СЛОЖНОСТ НА АЛГОРИТМИЧНИ ЗАДАЧИ

Алгоритмичните задачи, подобно на алгоритмите, имат сложност по време и по памет. Ако една задача е алгоритмично нерешима, за нея не се дефинира сложност. В противен случай има поне един алгоритъм за задачата. Тогава за нея съществуват безброй много алгоритми с произволно големи сложности по време и памет: към първоначалния алгоритъм добавяме код, който заделя ненужна памет и пише в нея единици. Количеството ненужни операции и памет определя сложността на такъв алгоритъм. Обратно, изобретяването на ефективен алгоритъм е значително по-трудно. Затова е съвсем естествено следното определение:

Сложност на алгоритмична задача е най-малката от сложностите на всички алгоритми, решаващи задачата. Записано с математически формули: $\widetilde{T}(n) = \min T(n)$, $\widetilde{M}(n) = \min M(n)$, където n е дължината на входа, $\widetilde{T}(n)$ и $\widetilde{M}(n)$ са сложностите на задачата по време и по памет, $T(n)$ и $M(n)$ са сложностите на алгоритмите по време и по памет съответно, а минимумите се взимат по всички алгоритми, решаващи задачата.

Това определение е продиктувано еднакво от практически и теоретически съображения. Изискванията на практиката са ясни: няма полза от бавни алгоритми с голям разход на памет. А теоретичните разсъждения от началото на темата показват, че сложността на алгоритмите, решаващи дадена задача, е неограничена отгоре; ето защо има смисъл да търсим най-малка, но не и най-голяма сложност.

Използването на минимум поставя въпроса за неговото съществуване. Възможно ли е да има безкрайна редица от алгоритми, решаващи една и съща задача, като всеки алгоритъм е с по-малка сложност от предишния в редицата? Нещо повече, някои функции са несравними по асимптотичен порядък. Възможно ли е да има два алгоритъма с несравними сложности, решаващи дадена задача по-ефективно от всички други алгоритми? В тези теоретични въпроси няма да навлизаме, тъй като са доста трудни, а нямат особено голямо значение за практиката. Рядко ще прилагаме даденото определение буквално, а когато правим това, то ще е само в задачи, в които съществуването на минимума е очевидно. По-често ще прилагаме не самото определение, а някои негови следствия.

Тъй като може да съществуват различни входни данни с една и съща дължина n, кои данни се подразбират в определението? Отговорът е, че не се подразбира нищо. Алгоритмичните задачи, както и алгоритмите, имат различни видове сложност: при най-добри и най-лоши входни данни, а също и средна сложност (тя е усреднена по всички входни данни с една и съща дължина).

Най-голямо значение за практиката има времевата сложност при най-лоши входни данни. Именно нея ще изследваме.

ГОРНИ ГРАНИЦИ НА СЛОЖНОСТТА НА АЛГОРИТМИЧНИ ЗАДАЧИ

От определението в началото на темата следва, че сложността на който и да е алгоритъм е горна граница за сложността на задачата, която той решава: $\widetilde{T}(n) \leq T(n)$, $\widetilde{M}(n) \leq M(n)$. Обикновено се интересуваме само от асимптотичния порядък на сложността; в такъв случай двете неравенства приемат вида: $\widetilde{T}(n) = \mathrm{O}\big(T(n)\big)$, $\widetilde{M}(n) = \mathrm{O}\big(M(n)\big)$.

Да разгледаме например времевата сложност при най-лоши входни данни на задачата за сортиране на масив с n елемента. Ако не познаваме нито един алгоритъм за тази задача, няма да можем да оценим сложността ѝ отгоре. След като научим обаче, че алгоритъмът сортиране чрез вмъкване работи за квадратично време, ще направим извод, че $\widetilde{T}(n) = \mathrm{O}\big(n^2\big)$. Ако знаем за пирамидалното сортиране, ще смъкнем горната граница до $\widetilde{T}(n) = \mathrm{O}\big(n \log n\big)$. Разбира се, действителната сложност на задачата не се променя, само оценката отгоре се мени с напредване на нашето познание. Обаче напредъкът не значи да надценяваме възможностите си. Човешката изобретателност е ограничена от обективната трудност на алгоритмичните задачи: не можем да създадем по-ефективен алгоритъм, отколкото допуска самата задача.

ДОЛНИ ГРАНИЦИ
НА ВРЕМЕВАТА СЛОЖНОСТ НА АЛГОРИТМИЧНИ ЗАДАЧИ

Долните граници на сложността на задачи се доказват значително по-трудно от горните. Горната граница е твърдение за съществуване на алгоритъм и доказването ѝ е сравнително лесно: достатъчно е да съставим ефективен алгоритъм.

Напротив, всяка долна граница е твърдение за несъществуване на ефективен алгоритъм, затова е трудно да бъде доказана. Не можем просто да посочим един неефективен алгоритъм; това не доказва нищо. Погледнато по друг начин, долната граница е твърдение за всеобщност: всеки алгоритъм за решаване на определена задача изразходва поне толкова време или памет. Ясно е, че не можем да изпробваме алгоритмите един по един, защото за всяка решима задача съществуват безброй алгоритми.

Следователно се нуждаем от специални методи за доказване на долни граници. Тези методи разчитат на теоретични познания. Долна граница може да се докаже само чрез разсъждение, обхващащо всички алгоритми за дадена задача, а тази всеобщност е постижима единствено чрез достатъчно разработена теория.

Оттук нататък ще се занимаваме само с един вид долни граници — за времева сложност при най-лоши входни данни. Тези долни граници се делят на тривиални и нетривиални.

Тривиални долни граници на времевата сложност:

- Тривиална долна граница по размера на входа: $\widetilde{T}(n) = \Omega(n)$. Тази граница е в сила само при задачи с неструктурирани входни данни, тоест прочитането на един елемент от входа не дава информация за другите елементи. Предполага се, че няма излишни входни данни. Алгоритъмът трябва да прочете всичките n елемента от входа, за което са нужни n стъпки. Оттук идва долната граница. Тя обаче не важи за структурирани входни данни. Например двоичното търсене притежава времева сложност $\Theta(\log\, n) = \mathrm{o}(n)$, което се дължи на факта, че входният масив е сортиран, следователно прочитането на произволен елемент от входа дава информация и за останалите елементи и тяхното прочитане може да стане излишно.

- Тривиална долна граница по размера на изхода: По порядък времевата сложност всякога е по-голяма или равна на размера на изхода. Тази оценка важи без никакво изключение: всеки алгоритъм изразходва единица време за изпращането на всеки бит към изхода.

- Тривиална долна граница по размера на паметта: $\widetilde{T}(n) = \Omega\Big(\widetilde{M}(n)\Big)$. Тази граница следва от такива съображения: по определение количеството памет в дясната страна е минимално, тоест всяка клетка от нея е нужна, следователно трябва да бъде поне инициализирана, така че алгоритъмът изразходва поне една единица време за всяка клетка от паметта. Съществуват изключения: някои алгоритми използват специална структура от данни за работа с неинициализирана памет.
  Забележка: Неравенството може да се обърне и да служи като горна граница за паметта по размера на времето: $\widetilde{M}(n) = \mathrm{O}\Big(\widetilde{T}(n)\Big)$.

Тривиалните граници се прилагат лесно, но чрез тях рядко се получава най-точната оценка. Сложността на повечето задачи надхвърля тривиалната долна граница и са нужни други способи.

Методи за доказване на нетривиални долни граници:
— рекурентни неравенства;
— дърво за взимане на решения;
— игра срещу противник;
— редукция.

При метода "игра срещу противник" алгоритъмът играе срещу генератора на входни данни. Генераторът е противник на алгоритъма, защото мерим сложността при най-лоши входни данни. Алгоритъмът се опитва да приключи играта по-скоро, а противникът му се опитва да го забави. Алгоритъмът няма печеливша стратегия $\iff$ противникът притежава печеливша стратегия. Дясната страна е по-удобна за работа: съществуването може да се докаже конструктивно.

ЗАДАЧИ

Задача 1. Да се намери времевата сложност на задачата за събиране на n числа.

Решение: Обхождаме числата и натрупваме сбора им в променлива, инициализирана с нула. Времето на този алгоритъм $T(n) = \Theta(n)$ е горна граница за времето на задачата: $\widetilde{T}(n) = \mathrm{O}(n)$. Сборът зависи от всяко събираемо и входните данни са неструктурирани, затова $\widetilde{T}(n) = \Omega(n)$. (тривиална долна граница по размера на входа). От двете граници следва, че $\widetilde{T}(n) = \Theta(n)$ при всякакви входни данни.

