Глава 1

Формални модели

Както вече споменахме, понятието *алгоритъм* е неформално. В тази глава ще се спрем на възможностите за формализация на това понятие и на свързаните с него – *задача* и *решение*. Така ще получим възможност за изследване с математически апарат на формалните модели на тези понятия. В частност – ще въведем понятието *сложност на алгоритъм* в различните формални модели и ще покажем връзката между сложностите на един и същ алгоритъм в различните модели.

1.1 Масови задачи и алгоритми

В тълковния *Речник на българския език* (т. 5, Издателство на БАН, 1987) намираме следното тълкуване на думата *задача* (в математически смисъл): „Упражнение по математика, физика и др., което се разрешава чрез разсъждения и изчисления“. Без преувеличение може да се каже, че същността на математиката е *решаването* на задачи. В Доклада си пред Втория международен конгрес на математиците в Париж, 6-12 август 1900г., Давид Хилберт посочва, че състоянието на всяка наука до голяма степен зависи от формулирането на сериозни задачи и от развитието на средствата за тяхното решаване. „Всяка научна област е жизнеспособна“, твърди Хилберт, „докато в нея има излишък от нови задачи. Недостигът от нови задачи означава отмиране или прекратяване на самостоятелното развитие.“ Друг от корифеите на съвременната математика, ненадминатият „решавач на задачи“ Дьорд Пойя пише: „Да имаме задача означава да търсим съзнателно някое действие, годно за постигането на една ясно схващана, но не и непосредствено достижима цел. Да се реши една задача означава да се намери това действие.“

В училищните курсове по математика и информатика, както и в изучаваните в университета дисциплини, читателят се е сблъскал с много задачи и техните решения. Тук ще обърнем внимание на една важна особеност на математическите задачи, като въведем неформалното понятие *масова задача*.

Нека е зададено произволно множество от математически обекти, с добре дефинирани свойства, и съвкупност от операции между обек-

ти, даващи в резултат обект от същото множество. Под масова задача ще разбираме всяка достатъчно ясна цел, състояща се в намиране (с последователно прилагане на допустими операции) на някакви обекти $\tilde{y} = (y_1, y_2, \ldots)$ от множеството, с предварително определени свойства, започвайки от зададени обекти $\tilde{x} = (x_1, x_2, \ldots)$, също с предварително определени свойства. Ще обърнем внимание, че под решение на дадена задача в математиката разбираме не само **крайния резултат** $\tilde{y} = (y_1, y_2, \ldots)$, но **и последователността от операции**, с които сме го намерили.

Стойностите на всеки от параметрите $\tilde{x} = (x_1, x_2, \ldots)$ са елементи на някакви, по-големи или по-малки, подмножества на множеството от обекти. С всяко стесняване на областта от стойности на някой (или някои) от параметрите (включително и ограничаването им до единствено възможно значение), получаваме нова масова задача, която наричаме *екземпляр* на първата масова задача. При стесняване на всички параметри до единствени значения ще стигнем до масова задача, която съвпада с единствения си екземпляр.

Ето някои примери на масови задачи:

Пример 1. „*Дадени са елементите* a, b *и* c *на полето* F. *Да се намерят всички такива елементи* x *на полето* F, *че* $ax^2 + bx + c = 0$.“ и „*Дадени са елементите* a, b, c *и* x *на полето* F. *Да се провери дали* $ax^2 + bx + c = 0$.“
са масови задачи, а
„*Да се намерят всички такива елементи* x *на полето* $GF(3)$, *че* $x^2+2 = 0$.“ и „*Да се провери, че за* $x = 2$ *в полето* $GF(3)$ *е в сила* $x^2 + 2 = 0$.“
са екземпляри на тези масови задачи.

Пример 2. „*Дадени са естествените числа* i *и* j. *Да се намери НОД на* i *и* j.“ и „*Дадени са естествените числа* i, j *и* $k > 0$. *Да се провери дали* k *е НОД на* i *и* j.“
са масови задачи, а
„*Да се намери НОД на 12 и 30*.“ и „*Да се провери дали 6 е НОД на 12 и 30*.“
са екземпляри на тези масови задачи.

Пример 3. „*Дадени са множество от елементи (линейни и ъглови) на равнинен (планиметричен) обект. Да се построи този обект само с линийка и пергел*.“
е масова задача. Задачата
„*Да се построи триъгълник по зададени две страни* a, b *и ъгъл* α *между тях*.“
е неин екземпляр и също е масова задача. При задаване на конкретни

две страни и ъгъл на търсения триъгълник, получаваме екземпляр на тази масова задача.

Масовите задачи в края на всеки от трите примера съвпадат с единствения си екземпляр, тъй като за всички параметри са фиксирани единствени стойности.

В първия пример множеството от обекти е съставено от елементите на някакво поле, във втория пример – от естествените числа, а в третия – от геометричните обекти в равнината: точки, прави, ъгли, отсечки и т.н. Операциите в първите два примера са аритметичните и операциите за сравняване, а в третия – изпълнимите с линийка и пергел: начертаване на (част от) права, начертаване на окръжност, определяне пресечната точка на два обекта и т.н.

В първите два примера са дадени двойки масови задачи, които твърде много си приличат, но имат и съществени различия. Д. Пойя нарича първите *задачи за намиране* (на решение, б.а.), а вторите - *задачи за доказателство*. Поради честата употреба на термина „доказателство“, могат да се посочат масови задачи за намиране на решение, във формулировката на които се среща фразата „докажете, че ...“. За да избегнем недоразуменията, ще наричаме задачите от втория тип *задачи за проверка* (на решение).

Не е трудно да се забележи, че в двойка подобни масови задачи – едната за намиране, а другата за проверка на решение – тази за намиране на решение, обикновено, е по-трудна. Така например, за да проверим дали едно число x е решение на уравнението $ax^2+bx+c=0$ е достатъчно да извършим прости пресмятания и едно сравнение. В същото време, ако не знаем формулата за намиране корените на квадратното уравнение, решаването на съответната задача за намиране на решение може да е сериозен проблем.

И така, когато теоретичният анализ на задача за намиране на решение е затруднен, ще я заменяме със съответната ѝ задача за проверка на решение. **Ако една задача за проверка на решение е алгоритмично трудна, тогава аналогичната ѝ задача за намиране на решение, вероятно, ще е още по-трудна.**

Не така стоят нещата с двете задачи от Пример 2. Да се провери дали зададено число k е най-голям общ делител на две зададени числа i и j е почти толкова трудно, колкото да се намери това число. Каква е причината? Ако сравним масовите задачи от Пример 1 с тези от Пример 2, ще забележим една разлика, важна за класифицирането на задачите. В Пример 2 се говори не просто за общ делител на две естествени числа, а за най-големия им общ делител, т.е. общ делител с екстремал-

ното свойство да бъде по-голям от всички останали общи делители на двете числа. Такива масови задачи наричаме *оптимизационни*. Читателят навярно познава много оптимизационни задачи, изучавани в други математически дисциплини. Задачите за оптимизация имат своя специфика и за нашите цели е по-удобно да ги заменим с не оптимизационни, като си даваме сметка, че получените задачи са различни от началните.

Една достатъчно обща техника за извършване на тази замяна се състои в следното: въвеждаме нов параметър B на задачата и вместо оптималното решение търсим решение, което е в лесно проверимо отношение с B, например, ограничено (отдолу или отгоре) от B. С тази техника задачите от Пример 2 се трансформират в: „Дадени са естествените числа i, j и B. Да се намери ОД k на i и j такъв, че $k \geq B$.“ и „Дадени са естествените числа $i, j, k > 0$ и B. Да се провери дали k е ОД на i и j такъв, че $k \geq B$.“ След трансформацията на опимизационните задачи в неоптимизационни, съотношението между задачата за намиране на решение и задачата за проверка на решение стана такова, каквото беше в Пример 1. За да се провери дали зададено число k е ОД на i и j такъв, че $k \geq B$ са достатъчни две деления, две сравнения на получения остатък с 0 и едно сравнение на k с B. В същото време, намирането на такова число k си остана също толкова трудно.

С оптимизационните задачи може да се наложи да постъпваме както със задачите за намиране на решение. Когато теоретичният анализ на една оптимизационна задача е затруднен, естествено ще прибягваме до нейната неоптимизационна версия. Ако една неоптимизационна задача е алгоритмично трудна, то тогава аналогичната ѝ оптимизационна ще бъде още по-трудна.

Понятието масова задача е неформално и не може да бъде изследвано с математически средства. За целта е необходимо да изберем математически формализъм, който достатъчно добре да отразява същността на това понятие. За формален модел на понятието масова здача ще използваме специален клас *функции*. По принцип, ще разглеждаме **само масови задачи с изброимо множество екземпляри**. Масови задачи с неизброимо множество екзмпляри ще си позволяваме да използваме само в някои примери, като тези по-горе, когато обсъждаме въпроси, които не са свързани с изброимостта на екзмплярите.

Нека Π е масова задача за намиране на решение, а X е крайна азбука. Да представим входните данни $\tilde{x}$ за произволен екземпляр π на Π и съответния резултат $\tilde{y}$ с думи $\alpha_{\tilde{x}}$ и $\alpha_{\tilde{y}}$ от X^*. Масовата задача Π можем да представим с функцията $f_\Pi : X^* \to X^*$ такава, че $f_\Pi(\alpha_{\tilde{x}}) = \alpha_{\tilde{y}}$, за всеки екземпляр на Π с входни данни $\tilde{x}$. Ако Π е задача за проверка на

решение, можем да я представим с функцията $f_\Pi : X^* \to \{true, false\}$ такава, че $f_\Pi(\alpha_{\tilde{x}}) = true$, за всички екземпляри на П с входни данни $\tilde{x}$, за които проверката завършва с положителен отговор и само за тях. Тъй като в този случай f_Π задава език над азбуката X, задачите за проверка на решение можем да наречем и *задачи за разпознаване на език* – термин, който читателят навярно познава добре от курса по *Езици, автомати и изчислимост*. За задачите за намиране на решение пък ще използваме по-естественото, от гледна точка на формализма понятие *задачи за пресмятане на функция*. Очевидно, разпознаването на език е частен случай на пресмятането на функция.

Намирането на стойността $f_\Pi(\alpha_{\tilde{x}})$ по зададена $\tilde{x}$, с помощта на последователност от допустими операции, е математическият модел на неформалното *решаване на задачата* П.

Ще класифицираме масовите задачи по много съществен признак. За масовите задачи за пресмятане на функции от Пример 1 и Пример 2 на предния раздел лесно можем да посочим процедури (последователности от допустимите операции), които ни позволяват с краен брой *стъпки* да намерим решение на задачата, независимо от това кой от екземплярите ѝ е зададен. В първия случай процедурата се задава от формулата за корените на квадратното уравнение, а за втория пример процедурата е известният алгоритъм на Евклид. Съответните им задачи за разпознаване на език се решават с елементарни проверки. За втората масова задача от Пример 3 съществува добре известна процедура за построяване на триъгълник по произволни зададени две страни и ъгъл между тях.

По-различно е положението с първата масова задача от Пример 3. Не е известна достатъчно обща процедура за решаването на тази масова задача, която не зависи от конкретния екземпляр. За много от екземплярите знаем специфични решения, но те са толкова различни едно от друго, че за всеки нов екземпляр решаването трябва да започне отново. Знаем също, че за някои екземпляри такова решение не съществува.

Безусловно процедурите, които решават масовите задачи от примерите по-горе, независимо от това кой екземпляр е зададен, са тези, които в Увода нарекохме *алгоритми*. Масова задача, за която съществува алгоритъм, който я решава ще наричаме *алгоритмически разрешима*, а такава, за която не съществува алгоритъм – *алгоритмически неразрешима*. За някои масови задачи е изключително трудно да се определи дали са алгоритмически разрешими. Затова, в класификацията на масовите задачи, освен алгоритмически разрешими и неразрешими, има и много такива, за които все още не знаем разрешими ли са или не.

Класификацията на масовите задачи според алгоритмичната им разрешимост се пренася и върху моделиращите ги функции. Функциите, съответни на алгоритмически разрешимите задачи ще наречем *ефективно* (*алгоритмически*) *изчислими*. Както не е съвсем ясно очертана съвкупността от алгоритмически разрешимите масови задачи, така не е добре дефинирана и съответната съвкупност от формалните им математически модели – множеството на ефективно изчислимите функции. С формалното дефиниране на функции, които да ни служат за математически модел на множеството от ефективно изчислимите функции с изброима дефиниционна област ще се занимаем в следващите раздели на тази глава.

С помощта на познанията, които ни дават курсовете по математика (в частност тези от областта на дискретната математика) лесно можем да посочим алгоритмични процедури за цели класове от функции и по този начин да установим тяхната ефективна изчислимост, т.е. алгоритмическата разрешимост на съответните масови задачи.

Например, нека масовата задача Π се представя с функция $f_\Pi : A \to X^*$ с **крайна** дефиниционна област $A \subseteq X^*$. Тогава f_Π може да бъде зададена с таблицата си с $|A|$ реда и два стълба. В първия стълб, на всеки отделен ред, е записан някой от елементите x на дефиниционната област, като в различните редове са записани различни елементи. Във втория стълб на съответния ред е записана стойността на функцията $f_\Pi(x)$.

Тривиалният алгоритъм за ефективното изчисляване на $f_\Pi(a)$ се състои в последователна проверка на стойностите от първия стълб, до откриване на реда, съдържащ a. Тогава съдържанието на втория стълб в този ред е значението на функцията $f_\Pi(a)$. Тази алгортмична процедура е известна като *пълно изчерпване*. При всичките си недостатъци, тя решава въпроса за ефективната изчислимост на функции с крайна дефиниционна област, зададени таблично.

Читателят, навярно, знае и други способи за изчисляване на такива функции. С подходящ избор на азбуката X, елементите на дефиниционната област и областта на изменение могат да бъдат кодирани така, че всяка функция с крайна дефиниционна област може да бъде представена като крайно множество *дискретни функции* (например булеви, когато $X = \{0, 1\}$), за които да бъдат построени съответни *формули* (ДНФ или съответните им q-ични аналози) или *схеми от функционални елементи*. Процедурите за пресмятане на функция по формула или чрез схема от функционални елементи, дискутирани в курса *Дискретна математика*, са примери на алгоритмични процедури.

От курса *Езици, автомати и изчислимост* познаваме алгоритмични процедури за решаване на някои масови задачи с **изброимо безкрайно** множество от екземпляри. Задачата за *разпознаване на* произволен *автоматен език* се решава от прост алгоритъм, който проследява работата на съответния *краен автомат*. С подобна процедура, моделираща работата на *недетерминиран стеков автомат*, разпознаваме *контекстно-свободните езици*.

