Европейски съюз

ОП “Развитие на човешките ресурси”
Договор BG051PO001-4.3.04-0018
Разработване на програми за електронни форми на дистанционно обучение във Факултета по математика и информатика

Европейски социален фонд

# Първа лекция по ДАА на втори поток КН
# Въведение в теорията на алгоритмите

24 февруари 2014

**Абстракт**

Въвеждаме понятията изчислителна задача и алгоритъм. Дискутираме елементарните инструкции, от които са съставени алгоритмите. Даваме като пример Евклидовия алгоритъм: най-старият известен истински алгоритъм. Въвеждаме големина на входа на алгоритъм и дискутираме условностите и опростяванията, които приемаме по отношение на големината на входа. Разглеждаме различните аспекти на анализа на алгоритмите: анализ на коректността и анализ на сложността, който на свой ред се разбива на анализ по време и анализ по памет. Накрая разискваме дали високата сложност е непременно лошо нещо.

## Съдържание



# 1 Алгоритъм

## 1.1 Опит за дефиниция

Понятието “алгоритъм” е първично. За него няма общоприета прецизна формална дефиниция, също както няма дефиниция на “множество”. Задълбочена дискусия на тема дефиниция на алгоритъм има в статията на Блас и Гуревич [BG03]. Въпреки отказа от строго формална дефиниция ще дадем интуитивно разяснение. Според Стив Скийна [Ski08],

> Алгоритъм е процедура, която решава дадена *задача*. Алгоритъм е идеята, която стои зад всяка смислена компютърна програма.
>
> За да бъде интересен, един алгоритъм трябва да решава задача, която е обща и добре формулирана. Всяка задача се определя от множеството на своите *примери* и от това, какъв трябва да бъде изходът върху всеки от тези примери.

## 1.2 Изчислителни задачи

Разликата между обща задача и конкретна задача е донякъде въпрос на мнение и интерпретация, защото всяка задача може да бъде обобщавана (докато стигнем до алгоритмично нерешима задача...), но с два примера ще покажем общоприетото разбиране за разликата между обща и конкретна задача.

- Да бъдат сумирани две конкретни естествени числа, примерно 7 и 5, е конкретна задача с фиксиран отговор 12. Да бъдат сумирани две произволни естествени числа $k$ и $m$ е по-обща задача. Да бъдат сумирани $n$ на брой естествени числа е още по-обща задача.
- Да бъдат сортирани две естествени числа, примерно 7 и 5, е конкретна задача с отговор векторът $(5, 7)$. Да бъдат сортирани две естествени числа $a_1$ и $a_2$ е по-обща задача, но не достатъчно обща. Нейният отговор е просто $(\min\{a_1, a_2\}, \max\{a_1, a_2\})$. Да бъдат сортирани $n$ на брой естествени числа $a_1, \ldots, a_n$ е обща задача.

Примерите (на английски, instances) на задачата за сортирането са всички $n$-елементни вектори от естествени числа, по всички допустими стойности на $n$, примерно $n \geqslant 1$. Очевидно става дума за безкрайно много примери (но *изброимо* безкрайно). При практически всички алгоритми, които ще разглеждаме, множеството от примерите на задачата е изброимо безкрайно. Затова въвеждаме термина *общ пример на изчислителна задача*, като общият пример е описание, от което можем да получим представа за всеки конкретен пример. Имайки предвид това, *изчислителна задача* е съвкупността от общ пример и самото задание: какво се иска да се направи. Терминът за последното е *изход* (на английски, *output*), а не отговор. Следователно, всяка изчислителна задача се определя напълно от описанието на общия пример и на изхода. Следният пример показва как ще описваме изчислителни задачи.

**Изчислителна задача** Числено Сортиране
**Общ пример:** естествени числа $a_1, a_2, \ldots, a_n$
**Изход:** пермутация $a'_1, a'_2, \ldots, a'_n$ на входа, такава че $a'_1 \leqslant a'_2 \leqslant \ldots \leqslant a'_n$

## 1.3 Алгоритъм и изчислителна задача

Изчислителна задача е нещо съвсем различно от алгоритъм. Изчислителната задача е *спецификацията*; това, което трябва да бъде реализирано. Можем да мислим за нея като за функция с домейн множеството от конкретните примери и кодомейн, който се задава еднозначно от изхода. В горния пример със Числено Сортиране, и домейнът, и кодомейнът са $\times_{i=1}^{n} \mathbb{N}$. Съществуват много алгоритми, които решават дадена задача. Всеки от тези алгоритми е *реализация* на въпросната функция[†]. Алгоритъмът е преобразувател на информация, той има *вход* и *изход*, като входът е общият пример[‡] на изчислителния проблем, а изходът е преработената информация – същото, което е изходът на изчислителната задача. Нагледно:


От ключово значение е това, че се интересуваме не просто от функцията, изобразяваща входа в изхода (описанието на изчислителната задача дефинира недвусмислено именно тази функция) а и от, както вече казахме, реализацията на функцията. Тоест, интересуваме от това, което е вътре в кутията.

[†]Забележете, че тези функции са повече от алгоритмите! Алгоритмите за безкрайно много, но *изброимо* безкрайно. Функциите, от друга страна, са *неизброимо* безкрайно: лесно се показва, че множеството от функции от вида $f : \mathbb{N} \to \{0, 1\}$ е неизброимо безкрайно, така че функциите от естествените числа в естествените числа също са неизброимо безкрайно. Следва, че възможните функции са повече от алгоритмите в същия смисъл, в който реалните числа са повече от естествените. Следователно, само безкрайно малко подмножество от тези функции имат алгоритми!

[‡]При всяко конкретно пускане на алгоритъма, входът е някой конкретен пример. Когато разглеждаме алгоритъма по принцип, входът е общият пример.

## 1.4 Елементарни инструкции

Реализацията на функцията е (крайна) последователност от елементарни инструкции. Това е съдържанието на кутията – последователност от елементарни инструкции. Какви инструкции са допустими? Няма общоприета схема за това, какви са елементарните инструкции, но е очевидно, че ако допуснем съвсем произволни инструкции, ще обезсмислим конструирането на алгоритми. Примерно, ако допуснем елементарна инструкция, която сортира $n$ числа за произволно $n$, въпросът за конструиране на сортиращ алгоритъм е "решен" – алгоритъмът се състои от тази единствена инструкция. Такова решение би било напълно безполезно на практика. Полезните, смислените елементарни инструкции са тези, които могат да бъдат реализирани *физически* така, че да се изпълняват за време, близко до единица (един машинен такт), върху реален компютърен процесор. Примерно,

> *събери числата, намиращи се в променливите* `a` *и* `b`*, и запиши резултата в* `a`

и

> *сравни променливите* `a` *и* `b` *и ако* `a` *е по-малко или равно на* `b`*, премини към изпълнение на инструкция номер* 45*, в противен случай премини към изпълнение на инструкция* 99

са смислени елементарни инструкции. С много примери нататък ще илюстрираме ясно какви инструкции и какво ниво на детайлизация имаме предвид. Засега ще кажем, че инструкциите се делят на императивни, условни и входно-изходни.

