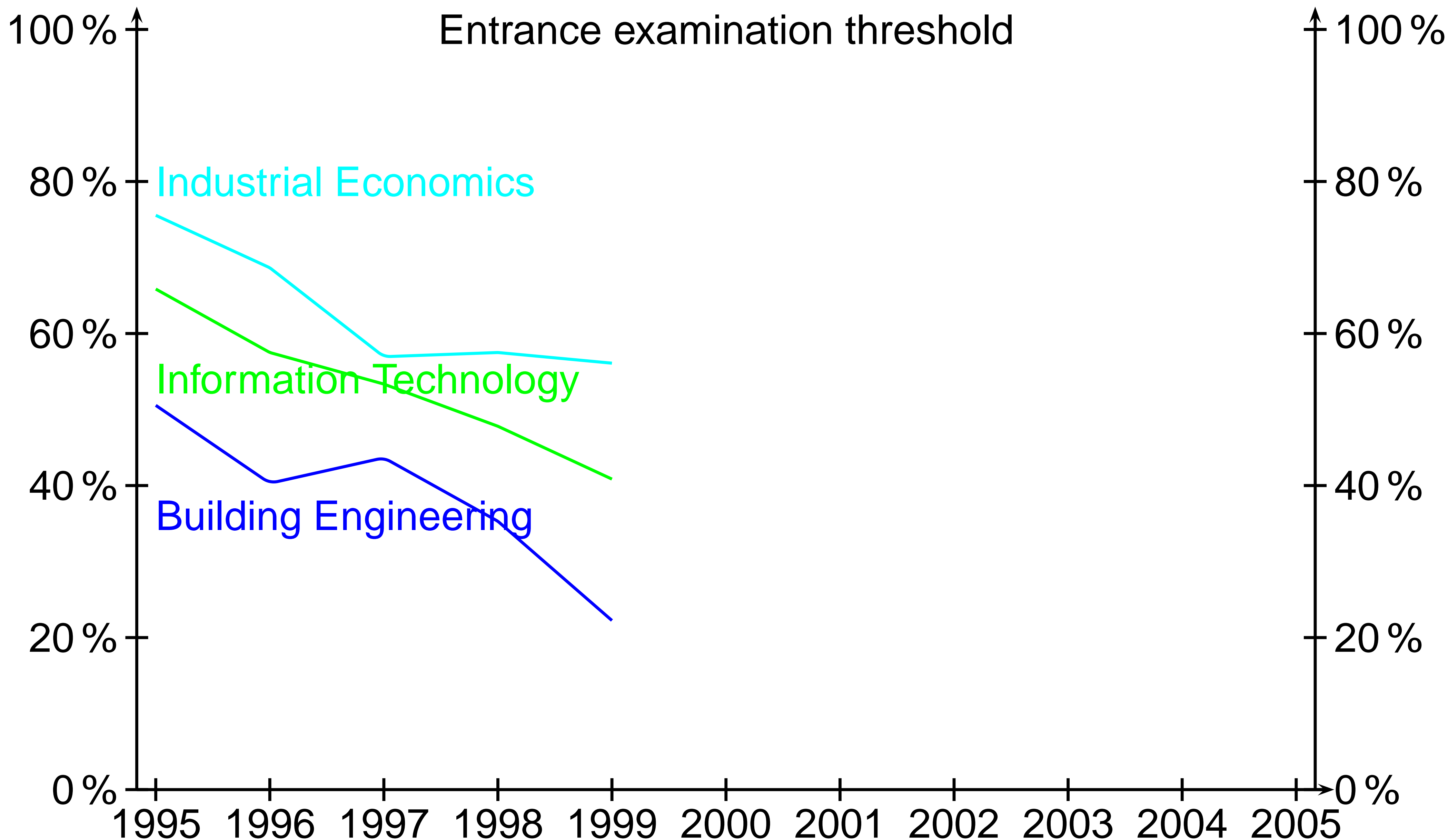
Why does this bother *me*?