Задача 2. Може ли за полиномиално време да се отпечатат всички пермутации на n елемента?

Решение: Не. Размерът на изхода е $n!\,n = \Theta\Big((n+1)!\Big)$, следователно $\widetilde{T}(n) = \Omega\Big((n+1)!\Big)$ при всякакви входни данни.

Задача 3. Да се докаже, че търсенето на стойност key в сортиран числов масив $A\big[1\ldots n\big]$ чрез алгоритъм, основан на сравнения, има времева сложност $\Omega(\log n)$ при най-лоши данни.

Решение: Задачата може да се реши по различни начини. Както и да подходим, съществено е разсъжденията да се отнасят за всички възможни алгоритми, а не за един конкретен алгоритъм!

Първи начин: чрез дърво за взимане на решения.

Всяко сравнение може да се разглежда като въпрос с два възможни отговора — "да" и "не". Въпросът "Вярно ли е, че $key < A\big[k\big]$?" има отговори "Да, $key < A\big[k\big]$." и "Не, $key \geq A\big[k\big]$." В програмен код на такъв въпрос съответства разклонение **if — then — else**.

На всеки алгоритъм (при всяко n) съответства дърво за взимане на решения, което описва изпълнението на алгоритъма (при конкретната стойност на n). Във всеки връх без листата стои въпрос (сравнение) и излизат две ребра, съответстващи на двата отговора — "да" и "не". Следователно имаме двоично кореново дърво. В листата стоят възможните позиции на key, тоест числата от 1 до n, или специалното съобщение "key липсва в масива A". На всеки връх можем да съпоставим и едно подмножество на $\Big\{\,1\,;\,2\,;\,3\,;\,\ldots\,;\,n\,\Big\}$, а именно множеството от индексите на онези елементи на масива A, които биха могли да съдържат стойността key към текущия момент от изпълнението на алгоритъма. Ето защо в корена на двоичното дърво стои пълното множество $\Big\{\,1\,;\,2\,;\,3\,;\,\ldots\,;\,n\,\Big\}$.

Всяко сравнение изисква константно време, затова за цялото изпълнение на алгоритъма е нужно време, равно по порядък на броя на сравненията, т.е. броя на ребрата в пътя от корена до листото, съответстващо на входните данни. Времевата сложност $T(n)$ е времето за изпълнение в най-лошия случай, т.е. дължината h на най-дълъг път от корена до листата:

$$T(n) \asymp h\,.$$

Числото h се нарича височина на дървото. Дървото е двоично, значи има най-много 2^h листа. От друга страна, дървото притежава поне $n+1$ листа — целите числа от 1 до n включително, както и специалната стойност "key липсва в масива A". Ето защо

$$2^h \geq n+1, \;\text{ тоест }\; h \geq \log_2(n+1).$$

Оттук следва редица от равенства и неравенства:

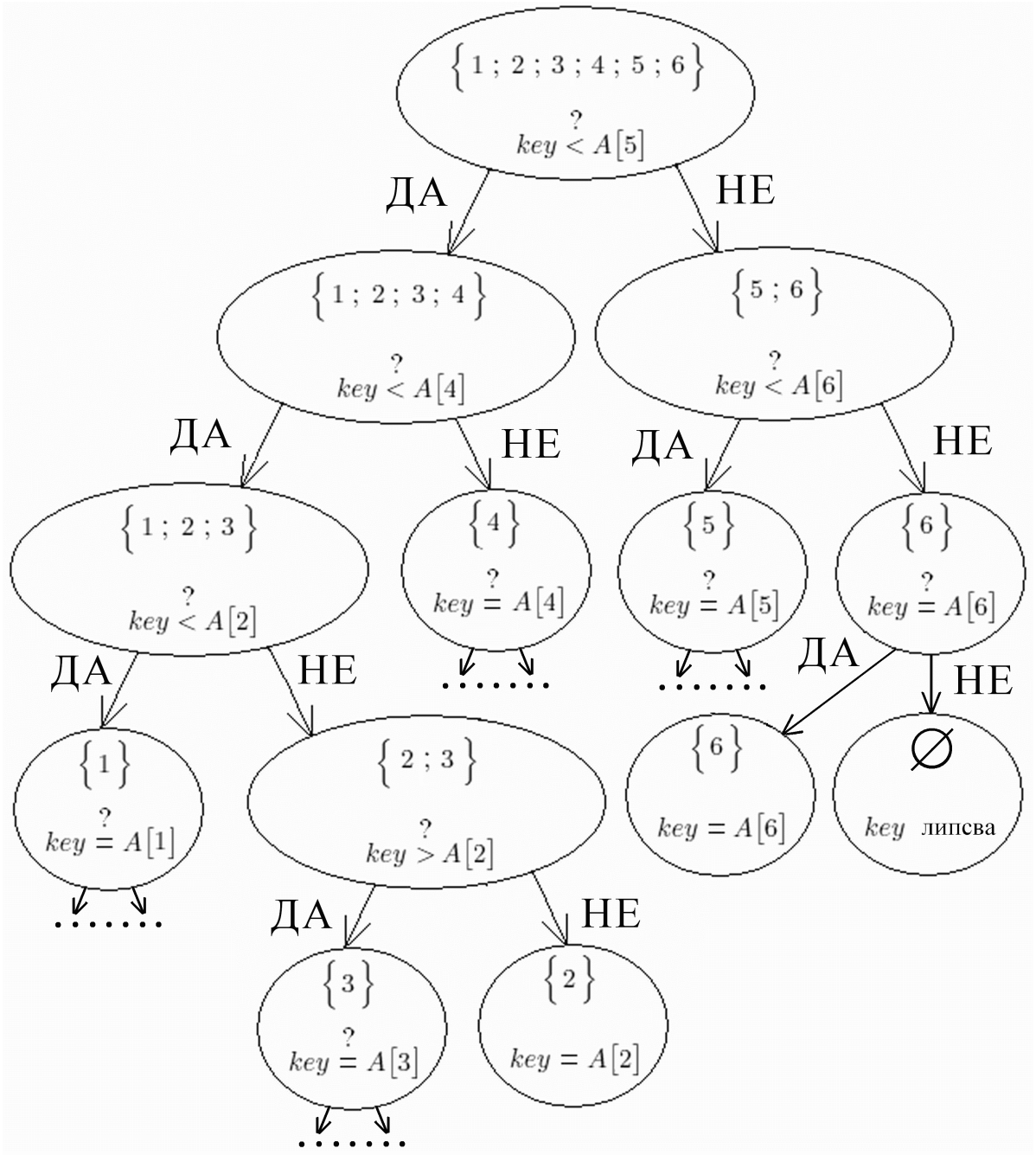
$$T(n) \asymp h \;\geq\; \log_2(n+1) \asymp \log\, n\,.$$

Като изоставим междинните членове, получаваме асимптотичното неравенство

$$T(n) \succeq \log\, n\,,$$

което трябваше да се докаже.

П р и м е р : Да разгледаме изпълнението на един конкретен алгоритъм при $n = 6$. За да подчертаем, че в конкретния алгоритъм няма нищо специално, съзнателно избираме алгоритъм, различен от стандартното двоично търсене.



Многоточията заместват разклонения, които не са изобразени, защото са подобни на други. Някои листа съдържат множество от вида $\{k\}$ и равенство $key = A[k]$ без въпросителна. Това е не сравнение, а констатация: ключът key е намерен на k-тата позиция в масива A. Други листа съдържат празното множество и констатацията "key липсва".

Възможно е малко по-различно тълкуване на сравненията: можем да си представяме всяко сравнение като въпрос от вида "Какъв е знакът между числата x и y ?", а отговорът е един от трите знака "по-голямо", "по-малко" и "равно". Тогава се получава троично дърво и навсякъде в решението основата 2 на степените и на логаритмите се заменя с 3.

Втори начин: чрез игра срещу противник.

Противник е генераторът на входните данни. При този метод трябва да си представяме, че елементите на входния масив не се задават предварително, а в момента на прочитането.

Разглеждаме множество $M \subseteq \Big\{ 1 \, ; \, 2 \, ; \, 3 \, ; \, \ldots \, ; \, n \Big\}$, което се променя по време на играта. M е множеството от индексите на онези елементи на масива A, които към текущия момент биха могли да съдържат стойността key. Ето защо играта започва с $M = \Big\{ 1 \, ; \, 2 \, ; \, 3 \, ; \, \ldots \, ; \, n \Big\}$. Алгоритъмът и неговият противник се редуват, като първият ход е на алгоритъма.