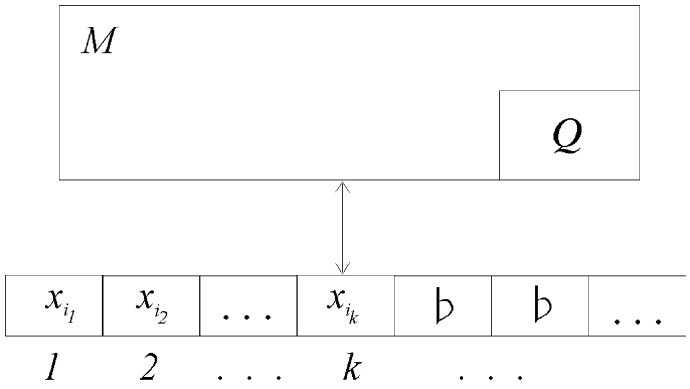
Както се вижда, някои масови задачи могат да бъдат решавани в рамките на ограничена съвкупност от алгоритми. В такъв случай казваме, че тези задачи са разрешими в рамките на тясната съвкупност. Например, задачата за разпознаване на автоматен език е *автоматно разрешима*. Задачата за разпознаване на произволен контекстно-свободен език не е автоматно разрешима, но е *разрешима с недетерминиран стеков автомат* и т.н.

Основен въпрос на теорията е, какъв формализъм за пресмятане на функции да се избере така, че с голяма доза увереност да може да твърдим, че множеството на изчислимите в рамките на този формализъм функции съвпада с множеството на ефективно (алгоритмично) изчислимите. Всеки такъв формализъм бихме могли да използваме като дефиниция на неформалното понятие алгоритъм. Съвременната математика предлага няколко такива формализма. В тази глава ще разгледаме три от тях, които са важни за изложението.

1.2 Машини на Тюринг

На Фиг. 1.1 е представена схематично абстрактна математическа машина. Тя чете входа си от безкрайно в едната посока (без ограничение на общноста сме избрали посоката на дясно) запомнящо устройство и записва изхода си на същото устройство, наричано за кратко *лента*. Входно-изходната лента е разделена на изброимо множество еднотипни *клетки*, във всяка една от които може да бъде записана някоя от буквите на крайна (входно-изходна) азбука. Клетките са номерирани (отляво на дясно) с естествените числа $1, 2, \ldots, k, \ldots$. Четенето от лентата и писането на нея се извършва от четящо-пишещо устройство, наричано *глава*. Главата е подвижна, във всеки момент се намира върху точно една клека на лентата и в края на всеки такт може да се премести върху една клетка наляво (ако не е била на най-лявата клетка), една клетка надясно или да остане на място. Означаваме с L, R и S тези възможности за преместване на главата.

Дефиниция. Петорката $M =< Q, X, q_0, \delta, F >$, където Q е крайно



Фигура 1.1: Машина на Тюринг.

множество от *състояния*, X е крайна *входна азбука*, $q_0 \in Q$ е *начално състояние*, F са *заключителни състояния*, $F \cap Q = \emptyset$, а $\delta : Q \times X \to (Q \cup F) \times X \times \{L, R, S\}$ е *функция на преходите*, наричаме *машина на Тюринг* (МТ) с безкрайна в едната посока лента.

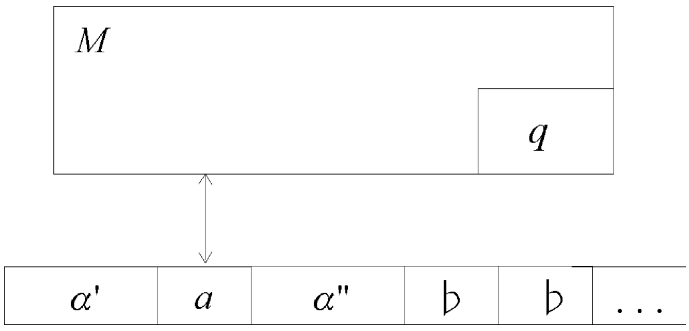
За разлика от някои други абстрактни машини, заключителните състояния на МТ са отделени от останалите състояния и са такива, че попадайки в заключително състояние машината не може да продължи да работи – *стоп-състояния*.

Една от буквите на азбуката X има специално предназначение. Наричаме я *запълваща буква* (или *бленк*) и я бележим с ♭. Ще разглеждаме машините на Тюринг с условието, че преди започване на работата, **крайно множество от клетки на лентата не съдържат бленк**. При това условие, след отработване на краен брой тактове, клетките на лентата, които не съдържат бленкове, ще продължат да бъдат крайно множество.

Дефиниция. Нека k е най-малкото естествено число такова, че всички клетки на лентата на МТ M с номера по-големи или равни на k съдържат бленк. Думата $x_{i_1}x_{i_2}\ldots x_{i_k} \in X^*$ такава, че x_{i_j} е записана в клетката с номер j, наричаме текуща *лентова дума* на M.

Дефиниция. Нека МТ $M =< Q, X, q_0, \delta, F >$ е в състояние $q \in Q$, текущата лентова дума е $\alpha' a \alpha'' \in X^*$, а главата на машината се намира над буквата a (вж. Фиг. 1.2). Тройката $(\alpha', q, a\alpha'')$ наричаме *конфигурация* на МТ M.

Очевидно, конфигурацията на МТ еднозначно определя бъдещото ѝ поведение. Конфигурациите от вида $(\varepsilon, q_0, \alpha)$ наричаме *начални конфи-*



Фигура 1.2: Конфигурация на машина на Тюринг.

гурации. Ще опишем формално работата на M чрез *трансформация на конфигурации*.

Дефиниция. Релацията $R_{\vdash} \subseteq (X^* \times Q \times X^*) \times (X^* \times Q \times X^*)$, съставена от наредени двойки конфигурации дефинираме така:

а) $(\alpha' b, q, a\alpha'') \vdash (\alpha', q', ba'\alpha'')$, ако $\delta(q, a) = (q', a', L)$;

б) $(\alpha', q, a\alpha'') \vdash (\alpha' a', q', \alpha'')$, ако $\delta(q, a) = (q', a', R)$;

в) $(\alpha', q, a\alpha'') \vdash (\alpha', q', a'\alpha'')$, ако $\delta(q, a) = (q', a', S)$.

С други думи, ако $\delta(q, a) = (q', a', m)$, то в края на такта машината на Тюринг записва a' на мястото на a, преминава в състояние q' и премества главата на една клетка в ляво, на една клетка в дясно или я оставя на място, ако $m = L, R$ или S, съответно. Ако κ_1 и κ_2 са две конфигурации на M, то трансфрмацията $\kappa_1 \vdash \kappa_2$ наричаме *непосредствен преход* на κ_1 в κ_2. Казваме още, че κ_1 *преминава непосредствено* в κ_2.

Нека $R_{\models} \subseteq (X^* \times Q \times X^*) \times (X^* \times Q \times X^*)$ е рефлексивно и транзитивно затваряне на $R_{\vdash}$. Трансформацията $\kappa' \models \kappa''$ наричаме *преход* на κ_1 в κ_2. Казваме още, че κ' *преминава* в κ''. Последователността от непосредствени преминавания $\kappa' \vdash \kappa_1 \vdash \kappa_2 \vdash \ldots \vdash \kappa_r \vdash \kappa''$ наричаме *изчисление* с машината на Тюринг M.

Естествен начин за завършване на едно изчисление на машината на Тюринг M е, тя да достигне някое от заключителните състояния. В такъв случай, казваме че M *спира нормално*. Машината може да *спре аварийно* – когато се намира в конфигурация $(\alpha, q, a\alpha')$, а стойността $\delta(q, a)$ не е дефинирана, или пък ако в конфигурация $(\varepsilon, q, a\alpha)$ имаме $\delta(q, a) = (q', a', L)$, т.е. машината се опитва да премести главата си вляво от най-лявата клетка на лентата.

Възможно е, за някоя лентова дума α, машината на Тюринг M да не спре работа никога. Например, ако машината се намира в състояние q, главата попадне над клетка, съдържаща буквата x и $\delta(q,x) = (q,x,S)$. В такъв случай казваме, че M *зацикля*.

Възможни са различни видове изчисления с машини на Тюринг. Ще се спрем на две от тях:

Дефиниция. 1) Нека M е МТ. Нека с α сме означили произволна лентова дума, за която има спиращо изчисление $(\varepsilon, q_0, \alpha) \models (\beta', q, \beta'')$ и $\beta = \beta'\beta''$. Тогава функцията $f_M : X^* \to X^*$ такава, че

$$f_M(\alpha) = \begin{cases} \beta & \text{ако } (\varepsilon, q_0, \alpha) \models (\beta', q, \beta'') \\ \text{неопределена} & \text{ако М зацикля} \end{cases}$$

наричаме *изчислима по Тюринг*.

2) Нека $F = \{q_y, q_n\}$. Дефинираме *език, разпознаван от МТ* M като множеството от думи $L_M \subseteq X^*$ такова, че $\forall \alpha \in L_M, (\varepsilon, q_0, \alpha) \models (\alpha', q_y, \alpha'')$, а $\forall \beta \notin L_M, (\varepsilon, q_0, \beta) \models (\beta', q_n, \beta'')$. Състоянието q_y наричаме *допускащо*, а състоянието q_n – *отхвърлящо*. Не е трудно да се трансформира една МТ в еквивалентна на нея със само две заключителни състояния – едно допускащо и едно отхвърлящо.

Както се вижда, машината на Тюринг е изчислителен формализъм, който е в състояние да реализира и двете интересуващи ни форми на изчисление – пресмятането на функции и разпознаването на език.

Машините на Тюринг ще задаваме с помощта на два типа оператори. Нека M е МТ с входна азбука $X = \{x_1, x_2, \ldots, x_n\}$. Първият тип оператор е за представяне на незаключителни състояния и има общ вид:

$$q)\ \delta(q,x_1); \delta(q,x_2); \ldots; \delta(q,x_n)$$

Той съдържа списък от всички стойности на функцията δ за състояието q, разделени с точка и запетая. Съответното състояние е изписано преди списъка и е отделено от него с дясна скоба. Ще го наричаме *етикет* на оператора.

Вторият тип оператор представя заключителни състояния и има общ вид:

$$q)\ stop$$

И при този тип съответното състояние е изписано като етикет, преди думата *stop*. Последователност от оператори, в която всяко $q_i \in Q \cup F$ е етикет на не повече от един оператор, наричаме *програма за МТ*.

За простота ще използваме азбуката с две букви $X = \{0,1\}$, в която 0 играе ролятя ня бленк. В теорията е доказано, че за всяка машина

на Тюринг съществува еквивалента (т.е. изчисляваща същата функция или разпознаваща същия език), азбуката на която е $\{0,1\}$.

Пример 1. Програмата

$$\begin{array}{lllll} M_1: & q_0) & (q_1,0,R) & ; & - \\ & q_1) & (q_2,1,R) & ; & (q_1,1,R) \\ & q_2) & (q_3,0,L) & ; & (q_3,0,L) \\ & q_3) & (q_4,0,S) & ; & (q_3,1,L) \\ & q_4) & \mathit{stop} & & \end{array}$$

винаги спира работа нормално, започвайки от лентовата дума $\alpha = 01^i 0a\beta$, $i = 0,1,2,\ldots$, като оставя на лентата думата $01^{i+1}0\beta$, $i = 0,1,2,\ldots$. За всяка друга дума машината спира още в началото аварийно, заради недефинираност. Следователно машината M_1 изчислява функцията

$$f_{M_1}(\alpha) = \begin{cases} 01^{i+1}0\beta & , \quad \alpha = 01^i 0a\beta \\ \text{недефинирана} & , \quad \alpha = 1\beta \end{cases}$$

Тъй като азбуката на машината M_1 се състои само от нули и единици, а нулата е бленк на тази азбука, можем да си мислим, че всеки блок от единици представя някакво естествено число – блок от 1 единица представя числото 0, блок от 2 единици – числото 1 и т.н. Такова представяне на естествените числа наричаме *унарно*. Забележете, че за да се представят унарно естествените числа, в началото и в края трябва непременно да поставим бленк. При това положение можем да кажем, че по зададено (с унарно представяне) естествено число n машината M_1 пресмята следващото естествено число – $n+1$, т.е. че пресмята функцията $f_{M_1}(n) = n+1$. Интересното тук е, че ако в началото на лентата е записана дума започваща с две нули, т.е. такава, че не представя никакво естествено число, машината също завършва нормално и оставя на лентата дума от вида 010β, т.е. машината „инициализира“ лентата с числото 0.

Пример 2. Нека сега разгледаме лентови думи от вида $01^{n+1}01^{m+1}0\beta$, които съгласно направената по-горе уговорка представят наредени двой-

ки от естествени числа (n, m). За всяка такава двойка програмата:

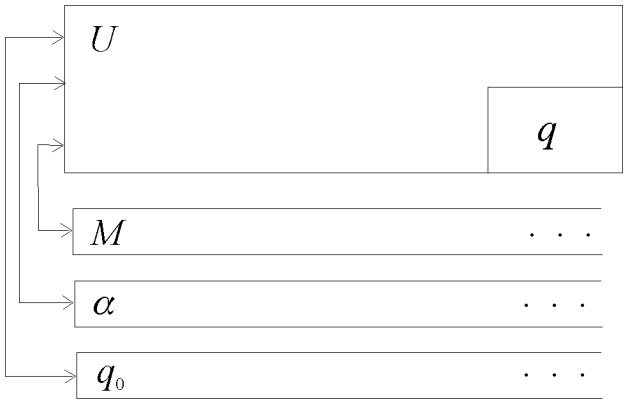
$$
\begin{array}{lllll}
M_2: & q_0) & (q_1, 0, R) & ; & - \\
 & q_1) & (q_2, 0, R) & ; & (q_1, 1, R) \\
 & q_2) & (q_2, 0, R) & ; & (q_3, 1, R) \\
 & q_3) & (q_4, 0, S) & ; & (q_5, 1, L) \\
 & q_4) & stop & & /*\, n \geq m \,*/ \\
 & q_5) & - & ; & (q_6, 0, L) \\
 & q_6) & (q_6, 0, L) & ; & (q_7, 1, L) \\
 & q_7) & (q_8, 0, S) & ; & (q_2, 0, R) \\
 & q_8) & stop & & /*\, n < m \,*/
\end{array}
$$

завършва нормално – в заключителното състояние q_4, ако $n \geq m$ и в заключителното състояние q_8, ако $n < m$. За да не усложняваме нещата, ще предполагаме, че машината M_2 работи само върху „неизродените“ лентови думи, описани по-горе. Ако приемем $q_4 = q_y$ (допускащо заключително състояние), а $q_8 = q_n$ (отхвърлящо заключително състояние), можем да заключим, че M_2 разпознава езика, състоящ се от всички такива наредени двойки естествени числа (n, m), че $n \geq m$. Ако разменим ролите на q_4 и q_8, ще получим нова машина M_2', която разпознава езика, състоящ се от всички такива наредени двойки естествени числа (n, m), че $n < m$.