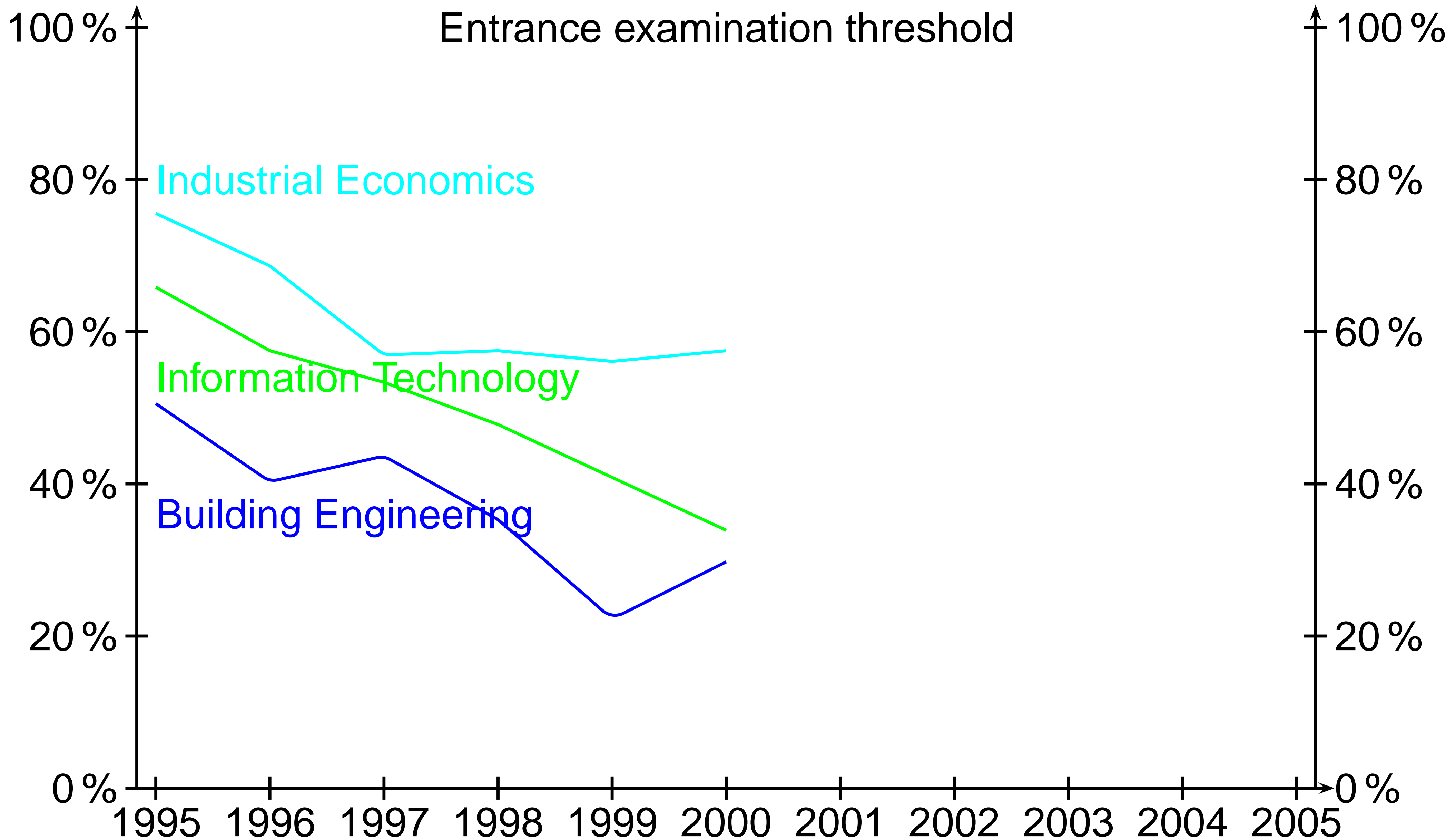
Why does this bother *me*?



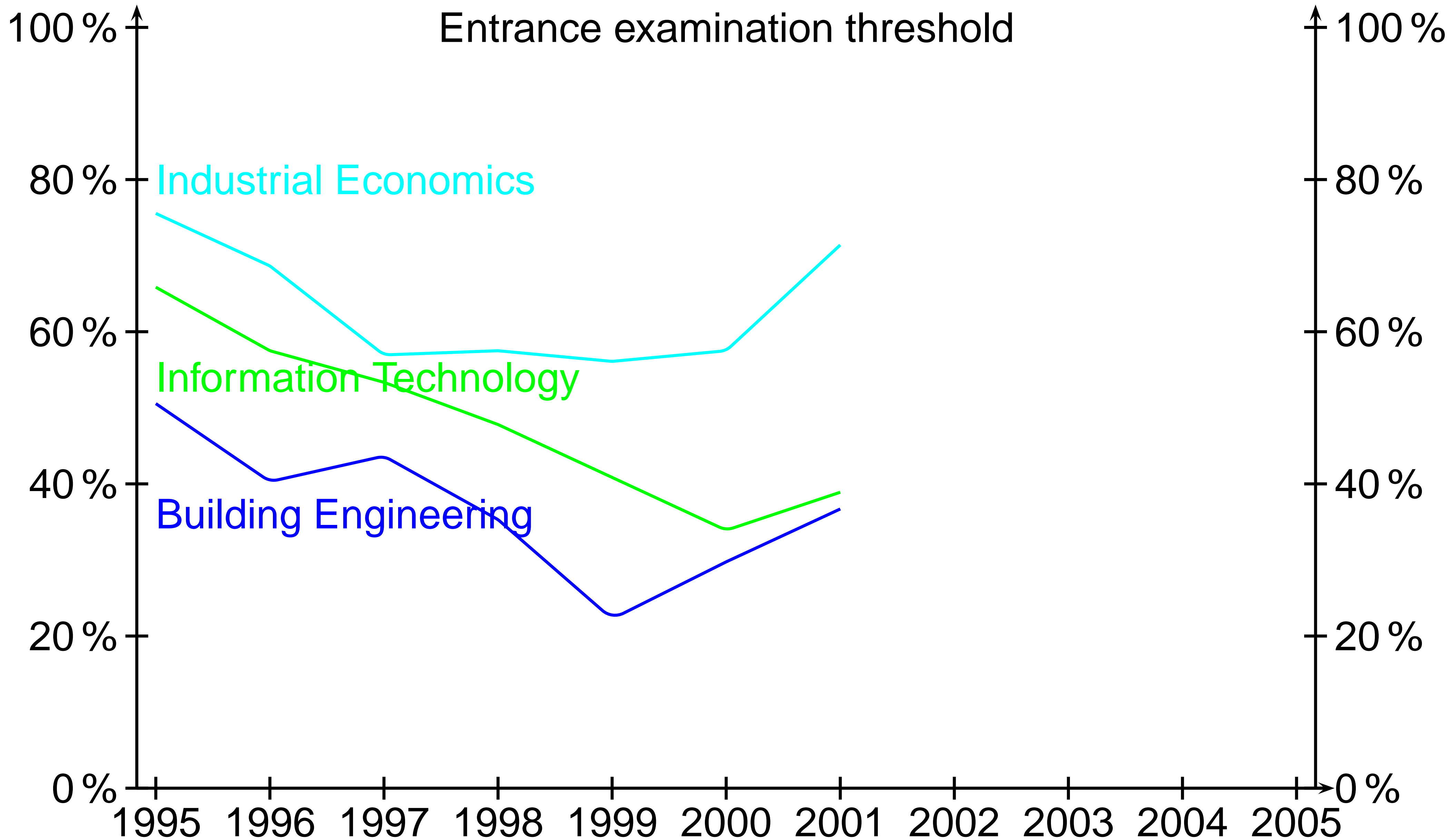
Why does this bother *me*?