Всяко сравнение разбива множеството M на две подмножества X и Y; едното съдържа индексите на елементите, удовлетворяващи сравнението, а другото — индексите на елементите, които не го удовлетворяват. "Разбиване" означава, че $X \cap Y = \varnothing$ (закон за непротиворечието) и $X \cup Y = M$ (закон за изключеното трето). Позволяваме участието на празни подмножества; такива разбивания съответстват на излишни сравнения (чийто резултат е известен отнапред); позволяваме ги за общност на разсъжденията — за да обхванем всички възможни алгоритми, включително някои, които са очевидно неоптимални.

Изборът на сравнение принадлежи на алгоритъма, тоест той избира някакво разбиване на текущото множество M.

Резултатът от сравнението зависи от входните данни, т.е. от противника. С други думи, противникът избира X или Y да бъде новото множество M.

Алгоритъмът приключва работа, щом намери търсената стойност или установи липсата ѝ. Тогава множеството M остава с един или нула елемента.

Правила на играта:

1) Играта започва с множеството $M = \Big\{ 1 \, ; \, 2 \, ; \, 3 \, ; \, \ldots \, ; \, n \Big\}$.
2) Двамата играчи — алгоритъмът и неговият противник — се редуват.
3) Алгоритъмът играе първи.
4) На всеки свой ход алгоритъмът избира две множества X и Y, такива че $X \cap Y = \varnothing$ и $X \cup Y = M$.
5) На всеки свой ход противникът избира или X, или Y да бъде новото M.
6) Играта приключва, когато множеството M остане с един елемент или без елементи.

Целта на алгоритъма е играта да свърши възможно най-бързо.
Целта на противника е да удължи играта колкото може повече.

Забележки по правилата:

— В правило № 6 се допуска играта да завърши и тогава, когато множеството M остане с един елемент. Тогава са възможни два случая: алгоритъмът може да е сигурен, че този елемент е търсеният индекс (т.е. намерил е търсената стойност в масива), но може и да не е сигурен (защото е възможно масивът изобщо да не съдържа търсената стойност). Във втория случай се налага едно допълнително сравнение, а такова не се предвижда от правилата на играта. Това не е проблем, защото всяко сравнение изисква константно време, което може да бъде пренебрегнато, щом правим оценка по порядък.

— В правило № 4 се допуска произволно разбиване, което е много повече, отколкото може да се постигне чрез едно сравнение. (Например няма как чрез едно сравнение да се разделят числата от 1 до 10 на четни и нечетни.) Това, че даваме на алгоритъма повече възможности, не е проблем, а е предимство, защото така задачата се решава за по-широк клас от алгоритми. Тук няма да уточняваме какво представлява споменатият по-широк клас, а ще се ограничим с очевидната забележка, че ако докажем, че дори той не съдържа достатъчно бърз алгоритъм, то заключението за несъществуването на такъв алгоритъм в първоначалния (по-тесния) клас ще последва автоматично.

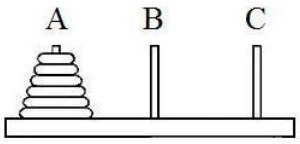
Тъй като всяко сравнение изразходва константно време, то общото време на алгоритъма е правопропорционално и затова равно по порядък на броя на сравненията, т.е. ходовете в играта. Понеже се интересуваме от времевата сложност в най-лошия случай, то трябва да покажем, че за всяко n съществуват входни данни, които изискват $\Omega(\log n)$ сравнения. Това е равносилно на съществуването на стратегия на противника, позволяваща му да изиграе $\Omega(\log n)$ хода.

Такава стратегия се построява лесно: понеже множеството M намалява по време на играта, а целта на противника е играта да продължи максимално дълго, то той трябва да се постарае да забави намаляването на множеството M. Затова противникът трябва всеки път да избира по-голямото от множествата X и Y (ако са еднакво големи, няма значение кое от тях ще избере).

Тази стратегия гарантира, че на всеки ход множеството M намалява най-много два пъти. Следователно за два хода M намалява най-много четири пъти, за три хода — осем пъти и т.н. По индукция следва, че за k хода множеството M намалява най-много 2^k пъти. Като следва описаната стратегия, противникът никога не избира празно множество, така че играта завършва, когато M остане с един елемент, т.е. намалее n пъти. Ако това стане за k хода, то $n \leq 2^k$, следователно $k \geq \log_2 n$, което трябваше да се докаже.

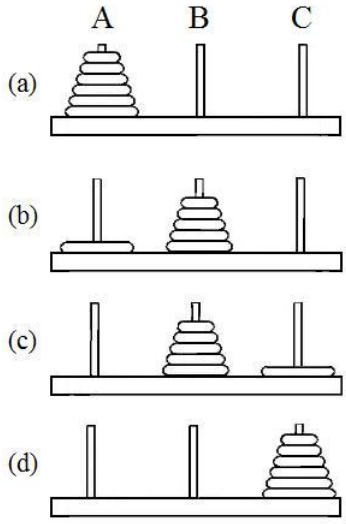
И тъй, установихме, че ако се решава чрез сравняване, разглежданата алгоритмична задача има времева сложност $\widetilde{T}(n) = \Omega(\log n)$ при най-лоши входни данни. От друга страна, знаем, че стандартният алгоритъм за двоично търсене решава поставената задача за време $\Theta(\log n)$, откъдето пък получаваме горната граница $\widetilde{T}(n) = \mathrm{O}(\log n)$ за времевата сложност на задачата, отново при най-лоши входни данни. Понеже горната и долната граница съвпадат по порядък, следва, че сме намерили точния порядък на времевата сложност при най-лоши входни данни на задачата за търсене по ключ в сортиран масив: $\widetilde{T}(n) = \Theta(\log n)$ сравнения.

Задача 4. Разглеждаме играта *ханойска кула*. Върху първия от три отвесни пръта са нанизани n диска, чиито размери намаляват отдолу нагоре. Целта на играта е да преместим всички дискове от първия на третия прът с най-малко ходове. На всеки ход местим по един диск и нямаме право да слагаме по-голям върху по-малък диск.



Решение: Да означим прътовете с A, B и C. Отначало кулата от дискове се намира върху A. Ще я преместим върху C по следния начин:

1) Като използваме C в ролята на помощен прът, преместваме горните $n-1$ диска от A на B.
2) Преместваме най-големия диск от A върху C.
3) Като използваме A в ролята на помощен прът, преместваме малките $n-1$ диска от B на C.



Първата и третата стъпка на този алгоритъм се изпълняват рекурсивно. Дъното на рекурсията е при $n = 0$; тогава алгоритъмът не прави нищо.

Нека $T(n)$ е броят на ходовете на алгоритъма. Тогава $T(n) = 2T(n-1) + 1$ за всяко цяло $n > 0$ и $T(0) = 0$. Оттук с характеристично уравнение намираме $T(n) = 2^n - 1$.

Сложността на алгоритъма е горна граница за сложността на задачата, която той решава. Затова $\widetilde{T}(n) \leq 2^n - 1$.

Долната граница се доказва по-трудно, с по-отвлечени разсъждения, защото е нужно разсъжденията да обхванат всички алгоритми. Нека $\widetilde{T}(n)$ е най-малкият възможен брой ходове. В задачата за ханойската кула съществуването на най-малък брой ходове е сигурно отнапред, тъй като във всяко непразно множество от цели неотрицателни числа има най-малко число. Става дума за множеството от бройките ходове на всички възможни алгоритми за задачата: бройките ходове са цели неотрицателни числа, а множеството, съставено от бройките ходове, е непразно, защото съществува поне един алгоритъм, решаващ задачата, както се уверихме.

Да изберем един произволен алгоритъм за задачата *ханойска кула* и да го фиксираме. Нека $T(n)$ е броят на ходовете на избрания алгоритъм.

Всеки диск, включително най-големият, се премества поне веднъж. Но той може да се мести само при много специални положения в играта. Прътът, върху който отива най-големият диск, трябва да бъде празен, защото не е позволено да се слага по-голям върху по-малък диск. А прътът, от който взимаме най-големия диск, не бива да съдържа други дискове, защото поради същото правило те ще затискат най-големия диск, а нямаме право да преместваме два или повече диска наведнъж. Ето защо останалите $n-1$ диска трябва да образуват кула върху прътa, който не е засегнат от движението на най-големия диск.

И така, най-големият диск се премества краен брой пъти, но поне веднъж. Следователно има първо и последно местене на най-големия диск (те съвпадат, ако той се мести само веднъж). Ходовете на произволно избрания алгоритъм се разделят на три етапа:
— преди първото местене на най-големия диск;
— от първото до последното местене на най-големия диск;
— след последното местене на най-големия диск.
Ще пресметнем най-малкия възможен брой ходове за всеки етап.