Известни са много опити за обобщаване на машините на Тюринг. Например, *машина с безкрайна в двете посоки лента*, *машина с двумерна лента* (матрица с безкрайно изброимо множество редове и безкрайно изброимо множество стълбове), *недетерминирана машина на Тюринг* и т.н. В теоретичен план е доказано, че всички тези машини са еквивалентни на МТ с едностранно безкрайна лента, т.е. изчисляват същите функции или разпознават същите езици. Ще се спрем накратко само на един от тези опити, който е важен за разглежданата в тази книга проблематика – *машина на Тюринг с k ленти* (k-МТ).

В общия случай k-лентовата МТ има k различни азбуки $X_1, X_2, \ldots, X_k$ (по една за всяка лента), всяка от които съдържа собствен бленк. Нека $X = X_1 \times X_2 \times \ldots \times X_k$. При това положение функцията на преходите на k-лентовата МТ е от вида $\delta : Q \times X \to (Q \cup F) \times X \times \{L, R, S\}^k$, т.е. всяка глава има свое самостоятелно поведение на всяка стъпка. Една трилентова машина е показана на Фиг. 1.3.

За да демонстрираме как работи една k-лентова машина на Тюринг, нека разгледаме двулентови машини с азбука $\{0, 1, \flat\}$. Изборът на специален бленк, различен от 0 и 1, ще ни улесни много в програмирането.



Фигура 1.3: k-лентова машина на Тюринг.

Вместо в унарно представяне, естествените числа ще представяме в двоична бройна система, като от двата му края непременно има бленк. В това представяне, всяка от двете лентови думи ще съдържа, в общия случай, по една редица от естествени числа в двоично представяне, разделени с бленкове – $\flat\alpha_1\flat\alpha_2\flat\ldots\flat\alpha_k\flat$. За по-просто такава дума ще записваме във вида $\alpha_1, \alpha_2, \ldots, \alpha_k$. Ако някоя от двете ленти съдържа само бленкове, тогава съответната и дума ще означаваме с ϵ. С $(\alpha;\beta)$ означаваме, че на първата лента е записана α, а на втората – β.

Поради големия брой входни букви (наредените двойки от елементи на азбуката $\{0, 1, \flat\}$ са 9), използваният по-горе начин за задавне на МТ е неудобен. Затова ще задаваме машината със списък от всички стойности на функцията δ, за които машината е дефинирана.

Пример 3. Нека разгледаме двулентова машина на Тюринг M_3, с начално състояние на лентите $(\alpha;\epsilon)$, т.е. на първата лента е поставена думата $\flat\alpha\flat$, на втората лента – думата ϵ. Функцията δ на M_3 е дефинирана както следва:

$$
\begin{aligned}
M_3: \quad & \delta(q_0,(\flat,\flat)) = (q_1,(\flat,\flat),(R,R)) \\
& \delta(q_1,(0,\flat)) = (q_1,(0,0),(R,R)) \\
& \delta(q_1,(1,\flat)) = (q_1,(1,1),(R,R)) \\
& \delta(q_1,(\flat,\flat)) = (q_2,(\flat,\flat),(L,L)) \\
& \delta(q_2,(0,0)) = (q_2,(0,0),(L,L)) \\
& \delta(q_2,(1,1)) = (q_2,(1,1),(L,L)) \\
& \delta(q_2,(\flat,\flat)) = (q_3,(\flat,\flat),(S,S)) \\
& q_3 = stop
\end{aligned}
$$

Не е трудно да се проследи работата на M_3 върху прост примерен вход, за да се види, че тръгвайки от ленти със съдържание $(\alpha;\epsilon)$ машината M_3 винаги завършва нормално работа, връща главите на двете ленти върху най-левите клетки, а съдържанието на двете ленти в момента на завършване е $(\alpha;\alpha)$.

Пример 4. Нека разгледаме двулентова машина на Тюринг M_4, с начално съдържание на лентите $(\alpha;\beta)$ (две естествени числа записани в двоична система така, че най-младшият им разряд е най-вляво на лентата) и функцията δ е дефинирана както следва:

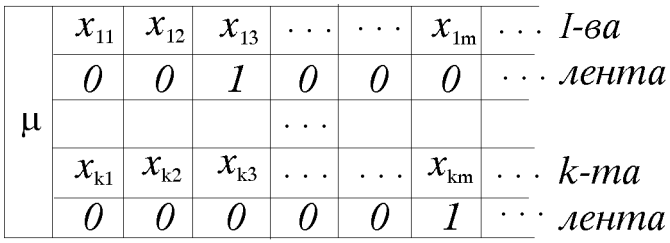
$$
\begin{array}{lll}
M_4: & \delta(q_0,(\flat,\flat)) = (q_1,(\flat,\flat),(R,R)) & \\
& \delta(q_1,(a,b)) = (q_1,(a,b),(R,R)) & /*\, a,b \neq \flat \,*/ \\
& \delta(q_1,(a,\flat)) = (q_y,a,\flat),(S,S)) & /*\, a \neq \flat \,*/ \\
& \delta(q_1,(\flat,a)) = (q_n,(\flat,a)),(S,S)) & /*\, a \neq \flat \,*/ \\
& \delta(q_1,(\flat,\flat)) = (q_2,(\flat,\flat),(L,L)) & \\
& \delta(q_2,(a,a)) = (q_2,(a,a),(L,L)) & /*\, a \neq \flat \,*/ \\
& \delta(q_2,(a,b)) = (q_y,(a,b),(S,S)) & /*\, a,b \neq \flat, a > b \,*/ \\
& \delta(q_2,(a,b)) = (q_n,(a,b),(S,S)) & /*\, a,b \neq \flat, a < b \,*/ \\
& \delta(q_2,(\flat,\flat)) = (q_n,(\flat,\flat),(S,S)) & \\
& q_y = q_n = stop &
\end{array}
$$

С проследяване работата на M_4 върху няколко примера можем да установим, че тя извършва сравняване на двете естествени числа n_α и n_β, представени с думите α и β и спира работа нормално, в състояние q_y, ако $n_\alpha > n_\beta$ и в състояние q_n, ако $n_\alpha \leq n_\beta$.

Дефинициите на двата вида изчиления за k-лентови машини – пресмятането на функция и разпознаването на език – не се различават съществено от тези, които дадохме за еднолентовите машини, затова ще ги пропуснем. Така, за машината M_3 можем да кажем, че пресмята функцията $f_{M_3}(\alpha,\epsilon) = (\alpha,\alpha)$, а за машината M_4, че разпознава езика $L_{M_4} = \{(\alpha,\beta) | n_\alpha > n_\beta\}$.

Еквивалентността на двата вида машини може да бъде доказана с моделиране на работата на всяка една от тях върху другата. Моделирането на еднолентова машина M върху k-лентова машина M_k е тривиално – използва се само първата лента. Моделирането на работата на k-лентовата M_k върху еднолентова M, разбира се, е доста по-сложно.

На Фиг. 1.4 е показан видът на лентата на моделиращата еднолентова машина M. За всяка от лентите на M_k, върху лентата на M са отделени по две подленти – първата е за думата от съответната лента на M_k, а втората показва мястото на главата за тази лента на M_k. Във



Фигура 1.4: Моделиране на $k-$МТ с МТ.

всеки момент, мястото на главата е определено с 1 в точно една от клетките на втората подлента, а всички останали нейни клетки съдържат 0. В най-лявата клетка на M записваме специална буква μ, ролята на която е да ни предпази от напускане на лентата наляво, по време на моделирането. Тази буква няма да се среща в друга клетка на лентата. Моделиращата машина ще има свой бленк $\flat$. Така за входна азбука на M имаме множеството $(X_1 \times \{0,1\} \times X_2 \times \{0,1\} \times \ldots \times X_k \times \{0,1\}) \cup \{\mu, \flat\}$.

Моделирането се извършва, като **за всеки такт** на M_k, M повтаря следните действия: k пъти (по един път за всяка лента на M_k) обхожда лентата си. Всяко обхождане започва от най-лявата клетка, стига до мястото на главата за поредната лента и се връща обратно в началото. Така, с помощта на голям брой предназначени за целта състояния, M достига до състояние, в което е „отразена“ текущата k-торка от букви на M_k – *прав ход* на моделирането. След като пресметне следващо състояние, k-торка от букви, които трябва да запише и k-торка от движенията на главите, M извършва още k обхождания на лентата си, от най-лявата клетка до мястото на съответната глава и обратно. При достигане мястото на главата на i-тата лента, M записва съответната буква и премества единицата, сочеща мястото на главата за i-тата лента на M_k – *обратен ход* на моделирането. За обратния ход също ще са необходими много голям брой, предназначени за целта състояния. Не е трудно да се прецени, че **за всяко състояние на** M_k в M ще са необходими около $|X_1||X_2|\ldots|X_k|$ състояния за правия и $|X_1||X_2|\ldots|X_k|.2^k.3^k$ състояния за обратния ход на моделирането. Така е доказано, че изчисляваните с k-лентова МТ функции (разпознаваните от тях езици) съвпадат с изчисляваните от МТ функции (разпознаваните от тях езици). Доказано е, че машините на Тюринг (и всички споменати опити за разширения, включително k-лентовие МТ) разпознават **езиците от общ тип** в йе-

рархията на Чомски.

Освен с МТ и формални граматики, са правени и много други опити за формализиране на неформалното понятие алгоритъм. Между най-известните са: *частично-рекурсивните функции*, въведени от Ст. Клини; λ-*смятането*, въведено от А. Чърч; *машините на Пост*, въведени от Е. Пост, *нормалните алгоритми на Марков*, въведени от А. А. Марков и др. Доказана е **еквивалентността** на всички известни до днес формални модели, т.е. функциите, които тези модели естествено пораждат (или изчисляват) са едни и същи – изчислимите по Тюринг функции. Еквивалентността на споменатите модели дава основание да бъде изказано следното неформално твърдение:

Тезис на Тюринг-Чърч: *Множеството на ефективно (алгоритмично) изчислимите функции съвпада с множеството на изчислимите по Тюринг функции.*

Това твърдение не може да бъде доказвано, заради използваното в него неформално понятие „ефективно (алгоритмично) изчислими функции“. Затова можем само да приведем доводи в негова полза. Най-естественият от практическа гледна точка формализъм за представяне на алгоритми е формално дефинираният език за програмиране (ще се върнем към него по-долу и ще го изплзваме в повечето случаи). Ще покажем, че създаването на програми за машини на Тюринг не се отличава съществено от процеса на реалното програмиране.

Операторът $q)\ (q', 0, S)$; $(q', 1, S)$, например, независимо от четената буква предизвиква преминаване в състояние q'. Затова можем да го означим с $q)\ \texttt{goto}\ q'$. Аналогично, операторът $q)\ (q', 0, S)$; $(q'', 1, S)$ можем да запишем като $q)\ \texttt{if}\ x = 0\ \texttt{then}\ q'\ \texttt{else}\ q''$. Следователно, МТ може да изпълнява управляващите конструкции на алгоритмичните езици.

В Пример 1 показахме, как можем да нулираме началото на лентата (т.е. да изчислим по Тюринг функцията $f(n) = 0$) и как можем да увеличим с 1 дължината на блок от единици (т.е. да изчислим по Тюринг функцията $f(n) = n + 1$). В Пример 2 показахме как може да се изчисли условие (в случая проверихме дали за две зададени естествени числа n и m е изпълнено условието $n \geq m$). За пресмятане на по-сложни функции се налага да се създават по-сложни МТ. По аналогия с използваните в програмирането техники, това може да става с композиране на готови вече машини (подпрограми) в нова машина (програма). В разглежданията по-долу, без ограничаване на общността, ще предполагаме че състоянията на всяка МТ са избрани последователно от множеството $\{q_0, q_1, q_2, \ldots\}$ и q_0 винаги е началното състояние. Без ограничение на общността можем да считаме също, че МТ, които пресмятат функции имат

точно едно, а тези, които разпознават езици – точно две заключителни състояния.

Дефиниция. Нека M е машина на Тюринг, зададена с програмата си. *Редактиране от адрес* r на M ще наричаме замяната на всяко състояние q_i на M със състояние q_{i+r}.

Нека M_1 и M_2 са МТ, пресмятащи функциите f_{M_1} и f_{M_2}, зададени със съответните си програми и състоянието с най-голям номер в M_1 е q_n. Да редактираме M_2 от адрес $n+1$ и да заменим заключителното състояние q_f) *stop* на M_1 с q_f) `goto` q_{n+1}. Получаваме нова машина на Тюринг M, която наричаме *последователна композиция* на M_1 и M_2. Очевидно, ако M_1 изчислява функцията f_{M_1}, а M_2 – функцията f_{M_2}, то M изчислява композицията $f_{M_2} \circ f_{M_1}$ на тези две функции.

Аналогично, можем да дефинираме и други композици на МТ (подпрограми), в подкрепа на твърдението, че създаването на програми на МТ е твърде близко до общоприетото понятие за програмиране. Разбира се средствата, с които се изграждат програмите на МТ са много ограничени, което прави процеса на програмиране много по-тежък, но това не променя принципния въпрос: може или не може една функция да бъде изчислена с МТ. След всичко казано, можем **да приемем верността на Тезиса на Тюринг-Чърч. Затова, в следващия раздел под алгоритъм ще разбираме машина на Тюринг**.

1.3 Сложност на алгоритми

В класическата математика, след като е установена **разрешимостта** на една масова задача, интересът към нея значително спада. Наличието на алгоритъм прави решаването на кой да е екземпляр на задачата, повече или по-малко, **рутинно**. С развитието на изчислителната техника стана ежедневие да се решават огромни по размери екземпляри на алгоритмически разрешими задачи, които не са по силите на никой човек или даже на голяма група от хора.

На фона на този значителен прогрес се открояват някои задачи с дискретен характер. Те са алгоритмически разрешими. Алгоритмичната схема „пълно изчерпване“, например, дава (**лесен, макар и не бърз**) алгоритъм за решаване на всяка такава задача. Проблемът е в това, че **когато расте размерът на входните данни, нараства твърде много времето**, необходимо за изпълнение на алгоритъма, построен по схемата „пълно изчерпване“, дори и на много мощен компютър. Например, за задачата да се провери дали един граф съдържа Хамилтонов цикъл не е известен алгоритъм, който принципно се различава от пъл-

ното изчерпване. Затова решаването на тази задача е възможно само за графи с неголям брой върхове.

Ще изложим основите на математическа теория, занимаваща се с посочения проблем. Ще въведем формална мярка за това, колко време и колко памет ще са необходими за работата на един алгоритъм върху произволен зададен екземпляр на масовата задача. От казаното по-горе е ясно, колко важен за разглежданията е *размерът на входните данни* на конкретен екземпляр. Формализацията на понятието размер на входните данни е много сложна задача. Затова, засега, ще поставим само едно важно условие, без което теоретичното изследване на проблема губи смисъл. Ще поискаме кодирането на екземплярите с думи над избрана азбука да е *разумно*, т.е. **всичко необходимо** за решаването на съответната задача да се съдържа в думата, представяща екземпляра и **да няма излишък**, т.е. такава част от входната дума, че ако я премахнем, същността на екземпляра не се променя. По-нататък ще разгледаме проблема за разумното кодиране на входните данни по-подробно.