Смятаме, че информацията на нашия алгоритъм—а именно информацията на входа, информацията на изхода и междинната информация по време на работа на алгоритъма—е структурирана в променливи. Всяко късче информация е част от някаква променлива. Има съставни променливи, примерно масиви, и атомарни (елементарни) променливи, примерно целите числа. Това, разбира се, е абстракция по отношение на работата на истински компютър. Всеки истински компютър работи не директно с числа, а с *представяне*, или *кодиране*, на числа. Тоест, със стрингове над азбуката $\{0,1\}$, които може да кодират числа. За целите на този курс е много удобно да избегнем разглеждането на кодиранията и да смятаме, че на най-ниското ниво имаме обекти като числа, с които се работи директно. Елементарните типове данни са `boolean`, `integer`, `real`, `char`, `pointer` и `vertex` (на граф). В това разделение има голяма доза условност, примерно върховете на графа може да се идентифицират с естествени числа, булевите стойности да са 0 и не-нула и т.н.

Приемаме, че основните операции се случват в една стъпка независимо от големината на операндите. Примерно, събирането и умножаването на числа стават за една стъпка *независимо от големината на числата.* От практическа гледна точка, разбира се, това е нереалистично допускане по отношение на много големите числа. За всеки реален компютър има числа, които са достатъчно големи, за да не се побират (по-точно, тяхното представяне да не се побира) в една машинна дума. Нещо повече, всеки реален компютър може да работи само с крайно подмножество на безкрайното множество на целите числа; независимо от избраното кодиране на числата, само безкрайно малка част от всички тях са достъпни за него. Но за целите на курса ще приемаме, че всяко число може да се представи върху нашата абстрактна машина и че основните операции върху кои да е две числа стават за единица време. Това допускане ни позволява да се съсредоточим върху основните аспекти на алгоритмите, а и днешните компютри са толкова мощни, че допускането не е нереалистично за типично възникващите на практика данни.

И така, всеки алгоритъм е последователност от инструкции. Изпълнението става в дискретно време (стъпки). Както казахме, инструкциите са подредени и са краен брой, така че може да бъдет номерирани и в изложението нататък ще смятаме, че са номерирани. Започвайки от първата инструкция, във всеки следващ момент (стъпка) се изпълнява точно една инструкция[†]. Коя ще е следващата инструкция след инструкция `i` се определя така:

- Ако инструкция `i` е императивна или входно-изходна, но не е инструкция за край, след нея се изпълнява инструкция `i + 1`.
- Ако инструкция `i` е условна—`if ... then ... else ...` или начало на цикъл, `while ... do ...` или `for ... do ...`—тогава следващата инструкция, която ще се изпълни, зависи от конкретния алгоритъм и стойността на променливите в момента.
- Ако инструкция `i` е за край на алгоритъма, след нея не се изпълнява нищо. Има две възможности за това: тази инструкция да е последната в алгоритъма, без да е част от цикъл, или да бъде специална инструкция за прекратяване на работата, наречена примерно `exit` или `halt`.

[†]С други думи, няма паралелизъм.

По принцип не е невъзможно алгоритъмът да се състои в изпълнението на първата инструкция, последвано от втората, третата и така нататък до последната, но това би бил един твърде прост и скучен алгоритъм. Тъй като имаме условни инструкции и възможност за преминаване към изпълнение на произволна инструкция, можем да реализираме цикли. Има изискване за крайност на изпълнението: всеки алгоритъм трябва да завършва изпълнението си за краен брой стъпки върху *всеки* възможен вход. Вече обяснихме при какво условия завършва своята работа алгоритъмът. Има два начина алгоритъм да не завърши:

- Може да “зацикли”, което означава да влезе повторно в конфигурация, в която вече е бил. *Конфигурация* е съвкупността от стойностите на всички използвани променливи, тоест състоянието на паметта. Прост пример за фрагмент от алгоритъм, написан на синтаксиса на C, който гарантира зацикляне, е

  ```
  for (;;);
  ```

- Може, без да влиза никога повторно в конфигурация, в която вече е бил, да увеличава някои променливи неограничено, примерно

  ```
  for (i = 0, j = 1; i < 2; j ++);
  ```

  Теоретично говорейки, върху истински компютър вторият сценарий не може да се реализира, тъй като всеки истински компютър има само крайна памет и краен, макар и невъобразимо голям, брой възможни състояния, така че ако бъде оставен да работи достатъчно дълго, той неизбежно ще влезе в състояние, в което вече е бил. В посочения пример, паметта, която реализира променливата `j`, рано или късно ще се препълни и `j` пак ще стане единица. От практическа гледна точка вторият сценарий може да се реализира дори върху истински компютър, защото, ако времето за повторно влизане в дадена конфигурация е по-голямо от човешки живот или дори живота на звездите във Вселената, то спокойно можем да смятаме, че до повторение на конфигурация няма да се стигне.

## 2 Евклидовият алгоритъм

Най-старият известен нетривиален алгоритъм е Евклидовият алгоритъм за най-големия общ делител на две цели положителни числа. Оригиналната формулировка се намира в седмата книга от Елементи [HH08]. Алгоритъмът се съдържа в Задача 1 и Задача 2.

**Задача 1** (стр. 296 в [HH08])**.** *Дадени са две числа, като по-малкото бива изваждано последователно от по-голямото. Ако числото, което се получава накрая, никога не дели по-голямото*[†]*, и най-накрая остане единица, оригиналните числа са взаимно прости.*

**Аргументация**

*Нека по-малкото от две неравни числа AB и CD*[‡] *бива изваждано последователно от по-голямото и нека получаваното число никога не дели това преди него, докато накрая се получи единица; твърдя, че AB и CD са взаимно прости, тоест, само единицата ги дели и двете.*

*Защото, ако AB и CD не са взаимно прости, някакво число ги дели*[§]*. Нека ги дели някакво число и нека то да е E; нека CD, което дели BF, оставя някакво FA, по-малко от себе си.*

*Нека AF, което дели DG, оставя по-малко от себе си GC, и нека GC, което дели FH, оставя единица HA.*

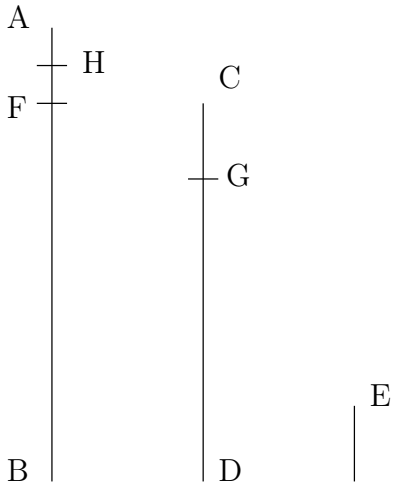
*Но тъй като E дели CD, и CD дели BF, то E дели BF.*

*Но то също така дели цялото BA; следователно, то дели и остатъка AF.*

*Но AF дели DG; затова E дели и DG.*

*Но то също така дели и цялото DC, затова дели и остатъка CG.*

*Но CG дели FH; затова E също дели и FH.*




[†] Има се предвид, в процеса на изваждане.