Вторият етап съдържа поне един ход — поне едно преместване на най-големия диск (то може да е едновременно първо и последно местене на този диск).

Първото местене на най-големия диск е от A на B или от A на C. Точно преди този ход кулата от останалите $n-1$ диска се намира върху C или B съответно. В началото на играта тази кула се е намирала върху A. Първият етап съдържа поне толкова хода, колкото са нужни за преместването ѝ от A върху друг прът, тоест най-малко $\widetilde{T}(n-1)$ хода.

Последното местене на най-големия диск е от A на C или от B на C. Точно преди това кулата от останалите $n-1$ диска се намира върху B или A съответно. Щом в края на играта тази кула се намира върху C, то третият етап съдържа поне толкова хода, колкото са нужни за преместването ѝ върху C от друг прът, тоест най-малко $\widetilde{T}(n-1)$ хода.

Сборът от дължините на трите етапа е оценка отдолу за броя на ходовете на алгоритъма:

$$T(n) \geq 2\,\widetilde{T}(n-1)+1 \text{ за всяко цяло } n>0.$$

Макар и фиксиран, алгоритъмът беше избран произволно, следователно неравенството важи за всеки алгоритъм, преместващ ханойската кула, в това число за алгоритъма с най-малко ходове (може да съществува повече от един такъв алгоритъм). Прилагаме доказаното неравенство за някой най-бърз алгоритъм, като заместваме $T(n)=\widetilde{T}(n)$ в лявата страна на неравенството:

$$\widetilde{T}(n) \geq 2\,\widetilde{T}(n-1)+1 \text{ за всяко цяло } n>0.$$

Оттук с помощта на очевидното равенство $\widetilde{T}(0)=0$ извеждаме по индукция долната граница

$$\widetilde{T}(n) \geq 2^n-1 \text{ за всяко цяло } n\geq 0.$$

Горната и долната граница съвпадат, откъдето заключаваме, че

$$\widetilde{T}(n) = 2^n-1 \text{ за всяко цяло } n\geq 0.$$

Това е най-малкият брой ходове, с които може да бъде преместена ханойска кула от n диска, следователно алгоритъмът, описан в началото на решението, е най-бърз. По-бърз не съществува, обаче не е изключено да има друг също толкова бърз алгоритъм.

В решението използвахме рекурентно неравенство. Съответното нерекурентно неравенство (тоест долната граница) е негово неравносилно следствие: обратната импликация не е вярна. Затова долната граница не се смята за "решение" на рекурентното неравенство.

Някои важни задачи
с времева сложност $\Theta(n \log n)$
при най-лоши входни данни

С помощта на редукция можем да сведем една алгоритмична задача до друга. По този начин се образуват класове от алгоритмични задачи с равна сложност, които се свеждат една до друга. Един от тези класове е особено важен: той съдържа задачата за сортиране и някои други задачи. Всички те, ако се решават чрез сравняване, дори по най-бърз начин, изискват време $\Theta(n \log n)$ при най-лоши входни данни; по-точно, нужни са $\Theta(n \log n)$ сравнения при най-лоши данни, а броят на другите операции може да бъде по-малък.

Има различни видове редукция; броят им зависи от основата на делението. За нашите цели е удобно да разглеждаме два вида редукция: на алгоритъма и на данните. Първата е по-трудна и се използва по-рядко. Втората се състои от две части: редукция на входа и редукция на изхода, които могат да присъстват едновременно или поотделно в решението на една задача.

Задача 1. Алгоритмичната задача за сортиране на n елемента, ако се решава чрез сравняване, изисква $\Omega(n \log n)$ сравнения между елементите при най-лоши входни данни. Следователно нейната времева сложност е $\Omega(n \log n)$, а тъй като има алгоритми за сортиране чрез сравняване, работещи за време $O(n \log n)$, то времевата сложност на задачата в крайна сметка е $\Theta(n \log n)$ при най-лоши входни данни, ако се решава чрез сравняване. Типът на елементите се предполага нареден (без линейна наредба задачата няма смисъл) и достатъчно голям (трябва да съдържа $\Omega(n)$ различни стойности, иначе долната граница може да не важи).

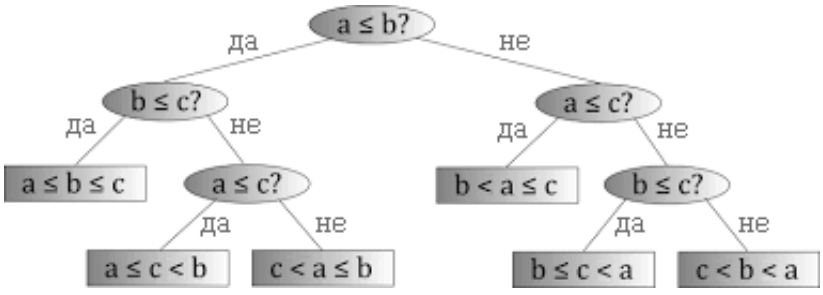
Решение: Преди същинското доказателство нека обърнем внимание на някои подробности.

Първо, споменатата долна граница $\Omega(n \log n)$ важи само за сортиране чрез сравняване. Сортирането чрез трансформация може да бъде по-бързо при определени условия.

Второ, долната граница $\Omega(n \log n)$ важи не просто за времето на сортировката като цяло, а по-конкретно за броя на сравненията. Броят на другите операции може да бъде по-малък; например сортирането чрез пряк избор прави $O(n)$ размествания. За практически приложения това най-често няма значение (обикновено се интересуваме от общото време на алгоритъма), обаче е важно за построяването на теорията: някои доказателства зависят от тази подробност.

Трето, долната граница $\Omega(n \log n)$ се отнася само за най-лоши входни данни. Тя не пречи алгоритъмът да бъде по-бърз в отделни случаи. Нещо повече, всеки алгоритъм притежава собствени най-лоши входни данни, тоест кои входни данни са най-лоши, зависи от алгоритъма. Ето защо твърдението, което трябва да докажем, може да бъде формулирано по следния начин: за всяко сортиране чрез сравняване има входни данни, изискващи $\Omega(n \log n)$ сравнения.

Доказателство: Задачата е първата от този тип, затова не можем да я решим с редукция (не разполагаме с друга задача). Вместо това ще използваме дърво за взимане на решения.



На чертежа е изобразено едно дърво за взимане на решения. То показва начина на работа на някоя конкретна сортировка върху вход от $n = 3$ елемента: a, b и c. Алгоритъмът започва от корена на дървото, изпълнява сравнението, записано там, и според резултата от проверката преминава по едно от излизащите ребра. Всеки път, когато се озове във връх, който не е листо, алгоритъмът изпълнява предписаното сравнение и излиза по реброто с надпис "да" или "не" според резултата от проверката. Когато стигне до листо, алгоритъмът подрежда елементите в реда, указан върху листото.

На всяка наредена двойка от алгоритъм за сортиране чрез сравняване и дължина n на входа съответства едно дърво за взимане на решения. Друга дължина на входа поражда друго дърво. Промяна в алгоритъма най-често води до промяна в дървото, но все пак това не е задължително: може да се случи два различни алгоритъма да имат едно и също дърво за взимане на решения при някои стойности на n.

Ако променим стойностите на входа, запазвайки дължината му и алгоритъма, ще установим, че дървото остава същото, може да се промени евентуално пътят, който алгоритъмът изминава, и листото, до което стига. Броят на сравненията е равен на броя на ребрата (дължината на пътя), следователно най-лоши входни данни съответстват на най-дълъг път от корена до листата. С други думи, броят на сравненията в най-лошия случай е равен на височината h на дървото. Тъй като дървото е двоично, то има не повече от 2^h листа.

От друга страна, на всяка пермутация на входните данни съответства различно листо, иначе алгоритъмът, стигайки до листо, не би знаел как да подреди елементите. Следователно листата са поне колкото пермутациите на n елемента. Тъй като типът на елементите съдържа $k = \Omega(n)$ различни стойности, то можем да изберем входни данни с $\ell = \min\{k\,;\,n\} = \Theta(n)$ различни стойности; тогава броят на пермутациите е поне $\ell\,!$.

От съотношенията

$$\ell\,! \le \text{брой на пермутациите на входните данни} \le \text{брой на листата} \le 2^h$$

следва неравенството

$$\ell\,! \le 2^h.$$

Като го решим относно h, получаваме долната граница

$$h \ge \log_2(\ell\,!) \asymp \ell \log \ell \asymp n \log n.$$

Изоставяйки средните членове, стигаме до оценката

$$h \succsim n \log n.$$

Понеже h съвпада с броя на сравненията при най-лоши входни данни, то този брой е $\Omega(n \log n)$, което трябваше да се докаже.