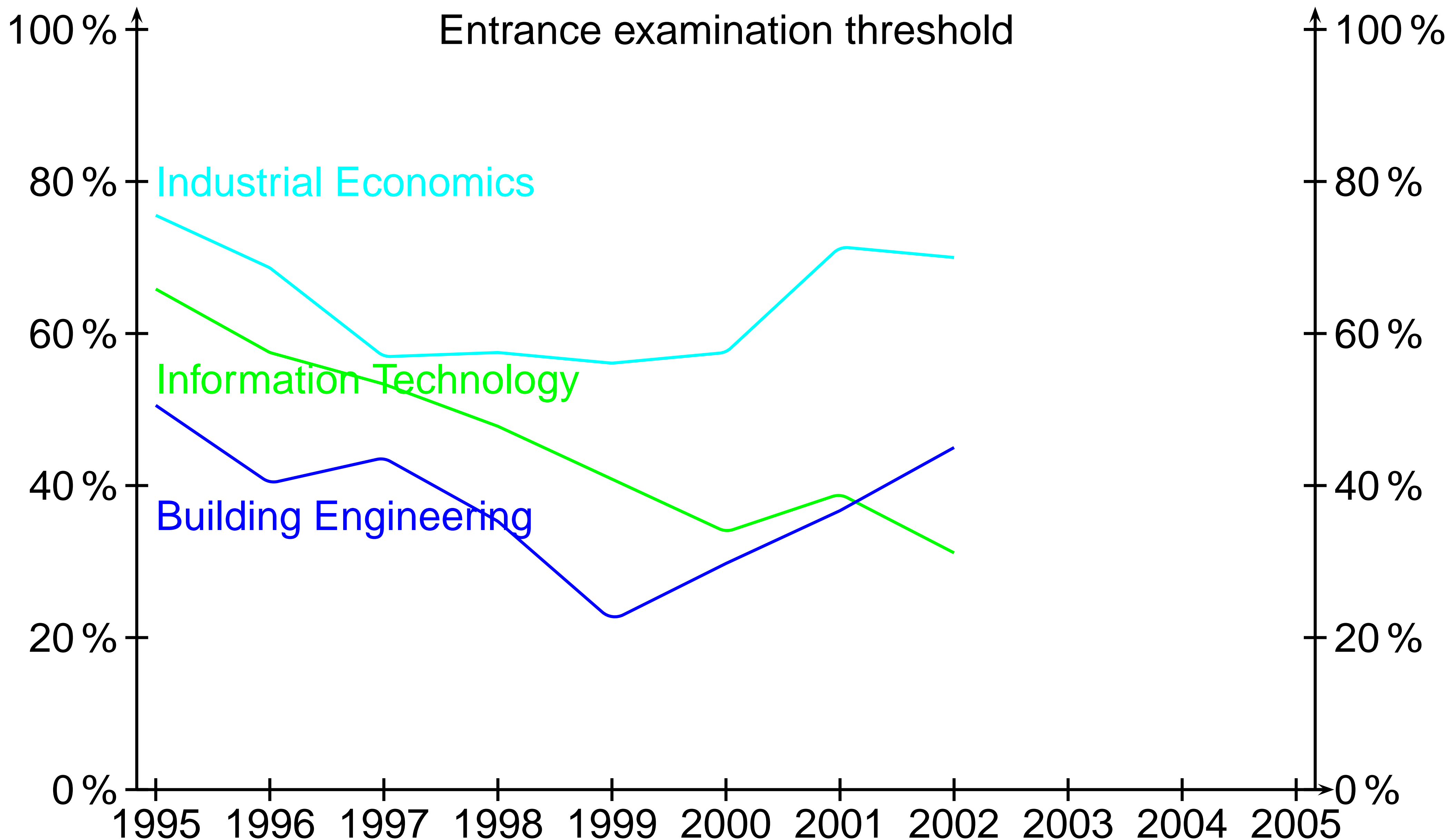
Why does this bother *me*?