Основен ресурс, спрямо който ще изследваме качествата на алгоритмите е *времето* за работа, но успоредно с това ще обърнем внимание и на количеството *използвана памет*, като дефинираме понятията сложност на алгоритмите, по отношение на необходимите им време и памет. За двете дефиниции по-долу, нека A е множество от думи над азбуката X, т.е. $A \subseteq X^*$. Ще означаваме дължната на думата $\alpha \in X^*$ с $d(\alpha)$, а с $A_n = \{\alpha | \alpha \in A, d(\alpha) = n\}, n = 0, 1, 2, \ldots$ – подмножеството от всички думи на A с дължина n.

Дефиниция. Нека $f_M : A \to X^*$ е тотална изчислима по Тюринг функция. Да означим с $\#_M(\alpha)$ броя на стъпките, които M прави, при работа върху лентовата дума α. Функцията

$$t_M(n) = \max_{\alpha \in A_n} \#_M(\alpha)$$

наричаме *сложност по време на M в най-лошия случай*, а функцията

$$\tilde{t}_M(n) = \frac{\sum_{\alpha \in A_n} \#_M(\alpha)}{|A_n|}$$

наричаме *средна сложност по време* на МТ M.

Аналогично за сложността по памет имаме

Дефиниция. Нека $f_M : A \to X^*$ е тотална изчислима по Тюринг функция. Да означим с $\&_M(\alpha)$ броя на клетките на лентата, които M

използва, при работа върху лентовата дума α. Клетките в които е разположена самата α винаги включваме в този брой, затова $\&_M(\alpha)$ не може да е по-малко от дължината на α. Функцията

$$s_M(n) = \max_{\alpha \in A_n} \&_M(\alpha)$$

наричаме *сложност по памет на M в най-лошия случай*, а функцията

$$\tilde{s}_M(n) = \frac{\sum_{\alpha \in A_n} \&_M(\alpha)}{|A_n|}$$

наричаме *средна сложност по памет на МТ M*.

Тъй като разпознаването на език е частен случай на изчисляване на функция ($f_M : A \to \{true, false\}$), горните дефиниции са валидни и в този случай. Да приложим дефинициите за примерите за МТ от предишния раздел.

Пример 1. Да започнем с алгоритъма (машината) M_1. При нея дължината на входните данни естествено се определя от дължината n на първия блок от единици, защото машината разглежда (освен този блок от единици) още само 3 клетки – първата и двете клетки след блока от единици. За пресмятане на сложността (и по време и по памет) в най-лошия случай е задължително да идентифицираме този най-лош случай. При алгоритъма M_1 имаме само един случай, затова той е и най-лош. При своята работа машината преглежда по един път всяка от $(n+3)$-те клетки при движение на главата надясно, а при връщането си назад – по един път всяка от първите $n+2$ клетки. Затова $t_{M_1}(n) = 2n+5$. Никоя друга клетка освен споменатите по-горе не участва в изчислението и затова $s_{M_1}(n) = n+3$ Тъй като имаме работа с единствен случай, средните сложности, по време и по памет, не се различават от тези в най-лошия случай, т.е. $\tilde{t}_{M_1}(n) = 2n+5$, а $\tilde{s}_{M_1}(n) = n+3$.

Пример 2. При алгоритъма M_2 нещата са малко по-сложни. Да означим с N сумата $n+m$ от дължините на двата блока с единици. Това е естественият размер на входа на този алгоритъм, тъй като освен двата блока от единици, машината разлежда само още 3 клетки, съдържащи в началото бленк – началната, разделящата двата блока и маркиращата края на втория блок. По-лесно е да пресметнем сложността по памет. И в най-лошия и в средния случай няма да имаме нужда от други клетки, освен споменатите вече $N+3$ и затова $s_{M_2}(N) = \tilde{s}_{M_2}(N) = N+3$.

Не е трудно да се съобрази, че най-лош случай за тази машина ще е един от тези, при които двете числа са приблизително еднакви, т.е.

$n = m$, $n = m - 1$ или $n = m + 1$. Да разгледаме първо случая $n = m$. В началото машината, премествайки главата надясно, отива до края на левия блок от единици, което става с $n = N/2$ стъпки. След това започва основният цикъл. Изтрива се единица отдясно, за което са необходими 3 стъпки – главата преминава през разделящата нула, единицата, която подлежи на изтриване и клетката, която е след нея и от съдържанието на която решаваме, дали не трябва да спрем цикъла. Следва движение в обратната посока, за изтриване на единица отляво, като при първата стъпка наляво изтриваме първата единица на дясната дума. На този етап машината вече ще направи 4 стъпки, за следващите две изтривания ще са необходми съответно 5 и 6 стъпки и т.н. Когато машината установи, че може да изтре предпоследната единица на лявата дума и тръгне надясно за да изтрие n-тата единица на дясната дума, тя ще направи точно $2n + 1 = N + 1$ стъпки – $n - 1$ стъпки през изтритите единици на лявата дума (включително последната изтрита), 1 стъпка през разделящата двете думи нула, $n - 1$ стъпки през изтритите единици на дясната дума, 1 стъпка през последната единица и една стъпка върху нулата след нея. Тогава машината установява, че двете думи са с равен брой единици и ще спре работа. Затова

$$\begin{aligned} \#_{M_2}(m = n) &= \tfrac{N}{2} + 3 + 4 + 5 + 6 + \cdots + (N + 1) = \tfrac{N}{2} + \textstyle\sum_{n=3}^{N+1} i = \\ &= \tfrac{N}{2} + \tfrac{(N+1)(N+2)}{2} - 3 = \tfrac{N^2+4N-4}{2}. \end{aligned}$$

В случая $m = n+1$, т.е. $N = 2n+1$, изтриването на n-та единица отдясно ще бъде успешно, а опитът за изтриване на съответна единица отляво ще установи, че тя е последна и алгоритъмът ще спре. Значи при последното движение на главата наляво машината ще направи $2n+2 = N+1$ стъпки - преминавайки през n изтрити единици на дясната дума, разделящата нула, $n - 1$ изтрити единици на лявата дума, последната единица на лявта думя и нулата след нея. Затова в този случай

$$\begin{aligned} \#_{M_2}(m = n + 1) &= \tfrac{N-1}{2} + 3 + 4 + \ldots + (N + 1) = \tfrac{N-1}{2} + \textstyle\sum_{n=3}^{N+1} i = \\ &= \tfrac{N-1}{2} + \tfrac{(N+1)(N+2)}{2} - 3 = \tfrac{N^2+4N-5}{2}. \end{aligned}$$

Аналогично за случая $m = n - 1$, т.е. $N = 2n - 1$, имаме $2n = N + 1$ стъпки при последното движение на главата надясно и

$$\begin{aligned} \#_{M_2}(m = n - 1) &= \tfrac{N+1}{2} + 3 + 4 + \ldots + (N + 1) = \tfrac{N+1}{2} + \textstyle\sum_{n=3}^{N+1} i = \\ &= \tfrac{N+1}{2} + \tfrac{(N+1)(N+2)}{2} - 3 = \tfrac{N^2+4N-3}{2}. \end{aligned}$$

Както се вижда, макар и със съвсем малко, но последният случай е най-лош от трите и затова $t_{M_2}(N) = \frac{N^2+4N-3}{2}$.

Намирането на сложността по време в средния случай е по-трудно. Първо, да видим какво трябва да се направи за да намерим сложността в средния случай и след това да направим съответните пресмятания. Когато двата блока имат общо N единици, тогава трябва да разгледаме $N-1$ случая – $N-1$ единици в лявата дума и 1 в дясната, $N-2$ единици в лявата дума и 2 дясната, ..., 1 единица в лявата дума и $N-1$ в дясната. За всеки от случаите трябва да направим пресмятания, аналогични на тези които направихме за най-лошия случай, да сумираме получените резултати и да разделим на $N-1$.

Нека в лявата дума има i единици, а в дясната – $j = N-i$. От направените по-горе ръзсъждения можем да заключим, че в началото машината ще напрви i стъпки за да стигне до края на лявата дума, след което ще прави 3 стъпки надясно за да изтрие първата единица на дясната дума, 4 съпки наляво за да изтрие първата единица на лявата дума,..., $k(i)$ стъпки в някоя от двете посоки, за да установи че от съответната дума не могат да се изтриват повече единици. Броят на стъпките $k(i)$ на последния етап, очевидно, зависи от i. Когато $i < N-i$, при последното движение на главата няляво машината ще направи $k(i) = 2i+2$ стъпки, а когато $i \geq N-i = j$ – при последното движение на главата надясно машината ще направи $k(i) = 2(N-i)+1 = 2j+1$ стъпки. Следователно при $i < j$ машината ще направи $3+4+5+\cdots+(2i+2) = \frac{(2i+2)(2i+3)}{2} - 3$ стъпки, а при $i \geq j$ ще направи $3+4+5+\cdots+(2j+1) = \frac{(2j+1)(2j+2)}{2} - 3$ стъпки. Така за общия брой T на стъпките по всичките $N-1$ случаи получаваме

$$T = \sum_{i<j}[i + \frac{(2i+2)(2i+3)}{2} - 3] + \sum_{i\geq j}[N-j + \frac{(2j+1)(2j+2)}{2} - 3].$$

При $N = 2p+1$, като сменим променливата j във втората сума с i, получаваме

$$\begin{aligned} T &= \textstyle\sum_{i=1}^{p}[i+(i+1)(2i+3)-3] + \sum_{i=1}^{p+1}[N-i+(2i+1)(i+1)-3] = \\ &= \textstyle\sum_{i=1}^{p}[N+2i^2+5i+2i^2+3i-2] + 2p^2+8p-3 = \\ &= (N-2)p + 4\textstyle\sum_{i=1}^{p} i^2 + 8\sum_{i=1}^{p} i + 2p^2 + 8p - 3 = \\ &= (N-2)p + \frac{4p(p+1)(2p+1)}{6} + \frac{8p(p+1)}{2} + 2p^2 + 8p - 3. \end{aligned}$$

Следователно, като вземем предвид, че $N-1 = 2p$, получаваме, че при нечетни $N > 1$

$$\begin{aligned} \widetilde{t}_{M_2}(N) &= \frac{T}{2p} = \frac{N-2}{2} + \frac{(p+1)(2p+1)}{3} + 2(p+1) + p + 4 - \frac{3}{p} = \\ &= \frac{N}{2} + \frac{(N+1)(N+9)}{6} + 2 - \frac{3}{p} = \frac{N^2+13N+21}{6} - \frac{6}{N-1}. \end{aligned}$$

Оставяме на читателя да намери съответната сложност в средния случай, когато N е четно.

Пример 3. За алгоритъма (машината) M_3 е очевидно, че ако означим с n дължината на входната дума α върху първата лента, то за всички думи с дължина n машината M_3 прави един и същ брой стъпки и използва един и същ брой клетки. Затова лесно установяваме, че

$$t_{M_3}(n) = \tilde{t}_{M_3}(n) = 2n + 2, s_{M_3}(n) = \tilde{s}_{M_3}(n) = 2n + 4.$$

Пример 4. За алгоритъма (машината) M_4 нека първо пресметнем сложността по памет и време в най-лошия случай. За дължина на входа да изберем дължината n на по-дългата от двете думи α и β. Най-лошият случай, и за сложността по памет и за сложността по време, е когато двете думи са еднакви. Затова, за най-лошия случай лесно получаваме

$$t_{M_4}(n) = n + 1, s_{M_4}(n) = 2n + 4.$$

За да пресметнем сложността по памет в средния случай, трябва да разгледаме всички възможни случаи. Когато първата дума е с дължина n, а имаме 2^n такива възможности, то за дължината d на другата е възможна всяка от дължините $1, 2, \ldots, n$. Така при дължина на втората дума d ще имаме $2^n.2^d$ случая, във всеки от които ще бъдат ангажирани по $n + d + 4$ клетки общо на двете ленти. Тъй като ситуацията, когато втората дума е с дължина n е симетрична, получаваме за общия брой на случаите $2\sum_{d=1}^{n-1} 2^n.2^d + 2^{2n}$, а за сумарния брой използвани клетки $2\sum_{d=1}^{n-1} 2^n.2^d(n + d + 4) + 2^{2n}(2n + 4)$. Така

$$\tilde{s}_{M_4}(n) = \frac{2\sum_{d=1}^{n-1} 2^n.2^d(n + d + 4) + 2^{2n}(2n + 4)}{2\sum_{d=1}^{n-1} 2^n.2^d + 2^{2n}}.$$

За да пресметнем сложността по време в средния случай, постъпваме аналогично. Когато едната от двете думи е с дължина $d < n$, тогава неравенството ще бъде установено при достигане на граничния бленк в десния край на по-късата дума и в такъв случай машината ще направи точно $d+1$ стъпки. Когато и двете думи са с дължина n, тогава неравенството ще бъде установено при обратния ход на главата на машината. Да преположим, че първите $i - 1$ двоични цифри, от дясно наляво, са еднакви и двете думи се различават в i-тите си цифри, $i = 1, 2, \ldots, n$. Тогава машината ще направи точно $n + i + 1$ стъпки. При $i = n + 1$ ще получим и случая при който двете думи са равни (тогава машината ще спре не заради несъвпадение на цифри, а поради достигане на граничните бленкове вляво). Възможни са 2^{i-1} начина за съвпадане на първите

$i-1$ цифри, два начина за несъвпадане на i-тите (първата цифра е 0, втората е 1 или обратно) и $2^{2(n-i)}$ за останалите цифри, които няма да бъдат сравнявани. Следователно общият брой случаи, в които машината ще завърши работата след $n+i+1$ стъпки, са $2^i 2^{2(n-i)} = 2^{2n-i}$. Затова, за сложността по време в средния случай ще получим

$$\tilde{t}_{M_4}(n) = \frac{2\sum_{d=1}^{n-1} 2^n.2^d(d+1) + \sum_{i=1}^{n+1} 2^{2n-i}(n+i+1)}{2\sum_{d=1}^{n-1} 2^n.2^d + 2^{2n}}.$$

Преобразуването на суми, подобни на тези по-горе, до „затворени“ изрази, които не съдържат суми, изисква специална техника, на която ще се спрем по-подробно в следващата глава. Затова ще отложим засега окончателното пресмятане на средната сложността по време и памет на машината M_4.

Както се вижда пресмятането на сложността в средния случай (както по време, така и по памет) може да се окаже нелека работа. Затова, с малки изключения, в тази книга ще се ограничим до пресмятане само на сложността в най-лошия случай. Дотолкова, доколкото ресурсът време е много по-ценен от ресурса памет (човешкият живот, за съжаление, е относително кратък), ще се ограничим при разглеждането на алгоритми само до тяхната сложност по време. Ще говорим за сложността по памет само в онези случаи, при които това е от решаващо значение за качествата на алгоритъма.

