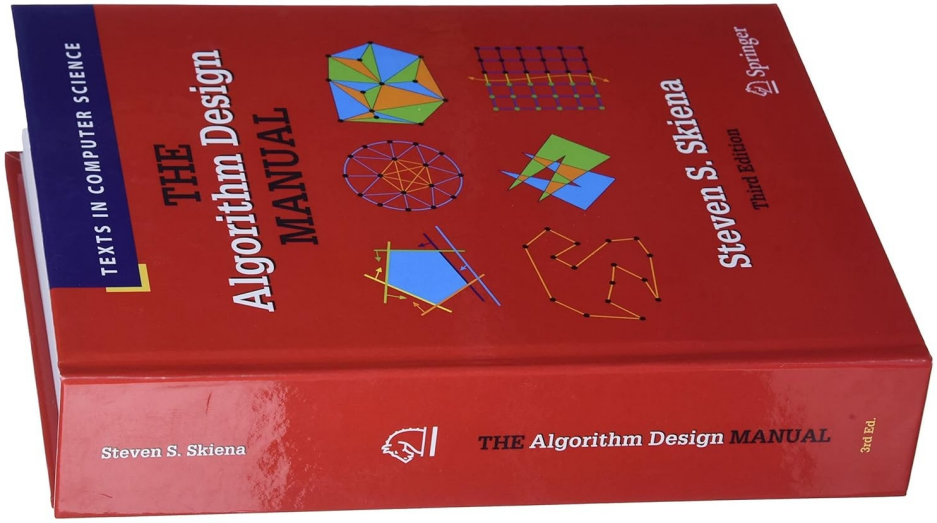
Introduktion och Kursplan

Tillämpad Algoritmisk Problemlösning (Våren 2026)

Martin Berglund, mbe@cs.umu.se

Kurslitteratur

"The Algorithm Design Manual" av Steven S. Skiena, tredje upplagan



Väldigt väldigt bra. 787:- på `adlibris.se`

- $\Rightarrow$ Andra upplagan liknande, *men inga garantier!*
- $\Rightarrow$ Kan vara enkelt att hitta åtminstone andra upplagan...

Ny kurs!

Detta är en *nästan* helt ny kurs, i *innehåll* och *typ*, som en följd:

- Vi kommer behöva känna oss för tillsammans: kommunicera!
- Jag är *övergripande* förberedd, men har undvikt att fixera detaljer
  - Er feedback kommer att forma kursen
  - Era framsteg och bakslag kommer att forma kursen
  - Jag kommer i senare delar låta er välja mellan olika ämnesområden (vill vi titta på sektion 21.3 om strängmatchning eller 17.9 om schemaläggning av jobb?)

Ny kurs!

Detta är en *nästan* helt ny kurs, i *innehåll* och *typ*, som en följd:

- Vi kommer behöva känna oss för tillsammans: kommunicera!
- Jag är *övergripande* förberedd, men har undvikt att fixera detaljer
  - Er feedback kommer att forma kursen
  - Era framsteg och bakslag kommer att forma kursen
  - Jag kommer i senare delar låta er välja mellan olika ämnesområden (vill vi titta på sektion 21.3 om strängmatchning eller 17.9 om schemaläggning av jobb?)
- För att möjliggöra: improvisation i kursmaterial, delar tagna från:
  - 'bokens' slides (se `algorist.com`)
  - en gammal kurs i komputationell geometri
  - sommarkursen *"programmering för effektiv problemlösning"*,
  - plus, förstås, mycket nytt.

Språk och material

Kursen kommer ha en spretig grafisk och språklig profil.

- Existerande slides från tre olika källor, kommer dessutom *handskriva* vissa (mer experimentella) slides
- Uppgifter från boken, Kattis, och nyskrivna.

Vi kommer att blanda *svenska* och *engelska* ganska fritt: bokens material och komputationell geometri på engelska.

Språk och material

Kursen kommer ha en spretig grafisk och språklig profil.

- Existerande slides från tre olika källor, kommer dessutom *handskriva* vissa (mer experimentella) slides
- Uppgifter från boken, Kattis, och nyskrivna.

Vi kommer att blanda *svenska* och *engelska* ganska fritt: bokens material och komputationell geometri på engelska.

En sak jag *inte planerar att ändra,* även om jag skulle få oändligt med tid: (minst) två programmeringsspråk: *C* och *Python*.