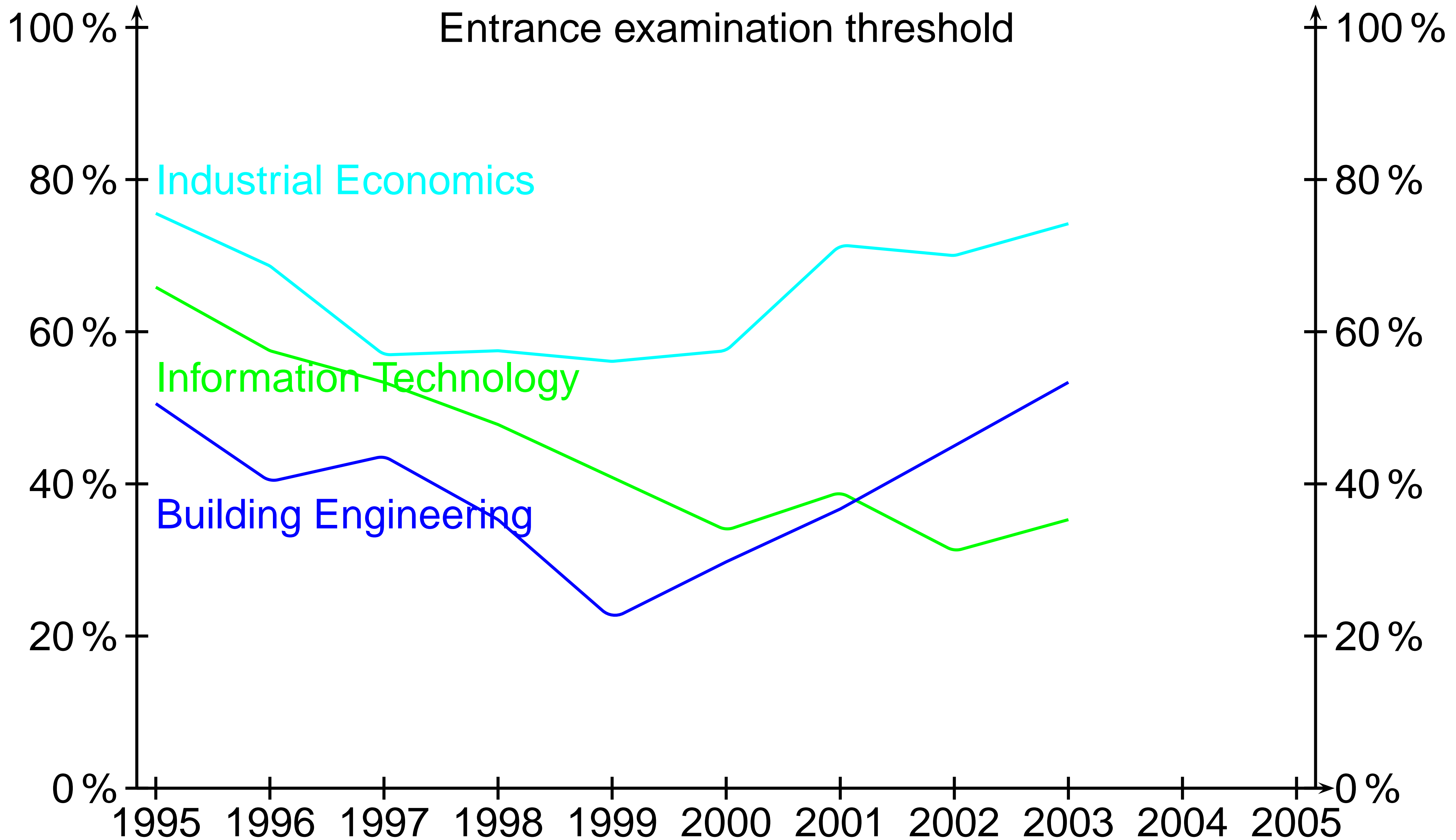
Why does this bother *me*?