[‡] Древните гърци са мислели за числата като за дължини на отсечки.

[§] Древните гърци не са смятали единицата за число, а за нулата дори не са имали понятие.

*Но то дели и цялото FA, затова то още дели и остатъка, а именно единицата AH, въпреки че то*[†] *е число. А това е невъзможно.*

*Затова никое число не дели числата AB и CD; затова те са взаимно прости.* □

**Задача 2** (стр. 298 в [HH08]). *Дадени са две числа, които не са взаимно прости, да се намери техният най-голям общ делител.*

**Аргументация**

*Нека AB и CD са две дадени числа, които не са взаимно прости.*

*Иска се да се намери най-големият общ делител на AB, CD.*

*Ако CD дели AB—тогава то дели и себе си—CD е общ делител на AB и CD.*

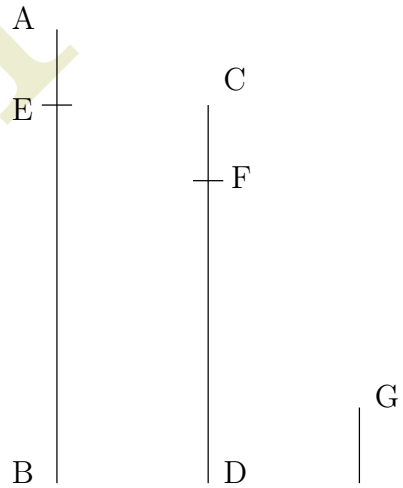
*И е очевидно, че то освен това е най-големият общ делител; защото никое число, по-голямо от CD, не дели CD.*

*Но ако CD не дели AB, тогава по-малкото от числата AB, CD, ако бъде изваждано последователно от по-голямото, то ще остане число, което дели това преди себе си.*

*Единица няма да остане; инак, AB, CD ще се окажат взаимно прости (виж Задача 1), което противоречи на хипотезата.*

*Затова ще остане число, което дели това преди него.*

*Нека CD, което дели BE, оставя EA, което е по-малко от него самото, което дели DF, оставяйки FC, което е по-малко от него самото*[‡]*, и нека CF дели AE.*




*Тъй като CF дели AE, и AE дели DF, следва, че CF дели DF.*

*Но то дели освен това себе си, затова дели цялото CD.*

*Но CD дели BE, следователно CF също така дели BE.*

*Но то*[§] *също така дели EA, затова то също така дели цялото BA.*

*Но то също така дели CD, затова CF дели AB, CD.*

*Затова CF е общ делител на AB, CD.*

*Сега аз казвам, че това число е най-голямото такова.*

*Защото, ако CF не е най-големият общ делител на AB, CD, някое число, по-голямо от CF ще дели числата AB, CD.*

*Нека едно такова число е G.*

*Понеже G дели CD, тъй като CD дели BE, G също така дели и BE.*

*Но то също дели и цялото BA; затова то дели и остатъка AE.*

*Но AE дели DF; затова и G дели DF.*

*Но то също дели и цялото DC; затова то дели и остатъка CF, тоест, по-голямото дели по-малкото: което е невъзможно.*

*Затова никое число, което е по-голямо от CF, не дели числата AB, CD.*

*Затова CF е най-големият общ делител на AB, CD.* □

Този текст ни звучи непривично заради особената нотация и конвенции. Очевидно по Евклидово време не са имали съвременната идея за индукция; той дори не ползва някаква форма на “и така нататък” при аргументацията, а разглежда трикратно прилагане на конструкцията на вземане на (това, което днес бихме нарекли) остатъци при деление и смята, че това е достатъчно убедително. В Задача 1, трикратното прилагане е:

$$\text{AB и CD дават остатък FA} \;\rightarrow\; \text{остатък GC} \rightarrow \text{остатък HA, равен на единица}$$

Въпреки необичайното за нас изразяване, тези две задачи от седмата книга на Евклид съдържат истинско описание на алгоритъм, напълно недвусмислено, с аргументация за привършване на процедурата и за коректност на получения резултат. Анализ на сложността по време, естествено, няма. В десетата книга на Евклид се дискутира същата идея, но върху числа-отсечки, които днес бихме нарекли “реални”; в Евклидовата терминология,

[†] Става дума отново за E.
[‡] Вече се има предвид от EA.
[§] CF

това са отсечки, чието отношение на дължините не е рационално число (*incommensurables* в английския превод). Изключително задълбочена дискусия за Евклидовия алгоритъм има във втория том от култовата книга на Доналд Кнут "The Art of Computer Programming" [Knu97].

Съвременната формулировка на Евклидовия алгоритъм ползва не последователни изваждания, а значително по-бързото изчисляване на остатък при деление (което може да се имплементира чрез изваждания-докато-е-възможно, но това би била много тромава от днешна гледна точка имплементация). Две възможни детайлни реализации са итеративната и рекурсивната, които ще даваме тук. Езикът, който ползваме за описание на двете разновидности, се нарича *псевдокод*. Това не е истински език за програмиране, а измислена от нас езикова конструкция, която е пределно изчистена от ненужни детайли (каквито неизбежно има в истински език за програмиране) и въпреки това е абсолютно прецизна и недвусмислена.

```
EUCLID, ITERATIVE(a, b ∈ ℕ, a > b)
1  while b > 0 do
2      t ← a
3      a ← b
4      b ← t mod b
5  return a
```

```
EUCLID, REC(a, b ∈ ℕ, a > b)
1  if b = 0
2      return a
3  else
4      return EUCLID, REC(b, a mod b)
```

## 3 Големина на входа на алгоритъм

*Големина на входа* (на английски *input size*) е ключово понятие за разбирането на сложността на алгоритмите, защото сложността е функция на размера на входа. Прецизното определение е следното.

**Определение 1** (прецизно определение, което няма да използваме)**.** Големина на входа *на алгоритъм, за конкретен вход, е броят символи, необходими за кодирането на този вход в избраната система за кодиране.*

Това определение е полезно за теория на сложността, която отчита и кодирането на данните. Тъй като вече приехме, че няма да разглеждаме кодирания, Определение 1 не върши работа, поне не за целите на този вид теория на изчислителната сложност. Терминът "големина на входа" ще дефинираме непрецизно, с голяма доза условност. Ако алгоритъмът е върху числен масив, примерно ако задачата е свързана със сортиране, то големината на входа е броят на числата в масива. Ако алгоритъмът е върху графи, големината на входа е сумата от броя на върховете и броя на ребрата. Това разбиране за големина на входа отговаря добре на модела, който вече приехме, в който числата са неделими елементи, а елементарните операции върху тях стават в единица време.

Практиката е показала, че това разбиране за големина на входа обикновено води до смислени резултати. Както ще видим в подсекцията за сложност по време, има и изключения.