Hela kursen *bör* gå att genomföra på endera, men det *enkla och tänkta* sättet är att välja det som passar det man gör:

- högpresterande program med profilering, optimering, analys: *C,*
- knacka ut något snabb för ett problem: *Python,*
- på slides: typiskt *Python.*

Jag tänker *anta* att ni snappar upp Python som självstudie! (inte svårt)

Förväntade studieresultat

Kunskap och förståelse

Efter avslutad kurs ska studenten kunna:

- (FSR 1) Beskriva ett antal kategorier av algoritmer (exempelvis giriga algoritmer, söndra & härska-algoritmer, dynamisk programmering, sökmetoder med bakåtspårning, kombinatorisk sökning) samt förklara ett urval av specifika algoritmer ur vardera kategori.

Färdighet och förmåga

Efter avslutad kurs ska studenten kunna:

- (FSR 2) Konkretisera ett informellt formulerat problem på ett sätt som gör det möjligt att behandla problemet algoritmiskt

Färdighet och förmåga

Efter avslutad kurs ska studenten kunna:

- (FSR 2) Konkretisera ett informellt formulerat problem på ett sätt som gör det möjligt att behandla problemet algoritmiskt.
- (FSR 3) Planera en lämplig algoritmisk strategi för att lösa ett konkret problem.
- (FSR 4) Designa, genomföra och dokumentera experiment som utvärderar en algoritmisk lösnings lämplighet i en informellt beskriven kontext.
- (FSR 5) Implementera och testa en algoritmisk lösning för ett informellt formulerat problem, samt på ett systematiskt sätt uppdatera lösningen för förändrade behov.

Värderingsförmåga och förhållningssätt

Efter avslutad kurs ska studenten kunna:

- (FSR 6) Vetenskapligt resonera om och värdera mått på en algoritmisk lösnings beteenden, inklusive både experimentella resultat och de olika formerna av ordo-notation.

Värderingsförmåga och förhållningssätt

Efter avslutad kurs ska studenten kunna:

- (FSR 6) Vetenskapligt resonera om och värdera mått på en algoritmisk lösnings beteenden, inklusive både experimentella resultat och de olika formerna av ordo-notation.

Examination

Examinationen sker dels genom en skriftlig salstentamen, dels genom ett antal skriftliga inlämningsuppgifter. På kursen ges något av betygen Väl Godkänd (VG), Godkänd (G) eller Underkänd (U). För att bli godkänd på hela kursen krävs att samtliga examinerande moment är godkända. Betyget

Examination

Examinationen sker dels genom en skriftlig salstentamen, dels genom ett antal skriftliga inlämningsuppgifter. På kursen ges något av betygen Väl Godkänd (VG), Godkänd (G) eller Underkänd (U). För att bli godkänd på hela kursen krävs att samtliga examinerande moment är godkända. Betyget

I praktiken är planen tre delar:

- Kattis-problem: designa och koda lösningar på informellt ställda *tävlingsprogrammeringsproblem*
- Lite större algoritmisk *programmeringsuppgift:* design, kod, profilering, analys, rapport. Tema kartor och geometri
- Skriftlig tentamen: problem liknande Kattis (på papper) och bokens

Vi kommer också titta på ett antal problem ur boken.

En vanhelgande kurs

Jag utgår från att ni kan hålla isär följande:

- När man experimenterar kanske man skriver ostrukturerad kod *men* lång-levande kod bör vara strukturerad
- När man experimenterar kanske man inte dokumenterar/kommenterar väl *men* lång-levande kod bör vara dokumenterad och kommenterad
- Ibland berättar asymptoterna (t.ex. tid $\mathcal{O}(n^2)$) inte hela historien *men* formell komplexitetsanalys är ofta viktigt
- Ibland är en egen enkel lösning vägen framåt *men* givet tid har någon annan oftast tänkt längre om de flesta problem

Å andra sidan:

- Diagnostiserbar kod är alltid viktigt. Loggnings-debuggning är faktiskt bra, kod för att formatera och visualisera information guld värd!
- Förståelse för vad ens verktyg gör är alltid bra

En konstruktiv kurs

En “problemlösning i praktiken”-kurs. Uppgifterna är individuella, men i all praktik samarbetar man med människor:

- Var *hjälpsammma* med varandra (inom rimliga gränser!)
- *Kommunicera* vid problem och motgångar
- Var *vänliga,* detta är ingen tävling

Detta är *realism:* i både industri och forskningsprojekt spelar det ingen roll hur smart eller effektiv man är om man är svår att arbeta med.

En konstruktiv kurs

En “problemlösning i praktiken”-kurs. Uppgifterna är individuella, men i all praktik samarbetar man med människor:

- Var *hjälpsammma* med varandra (inom rimliga gränser!)
- *Kommunicera* vid problem och motgångar
- Var *vänliga,* detta är ingen tävling

Detta är *realism:* i både industri och forskningsprojekt spelar det ingen roll hur smart eller effektiv man är om man är svår att arbeta med.