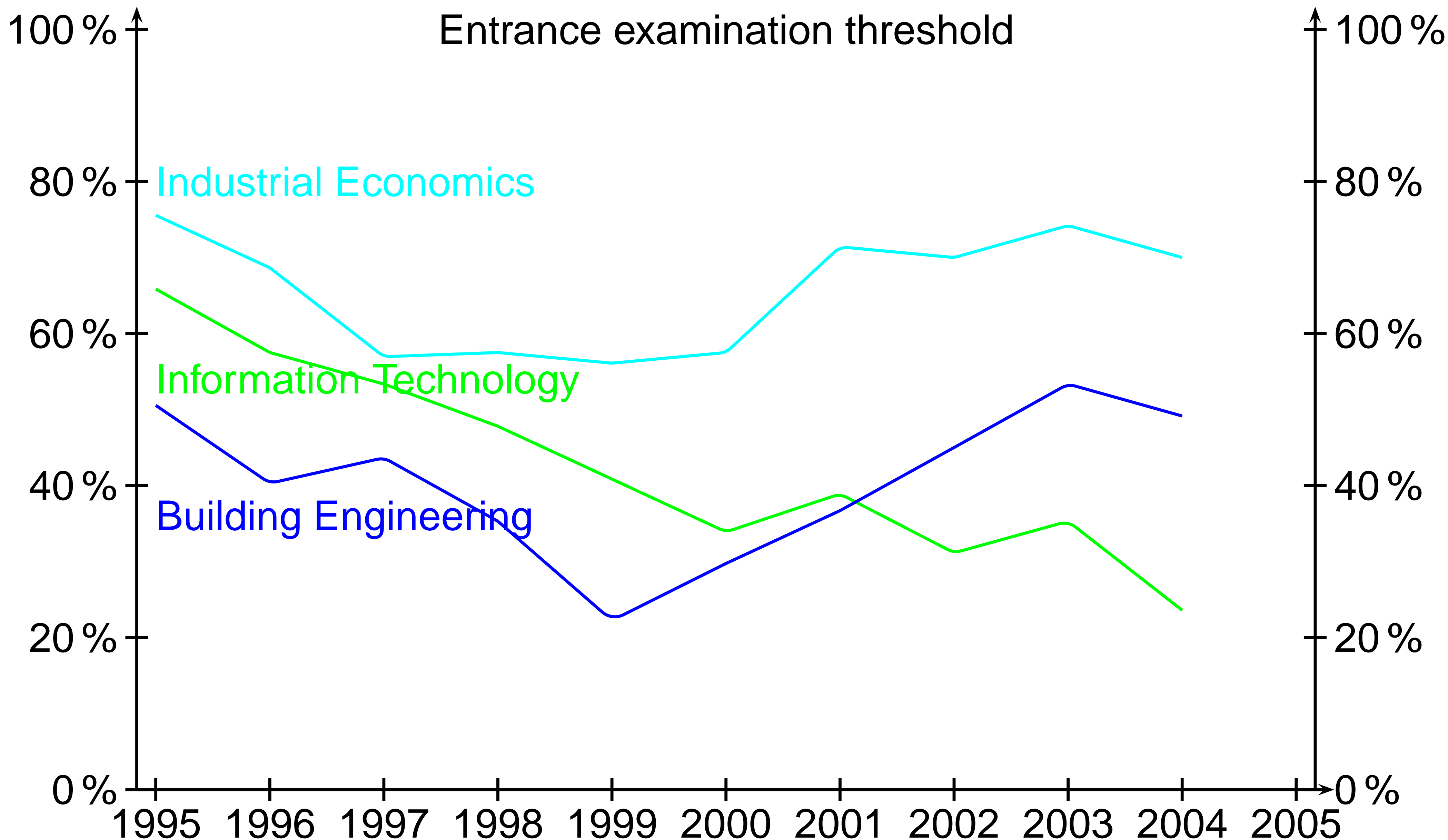
Why does this bother *me*?