Задача 2. Входът на алгоритмичната задача Липса на повторения е масив $A[1\ldots n]$, чийто базов тип има линейна наредба и поне n различни стойности. Масивът се проверява за повторения. Връща се логическа стойност: "истина" — ако масивът не съдържа повторения; "лъжа" — в противен случай. Да се докаже, че ако задачата се решава чрез сравняване, тя изисква $\Omega(n \log n)$ сравнения между елементите при най-лоши входни данни. Следователно времевата ѝ сложност е $\Omega(n \log n)$, а тъй като има алгоритми, които я решават чрез сравняване за време $\mathrm{O}(n \log n)$ — например пирамидално сортиране, последвано от сравнения само между двойките съседни елементи на сортирания масив, — то времевата сложност на задачата се оказва $\Theta(n \log n)$ при най-лоши входни данни, ако се решава чрез сравняване.

Решение: Трите забележки от началото на решението на задача 1 важат също и за задача 2 с подразбиращите се изменения.

Трябва да докажем, че за всеки алгоритъм, решаващ задачата Липса на повторения чрез сравняване на елементите на масива $A[1\ldots n]$, съществуват входни данни, които изискват $\Omega(n \log n)$ сравнения. Твърдението се доказва с помощта на редукция от задачата за сортиране. Ще използваме редукция на алгоритъма.

Избираме един произволен алгоритъм, който решава задачата Липса на повторения, сравнявайки елементите на масива $A[1\ldots n]$. Нека го наречем Алгоритъм 1. Разглеждаме го като бяла (прозрачна) кутия, променяме го и получаваме нов код, да го наречем Алгоритъм 2. Промените се състоят в следното.

В началото на алгоритъма копираме $A[1\ldots n]$ в нов масив $B[1\ldots n]$, в който добавяме сведения за първоначалните индекси на елементите във входния масив, тоест $B[k] = \langle A[k], k\rangle$. Всички други срещания на масива A в програмния код заменяме с B. Типа на всяка променлива, който съвпада с базовия тип на масива A, заменяме с базовия тип на масива B.

Предполагаме, че Алгоритъм 1 прави всички сравнения на величини от базовия тип на A чрез функция CompareA(x, y), връщаща -1, $+1$ и 0 при $x < y$, $x > y$ и $x = y$ съответно. При прехода от Алгоритъм 1 към Алгоритъм 2 заменяме всички извиквания на CompareA с нова функция CompareB, която сравнява стойности от базовия тип на B по следната схема: сравнява първите членове на наредените двойки по стария начин, а ако те се окажат равни, сравнява вторите членове. Освен това функцията CompareB води дневник за сравненията.

Въпреки че стойностите, сравнявани от CompareB, по принцип може да се окажат равни, това никога не се случва с елементи на масива B, защото всички те са различни по построение (различни са поне вторите елементи на наредените двойки — индексите от масива A). Ето защо алгоритъмът след направените промени винаги ще връща "истина", тоест върнатата стойност ще бъде безполезна. Затова правим промяна: Алгоритъм 2 ще връща дневника на сравненията.

```
Алгоритъм1 (A[1...n]: масив с базов тип T)
 1)  // T е нареден тип, т.е. върху него
 2)  // е дефинирана линейна наредба.
 3)  // Алгоритъм 1 проверява масива A
 4)  // за липса на повторения.
 5)  // Ако A не съдържа повторения,
 6)  // то Алгоритъм 1 връща "истина".
 7)  // Ако A съдържа повторения,
 8)  // то Алгоритъм 1 връща "лъжа".
 9)  // Начало на Алгоритъм 1:
10)  .....
11)  x, y: променливи от тип T.
12)  i, j, ℓ, r: цели числа.
13)  .....
14)  x ← A[ℓ]
15)  y ← A[r]
16)  if CompareA(x, y) = 0
17)      return false
18)  swap(A[i], A[j])  // Разменя елементи.
19)  .....
20)  if условие
21)      return true
22)  i ← j + 3
23)  .....
24)  return логически израз

CompareA (u, v: T)
 1)  // Реализира наредбата върху типа T.
 2)  .....
```

```
Алгоритъм2 (A[1...n]: масив с базов тип T)
 1)  // Имитира Алгоритъм 1.
 2)  Тип T2 = ⟨value: T; index: 1...n⟩.
 3)  B[1...n]: масив с базов тип T2.
 4)  for k ← 1 to n do
 5)      B[k].value ← A[k]
 6)      B[k].index ← k
 7)  Log: списък с резултати от сравнения.
 8)  Log ← празен списък.
 9)  // Начало на Алгоритъм 1:
10)  .....
11)  x, y: променливи от тип T2.
12)  i, j, ℓ, r: цели числа.
13)  .....
14)  x ← B[ℓ]
15)  y ← B[r]
16)  if CompareB(x, y, Log) = 0
17)      return Log
18)  swap(B[i], B[j])  // Разменя елементи.
19)  .....
20)  if условие
21)      return Log
22)  i ← j + 3
23)  .....
24)  return Log

CompareA (u, v: T)
 1)  // Реализира наредбата върху типа T.
 2)  .....

CompareB (s, t: T2; Log: списък)
 1)  q ← CompareA(s.value, t.value)
 2)  if q = 0
 3)      if s.index > t.index
 4)          q ← +1
 5)      else if s.index < t.index
 6)          q ← −1
 7)  Log.add(⟨s, t, q⟩)  // Запис в дневника.
 8)  return q
```

Когато масивът $A[1 \ldots n]$ не съдържа повторения, двата алгоритъма правят еднакъв брой сравнения между елементи на A, защото Алгоритъм 2 точно имитира работата на Алгоритъм 1.

Когато масивът $A[1 \ldots n]$ съдържа повторения, броят на сравненията може да е различен, защото Алгоритъм 2 имитира Алгоритъм 1 неточно: равните елементи в масива A се отчитат като равни от Алгоритъм 1, но като различни от Алгоритъм 2 (CompareB смята за по-малък елемента с по-малък индекс в A). С изключение на това различие всички останали сравнения имат еднакъв резултат в двата алгоритъма. Споменатата разлика не оказва съществено влияние върху най-лошия случай (който и да е той), защото се свежда до внасяне на малки изменения във входните данни.

Пример: Нека Алгоритъм 1 прави c_1 сравнения при входни данни $A_1 = (5\,;\,8\,;\,5)$ и c_2 сравнения при входни данни $A_2 = (5{,}001\,;\,8{,}002\,;\,5{,}003)$, където добавените дробни части имитират прибавянето на информация за индекса на елемента (замяната на типа T с типа T2). Масивът A_1 съдържа повторения, а A_2 — не. Следователно Алгоритъм 2 при входни данни A_2 прави c_2 сравнения, защото при липса на повторения имитира работата на Алгоритъм 1 точно. Алгоритъм 2 при входни данни A_1 прави пак c_2 сравнения, защото A_1 и A_2 изглеждат еднакво за Алгоритъм 2: макар че стойностите на елементите са различни, подредбата им е една и съща от гледна точка на функцията CompareB — първи, трети, втори, — а подредбата на елементите е всичко, което интересува Алгоритъм 2, тъй като той работи чрез сравняване на елементите, но не и чрез други операции върху тях, например аритметични (масивът може да не е числов).

Разсъжденията от примера важат и в общия случай: ако Алгоритъм 1 прави c_1 сравнения върху някой масив с повторения, то Алгоритъм 2 извършва c_2 сравнения върху същия масив, където c_2 е броят на сравненията, които Алгоритъм 1 прави върху друг масив (без повторения). Масив без повторения съществува, защото типът T съдържа поне n различни стойности.

Да означим с M_1 и M_2 множествата от бройките сравнения, които се извършват съответно от Алгоритъм 1 и Алгоритъм 2 върху всички възможни комплекти от входни данни. Възможно е да съществуват безброй много такива комплекти, обаче възможните подредби на елементите им са краен брой при всяко n, а понеже двата алгоритъма работят чрез сравняване на елементи, то има краен брой начини за реда на изпълнение на командите на алгоритмите. Следователно M_1 и M_2 са крайни множества, съставени от цели неотрицателни числа. Затова във всяко от тях съществува най-голямо число. Нека $m_1 = \max M_1$ и $m_2 = \max M_2$ са най-големите числа в двете множества.