Slutlig notis: systerkurs till två kurser:

- *5DV182:* Effektiva Algoritmer. Ett *komplicerat överlapp* planerat!
- *5DV232:* Programmering för effektiv problemlösning. *Stora likheter:* jobbase på. *Inte helt säkert* att den går att kombinera med denna i examen!

Från industri: prissättning sammansatta produkter

Inte alla algoritmer finns på hyllan, för en börs:

- Mary wants to buy (up to) 10 apples for at most $100 each
- John wants to trade away 4 apples for eggs, *paying* $2 each time

Från industri: prissättning sammansatta produkter

Inte alla algoritmer finns på hyllan, för en börs:

- Mary wants to buy (up to) 10 apples for at most $100 each
- John wants to trade away 4 apples for eggs, *paying* $2 each time
- ICA wants to sell 1000 cartons of 6 eggs for $590 each

Från industri: prissättning sammansatta produkter

Inte alla algoritmer finns på hyllan, för en börs:

- Mary wants to buy (up to) 10 apples for at most \$100 each
- John wants to trade away 4 apples for eggs, *paying* \$2 each time
- ICA wants to sell 1000 cartons of 6 eggs for \$590 each
- Ellen the baker trades away *a cake* for 8 eggs *charging* \$200

Från industri: prissättning sammansatta produkter

Inte alla algoritmer finns på hyllan, för en börs:

- Mary wants to buy (up to) 10 apples for at most \$100 each
- John wants to trade away 4 apples for eggs, *paying* \$2 each time
- ICA wants to sell 1000 cartons of 6 eggs for \$590 each
- Ellen the baker trades away *a cake* for 8 eggs *charging* \$200

How much would you have to *pay* to *get a cake?*

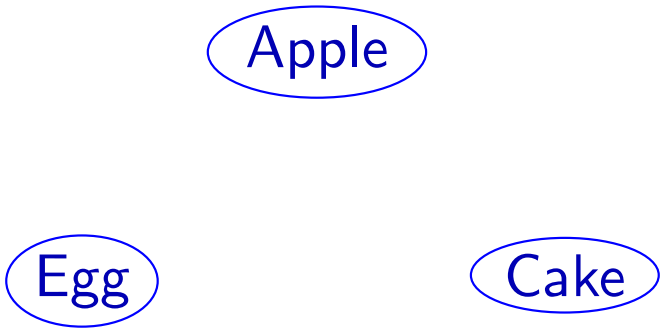
Från industri: prissättning sammansatta produkter

Inte alla algoritmer finns på hyllan, för en börs:

- Mary wants to buy (up to) 10 apples for at most \$100 each
- John wants to trade away 4 apples for eggs, *paying* \$2 each time
- ICA wants to sell 1000 cartons of 6 eggs for \$590 each
- Ellen the baker trades away *a cake* for 8 eggs *charging* \$200

How much would you have to *pay* to *get a cake?*

With *futures contracts* for e.g. *electricity* and *ore* this is ASX24.
Needed *new algorithms* for this. Begun with data structure:



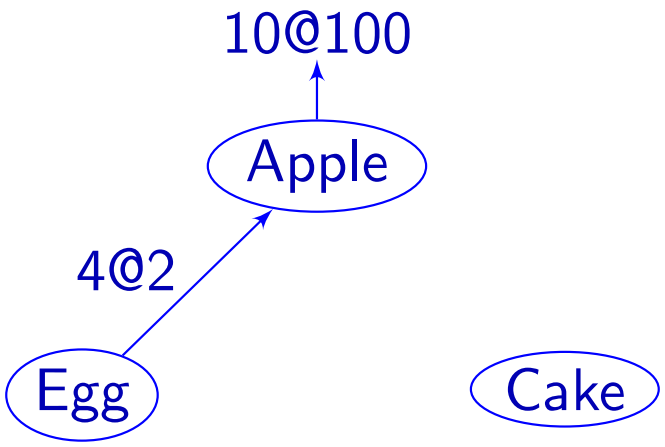
Från industri: prissättning sammansatta produkter

Inte alla algoritmer finns på hyllan, för en börs:

- Mary wants to buy (up to) 10 apples for at most \$100 each
- John wants to trade away 4 apples for eggs, *paying* \$2 each time
- ICA wants to sell 1000 cartons of 6 eggs for \$590 each
- Ellen the baker trades away *a cake* for 8 eggs *charging* \$200

How much would you have to *pay* to *get a cake?*

With *futures contracts* for e.g. *electricity* and *ore* this is ASX24.
Needed *new algorithms* for this. Begun with data structure:



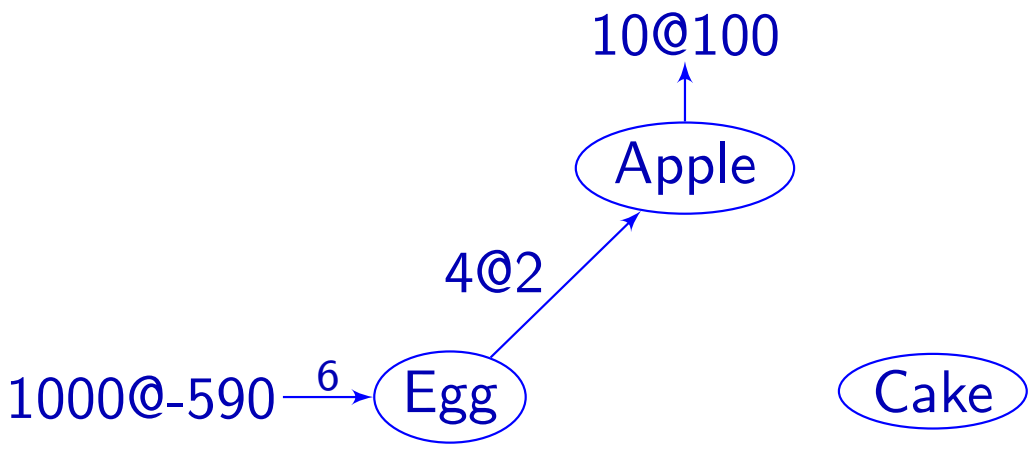
Från industri: prissättning sammansatta produkter

Inte alla algoritmer finns på hyllan, för en börs:

- Mary wants to buy (up to) 10 apples for at most \$100 each
- John wants to trade away 4 apples for eggs, *paying* \$2 each time
- ICA wants to sell 1000 cartons of 6 eggs for \$590 each
- Ellen the baker trades away *a cake* for 8 eggs *charging* \$200

How much would you have to *pay* to *get a cake?*

With *futures contracts* for e.g. *electricity* and *ore* this is ASX24.
Needed *new algorithms* for this. Begun with data structure:



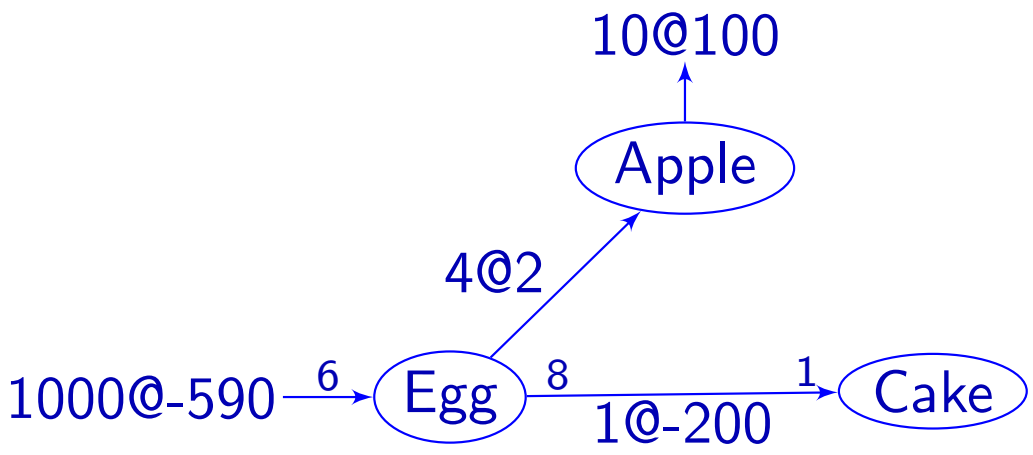
Från industri: prissättning sammansatta produkter

Inte alla algoritmer finns på hyllan, för en börs:

- Mary wants to buy (up to) 10 apples for at most \$100 each
- John wants to trade away 4 apples for eggs, *paying* \$2 each time
- ICA wants to sell 1000 cartons of 6 eggs for \$590 each
- Ellen the baker trades away *a cake* for 8 eggs *charging* \$200

How much would you have to *pay* to *get a cake?*

With *futures contracts* for e.g. *electricity* and *ore* this is ASX24.
Needed *new algorithms* for this. Begun with data structure:



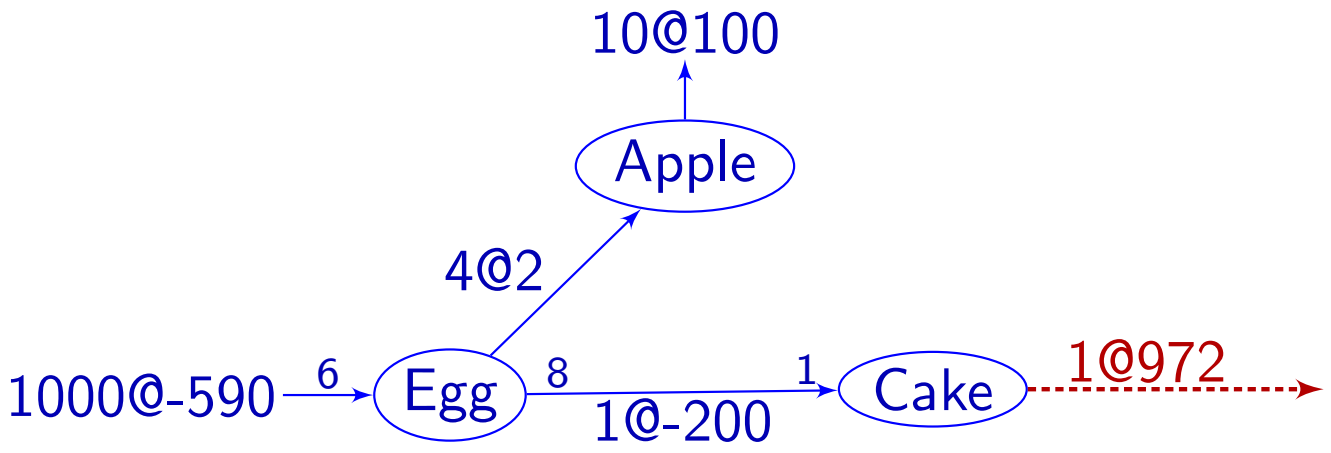
Från industri: prissättning sammansatta produkter

Inte alla algoritmer finns på hyllan, för en börs:

- Mary wants to buy (up to) 10 apples for at most \$100 each
- John wants to trade away 4 apples for eggs, *paying* \$2 each time
- ICA wants to sell 1000 cartons of 6 eggs for \$590 each
- Ellen the baker trades away *a cake* for 8 eggs *charging* \$200

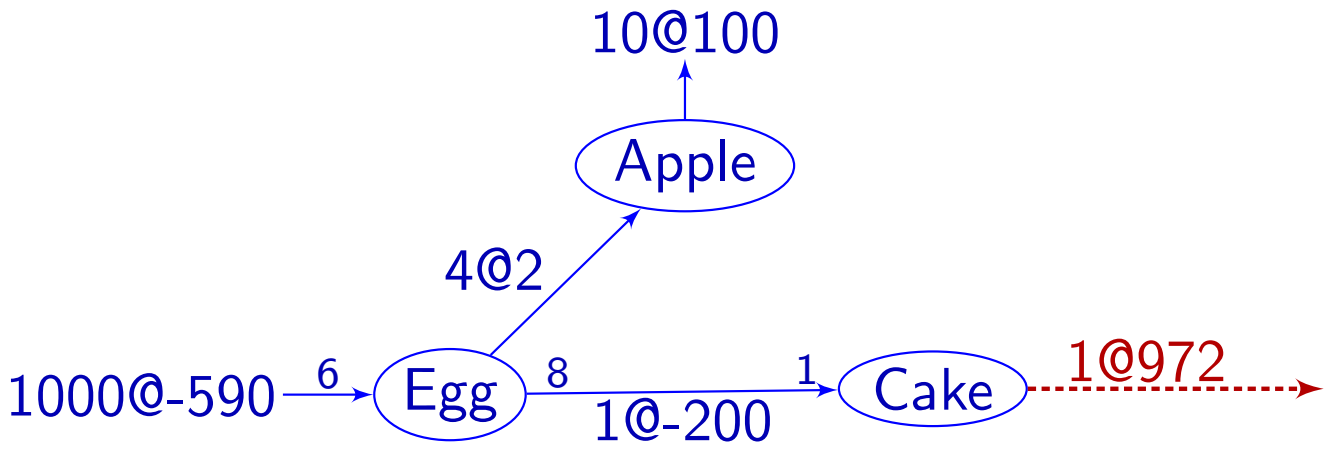
How much would you have to *pay* to *get a cake?*

With *futures contracts* for e.g. *electricity* and *ore* this is ASX24.
Needed *new algorithms* for this. Begun with data structure:



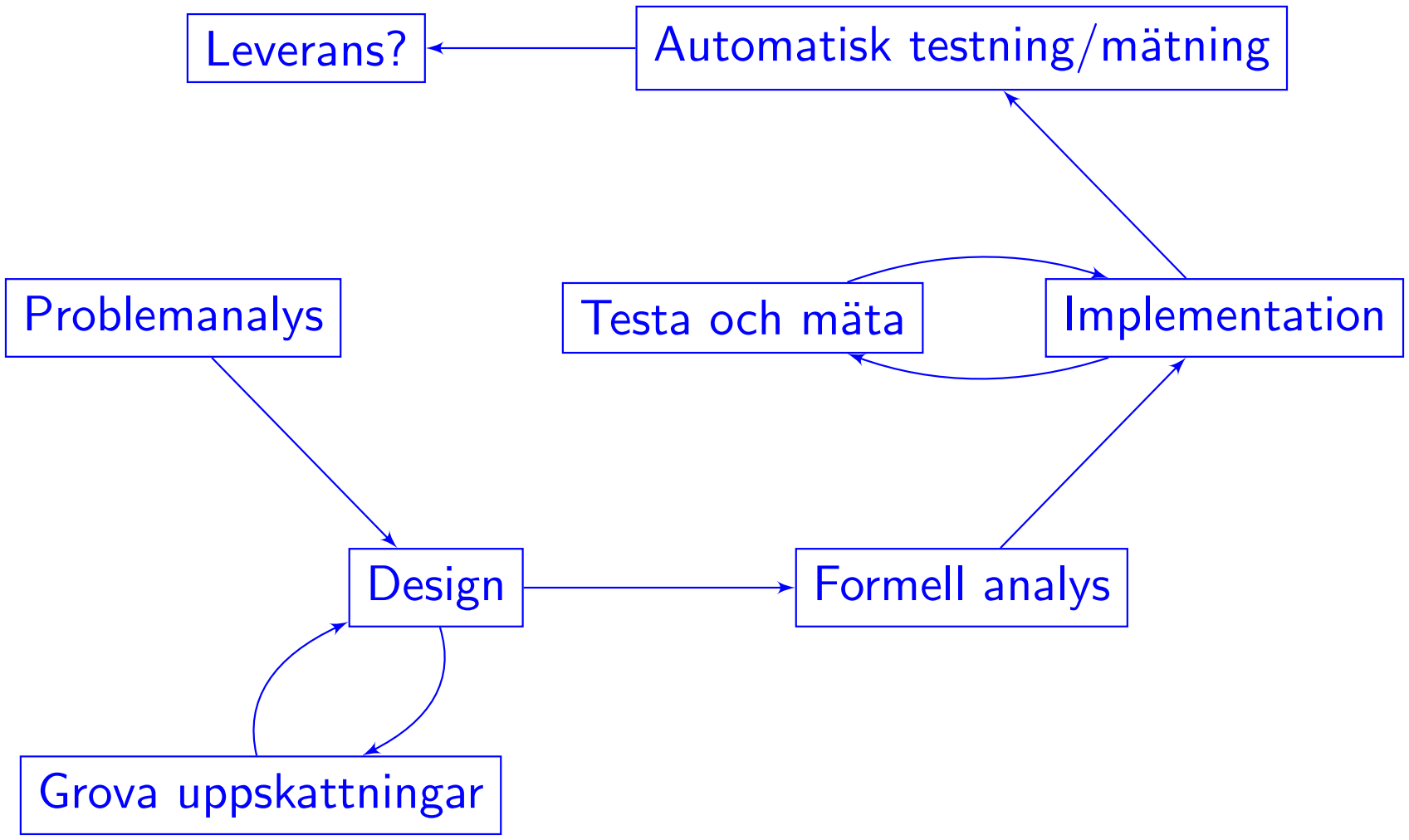
Från industri: prissättning sammansatta produkter

With *futures contracts* for e.g. *electricity* and *ore* this is ASX24.
Needed *new algorithms* for this. Begun with data structure:



Very hard in general, approximated: *modified Johnson's algorithm*, *Edmonds-Karp*, and *a detuning post-processing step.*

Utvecklingen av en algoritm



Plus då alla pilar bakåt...

Chapter 13

How to Design Algorithms

Designing the right algorithm for a given application is a major creative act—that of taking a problem and pulling a solution out of the air. The space of choices you can make in algorithm design is enormous, leaving you plenty of freedom to hang yourself.

This book has been designed to make you a better algorithm designer. The techniques presented in Part I provide the basic ideas underlying all combinatorial algorithms. The problem catalog of Part II will help you with modeling your application, and inform you what is known about the relevant problems. However, being a successful algorithm designer requires more than book knowledge. It requires a certain attitude—the right problem-solving approach. It is difficult to teach this mindset in a book, yet getting it is essential to becoming a successful algorithm designer.

