

СОРТИРАНЕ И ТЪРСЕНЕ

Сортирането и търсенето са често срещани задачи с разнообразни практически приложения и с множество разновидности:

- Структурата от входни данни може да бъде масив, списък и т.н.
- За стойностите може да има или да няма ограничения.
- Може да се търси елемент с дадена стойност (ключ), най-голям или най-малък елемент и изобщо k -ти по големина елемент.

Ето защо има различни алгоритми за сортиране и търсене. За да намалим неопределеността, приемаме, че входните данни се съдържат в масив $A[1 \dots n]$, а не в друга структура от данни. Допустимите индекси са от 1 до n , където n е броят на елементите на масива.

В практиката елементите обикновено са записи с няколко полета, едно от които е ключ от примитивен тип (най-често числов). Тук за простота ще приемем, че елементите на масива са реални числа — цели или дробни. Също за простота ще описваме алгоритмите на псевдокод.

Сортиран е този масив, чиито елементи образуват ненамаляваща редица. По-формално, масивът $A[1 \dots n]$ е сортиран тогава и само тогава, когато $A[1] \leq A[2] \leq A[3] \leq \dots \leq A[n]$. Алгоритмите за сортиране се различават по редица свойства — време, памет, устойчивост и др.

Алгоритъм, който размества елементите на масив, без да ги променя, се нарича устойчив, ако не размества равни елементи. Ако елементите са записи, алгоритъмът се нарича устойчив, когато не размества елементи с равни ключове. Не е задължително ключът да е числов.

Разместването на еднакви елементи е практически незабележимо, затова първият вариант на определението за устойчивост е безполезен. От значение е вторият вариант. Но за простота ще разглеждаме само числови масиви. Въпреки това всички алгоритми ще бъдат приложими и към масиви от записи.

РАЗДЕЛЯНЕ

Разделянето е операция, която понякога се използва самостоятелно, но по-често е стъпка от други алгоритми. Състои се в подреждането на n елемента спрямо даден разделител така, че елементите преди разделителя са по-малки, а следващите са по-големи или равни на него.

РАЗДЕЛЯНЕ ПО ЛОМУТО ($A[1 \dots n]$)

- 1) $S \leftarrow A[n]$ // Разделител е последният елемент на масива.
- 2) $r \leftarrow 1$
- 3) **for** $k \leftarrow 1$ **to** $n - 1$ **do**
- 4) **if** $A[k] < S$
- 5) $\text{swap}(A[k], A[r])$ // Разместване на два елемента.
- 6) $r \leftarrow r + 1$
- 7) $\text{swap}(A[n], A[r])$
- 8) **return** r // Алгоритъмът връща новия индекс на разделителя.

Анализ на алгоритъма:

— Сложност по памет: Разделянето работи на място, т.е. не копира елементите на масива. Допълнителна памет се заделя само за S , r , k и локалната променлива на разместването. Те са от примитивен тип (числа), затова сложността по памет е $M(n) = \Theta(1)$.

— *Полуинвариант*: броячът k . Расте с единица след всяко изпълнение на тялото на цикъла, затова то се изпълнява точно $n - 1$ пъти и алгоритъмът завършва.

— Сложността по време е $T(n) = \Theta(n)$, понеже тялото на цикъла се изпълнява $n - 1$ пъти.

— *Инвариант*: Всеки път, когато се изпълнява проверката за край на цикъла, $A[n] = S$ и са в сила неравенствата: $A[i] < S \leq A[j]$ за всички цели i и j , за които $1 \leq i < r \leq j < k$.

— Разделянето е неустойчиво: елементите $A[n]$ и $A[r]$ могат да се разместят с равни на тях.

Показаната схема за разделяне е съставена от Нико Ломуто. Тя е проста и бърза. Всъщност нейната времева сложност $\Theta(n)$ е оптимална по порядък, защото всеки алгоритъм за разделяне е длъжен да прочете всички елементи на масива. При все това тя не е най-бързата възможна схема за разделяне: ненаписаният константен множител пред n може да бъде намален.