M_2 се получава от M_1 чрез замяната на едни числа от M_1 с други числа от M_1 (да кажем, числото c_1 се заменя с числото c_2 в примера по-горе). Тъй като не работим с мултимножества, а с обикновени множества, то следва, че M_2 се получава от M_1 чрез премахване на елементи. Например с изтриването на c_1 от множеството

$$M_1 = \{c_1\,;\,c_2\,;\,c_3\,;\,c_4\,;\,\ldots\}$$

се получава множеството

$$M_2 = \{c_2\,;\,c_3\,;\,c_4\,;\,\ldots\}.$$

При премахване на елементи резултатът винаги е подмножество на първоначалното множество, тоест $M_2 \subseteq M_1$. От $m_2 \in M_2$ следва, че $m_2 \in M_1$, и от определението на m_1 заключаваме, че за най-големите числа в множествата важи неравенството $m_1 \geq m_2$: при изтриване на числа най-голямото число намалява или остава същото, но не нараства.

Извод: Алгоритъм 1 прави не по-малко сравнения от Алгоритъм 2 в най-лошия случай. (Само че най-лошите входни данни може да са различни за двата алгоритъма.)

За да решим задачата, трябва да докажем, че Алгоритъм 1 извършва $\Omega(n \log n)$ сравнения при най-лоши входни данни. От направения извод следва, че за целта е достатъчно да проверим, че Алгоритъм 2 прави $\Omega(n \log n)$ сравнения при най-лоши входни данни. Затова оттук нататък съсредоточаваме вниманието си върху Алгоритъм 2.

Алгоритъм 2 обработва масива $B[1 \ldots n]$, който по построение не съдържа повторения, но алгоритъмът не знае това отнапред, а трябва да го установи чрез сравняване на елементите.

Понеже масивът $B[1 \ldots n]$ не съдържа повторения, той може да бъде подреден (сортиран) по единствен начин относно наредбата в T2, зададена чрез програмната функция CompareB (лексикографска наредба). Подредбата на B поражда съответна подредба и на масива $A[1 \ldots n]$, който се оказва правилно подреден (сортиран) относно релацията на наредба, определена над T чрез програмната функция CompareA. Масивът A може да съдържа повторения, следователно той може да притежава няколко правилни подредби, обаче само една от всички тях е устойчива и тъкмо тя се поражда от съответната подредба на масива B. Устойчивостта ѝ следва от връзката между CompareA и CompareB: от два елемента на масива $A[1 \ldots n]$, равни относно CompareA, функцията CompareB поставя по-напред онзи, който отначало има по-малък индекс в масива A. При сортиран масив всеки елемент с изключение на крайните има точно два съседни елемента; крайните елементи имат по един съсед. Това е съседство по големина, не по начална позиция.

Нека x и y са два елемента на B. Може ли Алгоритъм 2 да установи, че x и y са различни, без да ги сравнява пряко? Понякога може. Ако например е установил вече, че $x < t$ и $t < y$, следва, че $x < y$, значи $x \neq y$. (Знакът "по-малко" означава строгата линейна наредба над T2, породена от програмната функция CompareB.)

Но ако елементите x и y са съседни по големина, описаният случай не може да възникне, защото сравненията им с всички други елементи на B са еднопосочни: ако $z < x$, то $z < y$; ако $x < w$, то $y < w$. Ето защо x и y са между z и w, но не следва нито $x = y$, нито $x \neq y$. Никоя от двете възможности не може да бъде отхвърлена.

Тук има тънкост: в зависимост от стойностите и типа на z и w понякога следва, че $x = y$. Ако например знаем, че x и y са цели числа между 3 и 5, то със сигурност е вярно, че $x = y = 4$. Обаче такова нещо не може да се случи при работата на Алгоритъм 2. Това твърдение следва от начина на построяване на масива B. Ние (за разлика от Алгоритъм 2) отнапред сме сигурни, че в масива B няма повторения. Ето защо за два негови елемента x и y, съседни по големина, не може да следва (тоест да бъде доказано), че $x = y$, въз основа на някакви други сравнения, защото няма как от верни предпоставки да бъде направен неверен извод.

Алгоритъм 2 не може също да заключи, че $x \neq y$, защото равенството $x = y$ е съвместимо с неравенствата $z < x$, $z < y$, $x < w$ и $y < w$: петте съотношения са удовлетворими съвкупно. Равенството $x = y$ е несъвместимо с построението на масива B (той не съдържа повторения), но Алгоритъм 2 не използва този факт, а обработвайки B, имитира работата на Алгоритъм 1, който, понеже е коректен, не предполага липса на повторения в масива A (такива може да има).

И така, Алгоритъм 2 не може да определи дали $x = y$, или $x \neq y$, само чрез сравнения между x и y, от една страна, и останалите елементи на масива B, от друга страна. Затова Алгоритъм 2 е принуден да сравни x и y пряко, тоест чрез обръщение CompareB(x, y, Log).

Казаното важи за произволни два елемента на B, съседни по големина: за да установи, че те са различни, Алгоритъм 2 трябва да ги сравни пряко (тоест един с друг).

Понеже Алгоритъм 2 води дневник на сравненията, то в края на работата на алгоритъма дневникът ще съдържа всички сравнения между съседни по големина елементи:

$$b_1 < b_2 < b_3 < \ldots < b_n \,,$$

където b_k е k-тият най-малък елемент на масива B. Разбира се, дневникът може да съдържа и други сравнения, тъй като алгоритъмът не знае отнапред кои елементи са съседни по големина и кои сравнения са излишни. Нещо повече, дневникът може да съдържа многократни сравнения между едни и същи елементи; това е така, защото Алгоритъм 2 използва дневника само за запис, но не и за четене, тоест може да не знае, че прави някое сравнение за втори път. Иначе казано, в дневника може да има много излишна информация, но това не е пречка за нашите разсъждения. Важното е, че дневникът съдържа всички сравнения между съседните по големина елементи, което е достатъчно за сортирането на масива B, а оттам — и на A.

Следователно можем да съставим алгоритъм за сортиране, който използва Алгоритъм 2. Схематично записана, сортировката изглежда по следния начин:

Сортиране($A[1 \ldots n]$: масив с базов тип T)
1) Log $\leftarrow$ Алгоритъм2($A[1 \ldots n]$)
2) $A[1 \ldots n]$ $\leftarrow$ Анализ_на_дневника(Log)

Тук няма нужда да описваме процедурата за анализ на дневника Log. Най-важното е, че тя не прави повече сравнения между елементи на масива: за нея масивът е изход, а не вход; освен това сравненията, извършени от Алгоритъм 2, са достатъчни за сортирането на масива.

Затова Алгоритъм 2 и сортировката правят еднакъв брой сравнения в най-лошия случай. За сортировката броят на сравненията е $\Omega(n \log n)$ според втората забележка към задача 1. Това е също броят на сравненията, извършени от Алгоритъм 2, което трябваше да се докаже.

От последното разсъждение става ясно значението на втората забележка към задача 1. Ако бяхме отнесли долната граница $\Omega(n \log n)$ към цялото време на алгоритъма за сортиране вместо към броя на сравненията, щеше да се наложи да извадим времето за анализ на дневника. Разликата (т.е. времето на Алгоритъм 2) би могла да бъде от малък порядък — не $\Omega(n \log n)$. Всъщност това не може да се случи, но няма как да го докажем по този начин, защото не знаем времето за анализ на дневника, а дори и да го знаехме, то можеше да не ни свърши работа, ако се окажеше например от порядък $\Omega(n \log n)$: порядъкът на разликата би бил неопределен.

Когато броим сравненията, вместо да мерим общото време, не възникват такива проблеми. Понеже анализът на дневника не сравнява елементите на масива, няма какво да вадим.

Забележка: В условието на задачата е скициран един алгоритъм, основан на сравняване, който я решава за време $\mathrm{O}(n \log n)$. Това обаче е възможно само защото базовият тип е нареден, тоест можем да извършваме сравнения от типа "по-малко" и "по-голямо". В противен случай можем да правим само сравнения от типа "равно" и "различно", затова времевата сложност става квадратична при най-лоши входни данни. Горната граница следва от пълното изчерпване (сравняват се всеки два елемента), а долната граница се доказва така: ако няма повторения, то единственият начин да установим това е да сравним всеки два елемента на масива, защото, пропуснем ли дори една двойка елементи, няма да сме сигурни, че те са различни помежду си, дори да са различни от другите елементи (от $x \neq y$ и $y \neq z$ не следва нито $x \neq z$, нито $x = z$). Затова е важно да умеем да дефинираме наредби върху различни множества.

Задача 3. Мода на едно мултимножество се нарича най-често срещаната стойност в него. Ако има няколко най-често срещани стойности, няма значение коя от тях ще смятаме за мода. Разглеждаме алгоритмичната задача Мода: да се намери модата на даден масив $A[1 \ldots n]$ с нареден базов тип. Докажете, че ако се решава чрез сравняване, задачата Мода изисква $\Theta(n \log n)$ сравнения при най-лоши входни данни.