The key to algorithm design (or any other problem-solving task) is to proceed by asking yourself questions to guide your thought process. "What if we do this? What if we do that?" Should you get stuck on the problem, the best thing to do

by asking yourself questions to guide your thought process. "What if we do this? What if we do that?" Should you get stuck on the problem, the best thing to do is move onto the next question. In any group brainstorming session, the most useful person in the room is the one who keeps asking "Why can't we do it this way?"; not the nitpicker who keeps telling them why. Because he or she will eventually stumble on an approach that can't be shot down.

Towards this end, I provide a sequence of questions designed to guide your search for the right algorithm for your problem. To use it effectively, you must not only ask the questions, but answer them. The key is working through the answers carefully by writing them down in a log. The correct answer to "Can I do it this way?" is never "no," but "no, because. . . ." By clearly articulating your reasoning as to why something doesn't work, you can check whether you have glossed over a possibility that you didn't think hard enough about. It is amazing how often the reason you can't find a convincing explanation for something is because your conclusion is wrong.

The distinction between *strategy* and *tactics* is important to keep aware of during any design process. Strategy represents the quest for the big picture—the framework around which we construct our path to the goal. Tactics are used

S. S. Skiena, *The Algorithm Design Manual*, Texts in Computer Science,
https://doi.org/10.1007/978-3-030-54256-6_13

"Should I write in C 2?"

to win the minor battles we must fight along the way. In problem-solving, it is important to repeatedly check whether you are thinking on the right level. If you do not have a global strategy of how to attack your problem, it is pointless to worry about the tactics. An example of a strategic question is "Can I model my application as a graph algorithm problem?" A tactical question might be "Should I use an adjacency list or adjacency matrix data structure to represent my graph?" Of course, such tactical decisions are critical to the ultimate quality of the solution, but they can be properly evaluated only in light of a successful strategy.

Too many people freeze up in their thinking when faced with a design problem. After reading or hearing the problem, they sit down and realize that they *don't know what to do next.* Avoid this fate. Follow the sequence of questions I provide below and in most of the catalog problem sections. I will try to *tell* you what to do next.

Obviously, the more experience you have with algorithm design techniques such as dynamic programming, graph algorithms, intractability, and data structures, the more successful you will be at working through the list of questions. Part I of this book has been designed to strengthen this technical background. However, it pays to work through these questions regardless of how strong your technical skills are. The earliest and most important questions on the list focus on obtaining a detailed understanding of your problem and do not require any specific expertise.

This list of questions was inspired by a passage in *The Right Stuff* [Wol79]—a wonderful book about the US space program. It concerned the radio transmis-

1. Do I really understand the problem?

   (a) What exactly does the input consist of?

   (b) What exactly are the desired results or output?

   (c) Can I construct an input example small enough to solve by hand? What happens when I try to solve it?

   (d) How important is it to my application that I always find the optimal answer? Might I settle for something close to the best answer?

   (e) How large is a typical instance of my problem? Will I be working on 10 items? 1,000 items? 1,000,000 items? More?

   (f) How important is speed in my application? Must the problem be solved within one second? One minute? One hour? One day?

   (g) How much time and effort can I invest in implementation? Will I be limited to simple algorithms that can be coded up in a day, or do I have the freedom to experiment with several approaches and see which one is best?

   (h) Am I trying to solve a numerical problem? A graph problem? A geometric problem? A string problem? A set problem? Which formulation seems easiest?

2. Can I find a simple algorithm or heuristic for my problem?

   (a) Will brute force solve my problem *correctly* by searching through all

formation seems easiest?

2. Can I find a simple algorithm or heuristic for my problem?

   (a) Will brute force solve my problem *correctly* by searching through all subsets or arrangements and picking the best one?

      i. If so, why am I sure that this algorithm always gives the correct answer?

      ii. How do I measure the quality of a solution once I construct it?

      iii. Does this simple, slow solution run in polynomial or exponential time? Is my problem small enough that a brute-force solution will suffice?

      iv. Am I certain that my problem is sufficiently well defined to actually *have* a correct solution?

   (b) Can I solve my problem by repeatedly trying some simple rule, like picking the biggest item first? The smallest item first? A random item first?

      i. If so, on what types of inputs does this heuristic work well? Do these correspond to the data that might arise in my application?

      ii. On what inputs does this heuristic work badly? If no such examples can be found, can I show that it always works well?

      iii. How fast does my heuristic come up with an answer? Does it have a simple implementation?

3. Is my problem in the catalog of algorithmic problems in the back of this

iv. Can I "eventually" tell if the correct choice was tried?

have a simple implementation?

3. Is my problem in the catalog of algorithmic problems in the back of this book?

   (a) What is known about the problem? Is there an available implementation that I can use?

   (b) Did I look in the right place for my problem? Did I browse through all the pictures? Did I look in the index under all possible keywords?