## 4 Анализ на алгоритми

Да бъде *анализиран* даден алгоритъм означава да бъде доказано, че той е коректен, и ако е коректен, да бъде предвидено колко ресурси ползва, като ресурсите са основно време и памет. Първият аспект е свързан с понятието *ефективност*, а вторият, с понятието *ефикасност*. На български е прието тези два термина да се смятат за синоними, но на английски има значителна разлика между съответните *effective* и *efficient*. Въпросът дали даден алгоритъм е *effective* има отговор или Да, или Не; ако алгоритъмът върши работата, която е определена от заданието (изчислителната задача), то той е ефективен, неговата работа дава желания ефект. При ефикасността въпросът не е дали, а до колко. Ефикасността е свързана с ресурсите, които алгоритъмът ползва. Терминологичната разлика между "ефективен" и "ефикасен" е много полезна на практика и ние ще я спазваме. Алгоритъм A е по-ефикасен от алгоритъм B, ако ползва по-малко ресурси, примерно работи по-бързо. От друга страна, A няма как да е по-ефективен от B, ако и двата са коректни, защото и тогава и двата са ефективни.

Съществуват сложни изчислителни задачи, за които са приемливи алгоритми, които не винаги работят коректно, но при условие, че можем да ограничим количествено грешката. Примерно, ако броят на входовете, върху които алгоритъмът не работи коректно, е пренебрежимо малък спрямо броя на всички входове. В такъв случай ефективността не би била булева величина Да/Не, а истинско число. В този курс такива задачи и алгоритми няма да разглеждаме, така че ефективността за нас ще е именно булева.

И така, анализирайки един алгоритъм, първо ще доказваме, че е ефективен, тоест коректен, и след това ще изследваме колко е ефикасен, тоест какви ресурси ползва. Когато говорим за алгоритми за дадена задача, искаме алгоритъмът да е колко е възможно по-ефикасен (естествено, бивайки ефективен). Когато говорим за сложност на задачи, фактът, че за дадена задача няма ефикасен алгоритъм[†] може да бъде от голям теоретичен интерес, но може и да е полезен на практика, както ще стане ясно в Подсекция 4.4 на стр. 13.

## 4.1 Анализ на коректността

Анализът на коректността е по-важният вид анализ. Даденият алгоритъм $A$ решава някаква изчислителна задача $\Pi$. Както казахме вече, можем да мислим за $\Pi$ като за функция от множеството на конкретните примери в множеството на възможните изходи, а алгоритъмът $A$ е една възможна реализация на тази функция чрез някакви основни елементи-инструкции. Анализът на коректността е доказателство, че наистина $A$ реализира именно *тази* функция, а не някаква друга. Важна част от доказателството за коректност е доказателството, че алгоритъмът завършва работата си върху всеки вход[‡]. Последното често се пренебрегва, защото бива считано за очевидно. В много случаи наистина е очевидно, че даден алгоритъм винаги завършва, но не винаги е така. Може да бъде изключително трудно да се отговори дали даден алгоритъм винаги завършва. Нещо повече: *не съществува* алгоритъм, който по дадено кодиране на произволен друг алгоритъм плюс вход да определи дали той (другият) ще завърши работата си върху този вход, или не. Този важен резултат от зората на Компютърните науки е доказан от Алан Тюринг [Tur36], а задачата да бъде определено дали алгоритъм завършва се нарича Стоп задача, на английски Halting Problem. Лесно могат да бъдат дадени примери за това, че да отговорим дали даден алгоритъм винаги завършва или не, е същото като да решим дадена математическа задача; верността на математическо твърдение може да бъде кодирана в това, дали някакъв алгоритъм завършва винаги или не. Примерно, широко известната голяма теорема на Ферма́:

**Теорема 1** (Fermat). *Уравнението* $a^n + b^n = c^n$ *няма решения в цели положителни числа за* $n \geqslant 3$.

има свой алгоритмичен аналог – да докажем, че следният алгоритъм, написан на C, не завършва, е същото като да докажем, че теоремата на Ферма́ е вярна:

```
k = 3;
for (;;) {
  for(a = 1; a <= k; a ++)
   for(b = 1; b <= k; b ++)
    for(c = 1; c <= k; c ++)
     for(n = 3; n <= k; n ++)
      if (pow(a,n) + pow(b,n) == pow(c,n))
       exit();
  k++; }
```

Теоремата на Ферма́ е изключително трудна за доказване. Доказателството ѝ е било непреодолимо предизвикателство в продължение на стотици години. През 90те години на 20 век най-накрая беше дадено доказателство [Wil95], чието прочитане (с разбиране...) е във възможностите на много малко хора. Извод: само един аспект от доказателството за коректност на даден алгоритъм, а именно това, че алгоритъмът изобщо завършва, може да бъде изключително труден и да изисква дълбоки познания по математика.

Алгоритмите, които ще разглеждаме в този курс, ще бъдат значително по-прости и доказателствата за тяхното завършване по правило ще са тривиални. Основно ще се интересуваме от доказване на същинската коректност – че алгоритъмът реализира именно тази функция, която трябва. Ще ползваме главно две техники: доказателства за коректност чрез индукция, когато става дума за рекурсивни алгоритми, и чрез инварианта на цикъл, когато става дума за итеративни алгоритми. Сега ще дадем пример за доказателство за коректност чрез индукция, а техниката, използваща инварианта, ще разгледаме в следваща лекция.

[†]"Ефикасен алгоритъм" без повече пояснения означава такъв с полиномиална сложност, която ще въведем в следваща лекция.

[‡]В термините на функциите, да бъде доказано, че алгоритъмът $A$ завършва винаги, е същото като да бъде доказано, че функцията, реализирана от $A$, е тотална, а не частична. Ако върху даден пример на задачата $\Pi$, тоест вход на алгоритъма $A$, той не завършва, можем да кажем, че на този елемент от домейна на функцията съответстват нула елемента от кодомейна – нещо, което е възможно при частичните функции. За да е коректен алгоритъмът, функцията, реализирана от него, трябва да е тотална.

НАРАСТВАНЕ С ЕДИНИЦА($n$: естествено число)
```
1  if n = 0
2      return 1
3  else
4      if n mod 2 = 0
5          return n + 1
6      else
7          return 2 × НАРАСТВАНЕ С ЕДИНИЦА(⌊n/2⌋)
```

**Теорема 2.** *Алгоритъмът* НАРАСТВАНЕ С ЕДИНИЦА *връща* $n+1$ *за всеки вход* $n$.

**Доказателство:**
С индукция по $n$.

**База.** $n = 0$. В такъв случай булевият израз на ред 1 е истина и алгоритъмът връща $0+1 = 1$ чрез присвояването на ред 2. ✓

**Индуктивно Предположение.** Нека за всяко $m < n$ алгоритъмът връща $m+1$.

**Индуктивна стъпка.** Да разгледаме работата на НАРАСТВАНЕ С ЕДИНИЦА върху вход $n \geqslant 1$. Първо да допуснем, че $n$ е четно. Тогава булевият израз на ред 4 е истина, следователно ред 5 се изпълнява и алгоритъмът връща $n+1$. ✓