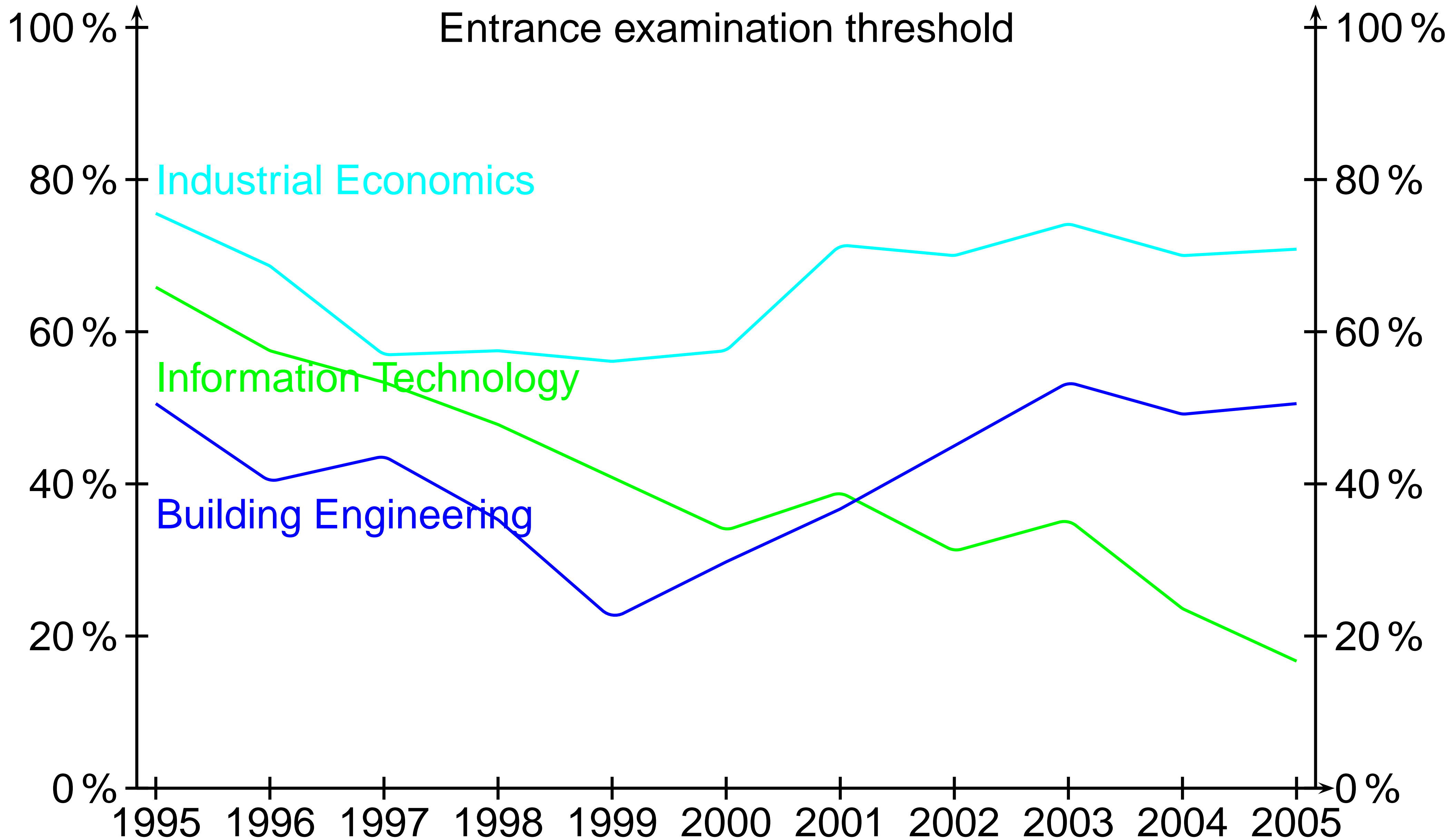
Why does this bother *me*?



Why does this bother *me*?



Why does this bother *me*?



2 Is mathematics necessary any more?

- Algorithm design and analysis?
  - Very good algorithms are available in standard libraries
  - Mediocre algorithms are fast enough most of the time on today's HW
  - Example: C++ algorithm project runtimes were often dominated by I/O

$\Rightarrow$ Programming changes: more libraries & design patterns & . . . , less coding

- What they don't learn does not benefit anyone

  $\Rightarrow$ It is ~~not worth teaching~~ frustrating to teach

- What do practicing engineers think?

  $\Rightarrow$ Lethbridge's survey . . .

Lethbridge's survey ...

- *IEEE Computer* May 2000 cover story
- Small ($<$ 200) and North America -dominated, but otherwise good

rank	topic	importance
1.	specific programming languages	3.8
2.	data structures	3.6
3.	software design and patterns	3.5
4.	software architecture	3.4
5.	requirements gathering & analysis	3.4
28.	programming language theory	2.7
30.	computational complexity & algorithm analysis	2.6

. . . Lethbridge's survey

rank	topic	importance
39.	predicate logic	2.4
48.	set theory	2.2
49.	automata theory	2.1
67.	differential and integral calculus	1.6
68.	combinatorics	1.6
69.	artificial intelligence	1.5
70.	analog electronics	1.5
71.	Laplace and Fourier transforms	1.3
72.	differential equations	1.3
73.	chemistry	1.3
74.	robotics	1.3
75.	VLSI	1.2

Really unnecessary, or taught in a wrong way?

- Public opinion: software engineering requires mathematical talent
- Universities should favour fundamental principles
  - Applicability often indirect

$\Rightarrow$ ***Influence on thinking***

- Influence on thinking had 50 % weight in the above ranking
- Observation: Influence rank seldom exceeded detail rank significantly
  - Exceeded mostly when both ranks were low

  $\Rightarrow$ People do not seem to appreciate or recognize influence on thinking
  - Or does it not exist?

My experience & opinion

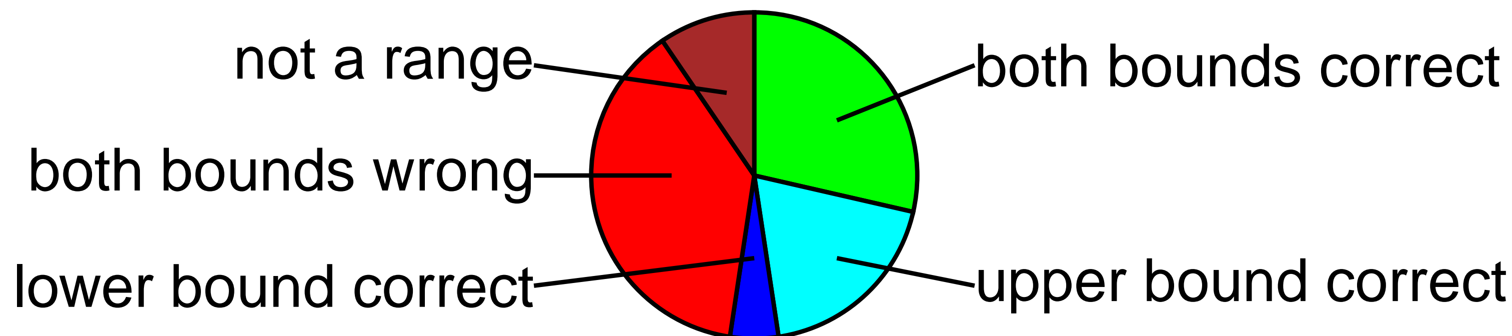
1. Traditional mathematics *is* unnecessary — even combinatorics!
   - Laaater . . .
2. Mathematical thinking *is* necessary — even in programming-in-the-small
   - Next topic
3. Mathematical thinking would be *very necessary* in programming-in-the-large
   - Later . . .