По-горе споменахме, че в сложността по памет непременно трябва да включим необходимия за разполагане на входните данни брой клетки. Не е изключено, обаче, някои алгоритми да не „преглеждат“ всички заделени за входните данни клетки. Затова ще наричаме *ефективна сложност по памет* броя на тези клетки, които действително са прегледани поне веднъж от алгоритъма е ще я означаваме с $e(n)$. Между сложността по време и ефективната сложност по памет в най-лошия случай съществува следната важна връзка, която е добре да имаме предвид, когато не се занимаваме специално със сложността по памет:

Лема 1.3.1 *Нека M е машина на Тюринг с k ленти, $k = 1, 2, \ldots$. Тогава $e_M(n) \leq kt_M(n), \forall n \in N$.*

Доказателство. Твърдението е очевидно, защото за една стъпка машината не може да използва повече от една клетка на всяка от лентите. □

Ще използваме този факт, за да оценим „сложността“ по време в най-лошия случай на показаното по-горе моделиране на еднолентова МТ

M с k-лентова МТ M_k и обратно. Да означим с $t_M(n)$ сложността по време на M, а с $T_{M_k}(n)$ сложността на моделиращата я M_k. Доколкото за моделирането на работата на M върху M_k не е нужно нищо друго, освен повтаряне дословно работата на M, с използване само на първата лента на M_k, получаваме, че $T_{M_k}(n) = t_M(n)$. Да означим сега с $t_{M_k}(n)$ сложността по време на M_k, а с $T_M(n)$ сложността на моделиращата я M_k. За всеки такт от работата на M_k моделиращата M трябва първо да намери мястото на всяка от главите. Не е възможно за i такта M_k да е преместила която и да е от главите по-далеч от клетката с номер i на съответната лента. Затова в първата фаза на моделиране на i-тия такт M ще направи не повече от $2ki$ стъпки. Във втората фаза, за всяка от лентите може да се наложи M да отиде до $(i+1)$-вата клетка, за да премести главата надясно и затова M ще направи не повече от $2k(i+1)$ стъпки в тази фаза. И така, като означим за по-просто $t_{M_k}(n)$ с $t(n)$, получаваме

$$T_M(n) \leq \sum_{i=1}^{t(n)} 2k(2i+1) = 2k[t(n)(t(n)+1) + t(n)] = 2k(t(n)^2 + 2t(n)).$$

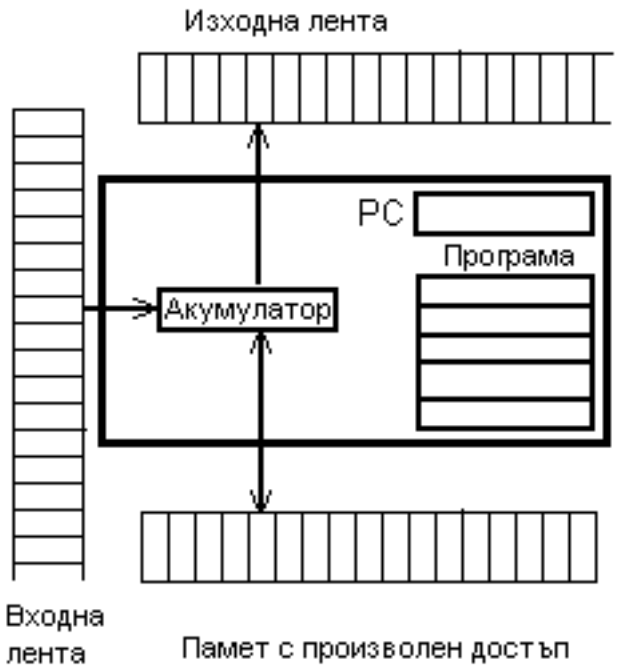
Така доказахме следната

Лема 1.3.2 *Ако M_k е k-лентова МТ със сложност по време в най-лошия случай $t_{M_k}(n)$, то съществува МТ M, която изчислява същата функция (разпознава същия език) както и M_k със сложност по време в най лошия случай $t_M(n) \leq 2k(t_{M_k}(n)^2 + 2t_{M_k}(n))$.*

Съвсем според очакванията, времето необходимо за моделирането на работта на M_k върху M нараства. Даже на пръв поглед изглежда, че нарастването е много голямо. От гледна точка на торията това не е съвсем така. Нека, например, сложността на M_k е полином от втора степен. Тогава сложността на моделиращия алгоритъм ще бъде **също полином**, макар и от четвърта степен. Както ще видим по-късно, такова увличаване на сложността при моделиране на един формализъм с друг е напълно приемливо.

1.4 Машини с произволен достъп до паметта

Машините на Тюринг са много добър формализъм за въвеждане на понятията на теорията и, както ще видим по-късно, за получаване на важни теоретични резултати относно сложността на агоритмите и задачите. Те, обаче, са трудно приложими за определяне на сложността на



Фигура 1.5: Машина с произволен достъп до паметта.

алгоритмите използвани в практиката. В този раздел ще въведем един изчислителен модел, който е много по-бизък до реалността от МТ.

На Фиг. 1.5 е показана схема на абстрактна математическа машина, наречена *машина с произволен достъп до паметта* (МПД). Машината с произволен достъп до паметта е съставена от Управляващ блок и три ленти, разделени на *клетки*. Всяка МПД има параметър R, наричан *разрядност*. МПД с разрядност R ще наричаме *R-разрядна МПД* или просто *R-МПД*. Входната и изходната лента на всяка МПД са безкрайни в едната посока, а паметта с произволен достъп на R-МПД се състои от 2^R клетки, номерирани с числата от 0 до $2^R - 1$. Номерът A на една клетка от паметта ще наричаме *адрес* на тази клетка. Всяка клетка от входната лента, от изходната лента и от паметта може да съдържа цяло число в интервала $[-2^{R-1}, 2^{R-1} - 1]$. *Съдържанието* на клетката от паметта с адрес A ще означаваме с $< A >$.

В Управляващия блок на R-МПД са разположени две специализирани клетки - *акумулаторът*, означаван накратко с AC и *броячът на команди*, означаван накратко с PC. Акумулаторът може да съдържа

всяко цяло число в интервала $[-2^{R-1}, 2^{R-1}-1]$, а броячът на команди – цяло число в интервала $[0, 2^R-1]$. Съдържанието на акумулатора ще означаваме с $< AC >$, а на брояча на команди – с $< PC >$.

В Управляващия блок на МПД е разположена и *програмата*, съставена от команди. Можем да си мислим, че програмата на МПД е разположена в последователност от клетки – *програмна памет*, подобна на паметта с произволен достъп. Клетките на програмната памет са номерирани с естествените числа $0, 1, 2, \ldots$. Всяка от клетките на програмната памет може да съдържа по една команда, а номерът B на клетката наричаме *адрес на командата*, съдържаща се в нея. Във всеки момент от времето МПД изпълнява една команда и нейният адрес се намира в брояча на команди, т.е. $< PC >$ винаги е адресът на изпълняваната от машината команда.

Всяка команда от програмата на МПД се състои от *код на командата* и *аргумент на командата*. За МПД, командите на която са с един аргумент казваме, че е *едноадресна*. Може да бъдат дефинирани и МПД с по-голяма адресност – двуадресни, триадресни и т.н. – но, от гледна точка на теорията за сложност на алгоритмите, те не се различават принципно от едноадресните МПД. Адресът на командата ще записваме пред нея, отделен с дясна скоба. Така общият вид на една команда е

```
адрес_на_команда) код_на_команда [ аргумент_на_команда ]
```

където със знаците `[` и `]` означаваме, че някои от командите не се нуждаят от аргумент.

Кодовете на командите на МПД са естествени числа, но за да не затрудняваме четенето, по традиция, ще заместваме всеки код на команда със съответна *мнемоника* – съкращение от английска дума, която ни напомня за предназначението на командата (виж списъка на командите на Фиг. 1.6). Например, за командите които извършват събиране ще използваме мнемониката `ADD` (от английската дума ADDition). Съществуват три различни начина за задаване на *операнд* в **командите на МПД, които извършват операции** – *непосредствен операнд*, *пряка адресация* и *косвена адресация* .

Команди с непосредствен операнд. При тези команди операндът е зададен в самата команда, на мястото за аргумент. Непосредственият операнд може да бъде произволно R-разрядно цяло число, с което може да се извърши операцията. За да укажем, че командата е с непосредствен операнд, поставяме знака `#` в края на мнемониката. Например

```
1000) ADD# -5
```

ще предизвика следното действие. Съдържанието на акумулатора ще се събере с числото -5 и полученият резултат ще замени в акумулатора старото му съдържание. Това означаваме накратко така `<AC> := <AC> + I`, където `I` е стойността на непосредствения операнд. След като изпълнението на командата завърши, броячът на команди се увеличава с 1 и се пристъпва към изпълнение на командата, разположена в следващата клетка на програмната памет. Това записваме наратко така `<PC> := <PC> + 1`.

Команди с пряка адресация. При тези команди операндът се намира в паметта, а в командата, на мястото за аргумента, се поставя адресът му. Пряк адрес може да бъде всеки адрес на клетка от паметта. Командите с непосредствен операнд задаваме без специален знак в края на мнемониката. Например

```
1000) ADD 245
```

ще предизвика следното действие. Съдържанието на акумулатора ще се събере с числото, съдържащо се в клетката с адрес 245 и полученият резултат ще замени в акумулатора старото му съдържание. Това означаваме накратко така `<AC> := <AC> + <A>`, където `A` е адресът на операнда. И в този случай, след като изпълнението на командата завърши, броячът на команди се увеличава с 1 и се пристъпва към изпълнение на командата разположена в следващата клетка – `<PC> := <PC> + 1`.

Команди с ксвена адресация. При тези команди операндът се намира в паметта, например в клетка с адрес X. В друга клетка на паметта, да речем с адрес A, трябва да се намира адресът X. В командата, на мястото за аргумента, се поставя адресът A. Косвен адрес може да бъде всеки адрес на клетка от паметта. За да укажем, че командата е с косвена адресация, поставяме знака `@` в края на мнемониката. Например

```
1000) ADD@ 245
```

ще предизвика следното действие. Съдържанието на акумулатора ще се събере с числото, съдържащо се в клетката с адрес $< 245 >$ и полученият резултат ще замени в акумулатора старото му съдържание. Това означаваме накратко така `<AC> := <AC> + «A»`, където `A` е косвеният адрес на операнда, или с други думи адресът на адреса на операнда. И в този случай, след като изпълнението на командата завърши, броячът на команди се увеличава с 1 и се пристъпва към изпълнение на командата разположена в следващата клетка.

Всички команди за **аритметичните операции** на МПД са дадени в таблицата на Фиг. 1.6. Освен командите за събиране (с мнемоника `ADD`),

Команда	**Действие**
LOAD# I	<AC> := I; <PC> := <PC> + 1
LOAD A	<AC> := <A>; <PC> := <PC> + 1
LOAD@ A	<AC> := <<A>>; <PC> := <PC> + 1
STORE A	<A> := <AC>; <PC> := <PC> + 1
STORE@ A	<<A>> := <AC>; <PC> := <PC> + 1
ADD# I	<AC> := <AC> + I; <PC> := <PC> + 1
ADD A	<AC> := <AC> + <A>; <PC> := <PC> + 1
ADD@ A	<AC> := <AC> + <<A>>; <PC> := <PC> + 1
SUB# I	<AC> := <AC> - I; <PC> := <PC> + 1
SUB A	<AC> := <AC> - <A>; <PC> := <PC> + 1
SUB@ A	<AC> := <AC> - <<A>>; <PC> := <PC> + 1
MUL# I	<AC> := <AC> * I; <PC> := <PC> + 1
MUL A	<AC> := <AC> * <A>; <PC> := <PC> + 1
MUL@ A	<AC> := <AC> * <<A>>; <PC> := <PC> + 1
DIV# I	<AC> := <AC> / I; <PC> := <PC> + 1
DIV A	<AC> := <AC> / <A>; <PC> := <PC> + 1
DIV@ A	<AC> := <AC> / <<A>>; <PC> := <PC> + 1
MOD# I	<AC> := <AC> % I; <PC> := <PC> + 1
MOD A	<AC> := <AC> % <A>; <PC> := <PC> + 1
MOD@ A	<AC> := <AC> % <<A>>; <PC> := <PC> + 1
JMP B	<PC> := B
JMPZ B	Ако <AC> = 0, то <PC> := B, иначе <PC> := <PC> + 1
JMPP B	Ако <AC> > 0, то <PC> := B, иначе <PC> := <PC> + 1
JMPN B	Ако <AC> < 0, то <PC> := B, иначе <PC> := <PC> + 1
INPUT	Поредната клетка на входната лента се прочита в AC, главата се мести на следваща клетка
INPUT	<AC> се записва в поредна клетка на изходната лента, а главата се мести на следваща клетка
STOP	Прекратява изпълнението на програмта

Фигура 1.6: Команди на МПД

това са командите за изваждане (с мнемоника `SUB`), за умножение (с мнемоника `MUL`), за целочислено деление (с мнемоника `DIV`) и намиране на остатъка при целочислено деление (с мнемоника `MOD`).

Не е проблем да бъдат добавени и други команди от „аритметичен“ тип като побитова конюнкция на акумулатора и операнда, побитова дизюнкция на акумулатора и операнда, или пък побитово изместване на съдържанието на акумулатора наляво или надясно на толккова позиции, колкото е стойността на операнда и т.н. Тъй като няма да използваме подобни команди в примерите, няма да ги въвеждаме формално.

Подобни на командите от аритметичен тип са **командите за работа с акумулатора**. Командите за поставяне на стойност в акумулатора са с мнемоника `LOAD`. Възможни са и трите типа адресация – стойността, която трябва да бъде поставена в акумулатора да бъде зададена непосредствено в командата, както и да бъде зададена с пряк или косвен адрес. Командите за запис на стойността от акумулатора в паметта са с мнемоника `STORE`. Логически не е възможно да съществува команда за запис на стойност в паметта без да е зададен адрес, затова тук имаме само две от версиите – с пряк и косвен адрес. Формалното описание на тези пет команди също е дадено в таблицата на Фиг. 1.6.

Командите за разклоняване на програмата се различават от командите от аритметичен тип и командите за работа с акумулатора по това, че аргументите на тези команди не са адреси в паметта или непосредствени стойности, а адреси на команди от програмата. Командата за безусловен преход е с мнемоника `JMP`. В резултат от изпълнението на командата, съдържание на брояча на команди става аргументът `B` на командата и затова следващата изпълнявана команда ще бъде тази, с адрес `B`. Съкратено това действие на машината означаваме с `<PC> := B`.

Командата за преход при нулево съдържание на акумулатора е с мнемоника `JMPZ`, тази за преход при положително съдържание на акумулатора - с менмоника `JMPP`, а при отрицателно съдържание на акумулатора – `JMPN`. Когато условието на командата е изпълнено (акумулаторът е с нулева, положитена или отрицателна стойност, съответно) тогава работата на програмата продължава с командата адресирана от аргумента. Ако условието не е изпълнено, работата на програмата продължава със следващата по ред команда. И тук, както при командите от аритметичен тип, можем да добавим и други команди за разклоняване на програмата – `JMPZP` за преход при неотрицателна стойност в акумулатора, `JMPNP` за преход при ненулева стойност в акумулатора и `JMPNZ` за преход при неположителна стойност в акумулатора, но това няма да промени съществено машината.