Сага да допуснем, че $n$ е нечетно. Булевият израз на ред 4 е лъжа, следователно ред 7 се изпълнява и алгоритъмът връща $2 \times$ НАРАСТВАНЕ С ЕДИНИЦА $(\lfloor \frac{n}{2} \rfloor)$. Тъй като $\lfloor \frac{n}{2} \rfloor < n$, може да приложем индуктивното предположение. Съгласно него, НАРАСТВАНЕ С ЕДИНИЦА $(\lfloor \frac{n}{2} \rfloor)$ връща $\lfloor \frac{n}{2} \rfloor + 1$. Тъй като $n$ е нечетно, $n = 2k+1$ за някои $k \in \mathbb{N}$, и изходът е

$$2 \times \left( \left\lfloor \frac{2k+1}{2} \right\rfloor + 1 \right) = 2 \times \left( \left\lfloor k + \frac{1}{2} \right\rfloor + 1 \right) = 2 \times (k+1) = 2k+2 = 2n+1 \qquad \checkmark$$

## 4.2 Анализ на сложността

Под "сложност на алгоритъм" обикновено се разбира колко бързо (или бавно...) работи алгоритъмът. "Сложен" в този смисъл е "бавен". Има и други видове сложност, които дават представа за качествата на алгоритъма, като основна сред тях е свързана с употребата на памет. Затова ще говорим за сложност по време и сложност по памет (на английски съответно *time complexity* и *space complexity*), които ще определим сега. Само ще споменем, че има и други видове сложност, примерно при разпределено изчисление (*distributed computation*) се говори за комуникационна сложност (*communication complexity*), измерваща комуникацията между независими изчисляващи агенти.

### 4.2.1 Въведение в сложността по време

Да разгледаме произволен прост алгоритъм, примерно сортиращия алгоритъм INSERTION SORT.

INSERTION SORT($A[1,2,\ldots n]$: масив от естествени числа)
```
1  for i ← 2 to n
2      key ← A[i]
3      j ← i − 1
4      while j > 0 and A[j] > key do
5          A[j + 1] ← A[j]
6          j ← j − 1
7      A[j + 1] ← key
```

Колко бързо работи той? От практическа гледна точка, този въпрос е за физическото време, което отнема пускането на алгоритъма, и по-точно на някаква негова практическа реализация, върху истински компютър. От тази гледна точка трябва да отчитаме следните фактори:

- За какъв вход става дума. На свой ред, този фактор има две компоненти:

  - Колко числа има във входа. Интуитивно е ясно, че колкото по-голямо е $n$ (големината на входа), толкова по-бавно работи алгоритъмът.
  - При една и съща големина, какви са конкретните стойности на числата. Лесно се забелязва, че Insertion Sort ще работи по-бързо, ако входът вече е сортиран, примерно $[1, 2, 3, 4]$. Нататък ще видим, че този алгоритъм работи най-бавно, когато входът е сортиран обратно, примерно $[4, 3, 2, 1]$.

- Каква е конкретната програмна реализация. По дефиницията на Скийна, алгоритъмът е идеята зад някаква програма. Но истинските програми са повече от идеите зад тях, те имат конкретна реализация на конкретен език за програмиране. Очевидно някои реализации ще са по-бързи от други, защото са по-грамотно написани. Компилаторът не може да компенсира неграмотно писане на софтуер.
- Какъв е компилаторът.
- Каква е виртуалната машина, ако има такава.
- Каква е операционната система.
- Каква е компютърната архитектура.

Да се даде *точен* отговор колко бързо би работила конкретна програма—примерно, с точност до милисекунда—реализираща Insertion Sort, на конкретен език, върху конкретен компютър, е практически невъзможно дори за фиксиран вход. А да се даде този отговор за всеки възможен вход е немислимо[†]. Толкова прецизен отговор не е и необходим.

За да може да правим смислени изводи за бързината на алгоритми, правим множество опростявания. Първото от тях е, че се фокусираме именно върху алгоритмите, а не върху програмните им реализации. Второто е допускането, че всички елементарни инструкции се изпълняват за единица време[‡], така че времето за изпълнение на алгоритъма върху някакъв вход се измерва с броя на елементарните инструкции, които се изпълняват преди завършването.

Разликата между бързодействието на две програми за една и съща задача, които реализират два различни алгоритъма, по правило са много по-драстични от разликите в бързодействието на две програми, реализиращи един и същи алгоритъм. **По отношение на сложните, нетривиални задачи, печелившата стратегия за бързодействието е подобрение на алгоритъма.** Печалбата от по-бързия алгоритъм е толкова по-видима, колкото по-голям е входът. Като пример да разгледаме задачата Сортиране. Алгоритъмът Insertion Sort, както ще видим в следваща лекция, в най-лошия случай работи във време, пропорционално на квадрата на размера на входа. Тоест, във време, пропорционално на $n^2$. В следващи лекции ще разгледаме по-бързи алгоритми за сортиране, работещи във време, пропорционално на произведението $n \times \log n$. Да разгледаме стойностите на $n^2$ и $n \times \log n$ за няколко различни $n$. За целите на този пример логаритъмът е с основа 10.

| $n$ | 10 | 100 | 1 000 | 100 000 | 1 000 000 | 100 000 000 |
|---|---|---|---|---|---|---|
| $n^2$ | 100 | 10 000 | 1 000 000 | 10 000 000 000 | 1 000 000 000 000 | 10 000 000 000 000 000 |
| $n \log n$ | 10 | 200 | 3 000 | 500 000 | 6 000 000 | 800 000 000 |

Разбира се, числата в тази таблица не са точният брой елементарни инструкции; точният брой, както казахме, е *пропорционален* на тези числа. Но на практика тези коефициенти на пропорционалност не са големи. И така, при размер на входа от сто милиона, огромната разлика между 10 000 000 000 000 000 и 800 000 000—**осем десетични порядъка**—води до това, че програмата на по-бързия алгоритъм е **смазващо** по-бърза[§] от тази на Insertion Sort независимо от неща като използван компилатор, ниво на оптимизация на компилирането, операционна система, модел на процесора, програмистки трикове за подобряване на скоростта и така нататък. Предимството, което дава по-бързият алгоритъм, по правило доминира с много над предимството на по-бързата компютърна технология, било хардуер, било софтуер.

Съществуват много по-екстремни примери за предимствата, които дават бързите алгоритми. За множество интересни и важни задачи наивният алгоритъм—който може да бъде предложен от най-общи съображения—се изпълнява в най-лошия случай за брой елементарни инструкции, който е пропорционален на експоненциална в $n$ функция, да кажем $2^n$, докато за същата задача има по-съвършени алгоритми, чието изпълнение като брой

[†] Съществуват специализирани компютри, изпълняващи критични дейности в реално време, при които наистина са необходими твърди гаранции за бързодействието на програмите, но това излиза далече извън границите на материала в този курс.

[‡] Инструкциите на истински процесор се изпълняват за различен брой тактове

[§] При нормално грамотна реализация на двата алгоритъма.