Решение: Горната граница доказваме, като съставим алгоритъм, решаващ задачата Мода чрез сравняване на елементите на масива: прилагаме пирамидално сортиране върху масива, при което равните елементи се подреждат един до друг; после с едно обхождане на масива намираме най-дълъг подмасив от равни елементи; тяхната еднаква стойност е търсената мода. И двата етапа на този алгоритъм се основават на сравняване на елементи. Времевата сложност е $\Theta(n \log n)$ за първия етап — сортирането на масива; $\Theta(n)$ за втория етап — обхождането му; $\Theta(n \log n) + \Theta(n) = \Theta(n \log n)$ общо за целия алгоритъм при най-лоши входни данни.

Времето на всеки алгоритъм е горна граница за времето на задачата, която той решава, следователно задачата Мода има времева сложност $\mathrm{O}(n \log n)$ при най-лоши входни данни. Тази оценка отгоре важи за общото време, а значи и за броя на сравненията.

Долната граница ще докажем с редукция от по-лесния тип — редукция на данните. За тази цел ще сведем алгоритмичната задача Липса на повторения до задачата Мода. Описание на редукцията:

Липса на повторения $(A[1 \ldots n])$
1) $m \leftarrow$ Мода $(A[1 \ldots n])$
2) $p \leftarrow 0$
3) **for** $k \leftarrow 1$ **to** n **do**
4) **if** $A[k] = m$
5) $p \leftarrow p + 1$
6) **return** $p = 1$

Коректност на редукцията: Нека алгоритъмът на ред № 1 от кода решава задачата Мода чрез сравняване на елементите на входния масив. Ще докажем, че целият текст на псевдокод образува алгоритъм, който решава задачата Липса на повторения посредством сравняване на елементите на масива $A[1 \dots n]$.

Че елементите на входния масив се използват само в сравнения с други негови елементи, личи от факта, че обръщения към елементи на масива има само на редове № 1 и № 4 от кода. Алгоритъмът от ред № 1 по предположение работи само със сравняване на елементите на A, а ред № 4 очевидно сравнява два елемента на масива — модата и текущия елемент.

Цикълът брои всяко срещане на модата и записва резултата от броенето в променливата p. Инвариант на цикъла: $p =$ броя елементи със стойност m в подмасива $A[1 \dots k-1]$ при всяка проверка за край на цикъла. Това се доказва с индукция по поредния номер на проверката. Полуинвариант: броячът k на цикъла. Тъй като k нараства с единица след всяко изпълнение на тялото на цикъла, като приема всички целочислени стойности от 1 до $n+1$ включително, то тялото на цикъла се изпълнява точно n пъти. При последната проверка за край $k = n+1$ и от инварианта следва, че $p =$ броя на елементите със стойност m в целия масив $A[1 \dots n]$. Ако $p > 1$, то масивът съдържа повторения (модата m се повтаря) и съгласно с постановката на задачата Липса на повторения алгоритъмът би трябвало да върне стойност "лъжа". Ред № 6 прави именно това, тъй като равенството $p = 1$ не е изпълнено. Обратно, ако $p = 1$, равенството е изпълнено и алгоритъмът връща стойност "истина" в знак, че няма повторения; това е точно така, понеже, щом модата (най-честата стойност) се среща в масива само веднъж, то и всички други стойности се срещат само по един път. Тоест върнатата стойност е правилна във всички случаи, следователно алгоритъмът за задачата Липса на повторения е коректен.

Бързина на редукцията: Редукцията се състои от следните редове: № 2, № 3, № 4, № 5, № 6; тоест всичко без обръщението към задачата Мода. То е в началото на алгоритъма (на ред № 1), входните данни се предават без изменения, обработва се само върнатата стойност, така че имаме редукция на изхода. Времето за тази обработка е от порядък $\Theta(n)$ при всякакви входни данни заради цикъла с начало на ред № 3; по-конкретно ред № 4 прави $\Theta(n) = \mathrm{o}(n \log n)$ сравнения. Според доказаното в задача 2 всеки алгоритъм, решаващ задачата Липса на повторения с помощта на сравняване, извършва $\Omega(n \log n)$ сравнения между елементите на входния масив. Следователно броят на сравненията, направени от алгоритъма за задачата Мода на ред № 1, е $\Omega(n \log n) - \Theta(n) = \Omega(n \log n) - \mathrm{o}(n \log n) = \Omega(n \log n)$.

За алгоритъма от ред № 1 на кода беше предположено само това, че решава задачата Мода чрез сравняване на елементи. Затова получената оценка важи за самата алгоритмична задача: задачата Мода, ако се решава чрез сравнения между елементите на входния масив $A[1 \dots n]$, изисква $\Omega(n \log n)$ сравнения при най-лоши входни данни.

Всяко доказателство за долна граница на времевата сложност на алгоритмична задача, провеждано чрез редукция на данните, трябва да отговаря на три задължителни изисквания:

- *Посока на редукцията:* Старата задача трябва да се решава чрез новата, а не обратното! Нова наричаме задачата, за чиято времева сложност искаме да докажем долна граница; стара наричаме задачата, за чиято времева сложност има вече доказана долна граница. Редукция в обратната посока би била само един конкретен алгоритъм за новата задача, което би дало нейна горна, а не долна граница.
- *Коректност на редукцията:* С некоректна редукция не може да се докаже нищо!
- *Бързина на редукцията:* Редукцията трябва да бъде достатъчно бърза; в противен случай умалителят няма да е по-малък по порядък от умаляемото и порядъкът на разликата ще бъде неопределен! "Достатъчно бърза" означава "по-бърза от желаната долна граница" (тоест $\mathrm{o}(n \log n)$ за разглеждания клас от алгоритмични задачи). Изискването за бързина важи само за броя на операциите, чиято долна граница доказваме, а не за общото време. Често и общото време на редукцията е малко, което гарантира, че са малко операциите от който и да било вид, включително вида, който ни интересува (например сравненията).

Задача 4. Разглеждаме алгоритмичната задача Най-малка разлика, която гласи следното: по даден масив $A[1 \dots n]$ от n реални числа, $n \geq 2$, да се намери $\min\limits_{1 \leq i < j \leq n} \left| A[i] - A[j] \right|$.

Докажете, че ако бъде решавана чрез сравняване, тази задача изисква $\Theta(n \log n)$ сравнения при най-лоши входни данни.

Забележка: При всички алгоритмични задачи с числови входни данни (включително тази) се броят не само преките сравнения между елементите на входния масив, но също и сравненията между стойностите на аритметични изрази, които зависят от входните данни пряко или косвено. Иначе долната граница $\Omega(n \log n)$ може да бъде заобиколена: ще сравним елементите x и y не направо, а косвено: като сравним стойностите на $2x$ и $2y$ или $x - z$ и $y - z$.

Решение: За да докажем, че броят на сравненията при най-лоши входни данни е $\Theta(n \log n)$, ще докажем долна граница $\Omega(n \log n)$ и горна граница $\mathrm{O}(n \log n)$.

Долната граница $\Omega(n \log n)$ доказваме с редукция от задачата Липса на повторения. Описание на редукцията:

Липса на повторения $\big(A[1 \dots n]\big)$
1) $d \leftarrow$ Най-малка разлика $\big(A[1 \dots n]\big)$
2) **return** $d > 0$

Коректност на редукцията: Понеже ни интересуват абсолютните стойности на разликите, то най-малката разлика $d \geq 0$. Ако $d > 0$, алгоритъмът връща логическата стойност "истина" в знак, че масивът не съдържа повторения. Правилността на върнатата стойност става ясна от следното разсъждение: щом като най-малката разлика е положителна, то другите разлики също са положителни (защото са по-големи от нея), следователно няма елементи с разлика нула, тоест липсват повторения. Обратно, ако $d = 0$, алгоритъмът връща стойност "лъжа" в знак, че има повторения, което е точно така: елементите с разлика нула са равни.

Бързина на редукцията: В решението на тази задача се използва редукция на изхода, която се състои в проверката на неравенството $d > 0$. Проверката на едно неравенство отнема константно време: $\Theta(1) = \mathrm{o}(n \log n)$. Следователно редукцията е достатъчно бърза.

Горната граница $\mathrm{O}(n \log n)$ доказваме чрез построяването на достатъчно бърз алгоритъм за новата алгоритмична задача.