Посследните три команди не се нуждаят от аргумент. Командата `INPUT` чете едно число от поредната клетка на входната лента, записва го в акумулатора и премества четящата глава върху следващата клетка. Командата `OUTPUT` извежда съдържанието на акумулатора в поредната клетка на изходната лента и премества четящата глава върху следващата клетка. Съдържанието на акумулатора остава непроменено. Командата `STOP` прекратява изпълнението на програмата. Ще илюстрираме с няколко примера работата на МПД.

Пример 1. Като първи пример да съставим програма за МПД, която въвежда от входната лента коефициентите a, b и c на квадратно уравнение, пресмята дискриминантата D на уравнението по формулата $b^2 - 4ac$ и извежда получения резултат на изходната лента. Първата работа при решаването на задача с МПД е, да определим къде ще поставим даните в паметта на машината. Трите коефициента на уравненението ще разположим в клетките с номера 0, 1 и 2, а дискриминантата – в клетката с номер 3. Клетката с номер 4 ще използваме за запазване на междинния резултат от пресмятането. Получаваме следното разпределение на паметта:

```
0) за коефициента a
1) за коефициента b
2) за коефициента c
3) за дискриминантата d
4) за междинен резултат
```

Ето и съответната програма (с // сме отделили командата от съответния коментар):

```
 0) INPUT     // въвеждаме a в AC
 1) STORE 0   // съхраняваме a в паметта
 2) INPUT     // въвеждаме b в AC
 3) STORE 1   // съхраняваме b в паметта
 4) INPUT     // въвеждаме c в AC
 5) STORE 2   // съхраняваме c в паметта
 6) LOAD# 4   // поставяме 4 в AC
 7) MUL 0     // умножаваме AC по a
 8) MUL 2     // умножаваме AC по c
 9) STORE 4   // запазваме 4ac в паметта
10) LOAD 1    // поставяме b в AC
11) MUL 1     // умножаваме AC по b
12) SUB 4     // изваждаме 4ac от AC
```

```
13) STORE 3   // запазваме резултата
14) OUTPUT    // извеждаме резултата
15) STOP      // прекратяваме изпълнението
```

Пример 2. Като втори пример да напишем програма за МПД, която въвежда от входната лента цели числа, докато срещне число равно на нула, съхранява въведените числа (без нулата) в последователни клетки на паметта и извежда на изходната лента броя на въведените числа, различни от 0. Можем да считаме, че разрядността на МПД е толкова голяма, че всяко от зададените на входната лента числа може да се запише в клетка от паметта и има достатъчно клетки за всички различни от нула числа, зададени на входната лента.

Нека отново започнем с разпределянето на паметта. Клетката с адрес 0 ще съдържа стойността $i + 1$, където i е броят на прочетените до момента числа. В началото нейната стойност ще бъде 1. След като прочетем число различно от нула, ще записваме стойността му в клетката с адрес $i + 1$ (**разглеждайки съдържанието на клетката с адрес 0 като адрес в паметта**) и ще увеличаваме стойността на клетката с адрес 0 с 1. Когато за пръв път прочетем от входната лента нула, в клетката с адрес 0 ще имаме точно $n + 1$, където n е броят на прочетените числа, различни от нула. Остава да извадим 1 от стойността на клетката с адрес 0 и да изведем полученото на изходната лента. В клетките с адреси $1, 2, \ldots, n$ ще са записани въведените числа – нещо като **масив** от цели числа, индексиран с числата от 1 до n.

Ето и програмата за МПД, която решава поставената задача

```
 0) LOAD# 1   // поставяме 1 в AC
 1) STORE 0   // съхраняваме 1 в клетка 0 (индекс)
 2) INPUT     // въвеждаме поредно число в AC
 3) JMPZ 9    // ако е въведена 0 - команда 9)
 4) STORE@ 0  // записваме числото в адреса от клетка 0
 5) LOAD 0    // стойността на брояча - в AC
 6) ADD# 1    // увеличаваме брояча с 1
 7) STORE 0   // нова стойност на брояча в клетка 0
 8) JMP 2     // връщаме се за ново четене
 9) LOAD 0    // стойността на брояча в AC
10) SUB# 1    // намаляваме брояча с 1
11) STORE 0   // крайна стойност на брояча в клетка 0
12) OUTPUT    // извеждаме резултата
13) STOP      // прекратяваме изпълнението
```

Забележете ролята на косвената адресация при организация на цикъла и достъпа до последователни клетки от паметта, както и използването на командите с непосредствени операнди за пресмятане на изрази, в които участват константи.

Понятията *изчислима с МПД функция*, *разпознаван от МПД език* и различните понятия за *сложност на МПД* – по време и памет, в най-лошия и средния случай, не се различават съществено от тези, които въведохме за машините на Тюринг. За целта е достатъчно да приемем целите числа в интервала от $[-2^{n-1}, 2^{n-1}-1]$ за букви на крайна азбука A. В такъв случай размер на входа при МПД M ще бъде броят n на клетките от входната лента, в които са записани входните данни.

Нека $\#_M(\alpha)$ е броят на изпълнените от МПД M команди, при работа върху думата α, а $\&_M(\alpha)$ е броят на използваните от МПД M клетки от паметта, при работа върху думата α. При това формулите, с които дефинирахме сложностите по време и памет за МТ, в най-лошия и в средния случай, са валидни и за МПД.

Да намерим функциите на сложност на МПД от двата примера по-горе, като означим с M_1 машината от Пример 1, а с M_2 – машината от Пример 2. Машината от Пример 1 е особена. Размерът n на входните данни при нея винаги е 3 – броят на коефициентите на квадратно уравнение. На пръв поглед функциите на сложност за M_1 изглеждат недефинирани за стойности на n различни от 3. Такива ситуации възникват нерядко, затова за пресмятането на сложноста в този случай да постъпим формално. Какво ще направи машината M_1, ако поставим на входната ѝ лента повече от 3 числа? Машината ще въведе първите три от тях и ще извърши предвидените пресмятания. Подобно ще бъде положението и ако поставим на входната лента по-малко от 3 числа. Машината ще въведе съдържанието на първите три клетки на лентата, каквото и да е то, и ще извърши пресмятането с въведените стойности.

И така, каквото и да поставим на лентата, машината ще въведе стойностите на първите три клетки и ще извърши пресмятанията, за което ще са необходими точно 16 команди. Тъй като броят на изпълнените команди не зависи от въведените три стойности, получаваме $t_{M_1}(n) = \tilde{t}_{M_1}(n) = 16$.

Както се вижда от разпределението на паметта, което направихме преди да напишем програмата за M_1, каквото и да поставим на лентата, машината ще използва само 5 клетки от паметта. Затова $s_{M_1}(n) = \tilde{s}_{M_1}(n) = 5$.

Нещата при машината M_2 от Пример 2 изглеждат по-нормално. Нека в началото на входната лента са поставени n числа, различни от нула,

след които, в клетката с номер $n+1$ е поставена 0. Независимо от това кои са числата върху входната лента, машината M_2 ще изпълни точно по 1 път командите с адреси 0 и 1. Командата за въвеждане от адрес 2 и командата за проверка дали въведеното в акумулатора число е равно на нула от адрес 3 ще бъдат изпълнени по $n+1$ пъти – по един за всяко различно от нула число и един път за нулата. Всяка от командите с адреси от 4 до 8 (тялото на цикъла) ще бъде изпълнена по един път за всяко от n-те различни от 0 числа, а всяка от командите с адреси от 9 до 13 ще бъде изпълнена точно по 1 път.

И така, при поставени върху входната лента n различни от нула числа, машината M_2 ще изпълни 7 команди по 1 път, 2 команди ще бъдат изпълнени по $n+1$ пъти, а 5 команди – по n пъти, независимо от това, кои са n-те чесла. Следователно

$$t_{M_2}(n) = \tilde{t}_{M_2}(n) = 7.1 + 2.(n+1) + 5.n = 7n + 9.$$

Също така, независимо от стойностите на n-те числа, програмата ще използва винаги $n+1$ клетки от паметта – една за индекса и n за съхраняване на самите числа. Затова $s_{M_2}(n) = \tilde{s}_{M_2}(n) = n+1$.

Особеността на тази машина е, че при нея всеки случай е най-лош и затова сложностите в средния случай съвпадат със сложностите в най-лошия случай. Затова да рзгледаме още един пример, в който, поне за пресмятане на сложността по време, не всеки случай е най-лош.

Пример 3. Нека в първите n клетки на входната лента на МПД M_3 са поставени числата от 1 до n в произволен ред. Да напишем програма, която определя мястото в редицата (поредния номер от 1 до n), в която се намира единицата.

```
 0) LOAD# 1   // поставяме 1 в AC
 1) STORE 0   // съхраняваме 1 в клетка 0 (брояч)
 2) INPUT     // въвеждаме поредно число в AC
 3) SUB# 1    // изваждаме 1
 4) JMPZ 9    // ако числото е било 1, AC ще бъде 0
 5) LOAD 0    // стойността на брояча в AC
 6) ADD# 1    // увеличаваме брояча с 1
 7) STORE 0   // нова стойност на брояча в клетка 0
 8) JMP 2     // връщаме се за ново четене
 9) LOAD 0    // стойността на брояча в AC
10) OUTPUT    // извеждаме резултата
11) STOP      // прекратяваме изпълнението
```

Да пресметнем сложността по време на M_3 в най-лошия случай. Не е трудно да се определи кой е този случай – когато единицата е на последно място в редицата. Сега командите с адреси 0, 1, 9, 10 и 11 ще бъдат изпълнени само по един път, командите с адреси 2, 3 и 4 – по n пъти, а командите с адреси 5, 6, 7 и 8 – по $n-1$ пъти. Затова $t_{M_3}(n) = 7n+1$. Формулата за сложността в най-лошия случай можем да използваме по следния начин. Ако единицата не беше на последната n-та позиция в редицата, а в позиция $i < n$, тогава машината щеше да изпълни точно $7i+1$ команди. Всички възможни подреждания на числата от 1 до n са $n!$, като подредженията, в които единицата е в i-та позиция са $(n-1)!, i = 1, 2 \ldots, n$. Затова

$$\tilde{t}_{M_3}(n) = \frac{\sum_{i=1}^{n}(n-1)!(7i+1)}{n!} = \frac{1}{n}\sum_{i=1}^{n}(7i+1) = \frac{1}{n}\frac{7n^2+9n}{2} = \frac{7n+9}{2}.$$

При всички случаи програмата на M_3 използва само една клетка от паметта и затова $s_{M_3}(n) = \tilde{s}_{M_3}(n) = 1$.

МПД са, безусловно, по-удобен механизъм за представяне на изчислителните процедури, които наричаме алгоритми. Да означим с Z_r множеството на r-разрядните цели числа от интервала $[-2^{r-1}, 2^{r-1}-1]$ и да разглеждаме елементите му като букви на крайна азбука. Дефинициите за изчислима функция и разпознаван език можем да пренесем върху МПД по следния начин:

Дефиниция. Функцията $f_M : Z_r^* \to Z_r^*$ наричаме *изчислима с r-разрядната МПД M*, ако като поставим в първите n клетки на входната лента числата $a_1, a_2, \ldots, a_n, a_i \in Z_r, i = 1, 2, \ldots, n$, и стартираме машината, тя завършва работа нормално (с команда `STOP`), а в първите m клетки на изходната лента е записала числата $b_1, b_2, \ldots, b_m$ такива, че $f_M(a_1 a_2 \ldots a_n) = b_1 b_2 \ldots b_m$.

Дефиниция. Казваме, че езикът L, определен от функцията $L_M : Z_r^* \to \{0,1\}$ *се разпознава от r-разрядната МПД M*, ако като поставим в първите n клетки на входната лента числата $a_1, a_2, \ldots, a_n, a_i \in Z_r, i = 1, 2, \ldots, n$, и стартираме машината, тя завършва работа нормално (с команда `STOP`), а в първата клетка на изходната лента е записала 1, когато думата $a_1 a_2 \ldots a_n$ е от L, или 0 – когато $a_1 a_2 \ldots a_n$ не е от L.

Не е трудно да се докаже следната

Теорема 1.4.1 *Функциите изчислими с МПД са точно изчислимите по Тюринг функции, а разпознаваните с МПД езици са точно езиците разпознавани с МТ.*

Доказателството на теоремата, по същество, се състои в моделиране работата на МТ с МПД и обратно, така както моделирахме работата на k-лентова МТ върху еднолентова и обратно. За да се направи това моделиране формално ще се изискват сериозни усилия. За нуждите на теорията за сложност на алгоритмите ще е достатъчно да опишем неформално двете моделирания и да се опитаме да оценим сложността при моделиращата машина, като функция от сложността на моделираната машина. Ще се ограничим само до сложността по време в най-лошия случай, като оставим останалите оценки на читателя за упражнение.

Да започнем с моделирането на МТ $M(Q, X, q_0, \delta, F)$ върху МПД M'. Нека $t_M(n)$ е сложността по време, а $s_M(n)$ е сложността по памет на M в най-лошия случай. При моделиране на M върху МПД, първо трябва да разрешим два проблема – с броя $|X|$ на буквите на азбуката X и броя $|Q|$ на състоянията на M, които може да са произволно големи и с безкрайността на лентата при МТ.

Входните букви и състоянията на M ще представяме с последователни r-разрядни цели числа, започвайки от 0, а движенията на главата, както обикновено с $-1, 0$ и 1. Доколкото броят на буквите и състоянията, с които ще може да борави r-разрядната МПД е по-малък от или равен на 2^r, то трябва да изберем такова r, че $|X| \leq 2^r$ и $|Q| \leq 2^r$. При този избор дефиницията на функцията δ ще запишем в паметта на МПД, като $\forall q \in Q$ и $\forall x \in X$ заделим по три последователни клетки от паметта, в които да поставим q', y, m, където $\delta(q, x) = (q', y, m)$. Ако подредим тройките клетки плътно една до друга, представянето на функцията δ ще заеме $3|Q||X|$ клетки от паметта (за да упростим нещата, ще предполагаме, че функцията δ е тотална).

За нуждите на моделирането ще ни трябват още няколко клетки, които ще поставим в началото на паметта – преди таблицата на δ. В клетката с номер 0, например, ще помним текущото състояние на МТ, а в клетката с номер 1 – позицията на главата (като адрес в паметта). За да упростим нещата, ще предполагаме, че заключителни състояния на МТ са състоянията с номера по-големи от или равни на F – това може да се постигне лесно с преномериране на състоянията. Числото F ще поставим в клетката с номер 2. В клетката с номер 3 ще поставим броя $|X|$ на буквите във входната азбука на МТ умножен по 3, за по-бърз достъп до таблицата на функцията δ. Клетките 4 и 5 ще използваме за запазване на междинни резултати от пресмятания на адреси. И така, $S = 6 + 3|Q||X|$ е броят на заетите клетки, започвайки от началото на паметта.