елементарни инструкции е пропорционален на, примерно, $n^2$. Разликата между нарастването на $2^n$ и $n^2$ е много, много по-драстична от разликата между $n^2$ и $n \lg n$. Както ще стане ясно от някоя следваща лекция[†], $2^{200}$ като брой инструкции е **далече отвъд възможностите на всеки истински компютър**, както в момента, така и в бъдеще. Това заслужава да бъде повторено: в реалния свят, програма, която трябва да изпълни $2^{200}$ стъпки, няма да завърши работата си **никога**, като това не зависи от използваната технология; ограничението идва от фундаментални принципи на физиката и не може да бъде заобиколено чрез никакви технологични иновации и подобрения.

Следователно, опростеният модел, в който първо разглеждаме само алгоритъма, а не програмната му реализация, и второ всяка елементарна инструкция се изпълнява за една стъпка, е полезен от практическа гледна точка. Ако в този модел алгоритъм A е по-бърз от алгоритъм B за една и съща задача, най-вероятно програмната реализация на A ще е по-бърза от тази на B, като разликата ще е толкова по-очевидна, колкото по-голям е входът.

### 4.2.2 Определение на сложността по време

Сложността по време е функция на големината на входа. Но има много входове с една и съща големина. По-лошо, в нашия опростен модел, в който всяко число има един и същи размер (единица), има безброй много входове за всяка големина на входа. Да разгледаме отново Сортиране. Големината на входа $n$ може да е всяко естествено число. Множеството от входовете с големина 1 е $\mathbb{N}$. Множеството от входовете с големина 2 е $\mathbb{N} \times \mathbb{N}$. Множеството от входовете с големина 3 е $\mathbb{N} \times \mathbb{N} \times \mathbb{N}$. И така нататък. Множеството от входовете е изброимо безкрайно за всяко $n$.

Следното съображение ни позволява да разглеждаме само краен брой входове. Съображението е по отношение на Сортиране, но лесно може да бъде пренесено и при други изчислителни задачи. Разглеждаме само такова сортиране, което се базира на директно сравнение на числата (които биват сортирани)[‡]. Лесно се вижда, че в такъв случай сортирането на следните входове

$$[2, 1, 3]$$
$$[2000, 1000, 3000]$$
$$[2 \times 10^9, 1, 10^{10^{10}}]$$

ще протече по един и същи начин, и то в много силен смисъл. Всеки сортиращ алгоритъм, базиран на директни сравнения, ще завърши работата си в същия брой стъпки за всеки от тези три входа, като на всяка стъпка ще изпълнява една и съща инструкция. Разбира се, това е при не много реалистичното допускане, че $10^{10^{10}}$ има размер единица и действията върху него стават за единица време.

Сега ще дефинираме неразличими по отношение на бързодействието входове. За всяка големина на входа $n$, за всеки два входа $X$ и $Y$ с големина $n$, нека $X$ и $Y$ са в релация $R$ тогава и само тогава, когато имат един и същи брой $m$ различни елементи[§], $1 \leqslant m \leqslant n$, и $k$-ият по големина елемент във всеки от тях е на едни и същи позиции, за $1 \leqslant k \leqslant m$. Примерно, всеки два от горните три входа са в тази релация. Тривиално се показва, че $R$ е релация на еквиваленост. Нейните класове на еквивалентост са само краен брой за всяко $n$. Интересен въпрос, на който сега ще отговорим, е колко точно са тези класове на еквивалентност. Ако не допускаме повтаряне на елементи, то класовете на еквивалентост са $n!$, защото всеки клас се определя от това на коя позиция е първият елемент, на коя е вторият, и така нататък до $n$-ия. В по-общия случай, когато допускаме повтаряне на елементи, нека броят на уникалните елементи е $m$, където $1 \leqslant m \leqslant n$. Броят на начините числата $a_1, a_2, \ldots, a_n$ да бъдат групирани в точно $m$ групи, във всяка от които числата са равни, а числата от две различни групи са различни, е $\left\{ {n \atop m} \right\}$. Да си припомним, че $\left\{ {n \atop m} \right\}$ за $1 \leqslant m \leqslant n$ е броят на разбиванията на $n$-елементно множество на точно $m$ подмножества. Примерно, $\left\{ {4 \atop 2} \right\} = 7$, и действително има точно 7 начина $a_1$, $a_2$, $a_3$ и $a_4$ да имат точно две

[†] Става дума за принципа на Ландауер [Lan61], който налага долна граница за енергията, необходима за обръщането на един бит от нула в единица или обратно.

[‡] Съществуват и други възможности за сортиране, ако са известни някакви ограничения за входа. Примерно, ако сортираме само нули и единици, може просто да преброим колко са нулите.

[§] Тук разглеждаме общия случай, в който във входа може да има повтарящи се елементи. Ако елементите на входа са уникални, то $m = n$.

различни големини:

$a_1, a_2$ и $a_3$ с една и съща големина, $a_4$ с друга големина
$a_1, a_2$ и $a_4$ с една и съща големина, $a_3$ с друга големина
$a_1, a_3$ и $a_4$ с една и съща големина, $a_2$ с друга големина
$a_2, a_3$ и $a_3$ с една и съща големина, $a_1$ с друга големина
$a_1$ и $a_2$ с една и съща големина, $a_3$ и $a_4$ с друга големина
$a_1$ и $a_3$ с една и съща големина, $a_2$ и $a_4$ с друга големина
$a_1$ и $a_4$ с една и съща големина, $a_2$ и $a_3$ с друга големина

Броят на класовете на класовете на еквивалентност на $R$ за дадено $m$ е $m!\left\{ {n \atop m} \right\}$. Примерно, при $n = 4$ и $m = 2$, има точно 14 начина за подредбата на различните големини на числата:

$a_1, a_2$ и $a_3$ с една и съща големина, $a_4$ с друга големина, и $a_1, a_2, a_3 < a_4$
$a_1, a_2$ и $a_3$ с една и съща големина, $a_4$ с друга големина, и $a_1, a_2, a_3 > a_4$
$a_1, a_2$ и $a_4$ с една и съща големина, $a_3$ с друга големина, и $a_1, a_2, a_4 < a_3$
$a_1, a_2$ и $a_4$ с една и съща големина, $a_3$ с друга големина, и $a_1, a_2, a_4 > a_3$
$a_1, a_3$ и $a_4$ с една и съща големина, $a_2$ с друга големина, и $a_1, a_3, a_4 < a_2$
$a_1, a_3$ и $a_4$ с една и съща големина, $a_2$ с друга големина, и $a_1, a_3, a_4 > a_2$
$a_2, a_3$ и $a_3$ с една и съща големина, $a_1$ с друга големина, и $a_2, a_3, a_4 < a_1$
$a_2, a_3$ и $a_3$ с една и съща големина, $a_1$ с друга големина, и $a_2, a_3, a_4 > a_1$
$a_1$ и $a_2$ с една и съща големина, $a_3$ и $a_4$ с друга големина, и $a_1, a_2 < a_3, a_4$
$a_1$ и $a_2$ с една и съща големина, $a_3$ и $a_4$ с друга големина, и $a_1, a_2 > a_3, a_4$
$a_1$ и $a_3$ с една и съща големина, $a_2$ и $a_4$ с друга големина, и $a_1, a_3 < a_2, a_4$
$a_1$ и $a_3$ с една и съща големина, $a_2$ и $a_4$ с друга големина, и $a_1, a_3 > a_2, a_4$
$a_1$ и $a_4$ с една и съща големина, $a_2$ и $a_3$ с друга големина, и $a_1, a_4 < a_2, a_3$
$a_1$ и $a_4$ с една и съща големина, $a_2$ и $a_3$ с друга големина, и $a_1, a_4 > a_2, a_3$