Тони Хоор е предложил по-бърз алгоритъм за разделяне. С два индекса едновременно обхождаме дадения масив в противоположни посоки: с малкия индекс търсим големи елементи, а с големия индекс — малки елементи. Намерим ли такава двойка елементи, разменяме ги.

РАЗДЕЛЯНЕ ПО ХООР ($A[1 \dots n]$)

```

1)  $S \leftarrow A[n]$  // Разделител е последният елемент на масива.
2)  $r \leftarrow 0$ 
3)  $k \leftarrow n$ 
4) repeat
5)    $r \leftarrow r + 1$ 
6) until  $S \leq A[r]$ 
7) repeat
8)    $k \leftarrow k - 1$ 
9)   if  $k \leq r$ 
10)     $\text{swap}(A[n], A[r])$  // Разместване на два елемента.
11)    return  $r$  // Алгоритъмът връща новия индекс на разделителя.
12) until  $S > A[k]$ 
13)  $\text{swap}(A[k], A[r])$ 
14) go to 4 // Управлението се предава на ред № 4.
```

Това разделяне работи на място, следователно има константна сложност по памет: $M(n) = \Theta(1)$. Сложността по време също е с оптимален порядък при всякакви входни данни: $T(n) = \Theta(n)$, защото общият брой изменения на индексите r и k е равен на n .

При алгоритъма на Ломуто общият брой изменения на индексите r и k е по-голям от n в повечето случаи; само по изключение може да бъде по-малък. Индексът k нараства с $n - 2$, а нарастването на r е равно на броя на малките елементи. Ако подреждането на входния масив се приеме за случайно, то очакваният брой малки елементи е половината от дължината на масива, затова средният брой изменения на r и k е около $1,5n$.

Различен е също броят на разместванията. Предимство има отново алгоритъмът на Хоор. Разделянето по Ломуто размества всички малки елементи, а разделянето по Хоор — само онези малки елементи, които са заели местата на големи елементи. Средно взето, броят размествания в алгоритъма на Хоор е половината от броя на разместванията в алгоритъма на Ломуто.

Следователно алгоритъмът на Хоор е между 1,5 и 2 пъти по-бърз от алгоритъма на Ломуто в смисъл на средна времева сложност.

И двата алгоритъма са неустойчиви, тоест понякога разместват равни елементи.

Ако държим разделянето да бъде и устойчиво, и бързо, можем лесно да постигнем това с помощта на допълнителна памет. Няма как да спазим едновременно трите изисквания дотук: — разделител да е последният елемент на входния масив; — в изхода разделителят да се намира преди всички елементи, по-големи или равни на него; — разделянето да е устойчиво.

Трите изисквания са несъвместими, в случай че стойността на разделителя се повтаря в масива. Следователно трябва да се откажем от някое изискване. Най-лесно е да пожертваме първото. Алгоритъмът, чийто код предстои да опишем, ще притежава допълнителен входен параметър S , който ще играе ролята на разделител, но ще бъде само стойност, а не конкретен елемент на масива (масивът може да не съдържа елементи със стойност S). Както и досега, алгоритъмът ще връща най-малкия индекс, чийто елемент има стойност поне S , когато алгоритъмът приключва работа. Ако стойностите на всички елементи са по-малки от S , алгоритъмът ще връща $n + 1$.

УСТОЙЧИВО РАЗДЕЛЯНЕ ($A[1 \dots n]$, S)

```
1)  $B[1 \dots n]$ : масив със същия базов тип като  $A$ 
2)  $r \leftarrow 0$ 
3) for  $k \leftarrow 1$  to  $n$  do
4)   if  $A[k] < S$ 
5)      $r \leftarrow r + 1$ 
6)      $B[r] \leftarrow A[k]$ 
7)  $r \leftarrow n + 1$ 
8) for  $k \leftarrow n$  downto  $1$  do
9)   if  $A[k] > S$ 
10)     $r \leftarrow r - 1$ 
11)     $B[r] \leftarrow A[k]$ 
12) for  $k \leftarrow n$  downto  $1$  do
13)   if  $A[k] = S$ 
14)     $r \leftarrow r - 1$ 
15)     $B[r] \leftarrow A[k]$ 
16) for  $k \leftarrow 1$  to  $n$  do
17)    $A[k] \leftarrow B[k]$ 
18) return  $r$ 
```