Ще построим таблицата на функцията δ като запълним съответните

стойности по най-елементарния начин. Да започнем със стойностите на δ за началното състояние 0. Нека $\delta(0,0) = (q, y, m)$. Съответният програмен код ще започва така:

```
 0) LOAD# 6
 1) STORE 4    // индекс в клетка 4
 2) LOAD# q
 3) STORE@ 4   // въвеждане на q
 4) LOAD 4
 5) ADD# 1
 6) STORE 4
 7) LOAD# y
 8) STORE@ 4   // въвеждане на y
 9) LOAD 4
10) ADD# 1
11) STORE 4
12) LOAD# m
13) STORE@ 4   // въвеждане на m
14) LOAD 4
15) ADD# 1
16) STORE 4
. . .
```

И така за въвеждане на таблицата на функцията δ ще бъдат изпълнени 2 команди за инициализиране на индекса и по 15 команди за въвеждане на стойността на δ за всяка комбинация от състояние и входна буква – общо $3|Q||X| + 2$ команди – стойност, която не зависи от n.

Доколкото паметта на МПД е крайна, проблемът с моделиране на безкрайната входно-изходна лента на M можем да разрешим така. В началото, входната дума на M е поставена върху входната лента на МПД, а за моделиране на променящото се състояние на лентата на M ще използваме клетките от паметта на МПД, започвайки от първата клетка след описанието на функцията δ (с адрес S). При работата си върху входна дума с дължина n, машината M не може да промени сдържанието на повече от $t_M(n)$ последователни клетки на входно-изходната лента (вж. Лема 1.3.1). Затова моделирането на работата на M върху произволна входна дума с дължина n може да се извърши върху r-разрядна МПД такава, че освен поставените по-горе условия за r, трябва да е изпълнено и $S + t_M(n) \leq 2^r$.

Преди да започнем моделирането, ще прочетем $t_M(n)$ клетки от входната лента в определената за целта памет на МПД. Нека първият сво-

боден адрес на команда, след като сме завършили въвеждането на δ е a. Съответният фрагмент от програмата на МПД ще изглежда приблизително така (за по-лесно четене на програмата сме заменили адресите в паметта и адресите на програмните оператори със символични имена):

```
  a) LOAD# S    // началото на лентата в AC
a+1) STORE I    // инициализация на индекса
a+2) INPUT      // въвеждане на поредната клетка
a+3) STORE@ I   // запис в паметта
a+4) LOAD I     //
a+5) ADD# 1     // увеличаване на индекса
a+6) STORE I    //
a+7) SUB T      // сравняване с крайната стойност
a+8) JMPN a     // зацикляне на четенето
```

Следователно за пренасяне на входната дума на МТ в паметта на МПД ще бъдат изпълнени $7t_M(n) + 2$ команди.

Сега вече сме готови да започнем моделирането. Инициалиацията на необходимите за целта кетки ще извършим със следния фрагмент, всяка от деветте команди на който ще се изпълни по 1 път.

```
 a+9) LOAD# S    // началото на лентата в AC
a+10) STORE 0    // позиция на главата в клетка 0
a+11) LOAD# 0    //
a+12) STORE 1    // текущо състояние q в клетка 1
a+13) LOAD F     // последно незаключително състояние
a+14) STORE 2    // в клетка 2
a+15) LOAD# |X|  //
a+16) MUL# 3     //
a+17) STORE 3    // 3.|X| в клетка 3
```

А ето и прогамният фрагмент, който буквално повтаря стъпките на на МТ M:

```
a+18) LOAD 3     // 3.|X| в AC
a+19) MUL 0      // 3.|X|.{тек. състояние}
a+20) STORE 4    //
a+21) LOAD@ 0    // тек. буква x в AC
a+22) MUL# 3     //
a+23) ADD 4      //
a+24) ADD p      //
```

```
 a+25) STORE 4    // позиция на (q',y,m)
b+17) ADD# 1     // позиция на y
b+18) STORE 5    //
b+19) LOAD@ 5    // y в AC
b+20) STORE@ 0   // y замества x
b+21) LOAD 4     //
b+21) ADD# 2     // позиция на m
b+22) ADD 0      // + стара позиция на главата
b+23) STORE 0    // = нова позиция на главата
b+24) LOAD@ 4    // q' в AC
b+24) STORE      // q' става текущо
b+25) SUB 2      // сравнявяне с F
b+25) JMPN b+6   // следваща стъпка.
```

Както се вижда за всяка стъпка на M в МТ ще бъдат изпълнени точно 20 команди. Следователно общият брой команди, изпълнени по време на моделирането на стъпките на M ще бъде $20t_M(n)$. За да завършим работата, остава да изведем върху изходната лента на МПД съдържанието на МТ такова, каквото се е получило в резултат на моделирането. Тази част на програмата не се различава съществено от часта, в която въведохме съдържанието на лентата на M в паметта на МПД и ще изисква изпълнението на още $7t_M(n) + 1$ команди. Така, за сложността $T_{M'}(n)$, с която МПД M' моделира работата на M поулчаваме:

$$T_{M'}(n) = 3|Q||X| + 2 + 2(7t_M(n) + 1) + 9 + 20t_M(n) = 34t_M(n) + C,$$

където константата $C = 3|Q||X| + 13$. С това е доказана следната

Теорема 1.4.2 *Нека M е МТ, която изчислява функция (разпознава език) със сложност по време в най-лошия случай $t_M(n)$. Тогава, за всяко $n \in N$, съществува МПД M', която изчислява същата функция (разпознава същия език) със сложност $T_{M'}(n) = 34t_M(n) + C$, където C е константа, не зевисеща от n.*

Моделирането в другата посока е доста по-трудно, затова ще го опишем без да влизаме в подробности и ще направим груба оценка на неговата сложност. Ще моделираме работата на МПД M със сложност по време в най-лошия случай $t_M(n)$ върху 4-лентова МТ M'. Предназначението на четерите ленти е следното. Ще считаме, че първата лента на M' има същото съдържание, като входната лента на M. На втората лента ще записваме това, което M пише върху изходната си лента. С третата

лента на M' ще моделираме паметта на M, а четвъртата ще използваме за записване на междинно пресметнати стойности, които ще ни бъдат необходими за по-късни моменти на моделирането.

Основен проблем на моделирането в тази посока е отсъствието на пряк достъп до паметта в МТ. Казано по-прсто, за да може машината M' да достигне до аргумента на една команда, намиращ се на третата лента, тя трябва да започне от най-лявата клетка на тази лента и със помощта на съответни състояния да „преброи“ клетките, който трябва да пропусне, за да стигне до търсената. В най-лошия случай, на всеки такт от работата на M машината M' ще трябва да направи $t_M(n)$ стъпки в търсенето на аргумента и още $t_M(n)$ стъпки за да върне главата в началото и да се приготви за следващия такт. Заради наличието на косвена адресация, може да се наложи пътешествието да се извърши два пъти. И така за всяки изпълнена от M команда, на M' ще се наложи да направи по $t_M(n) + C$ стъпки, кдето C е някаква константа.

В сила е следната

Теорема 1.4.3 *Нека M е МПД, която изчислява функция (разпознава език) със сложност по време в най-лошия случай $t_M(n)$. Тогава, за всяко $n \in N$, съществува МТ M', която изчислява същата функция (разпознава същия език) със сложност $T_{M'}(n) = t_M^2(n) + Ct_M(n)$, където C е константа, не зависеща от n.*

1.5 Език за програмиране

Последният формализъм, който ще разгледаме, е *език за програмиране*. Кой от съвременните езици за програмиране ще използваме за целта е без значение, затова сме се спряли на езика C. Ще зададем сложност на всяка от конструкциите на езика C така, че сложността на една програма написана на C да е равна на сложността на съответната ѝ програма, написна за МПД, плюс-минус някаква константа.

Сложност на израз. Ще започнем с най-простите конструкции на езика C – изразите. Нека първо разгледаме няколко примера. За израза `a + b - c`, съответният фрагмент от програма на МПД ще бъде

```
  a) LOAD a
a+1) ADD b
a+2) SUB c
```

и значи броят на командите (в случая 3) може да изразим с броя на участващите в израза операции плюс 1. Този израз е твърде прост – в него

всички операции са с един и същ приоритет. Какво ще стане с броя на необходимите команди, ако в израза има различни приоритети. В израза `a * b - c` има различни приоритети, но ако изпълним пресмятанията отляво надясно ще получим верен резултат. Съответната програма за този израз също е от 3 команди, т.е. броят на командите отново е равен на броя на операциите плюс 1.

В израза `a + b * c`, обаче, не можем да изпълним операциите отляво надясно заради по-високия приоритет на операцията умножение. Ако не използваме приоритети, този израз би трябало да се запише като `a + (b * c)`. Съответната му програма за МПД ще бъде:

```
  a) LOAD b
a+1) MUL c
a+2) STORE x
a+3) LOAD a
a+4) ADD x
```

и броят на командите е 5 – равен на броя на операциите (2), плюс броя на задължителните в случая скоби (една лява и една дясна), плюс 1.

Ето още един пример. Без да използваме приоритети, изразът `a*b+c*d` може да се запише като `a*b+(c*d)`, а съответната му програма за МПД:

```
  a) LOAD c
a+1) MUL d
a+2) STORE x
a+3) LOAD a
a+4) MUL b
a+5) ADD x
```

има 6 команди – отново броят на командите е равен на броя на операциите (3), плюс броя на задължителните скоби (2), плюс 1.

Как ще изглеждат нещата, ако някой от операндите е елемент на масив. Например, `a[b]+c*d=a[b]+(c*d)`. Програмата:

```
  a) LOAD c
a+1) MUL d
a+2) STORE x
a+3) LOAD a   // началото на масива
a+4) ADD b    // адреса на a[b] в AC
a+5) STORE y  // адреса на a[b] в y
a+6) LOAD@ y  // a[b] в AC
a+7) ADD x    // a[b]+c*d в AC
```

има 8 команди и излиза, че в сложността на такъв израз трябва, освен споменатите вече елемети, да участва сложността на всеки индексен израз, като към него трябва да прибавим още 2, за двете индексни скоби. Действително сложността на индексния израз `b` е 1 (нула знаци за операции плюс 1) и като добавим два знака за операции, две кръгли скоби, две индексни скоби и още 1 за целия израз, получаваме точно 8.

Тези наблюдения ни дават основание да формулираме следното правило:

Сложност на израз. *Сложността* t_{expr} *на израза* `expr` *е равна на сумата от сложностите на участващите в израза индексни изрази, броя на знаците за операции, броя на задължителните кръгли скоби и броя на индексните скоби, плюс 1.*

Сложност на оператор за присвояване. В езика C **присвояването е операция** и операторът за присвояване не е нищо повече от израз, в края на който е поставен знакът `';'` за край на оператор. Затова би трябвало да очакваме, че правилото за сложност на оператор за присвояване няма да се различава от правилото за сложност на израз.

Да разгледаме, например, оператора за присвояване `a[b]=c*d;`. Очакваната сложност е 6 – индексен израз със сложност 1, два знака за операции, две скоби, плюс 1. И действително, съответната му програма ще изглежда така:

```
  a) LOAD a
a+1) ADD b
a+2) STORE x
a+3) LOAD c
a+4) MUL d
a+5) STORE@ x
```

Тук е мястото да обърнем внимание на някои специфични за езика C операции, които са комбинация на операция от аритметичен тип и присвояване (`++`, `--`, `+=`, `-=`, и т.н.). Да разгледаме, например, оператора `i++;`. Ако гледаме на `++` като на една операция, ще получим несъответствие с формулираното по-горе правило, защото сложността ще излезе 2, докато съответната МПД програма е със сложност 3:

```
  a) LOAD i
a+1) ADD# 1
a+2) STORE i
```

Причината е в това, че операторът `i++;` всъщност е кратък запис на оператора `i=i+1;`, който според правилото е със сложност 3. Същото

важи и за останалите операции, които са комбинация на операция от аритметичен тип и присвояване. Тъй като всяка от тези операции се представя с два знака, не е необходимо да правим никакви промени в правилото за сложност на аритметичен израз – дстатъчно е просто да броим и двата знака, представящи съответната операция.

При пресмятане на изрази, в които участват операциите за сравняване (`<`, `<+`, `==` и т.н.), трябва да вземем предвид и следната особеност. Командите за условни преходи извършват сравняване на стойността на акумулатора с 0. Затова пресмятането на сложността на изрази като `expr1<expr2` ще свеждаме до пресмятане сложността на `expr1-expr2<0`.

Можем да формулираме следното правило:

Сложност на оператор за присвояване. *Сложността t_{assgn} на оператора за присвояване* `assgn` *е равна на сложността на съставящия го израз.*

В допълнение към това правило ще формулирме и правилото за сложност на оператора `return expr;` както следва:

Сложност на оператора за връщане на стойност. *Сложността t_{return} на оператора за връщане на стойност* `return` *е $t_{return} = t_{expr} + 1$.*

Сложност на блок от оператори. Правилото за пресмятане сложността на блок от оператори (съставен оператор):

```
{ op_1  op_2 ... op_k }
```

е съвсем просто:

Сложност на блок от оператори. *Сложността t_{bl} на блок от оператори (съставен оператор)* `bl` *е равна на сумата $t_{op\_1} + t_{op\_2} + \cdots + t_{op\_k}$ от сложностите на съставящите го оператори.*

Сложност на условни оператори. Ще разгледаме отделно `if`-оператора

```
if (expr) oper
```

и `if-else`-оператора

```
if (expr) oper_1 else oper_2
```

И в двата варианта на оператора, първо трябва да се пресметне сложността на израза *expr*.

За `if`-оператора, най-лош е случаят, когато изразът има стойност `true` и трябва да се изпълни оператора *oper*. За `if-else`-оператора, най-лош е случаят, когато изразът има такава стойност, че трябва да

се изпълни този от операторите *oper_1* и *oper_2*, който е с по-голяма сложност. Получаваме следното правило:

Сложност на условни оператори. *Сложността на условните оператори* `if` *и* `if-else` *се определя както следва*: $t_{if} = t_{expr} + t_{oper}$, $t_{if-else} = t_{expr} + \max\{t_{oper\_1}, t_{oper\_2}\}$.

Сложност на оператори за цикъл. Да започнем с `while`-оператора

```
while (expr) oper
```

и `do`-оператора

```
do oper while (expr);
```

При пресмятане на сложността на оператори за цикъл, можем да постъпим по два начина – „груб“ и „прецизен“. При грубата оценка намираме сложността t_{expr} на израза `expr`, сложността t_{oper} на оператора `oper` в най-лошия случай и се опитваме да оценим броя $c(n)$ на повторенията, които операторът ще направи. В такъв случай сложността t_{while} на оператора за цикъл е равна на $c(n)(t_{expr} + t_{oper}) + t_{expr}$. Когато сложността на оператора не зависи от n, това оценяване е достатъчно, но ако сложността на оператора зависи от n, грубата оценката наистина може да се различава съществено от прецизната. За случая, когато сложността на оператора зависи от n, може да се окаже, че е по-добре да намерим сложностите $t_1(n), t_2(n), \ldots, t_{c(n)}(n)$ на оператора, който е тяло на цикъла, поотделно за всяка от итерационните стъпки и сложността t_{while} на оператора за цикъл да се пресметне по формулата $(c(n)+1)t_{expr} + t_1(n) + t_2(n) + \cdots + t_{c(n)}(n)$.