Броят на класовете на еквивалентост на $R$ за дадено $n$ е

$$\sum_{m=1}^{n} m!\left\{ {n \atop m} \right\}$$

Да резюмираме. В нашия модел, в който всички числа са с големина единица и елементарните операции върху всяко число стават за единица време, по отношение на произволен сортиращ алгоритъм $A$, базиран на директни сравнения, множеството от входовете $\times_{i=1}^{n} \mathbb{N}$ се разбива на $\sum_{m=1}^{n} m!\left\{ {n \atop m} \right\}$ класа, като за всеки два входа $X$ и $Y$ от един и същи клас, $A(X)$ и $A(Y)$ извършва в един и същи брой стъпки. От гледна точка на скоростта на изпълнение, $X$ и $Y$ са *неразличими*.

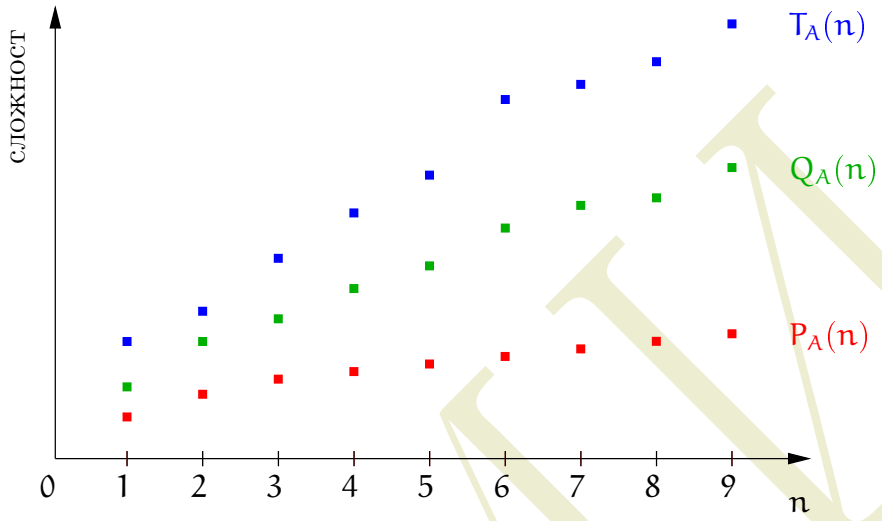
Очевидно за други изчислителни задачи разбиването на (изброимо) безкрайното множество от входовете на краен брой аналогични класове на еквивалентост става може да е по друг начин, но за всяка задача има такова разбиване, като за всеки два входа $X$ и $Y$ от един и същи клас, $A(X)$ и $A(Y)$ извършва в един и същи брой стъпки за всеки алгоритъм $A$ за нея. Това ни позволява, когато разглеждаме сложността по време, да смятаме, че входовете от дадена големина са само краен брой.

**Определение 2** (сложност по време)**.** *Нека $\Pi$ е изчислителна задача и $A$ е алгоритъм за нея. За всяка големина на входа $n \in \mathbb{N}^+$, нека $\mathcal{I}(n)$ е крайното множество от различните входове с големина $n$ и за всеки вход $\kappa$ нека $f(\kappa)$ е броят стъпки, които се изпълняват от $A$ върху него. Тогава* сложността по време на $A$ в най-лошия случай *е*

$$T_A(n) = \max\{f(\kappa) \mid \kappa \in \mathcal{I}(n)\}$$

сложността по време на $A$ в най-добрия случай *е*

$$P_A(n) = \min\{f(\kappa) \mid \kappa \in \mathcal{I}(n)\}$$




Фигура 1: Сложност по време в най-лошия случай $T_A(n)$, средна сложност $Q_A(n)$ и сложност по време в най-добрия случай $P_A(n)$. Ординатата е сложността като брой стъпки.

*и* средната сложност по време на $A$ *е*

$$Q_A(n) = \frac{1}{|\mathcal{I}(n)|} \sum_{\kappa \in \mathcal{I}(n)} f(\kappa)$$

Забележка: използваната нотацията, примерно $T_A(n)$ и т.н., не е общоприета. Фигура 1 показва нагледно трите вида сложност. Средната сложност винаги е между сложността в най-лошия и сложността в най-добрия случай. По правило функциите на сложността са строго нарастващи, защото (по правило) един и същи алгоритъм работи по-бавно върху по-голям вход, но има и изключения, както ще видим надолу в примера с Евклидовия алгоритъм.

На практика най-често използвана е сложността по време в най-лошия случай (*worst-case time complexity* на английски), по две основни причини. Първо, изследването на сложността в най-лошия случай е много по-лесно от изследването на средната сложност, в което ще се убедим в следваща лекция при анализа на QUICKSORT. Анализирането на средната сложност изисква значително по-задълбочени математически познания и значително по-сложни техники, дори при неявното допускане в определението на $Q_A(n)$, че всички входове с дадена големина са еднакво вероятни. Втората причина е, че резултатът за най-лошия случай е твърда гаранция, че по-лошо не може да бъде.

Сложността в най-добрия случай не се ползва, тъй като практически всеки важен алгоритъм можа да бъде подобрен като скорост за само един вход (по един такъв вход за всяка големина). Примерно, всеки алгоритъм $A$ за задачата ХАМИЛТОНОВ ПЪТ, която ще дефинираме нататък, може да бъде направен много бърз върху един конкретен вход (за всяка големина). По-точно, независимо как точно работи $A$, той може да бъде модифициран, като в самото му начало се добави тестване на една конкретна пермутация на върховете, дали задава хамилтонов път. Ако се окаже, че тя задава хамилтонов път, то отговорът на въпроса дали има хамилтонов път очевидно е утвърдителен и прекратяваме изпълнението. Ако се окаже, че тя не задава хамилтонов път, от това не следва, че няма такъв, и продължаваме с това, което $A$ прави. С други думи, към всеки алоритъм, дори най-бавния, може да се пришие напълно изкуствено тестване на някакъв конкретен вход. Този трик би подобрил изкуствено сложността в най-добрия случай до теоретично оптималната, без това да ни казва нищо съществено за това, колко е бърз $A$. В примера с ХАМИЛТОНОВ ПЪТ, според преобладаващото мнение, всеки алгоритъм за тази задача е с експоненциална сложност както в най-лошия, така и в средния случай, но чрез споменатия трик може да направим кой да е алгоритъм за тази задача да работи в линейно време в най-добрия случай. Такова изкуствено увеличаване на бързодействието чрез подобрение върху само един вход е напълно безполезно.