Сложността по памет е $M(n) = \Theta(n)$ заради локалния масив B . Времовата сложност $T(n) = \Theta(n)$ е оптимална по порядък, но не и по константата пред n : четирите цикъла правят това разделяне няколко пъти по-бавно от другите две.

ТЪРСЕНЕ

Търсене по ключ

Има два начина за търсене по ключ в масив — последователно търсене и двоично търсене. Последователното търсене работи върху масиви и списъци. Двоичното търсене работи правилно само върху сортирани масиви. Затова нека входните данни са представени в масив $A[1 \dots n]$. Търсенето по ключ връща индекса (от 1 до n включително) на първия намерен елемент, чиято стойност съвпада с ключа. Ако ключът липсва в масива, резултатът е минус единица.

ПОСЛЕДОВАТЕЛНО ТЪРСЕНЕ ($A[1 \dots n]$, key)

```
1) for  $k \leftarrow 1$  to  $n$  do
2)   if  $A[k] = key$ 
3)     return  $k$ 
4) return  $-1$ 
```

Сложност на последователното търсене:

- по време: $T(n) = \Theta(n)$ в най-лошия случай: когато ключът липсва в масива;
- по памет: $M(n) = \Theta(1)$ при всякакви входни данни, защото единствена локална променлива е броячът k , а той е от примитивен тип (цяло число).

Алгоритъмът връща най-малкия индекс, чийто елемент има стойност, равна на ключа.

Полуинвариант е броячът k . Той расте от 1 до n с една единица след всяко изпълнение на тялото на цикъла, затова цикълът приключва след най-много n изпълнения на тялото.

?

Инвариант: Всеки път, когато се изпълнява проверката $k \leq n$ за край на цикъла, е в сила следното твърдение: $A[1] \neq key$ и $A[2] \neq key$, и $A[3] \neq key \dots$ и $A[k-1] \neq key$.

Доказателството на инварианта се извършва с математическа индукция по поредния номер на проверката за край на цикъла.

База: Първата проверка се извършва при влизане в цикъла. От инициализацията на брояча на ред № 1 от алгоритъма следва, че при влизане в цикъла броячът k получава стойност 1, конюнкцията в инварианта се състои от $k - 1 = 0$ операнда, тоест тя е празна, поради което е тривиално вярна.

Индуктивна стъпка: Нека инвариантът е в сила при някоя проверка за край на цикъла, която не е последна, тоест нека $A[1] \neq key$ и $A[2] \neq key$, и $A[3] \neq key \dots$ и $A[k-1] \neq key$. Трябва да докажем, че инвариантът ще остане в сила и при следващата проверка за край, тоест ще важат следните неравенства: $A[1] \neq key$ и $A[2] \neq key$, и $A[3] \neq key \dots$ и $A[\tilde{k}-1] \neq key$, където \tilde{k} е стойността на брояча при следващата проверка.

Действително, щом сегашната проверка за край на цикъла не е последна, то тялото му ще се изпълни с текущата стойност на брояча k , а ред № 3 няма да бъде изпълнен. От ред № 2 следва, че $A[k] \neq key$. Добавяме неравенствата от индуктивното предположение и получаваме удължена редица от неравенства: $A[1] \neq key$ и $A[2] \neq key$, и $A[3] \neq key \dots$ и $A[k] \neq key$. След това броячът k нараства с единица: $\tilde{k} = k + 1$, следователно $k = \tilde{k} - 1$. Сега идва новата проверка за край на цикъла. В сила е удължената редица от неравенства. Заместваме $k = \tilde{k} - 1$ и редицата приема желанния вид: $A[1] \neq key$ и $A[2] \neq key$, и $A[3] \neq key \dots$ и $A[\tilde{k}-1] \neq key$.