Math. thinking vanishes ⇒ progr. skills vanish

- 2nd year SW mathematics course examination question:
  "What values can j have at the end of the following program?"

  j := 1; *limit* := $A[n]$
  for k := 1 **to** n **do**
  if $A[k] \leq$ *limit* **then** *tmp* := $A[j]$; $A[j]$:= $A[k]$; $A[k]$:= *tmp*; j := j+1 **endif**
  endfor

- 21 students in the examination
  - most had already failed ≥ 1 exam

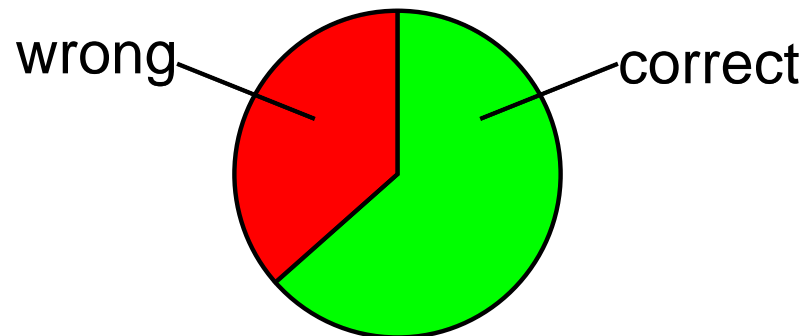


Another examination question

- "What does the following print?"

```
x = 8; y = 3; cout << x- - -y; cout << x << y;
```

- 115 students in the examination
  - Most of those who took the course that year



- Apparent explanation:
  - Students learn programming structures as patterns
  - Double negation is not useful $\Rightarrow$ not in memoized patterns
  - Unusual spacing can break a pattern
  - Angry student comment: "That was of strictly forbidden style!"

It seems as if

- Many cannot reason about simple programs
- Many fail to master simple syntactical structures of their main programming language
- Instead of trying to ***understand abstract structures***, many try to ***memoize the most common patterns***
  - A similar observation was made in the Maths Department
  - Such students can solve problems only as far as their memoized collection of patterns allows
  - ⇒ When problem complexity grows, at some point the performance of such students decreases sharply
  - Hypothesis: this phenomenon can be demonstrated

What about programming-in-the-large? ...

- Many (most?) expensive programming errors were ultimately made during requirements analysis and specification phases

⇒ It would be important to understand even subtle details well early on

- It is the task of software professionals, not customers, to understand logical subtleties!

  The computer is a hair-splitter, but (software) systems are built for humans. Therefore, a software designer must act as an arbitrator between the computer and the users.

- Unlike programming, mercyless tool support is not available
  - UML tools don't say "syntax error" or "core dumped"

⇒ Thinking skills are even more important than in programming-in-the-small

... What about programming-in-the-large?

The most valuable service that mathematical education can do to the software profession is to teach the students to recognise, formulate and reason with abstractions.

The challenge is to get important details correct, even when they are subtle and the end users or customers do not see any problem with them.

3 Is there any hope?

- I certainly do not claim to have a solution!
- I try to point to directions where not everything has been tried yet

What and how to teach? . . .

- Better to admit it: differential calculus, . . . are waste of time
  - No direct applicability to software engineering
  - Teaching focuses on ***applications*** in electrical engineering, etc.; not on the nature of definitions, proofs, . . .

  $\Rightarrow$ Fails to develop ***abstraction manipulation skills***

- Combinatorics is useful in specialised algorithm optimisation
  - Very few do that nowadays

. . . What and how to teach?

- Logic, set theory, automata theory, etc. are right topics, but taught wrong
  - Often taught as branches of mathematics, not as tools for manipulating abstractions
  - "Advanced Engineering Calculus" $\sim$ "Set Theory for SW Engineers"
- Such books actually exist, e.g.:

  Gries, D. & Schneider, F. B.:
  A Logical Approach to Discrete Math.
  Springer-Verlag, 1994, 497 p.

  - User opinions very divided!

  ". . . it really taught me how to use propositional calculus while applying it to computer programming." "In 3 Words, This book sucks."