### 4.2.3 Анализ на сложността по памет

Сложността по памет на даден алгоритъм $A$ върху даден вход е броят на елементите памет, които $A$ ползва, без да броим паметта, в която се разполага входа. С други думи, гледаме само работната памет на алгоритъма.

Сложност по памет в най-лошия, средния и най-добрия случай се дефинира по начин, аналогичен на сложността по време.

Една значителна принципна разлика между сложността по време и сложността по памет е, че сложността по време (било в най-лошия случай, било средната) най-често е строго растяща функция на големината на входа, докато по отношение на сложността по памет е напълно възможно даден алгоритъм да ползва само константна работна памет, за всяка големина на входа. Алгоритми, които ползват константна работна памет, се наричат *in-place*.

## 4.3 Проблеми със сложността по време, до които води нашия модел

Да разгледаме отново Евклидовия в модерната му формулировка, независимо дали рекурсивната или итеративната. Да се опитаме да направим груб анализ на сложността му по време в най-лошия случай. Очевидно, за някои двойки входни числа, а именно тези, за които $\mathtt{a}$ е кратно на $\mathtt{b}$, алгоритъмът ще завършва много бързо, за не повече от 5-6 елементарни инструкции. За други двойки входни числа, алгоритъмът ще работи много повече. От [Knu97] става ясно, че за за всяко цяло положително число $\mathtt{m}$, колкото и голямо да е, има двойка $(\mathtt{a}, \mathtt{b})$, такава че Евклидовият алгоритъм с този вход работи за повече от $\mathtt{m}$ стъпки. Но в нашия опростен модел, всички двойки $(\mathtt{a}, \mathtt{b})$ имат една и съща големина, а именно 2, защото се състоят от две числа, а ние приехме, че всяко число има големина единица. Тогава всеки вход има големина точно 2. Следва, че функциите на сложността, които имат аргумент-големината на входа, не могат да бъдат дефинирани смислено, защото имаме само една големина на вход, а именно 2. За тази големина имаме входове, за които алгоритъмът работи в повече стъпки от всяко предварително избрано число. Излиза, че сложността по време е безкрайност ...

Този парадоксален извод се дължи само на допускането, че всяко число има големина единица. Ако приемем обаче по-сложния модел, в който всяко число има големина, пропорционална на логаритъм от него при основа 2[†], парадоксът изчезва. Тогава Евклидовият алгоритъм работи във време, пропорционално на квадрата на големината на входа (вж. [Knu97]).

Моделите налагат някакви опростявания, иначе нямаше да са модели. Доколкото тези опростявания са смислени и полезни на практика, съответните модели също са смислени и полезни. Но в случаите, в които опростяванията и допусканията водят до безсмислени и безполезни резултати, съответните модели стават неизползваеми и трябва да се търсят други модели, по-сложни и детайлни. Така че моделът, в който всяко число има големина единица, не е имманентно лош или сбъркан, просто в някои случаи е полезен (примерно, сортиращите алгоритми), в други, не е (примерно, Евклидовият алгоритъм).

Аналогично, пътната карта на България е модел на истинската пътна мрежа (защото съдържа само най-важната информация). Пътната карта, в която градовете са отбелязани с точки, е много полезна в някои случаи и напълно безполезна в други случаи. Ако искаме са стигнем с кола от София до Варна, примерно, и не знаем пътя, пътната карта, в която Варна е точка, е полезна до момента, в който влезем във Варна. За отиването до конкретен адрес там, моделът, в който Варна е точка, вече не върши работа. Но това не означава, че пътната карта е излишна. Просто в някои случаи тя е полезен модел, в други, не. По същия начин, моделът, в който числата имат големина единица, е полезен в някои случаи и безполезен в други.

## 4.4 Винаги ли е лоша високата сложност

В заключение ще разискваме доколко бавно е синоним на лошо в света на алгоритмите. Наистина, ние търсим колкото е възможно по-бързи алгоритми и свикваме да мислим, че бавен алгоритъм е лош алгоритъм, а бърз алгоритъм е добър алгоритъм. Както ще стане ясно в следващи лекции, редица полезни изчислителни задачи нямат бързи алгоритми (при определени допускания ...). Този феномен на неизбежна висока сложност по време е силно негативен, ако смятаме, че бърз=добър и бавен=лош; високата неизбежна сложност е нещо като проклятие.

Има обаче друга гледна точка, от която високата неизбежна сложност е благословия. Цялата криптография с отворен ключ е базирана на хипотезата (недоказана засега), че има задачи, които имат висока неизбежна сложност, докато техните обратни задачи (става дума за обръщане на функцията, дефинирана от задачата) са бързо решими. Криптографията с отворен ключ има грамадно значение за (почти) сигурното комуникиране в несигурна, публично достъпна среда. Ако не съществуваше феномена на високата неизбежна алгоритмична сложност, нямаше да е възможна криптография с отворен ключ[‡] и услуги като онлайн банкиране щяха да са невъзможни.

[†] $\log_2 \mathtt{n}$ е горе-долу броят на битовете, необходими за записването на $\mathtt{n}$ в двоична позиционна бройна система.

[‡] вж. http://www.cs.duke.edu/courses/fall09/cps296.1/CACM-Fortnow.pdf

# Литература

[BG03] Andreas Blass and Yuri Gurevich. Algorithms: a Quest for Absolute Definitions. *Bulletin of the European Association for Theoretical Computer Science*, 2003. Достъпна онлайн на http://research.microsoft.com/en-us/um/people/gurevich/Opera/164.pdf.

[HH08] T.L. Heath and J.L. Heiberg. *Books 3-9.* The Thirteen Books of Euclid's Elements. The University Press, 1908. Достъпна онлайн на http://www.wilbourhall.org/pdfs/Heath_Euclid_II.pdf.

[Knu97] Donald E. Knuth. *The Art of Computer Programming, Volume 2 (3rd Ed.): Seminumerical Algorithms.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[Lan61] Rolf Landauer. Irreversibility and Heat Generation in the Computing Process. *IBM Journal of Research and Development*, 5:183–191, 1961.

[Ski08] Steven S. Skiena. *The Algorithm Design Manual.* Springer Publishing Company, Incorporated, 2nd edition, 2008.

[Tur36] Alan M. Turing. On Computable Numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 42:230–265, 1936. Достъпна онлайн на http://www.dna.caltech.edu/courses/cs129/caltech_restricted/Turing_1936_IBID.pdf.

[Wil95] Andrew John Wiles. Modular Elliptic Curves and Fermat's Last Theorem. *ANNALS OF MATH*, 141:141, 1995. Достъпна онлайн на http://citeseer.ist.psu.edu/viewdoc/download;jsessionid=081B8BC551F2A8589C87BA556DCB9096?doi=10.1.1.152.9137&rep=rep1&type=pdf.