Дотук доказахме инварианта. С негова помощ ще изведем коректността на алгоритъма, тоест ще се убедим, че последователното търсене, описано по-горе, връща най-малкия индекс, чийто елемент има стойност, равна на ключа (връща -1 , ако няма такъв елемент).

Вече установихме чрез подходящ полуинвариант, че алгоритъмът завършва. Да разгледаме последната проверка на условието за край на цикъла. Има две възможности.

Първи случай: Условието за край на цикъла е удовлетворено, т.е. $k > n$. Тогава $k = n + 1$, защото броячът k приема само целочислени стойности и на предишната проверка за край е бил по-малък или равен на n . Доказания инвариант прилагаме към текущата (последната) проверка, замествайки $k = n + 1$ в инварианта: $A[1] \neq key$ и $A[2] \neq key$, и $A[3] \neq key \dots$ и $A[n] \neq key$. Тази конюнкция означава, че ключът key не се среща в масива $A[1 \dots n]$, затова алгоритъмът трябва да върне минус единица. Наистина, щом $k > n$, то тялото на цикъла не се изпълнява; вместо това се изпълнява ред № 4 и алгоритъмът връща минус единица, както трябва.

Втори случай: Условието за край на цикъла не е удовлетворено, т.е. $k \leq n$. Следователно тялото на цикъла се изпълнява с текущата стойност на брояча k . Щом тази проверка е последна, алгоритъмът не се връща на ред № 2, а излиза през ред № 3. От ред № 2 следва, че $A[k] = key$. Към това равенство присъединяваме инварианта от последната проверка за край на цикъла и научаваме, че $A[1] \neq key$ и $A[2] \neq key$, и $A[3] \neq key \dots$ и $A[k-1] \neq key$, но $A[k] = key$. Ред № 3 връща това k — най-малкия индекс, чийто елемент има стойност, равна на ключа.

По подобен начин може да се анализира и двоичното търсене на ключ в сортиран масив. То се описва най-естествено чрез рекурсия.

ДВОИЧНО ТЪРСЕНЕ ($A[\ell \dots h]$, key)

- 1) **if** $\ell > h$
- 2) **return** -1 // празен масив
- 3) $m \leftarrow \left\lfloor \frac{\ell + h}{2} \right\rfloor$
- 4) **if** $A[m] = key$
- 5) **return** m
- 6) **if** $A[m] > key$
- 7) **return** ДВОИЧНО ТЪРСЕНЕ ($A[\ell \dots m-1]$, key)
- 8) // $A[m] < key$
- 9) **return** ДВОИЧНО ТЪРСЕНЕ ($A[m+1 \dots h]$, key)

Параметрите ℓ и h определят границите на подмасива, подлежащ на изследване, $\ell \leq h$. Първия път алгоритъмът се вика с целия масив $A[1 \dots n]$, тоест границите са $\ell = 1$ и $h = n$. При рекурсивните извиквания границите ℓ и h се променят. По-общо, можем да означим с n дължината на подмасива, тоест $n = h - \ell + 1$. По принцип $n \geq 0$. Тук за удобство допускаме отрицателни дължини, които подобно на нулевата дължина означават празен масив.

Двоичното търсене връща индекс от ℓ до h , чийто елемент има стойност, равна на ключа (връща -1 , ако няма такъв индекс). Ако съществуват няколко индекса с описаното свойство, алгоритъмът връща някой от тях (не непременно най-малкия или най-големия).

Полуинвариант е дължината на масива, тоест $h - \ell + 1$. Дължината винаги е цяло число. Ако това число е нула (или отрицателно), алгоритъмът не извършва рекурсивно обръщение, а приключва веднага — това е дъното на рекурсията на редове № 1 и № 2 от кода на алгоритъма. Ако дължината е положително число, тя намалява два пъти при всяко рекурсивно обръщение; рекурсията не е бездънна, тоест алгоритъмът завършва рано или късно, защото не съществува безкрайна строго намаляваща редица от цели положителни числа (дължините на подмасивите). Полуинвариант е също разликата $h - \ell$, която има подобни свойства.