При `do`-оператора имаме само тази особеност, че условието се проверява след изпълнение на тялото на цикъла и затова проверките са с една по-малко. Формулите са подобни на тези при `while`-оператора: $c(n)(t_{expr} + t_{oper})$ за грубия и $c(n)t_{expr} + t_1(n) + t_2(n) + \cdots + t_{c(n)}(n)$ за прецизния случай.

Малко по-сложни са нещата при `for`-оператора

```
for(expr1;expr2;expr3) oper
```

Нека отново с $c(n)$ сме означили броя на итерациите на цикъла. Изразът `expr1` се пресмята еднократно, изразът `expr2` се пресмята $c(n)+1$ пъти, а изразът `expr3` и операторът `oper` – по $c(n)$ пъти. Затова, при грубо оценяване, сложността на `for`-оператора ще пресмятаме по формулата $t_{expr1} + (c(n)+1)t_{expr2} + c(n)(t_{expr3} + t_{oper})$. За прецизното пресмятане на сложността, формулата ще бъде $t_{expr1} + (c(n)+1)t_{expr2} + c(n)t_{expr2} + t_1(n) + t_2(n) + \cdots + t_{c(n)}(n)$.

Сложност на функция. Да разгледаме отделно нерекурсивните и рекурсивните функции. Тялото на нерекурсивна функция е блок от оператори, а ние вече имаме правило за пресмятане на нейната сложност. Остава само да спомена, че когато оценяваме сложността на извикване на функцията, трябва да определим внимателно размера на входните данни с които е направено извикването. Добре е да се има предвид и времето необходимо на ОС да осъществи извикването, което до голяма степен зависи от броя на предаваните при извикването параметри.

Всяка рекурсивна функция f трябва да има клон (за по-просто нека си мислим, че е един), който завършва без да влиза в рекурсия и клон (нека отново си мислим, че е един), в който има едно или повече рекурсивни извиквания на функцията. Очевидно е, че за да приключи изпълнението на рекурсивната функция върху входни данни с размер n, рекурсивните извиквания в тялото ѝ трябва да са върху входни данни с размери по-малки от n. Нерекурсивният клон на функцията, пък, се изпълнява за някакви крайни, обикновено много малки стойности на n. Нека в тялото на функцията има k рекурсивни извиквания с размери $n_1, n_2, \ldots, n_k$, съответно, $n_i < n, i = 1, 2, \ldots, k$. Да означим с $t_f(n)$ сложността на f в най-лошия случай и нека $g(n)$ е сложността на нерекурсивния клон на функцията. Нека $h(n)$ е сложността на рекурсивния клон, без сложностите на рекурсивните извиквания. Очевидно $t_f(n)$ може да се изрази със следното рекурентно отношение:

$$t_f(n) = \begin{cases} g(n), & n \leq C \\ t_f(n_1) + t_f(n_2) + \cdots + t_f(n_k) + h(n), & n > C \end{cases}$$

където C е някаква цяла положителна константа. Така, намирането на сложността на рекурсивни функции по естествен начин се свежда до решаване на рекурентни отношения – техника на която ще обърнем специално внимание в следващата глава.

Сложност на програма на C. Сложност на програма, написана на езика C е сложността на нейната `main`-функция.

Да разгледаме няколко примера за пресмятане сложност на функции, написани на C.

Пример 1. В масив `a` от тип `int` са зададени n цели числа (по различни причини ще използваме елементите на масива с индекси `1,2,...,n`). Да напишем функция `search`, с параметър едно число `x` от тип `int`, която да проверява има ли елемент в масива, който е равен на `x`. Ако такъв елемент има, функцията трябва да върне индекса на този елемент, а иначе – да върне 0. Ето и съответната функция:

```
int a[],n;
```

```
int search(int x)
{
   int i;
   for(i=1;i<=n;i++)
      if(a[i]==x) return i;
   return 0;
}
```

За размер на входните данни да изберем броя n на числата в масива и да намерим сложността по време в най-лошия случай на тази функция. Най-лош случай ще имаме, когато търсеното число не се среща в масива и цкълът `for` направи всички придвидени стъпки – $c(n) = n$. Освен това, не е трудно да се види, че в най лошия случай операторът `return` в тялото на цикъла няма да се изпълни нито един път, защото сравнението `a[i]==x` винаги завършва със стойност `false`. Следвайки дефинираните по-горе правила получаваме:

$$\begin{aligned} t_{search}(n) &= t_{for}(n) + t_{return} = \\ &= t_{expr1} + (c(n)+1)t_{expr2} + c(n)(t_{expr3} + t_{oper}(n)) + t_{return} = \\ &= 2 + (n+1).3 + n(3 + t_{if}(n)) + 2 = \\ &= 3n + 7 + n(3 + t_{expr} + t_{oper}(n)) = \\ &= 3n + 7 + n(3 + 5 + 0) = 11n + 7 \end{aligned}$$

За да покажем как някои изменения в алгоритъма могат да се отразят на сложността на програмата, да напишем нова версия на алгоритъма за търсене в масив с използване на *сентинел* (пазач) – така наричат търсеното число `x`, поставено като допълнителен (например нулев) елемент на масива.

```
int a[],n;
int search1(int x)
{
   int i;
   a[0]=x;
   for(i=n;;i--)
      if(a[i]==x) return i;
}
```

Най-лош случай отново е отсъствието на търсеното число в масива и тогава цикълът ще направи $c(n) = n+1$ стъпки. В този случай, операторът `return` в тялото на цикъла ще се изпълни точно един път и затова ще извадим сложността му извън сложността на цикъла. Така получаваме:

$$\begin{aligned} t_{search1}(n) &= t_{assgn} + t_{for}(n) = \\ &= 4 + t_{expr1} + (c(n)+1)t_{expr2} + c(n)(t_{expr3} + t_{oper}(n)) = \\ &= 4 + 2 + (n+2).0 + (n+1)(3 + t_{if}(n)) = \\ &= 6 + (n+1)(3 + t_{expr}) + t_{return} = \\ &= 6 + (n+1)(3+5) + 2 = 8n + 16 \end{aligned}$$

Разликата между поведението на функциите $11n + 7$ и $8n + 16$ изглежда незначителна и по-нататък в нашите резглеждания по-скоро ще я пренебрегваме. Но от друга страна, при търсене в масив с 333 333 333 елемента например, вторият алгоритъм ще направи около 1 милиард операции по-малко.

Нека пресметнем и сложността по време в средния случай $\widetilde{t}_{search}(n)$. Различните случаи които трябва да разгледаме са $n + 1$, в зависимост от това на коя позиция в масива се намира търсеното число `x`. Във всеки случай програмата ще направи 2 стъпки за присвояването `i=1;`. Ако търсеното число се намира в първия елемент на масива, тогава програмата ще направи още 3+5+2 стъпки – за проверката `i<=n`, за проверка на условието в оператора `if` и за оператора `return`. Или общо 12 стъпки.

Нека търсеното число се намира в $(i+1)$-вия елемент на масива или, казано по друг начин – не е сред първите i елемента, $i = 1, 2, \ldots, n-1$. Тогава, подобно на пресмятанията за най-лошия случай можем да заключим, че до итерацията на цикъла, в която числото ще бъде намерено, програмата ще направи $2 + i(3 + 5 + 3) = 11i + 2$ стъпки. В последната итерация ще бъдат изпълнени 10 стъпки – 3 за сравняването `i<=n`, 5 за проверката `a[i]==x` и 2 за `return`, или общо $11i + 12$. Случаят, когато числото не се намира в масива е вече разгледаният най-лош случай и за него броят на стъпките е $11n + 7$.

Така получаваме:

$$\widetilde{t}_{search}(n) = \frac{1}{n+1}[12 + \sum_{i=1}^{n-1}(11i + 12) + 11n + 7] = \frac{11n^2 - 35n + 14}{2(n+1)}$$

Както се вижда, в средния случай алгоритъмът ще направи около $11n/2$ стъпки, което е половината на броя на стъпките в най-лошия случай - нещо което трябваше да се очаква.

Пример 2. Да напишем програма за сортиране на n числа от тип `int`, разположени в елементите с индекси `1,2,...,n` на масив `a`. Ще използваме популярния „алгоритъм на мехурчето“:

```
int a[],n;
void swap(int i,int j)
{int t=a[i];a[i]=a[j];a[j]=t;}
int bubble()
{
   int i,j;
   for(i=n-1;i>=1;i--)
      for(j=1;j<=i;j++)
         if(a[j]>a[j+1]) swap(i,j);
}
```

Да намерим сложността на функцията `bubble` по време в най-лошия случай. Да започнем със сложността на функцията `swap`, която очевидо не зависи от n: $t_{swap} = t_{assgn1} + t_{assgn2} + t_{assgn3} = 4 + 6 + 4 = 14$. Затова сложността на оператора `if` ще бъде $t_{if} = t_{expr} + t_{swap} = 8 + 14 = 22$. Сложността на вложения оператор `for` да пресметнем, като за размер на входа приемем параметъра `i`. Като приложим правилото за сложност на цикъл в грубия вид (защото сложността на тялото тук е константа), получаваме $t_{for2}(i) = 2 + (i+1)3 + i(3+22) = 28i + 5$.

Слжността на функцията ще бъде равна на сложността на външния оператор `for`. При него сложността на тялото е различна за различните итерации, затова ще изпозваме прецизната формула:

$$t_{bubble}(n) = t_{for}(n) = 3 + 3n + 3(n-1) + \sum_{i=1}^{n-1}(28i + 5) = 14n^2 - 3n - 5.$$

Намирането на сложността на алгоритъма на мехурчето в средния случай е трудна задача. Няма да се спираме на нея тук, но я препоръчваме на читателя като трудно упражнение.

Пример 3. Като трети пример да разгледаме отново задачата за търсене на число `x` в масив `a` с `n` елемента, но при условие, че елементите на масива са сортирани в нарастващ ред. Програмата дадена по-долу извършва т.н. „двоично търсене“ в сортирания масив. Не е проблем алгоритъмът да бъде имплементиран без рекурсия, но за целите на изложението ще реализираме рекурсивния варант на функцията, която извършва двоично търсене в интервал на масива от i-тия до j-тия елемент, $i \leq j$:

```
int a[],n;
int binsrch(int i,int j)
{
```

```
    int m,l,r;
    l=i;r=j;
    while(l<=r)
    {
       m=(l+r)/2;
       if(a[m]==x) return m;
       if(x<a[m]) return binsrch(l,m-1);
       else return binsrch(m+1,r);
    }
    return 0;
}
```

Да пресметнем сложността по време в най-лошия случай на функцията

```
int f(){ return binsrch(1,n);
```

която е практически равна на сложността $t_{binsrch}(n)$ на `binsrch`, извикана върху масив с n елемента. Най-лош случай винаги ще бъде случаят, при който търсеното число не се среща в масива. Когато $n = 1$ функцията ще влезе в цикъла `while` и, в зависимост от това, дали стойността на `x` е по-малка или по-голяма от `a[1]`, ще направи рекурсивното извикване `binsrch(1,0)` или рекурсивното извикване `binsrch(2,1)`. Което от двете и да се случи, условието на цикъла `while` няма да е изпълнено и функцията ще върне 0. При това програмата ще направи някакъв брой стъпки, който не зависи от входните данни. Да означим този брой с C.

Когато $n > 1$ нещата са подобни, но което и от двете рекурсивни ивиквания в тялото на цикъла `while` да се изпълни, то ще предизвика ново рекурсивно извикване и т.н. Тъй като всеки от двата възможни интервала `[1,m-1]` и `[m+1,n]` ще бъде с дължина не надхвърляща $n/2$, сложността $t_{binsrch}(n)$ може да се представи със следното рекурентно отношение:

$$t_f(n) = \begin{cases} C, & n = 1 \\ t_f(n/2) + C, & n > 1 \end{cases}$$

Както се вижда, сложността на рекурсивни процедури се изразява по естествен начин с рекурентни отношения. Ще се спрем доста подробно на решаването на рекурентни отношения в следващата глава и ще отложим засега намирането на сложноста на алгоритъма за двоично сортиране. Любознателният читател, обаче, би могъл да напише итеративна версия на този популярен алгоритъм и да се опита да определи сложността ѝ с изученото до момента.

Пример 4. Като последен пример да разгледаме една версия на алгоритъма на Евклид за намиране на най-голям общ делител на две естествени числа $a \geq b > 0$. За целите на изложението отново ще разгледаме рекурсивна версия на алгоритъма, а ще оставим по-лесната за анализ итеративна версия за упражнение:

```
int gcd(int a,int b)
{
   int x,y,q;
   x=a;y=b;q=a%b;
   if(q==0) return b;
   if(y%q==0) return q;
   return gcd(q,y%q);
}
```

Ако подходим рутинно при определяне размера на входа на тази масова задача, ще се окаже, че всички нейни екземпляри имат размер на входните данни 2. Това очевидно е погрешен подход, защото не е трудно да се забележи върху няколко примера, че броят на стъпките на алгоритъма на Евклид нараства при нарастване на числата. Затова в този случай, за намиране на адекватна функция на сложност, трябва да изберем размер на входните данни $n = a$.

При $n \leq 2$ функцията винаги ще завърши с първия `return` и няма да направи рекурсивното изивикване. При това тя ще изпълни определен брой стъпки, да го означим с C_1, който не зависи от входните данни.

При $n > 2$ функцията ще направи някакъв брой стъпки C_2, който също не зависи от входните данни, и след това ще направи рекурсивното извикване. Интересно е, какъв ще бъде размерът q на новата задача при това извикване. Ще докажем че $\forall a \geq b > 0$, за остатъкът q от целочисленото деление a/b е в сила $q \leq a/2$. Действително, $\forall a \geq b > 0, a = pb + q, 0 \leq q < b$. Ако $b \leq a/2$ тогава $q < b \leq a/2$. Ако пък $b > a/2$, тогава $a = b + q, q = a - b < a - a/2 = a/2$.

Това ни позволява да се опитаме да немерим сложността на алгоритъма на Евклид с помощта на рекурентното отношение:

$$t_{gd}(n) = \begin{cases} C_1, & n \leq 2 \\ t_f(n/2) + C_2, & n > 1 \end{cases}$$

което практически не се отличава от полученото по-горе за алгоритъма за двоично търсене.

И така за нуждите на практиката е важно да се дефинират адекватно размер на входа и, когато сравняваме сложността на два алгоритъма за една и съща задача, да прилагаме и към двата един и същ подход.