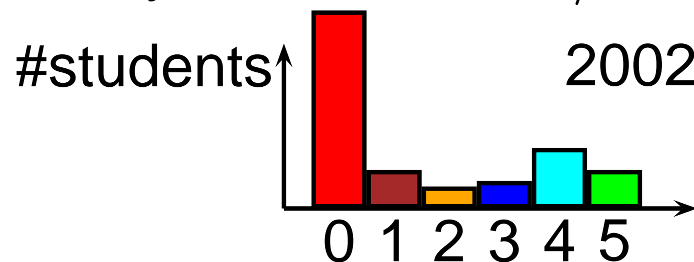
$\Rightarrow$ More work needed to find the right way of teaching

My experience on 2nd year obligatory software mathematics course

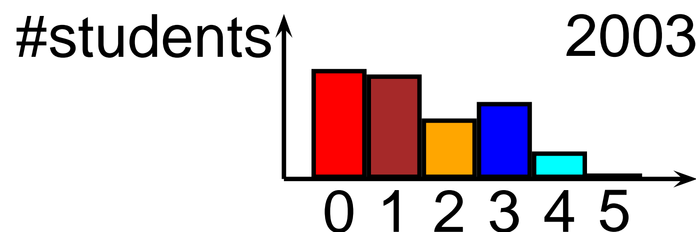
- Introduced in 2002
- Observation: it is next to impossible to make those students do homework who need it most!
  - In Finland, students may usually postpone unpleasant courses for ever
  - $\Rightarrow$ Some students take the examination year after year without ever seriously studying
  - Finnish students have a lot of part-time (or “part”-time) jobs
  - Univ. Jyväskylä survey: typical student studies only $\approx$ 20 hours / week

Illusions crumbling away ...

- I had always disliked scheduled obligatory work
  - I preferred voluntary work and extra points
- Examination results with various extra point systems
  - Good system $\Rightarrow$ Best $1/4$ of students do homework seriously



  - Bad system $\Rightarrow$ Nobody does homework seriously



- Some students apparently plagiarize solutions

. . . Illusions crumbling away

- I had always considered syntax a secondary issue
  - I thought that teachers should leave the hair-splitting on missing commas to compilers
- Observation: many students cannot write a readable logical formula at a late stage of the course
  - They have already taken (have they?) a course on logic and set theory
  - Logic has been extensively refreshed in my course
  - Most of the course material and exercises use logical formulas
- I do not believe that they cannot learn the syntax of logic, after learning (?) the syntax of C++

$\Rightarrow$ They have not tried enough

Why to promote learning the syntax?

- Semantics cannot be understood without some idea of syntax
- It is easier to learn syntax first

How?

- Announcing that syntax errors will be treated harshly helped a bit
- Students do not have enough opportunities to get feedback
  - Teaching assistants' time is limited and only available at certain times
  - There is no logic syntax checker

$\Rightarrow$ Let's make a logic syntax checker!

- Possible, but nontrivial
  (e.g., precedence in $\sin 2x = 2 \sin x \cos x$)

One more observation

- "Each of next examinations will almost certainly contain a question on expression trees"

  ⇒ Students knew expression trees pretty well

- "Each of next examinations will almost certainly contain a question on pumping lemma"

  ⇒ Very few mastered pumping lemma

4 Concurrency

- Concurrent programming is more important than ever:
  the net, multi-core processors
- It is less mature than sequential programming
  - Its programming primitives vary a lot compared to **if**, **while**, **class**
  - Primitives are usually in libraries, not in the language
- Concurrent program behaviour is hard to grasp
  - SW engineering tries to grasp it as collections of sequential objects (e.g., message sequence chart, use case)
- Theory and practice are further away from each other than in sequential
  - No similar continuum as programs — algorithms — complexity — **NP**-completeness — Turing machines
  - No concurrency theory is as universally accepted as Turing machines

Change in teaching

- Traditionally concurrency has been taught in operating system courses
- Now more and more books and courses promote it to a topic in its own right
- My local colleague's experience:
  - Students take it more seriously now that it is a course of its own at TUT
  - It is challenging enough to justify devoting a full course to it

My guess

- Confusion will prevail for some years
- Gradually a new line of thought will develop, which makes concurrent programming accessible to the masses
- I think that we have seen it happen before
  - 1985: Pointers and dynamic memory are too difficult for the masses
  - 2006: Anyone can use Java objects

My recommendation ...

- When teaching concurrency to advanced students, teach ***more than one theory***
  - Students realise that no theory is universally adopted
  - They learn to distinguish general issues from theory-specific
  - They get more ideas that they can apply later on
- General students should be taught to appreciate that concurrent programming is very difficult

. . . My recommendation

- State space is a useful concept to teach also to general students
  - Although impractical, helps a lot in understanding concurrent behaviour
  - Drawing a small state space forces to think carefully about the operational semantics of the conc. programming notation in question: atomicity, state-storing entities

$\Rightarrow$ Teaching the fundamentals of some "impractical" concurrency theory to all SW engineering students might be a good idea

  - It does not matter much which theory, but . . .
  - . . . even so, I recommend state machines: used also elsewhere in SW
  - Could perhaps replace some less useful mathematics in 2nd year theoretical background studies

Thank you for attention!