Никоя от границите ℓ и h не е полуинвариант сама по себе си: те се изменят монотонно (ℓ расте, а h намалява при всяко рекурсивно обръщение), но тази монотонност не е строга. Например ℓ остава същата на ред № 7, а пък h запазва стойността си на ред № 9.

Рекурсивните алгоритми нямат инварианти. Такива имат само итеративните алгоритми (по-точно, всеки цикъл има собствен инвариант). Коректността на рекурсивните алгоритми се доказва с математическа индукция по подходящ параметър — най-често полуинварианта. В тази задача провеждаме структурна математическа индукция по дължината $n = h - \ell + 1$ на входния масив $A[\ell \dots h]$. Тъй като дължината на масива се изменя с повече от една единица при рекурсивните обръщения, удобно е да използваме т. нар. силна индукция.

База: $n \leq 0$, тоест $\ell > h$. Масивът $A[\ell \dots h]$ е празен, следователно не съдържа ключа *key* и алгоритъмът трябва да върне минус единица. От редове № 1 и № 2 на програмния код следва, че алгоритъмът връща -1 , тоест върнатата стойност е правилна.

Индуктивна стъпка: Нека $n > 0$, тоест $\ell \leq h$. Да предположим, че алгоритъмът е верен при масиви с всякакви дължини, по-малки от n . Ще докажем, че е коректен и при дължина n .

Първо, всички операции в кода на алгоритъма се изпълняват върху допустими стойности:

— Тъй като $m = \left\lfloor \frac{\ell + h}{2} \right\rfloor$, то $\ell \leq m \leq h$. Ето защо $A[m]$ е елемент на масива $A[\ell \dots h]$,

тоест имаме право да извличаме стойността на $A[m]$ на редове № 4 и № 6.

— От $\ell \leq m \leq h$ следва, че $m - 1 < h$ и $\ell < m + 1$. Тогава $A[\ell \dots m - 1]$ и $A[m + 1 \dots h]$ са подмасиви (може и празни подмасиви) на сортирания масив $A[\ell \dots h]$, затуй и те са сортирани. Ето защо редове № 7 и № 9 от кода на алгоритъма се изпълняват върху допустими входни данни.

Второ, върнатата стойност е коректна. Това се доказва чрез трасиране на алгоритъма. Тъй като $\ell \leq h$, то ред № 2 няма да се изпълни, а ще се изпълнят ред № 3 и следващите редове:

— Ако $A[m] = \text{key}$, то индексът m ще бъде върнат от ред № 5. Това действие ще е правилно, защото индексът m има желаното свойство: стойността на неговия елемент е равна на ключа.

— Ако $A[m] > \text{key}$, то $\text{key} < A[m] \leq A[m + 1] \leq A[m + 2] \leq \dots \leq A[h]$. Тоест ключът *key* не се среща в подмасива $A[m \dots h]$, затова остава да търсим ключа в подмасива $A[\ell \dots m - 1]$. Ред № 7 прави точно това. Че търси правилно, следва от индуктивното предположение. То може да се приложи, защото дължината на подмасива е по-малка от n : от $m \leq h$ следва, че $m - 1 < h$, затова дължината на подмасива $(m - 1) - \ell + 1 < h - \ell + 1 = n$.

— Ако $A[m] < \text{key}$, то $\text{key} > A[m] \geq A[m - 1] \geq A[m - 2] \geq \dots \geq A[\ell]$. Тоест ключът *key* не се среща в подмасива $A[\ell \dots m]$, затова остава да търсим ключа в подмасива $A[m + 1 \dots h]$. Ред № 9 прави точно това. Че търси правилно, следва от индуктивното предположение. То може да се приложи, защото дължината на подмасива е по-малка от n : от $m \geq \ell$ следва, че $m + 1 > \ell$, затова дължината на подмасива $h - (m + 1) + 1 < h - \ell + 1 = n$.