   (c) Are there relevant resources available on the World Wide Web? Did I do a Google Scholar search? Did I go to the page associated with this book: www.algorist.com?

4. Are there special cases of the problem that I know how to solve?

   (a) Can I solve the problem efficiently when I ignore some of the input parameters?

   (b) Does the problem become easier to solve when some of the input parameters are set to trivial values, such as 0 or 1?

   (c) How can I simplify the problem to the point where I *can* solve it efficiently? Why can't this special-case algorithm be generalized to a wider class of inputs?

   (d) Is my problem a special case of a more general problem in the catalog?

5. Which of the standard algorithm design paradigms are most relevant to my problem?

(a) Is there a set of items that can be sorted by size or some key? Does this sorted order make it easier to find the answer?

(b) Is there a way to split the problem into two smaller problems, perhaps by doing a binary search? How about partitioning the elements into big and small, or left and right? Does this suggest a divide-and-conquer algorithm?

(c) Does the set of input objects have a natural left-to-right order among its components, like the characters in a string, elements of a permutation, or the leaves of a rooted tree? Could I use dynamic programming to exploit this order?

(d) Are there certain operations being done repeatedly, such as searching, or finding the largest/smallest element? Can I use a data structure to speed up these queries? Perhaps a dictionary/hash table or a heap/priority queue?

(e) Can I use random sampling to select which object to pick next? What about constructing many random configurations and picking the best one? Can I use a heuristic search technique like simulated annealing to zoom in on a good solution?

(f) Can I formulate my problem as a linear program? How about an integer program?

(g) Does my problem resemble satisfiability, the traveling salesman problem, or some other NP-complete problem? Might it be NP-complete and thus not have an efficient algorithm? Is it in the problem list in the back of Garey and Johnson [GJ79]?

integer program?

(g) Does my problem resemble satisfiability, the traveling salesman problem, or some other NP-complete problem? Might it be NP-complete and thus not have an efficient algorithm? Is it in the problem list in the back of Garey and Johnson [GJ79]?

6. Am I still stumped?

(a) Am I willing to spend money to hire an expert (like the author) to tell me what to do? If so, check out the professional consulting services mentioned in Section 22.4 (page 718).

(b) Go back to the beginning and work through these questions again. Did any of my answers change during my latest trip through the list?

Öva, öva, öva: Kattis

Övning är viktigt: Vi borde nog alla lösa fler problem från grund (hur många gånger har ni hunnit med under utbildningen?)

En stor del av kursen är problemlösning på *umu.kattis.com* registrera er på:

`https://umu.kattis.com/courses/5DV239/VT2026`,

registreringskoden är *5DV239X*

Lös *tre problem* innan föreläsningen nästa tisdag!

Titta igenom alla uppgifterna och välj de ni föredrar

Cetiri

Mirko has chosen four integers which form an arithmetic progression. In other words, when the four numbers are sorted, then the difference between each pair of **adjacent** elements is **constant**.

As has become usual, Mirko **lost** one of the numbers and also is not sure whether the remaining three are in the correct (sorted) order.

Write a program that, given the three remaining numbers, finds the

Mirko has chosen four integers which form an arithmetic progression. In other words, when the four numbers are sorted, then the difference between each pair of **adjacent** elements is **constant**.

As has become usual, Mirko **lost** one of the numbers and also is not sure whether the remaining three are in the correct (sorted) order.

Write a program that, given the three remaining numbers, finds the fourth number.

Input

The input contains 3 integers between -100 and 100 on a single line, separated by single spaces.

Note: the input data will guarantee that a solution, although not necessarily unique, will always exist.

Output

Output any number which could have been the fourth number in the sequence.

Sample Input 1	Sample Output 1
4 6 8	10

fourth number.

Input

The input contains 3 integers between -100 and 100 on a single line, separated by single spaces.

Note: the input data will guarantee that a solution, although not necessarily unique, will always exist.

Output

Output any number which could have been the fourth number in the sequence.

Sample Input 1

```
4 6 8
```

Sample Output 1

```
10
```

Sample Input 2

```
10 1 4
```

Sample Output 2

```
7
```

Cetiri lösning

Vi observerar: förenklar att *sortera* input, sedan *tre fall*

```
import sys

# read and sort input integers (only last line matters)
vals=sys.stdin.readlines()[-1].split()
x,y,z=sorted([int(val) for val in vals])

# three cases
d1=y-x
d2=z-y
if d1==d2:
    # input in progression, create next larger (smaller eqv.)
    print(z+d1)
elif d1<d2:
    # d2 larger gap, new value goes there
    print(y+d1)
else:
    # d1 larger gap, new value goes there
    print(x+d2)
```