Най-малка разлика $\big(A[1 \dots n]\big)$
1) Сортиране $\big(A[1 \dots n]\big)$ // Някоя бърза сортировка, например пирамидално сортиране.
2) $d \leftarrow A[2] - A[1]$
3) **for** $k \leftarrow 3$ **to** n **do**
4) $\quad \delta \leftarrow A[k] - A[k-1]$
5) $\quad$ **if** $\delta < d$
6) $\quad\quad d \leftarrow \delta$
7) **return** d

Коректност на алгоритъма: Най-малката разлика на един елемент с всички останали е измежду разликите му със съседните по големина елементи. След сортирането на масива съседните по големина елементи заемат съседни места. Затова в търсене на най-малка разлика е достатъчно да проверим разликите на съседните елементи. Ред № 2 и ред № 4 от алгоритъма пресмятат абсолютните стойности на разликите, защото вадят по-малкия от по-големия елемент.

Бързина на алгоритъма: На ред № 1 от псевдокода използваме някоя бърза сортировка, например пирамидално сортиране или сортиране чрез сливане; тя изразходва време $\Theta(n \log n)$ при най-лоши входни данни. Ред № 2 и ред № 7 отнемат време $\Theta(1)$, а цикълът — време $\Theta(n)$. Общото време на алгоритъма се получава равно на $\Theta(n \log n) + \Theta(n) + \Theta(1) = \Theta(n \log n)$ при най-лоши входни данни.

Много често една алгоритмична задача е частен случай на друга алгоритмична задача. Всеки алгоритъм, който решава общия случай, решава и частния, но обратното не е вярно. Затова частният случай е не по-труден от общия. Понякога частният случай е по-лесен от общия, друг път са еднакво трудни, но е невъзможно частният случай да е по-труден от общия.

Обратно, обобщението на една алгоритмична задача е не по-лесно от нея. Това се използва за доказване на долни граници.

Задача 5. Алгоритмичната задача Най-малко разстояние има следната формулировка: да се намери най-малкото разстояние между n точки в равнината, зададени с координатите си. Докажете, че ако бъде решавана чрез сравняване, тази задача изисква $\Omega(n \log n)$ сравнения при най-лоши входни данни.

Забележка 1: Долната граница е точна, тоест може да се замени с $\Theta(n \log n)$, защото има алгоритъм от тип "разделяй и владей", решаващ задачата за време $\mathrm{O}(n \log n)$ във всички случаи. Съставянето на алгоритъма обаче е трудно, поради което го пропускаме.

Забележка 2: Задачата се формулира и решава аналогично за точки в пространството.

Решение: Долната граница следва от простия факт, че задачата Най-малко разстояние е обобщение на задачата Най-малка разлика. Това може да бъде казано в обратна посока: задачата Най-малка разлика се явява частен случай на задачата Най-малко разстояние, а именно случаят, когато дадените точки са разположени върху една права — абсцисната ос. По-точно, числовите данни на задачата Най-малка разлика смятаме за точки от числова ос, която отъждествяваме с абсцисната ос на правоъгълна координатна система в равнината.

Изложеното интуитивно обяснение може да се представи формално като редукция на входа. Описание на редукцията:

Най-малка разлика $\big(X\big[1 \ldots n\big]$: масив от реални числа$\big)$
1) $Y\big[1 \ldots n\big]$: масив от реални числа.
2) **for** $k \leftarrow 1$ **to** n **do**
3) $\quad Y\big[k\big] \leftarrow 0$
4) **return** Най-малко разстояние $\big(X\big[1 \ldots n\big], Y\big[1 \ldots n\big]\big)$

Бързина на редукцията: В тази задача се използва редукция на входа, която се състои в добавянето на ординати. Цикълът от редове № 2 и № 3 изразходва време $\Theta(n) = \mathrm{o}(n \log n)$. Следователно редукцията е достатъчно бърза за целите на доказателството.

Коректност на редукцията: Тъй като ред № 4 от кода връща резултата от подалгоритъма без по-нататъшна обработка, трябва да докажем, че двете множества притежават една и съща най-малка стойност. Става дума за множеството от абсолютните стойности на разликите между елементите на масива $X\big[1 \ldots n\big]$ и множеството от разстоянията между точките в равнината. Всъщност вярно е нещо повече: двете множества съвпадат. По-точно, абсолютната стойност на разликата на две числа от масива X съвпада с разстоянието между съответните им точки:

$$\sqrt{\Big(X\big[i\big] - X\big[j\big]\Big)^2 + \underbrace{\Big(Y\big[i\big] - Y\big[j\big]\Big)^2}_{0}} = \sqrt{\Big(X\big[i\big] - X\big[j\big]\Big)^2} = \Big|X\big[i\big] - X\big[j\big]\Big|.$$

Задача 6. Алгоритмичната задача Сортиране по абсолютна стойност се поставя така: даден масив $A\big[1 \ldots n\big]$ от n реални числа да се сортира по абсолютна стойност, тоест изходът да удовлетворява неравенствата $\Big|A\big[1\big]\Big| \leq \Big|A\big[2\big]\Big| \leq \ \ldots \ \leq \Big|A\big[n\big]\Big|$ и да бъде пермутация на входа. Да се докаже, че ако тази задача се решава чрез сравняване, тя изисква $\Theta(n \log n)$ сравнения при най-лоши входни данни.

Решение: Горната граница $\mathrm{O}(n \log n)$ доказваме, като съставим достатъчно бърз алгоритъм. Например сортиране чрез сливане, което сравнява абсолютните стойности на елементите.

Долната граница $\Omega(n \log n)$ се доказва с подходяща редукция, например от задачата за сортиране на реални числа. Че този вариант на сортирането изисква $\Omega(n \log n)$ сравнения, следва от задача 1, чиито изисквания се удовлетворяват от избрания базов тип на масива: множеството $\mathbb{R}$ на реалните числа е линейно наредено и е достатъчно голямо (то е безкрайно). Описание на редукцията:

Сортиране $\big(A[1 \ldots n]$: масив от произволни реални числа$\big)$

```
1)  x ← A[1]
2)  for k ← 2 to n do
3)      if A[k] < x
4)          x ← A[k]
5)  for k ← 1 to n do
6)      A[k] ← A[k] − x
7)  Сортиране по абсолютна стойност (A[1 ... n])
8)  for k ← 1 to n do
9)      A[k] ← A[k] + x
```

Бързина на редукцията: Това решение използва редукция както на входа, така и на изхода. Редукцията на входа обхваща редовете от № 1 до № 6 вкл. Тя се състои от едно присвояване и два цикъла, всеки от които изразходва време $\Theta(n)$. Редукцията на изхода се състои само от цикъла на редове № 8 и № 9, който също отнема време $\Theta(n)$. Общото време на редукцията е $\Theta(n) + \Theta(n) = \Theta(n) = \mathrm{o}(n \log n)$, тоест тя е достатъчно бърза за целите на доказателството.

Коректност на редукцията: Редукцията на входа намира най-малкото от дадените числа и го изважда от всички елементи на масива. Сега всички числа в масива са неотрицателни, затова съвпадат с абсолютните си стойности и сортирането по абсолютна стойност на ред № 7 работи като обикновена сортировка. Редукцията на изхода събира всички числа с изваденото и така възстановява първоначалните стойности, следователно изходът е пермутация на входа. Не е пречка, че сортирането на ред № 7 работи с други числа: те са получени от входните данни чрез изваждане на едно и също число, а това действие не променя тяхната подредба.

Долната граница на алгоритмичната задача Сортиране по абсолютна стойност може да се докаже и с по-проста редукция: от сортирането на *неотрицателни* реални числа. Описание на редукцията:

Сортиране $\big(A[1 \ldots n]$: масив от неотрицателни реални числа$\big)$

```
1)  Сортиране по абсолютна стойност (A[1 ... n])
```

Времето за тази редукция е нула: не се обработват нито входът, нито изходът на подзадачата. Редукцията е коректна, защото за неотрицателни числа сортирането по абсолютни стойности съвпада с обикновеното сортиране.

Ограничаването на допустимите стойности за входните данни до неотрицателни числа е наложително: иначе редукцията няма да бъде коректна (ако числата са с произволни знаци, сортирането по абсолютна стойност не съвпада с обикновеното сортиране).

Този трик се прилага често, но крие опасност: ако множеството от допустими стойности стане прекалено малко, долната граница може да спре да важи, защото можем да попаднем в частен случай, който притежава бързо решение. В конкретната задача такава опасност няма: множеството на неотрицателните реални числа е линейно наредено и голямо (безкрайно), тоест изпълнява изискванията на задача 1.

В други задачи може да бъдат използвани други ограничения. Долната граница $\Omega(n \log n)$ на сортирането чрез сравняване остава в сила, ако допустимите стойности бъдат ограничени до положителните реални числа или до отрицателните реални числа и т.н. Това се отнася за всичките шест задачи от настоящия раздел.