Времева сложност на двоичното търсене може да се намери с рекурентно уравнение. Този подход е естествен, защото двоичното търсене е рекурсивен алгоритъм. Да означим с $T(n)$ времева сложност на алгоритъма при най-лошите възможни входни данни: когато ключът key не се съдържа в масива $A[\ell \dots h]$. Както обикновено, $n = h - \ell + 1$ е дължината на масива. Функцията $T(n)$ удовлетворява рекурентното уравнение

$$T(n) = T\left(\frac{n}{2}\right) + \Theta(1).$$

Първото събираемо в дясната страна е времето за рекурсивното извикване на алгоритъма. Коефициентът пред това събираемо е единица, защото всеки път се изпълнява само едно от двете рекурсивни обръщения на редове № 7 и № 9. Делението на две в аргумента на T съответства на делението на масива на две части с (почти) равни дължини.

Събираемото $\Theta(1)$ е времето за всички останали действия — пресмятането на индекса m , извличането на стойността на елемента $A[m]$, сравняването на тази стойност с ключа key , прибавянето или изваждането на единица от индекса m . Общото време е ограничено отгоре, защото всяко от тези действия се извършва само веднъж: няма цикъл.

Двоичното търсене е алгоритъм от тип “разделяй и владей”, затуй рекурентното уравнение, което описва неговата времева сложност, се решава с мастер-теоремата. Конкретното уравнение попада във втория случай на теоремата, следователно времева сложност е $T(n) = \Theta(\log n)$ при най-лоши входни данни.

Анализът на сложността по памет съдържа някои тънкости. На пръв поглед е логично да разсъждаваме по следния начин. Всяко равнище на рекурсията само по себе си изразходва константно количество памет c — за три променливи от примитивен (целочислен) тип: за индекса m , който е локална променлива, и за границите ℓ и h на подмасива $A[\ell \dots h]$. По принцип паметта за входните данни не се брои, защото тя се заделя от извикващия код, обаче рекурсивните алгоритми са едновременно извиквани и извикващи, затова броим паметта за параметрите ℓ и h . При правилна (т.е. възможно най-икономична) реализация на алгоритъма няма нужда да слагаме в сметката паметта за елементите на масива $A[\ell \dots h]$, нито за указателя към първия елемент на масива, нито за ключа key : те не се менят при рекурсивните извиквания, затова не е нужно да се копират и предават между равнищата на рекурсията, а вместо това могат да се реализират като глобални променливи. (По принцип трябва да добавим малко памет за резултатите на междинните операции, например за пресмятане на сбора на ℓ и h на ред № 3, но това не променя разсъжденията по същество, а само увеличава стойността на константата c .)

Всяко равнище на рекурсията ползва различна памет за локалните променливи. Ето защо е важно да знаем броя на равнищата (дълбочината на рекурсията). Ако дължината на масива отначало е n , то след първото рекурсивно извикване тя е около $n/2$, след второто е около $n/4$, след k -тото е около $n/2^k$. Най-лошият случай (с най-голяма дълбочина на рекурсията) съответства на масив $A[\ell \dots h]$, несъдържащ ключа key . Дълготата на рекурсията се достига, когато масивът остане празен. На предишната стъпка масивът е имал дължина 1 или 2. Тоест $n/2^k \approx 1 \iff 2^k \approx n \iff k \approx \log_2 n$. Щом всяко равнище на рекурсията има собствена памет, то количеството памет се увеличава с навлизане в по-дълбоките равнища и трябва да сумираме: $M(n) = \underbrace{c + c + \dots + c}_{k \text{ пъти}} = k \cdot c \approx c \cdot \log_2 n = \Theta(\log n)$.

Този извод щеше да е верен, ако рекурсията беше неопашкова. Но в конкретния алгоритъм всички рекурсивни обръщения са опашкови, защото техните резултати не се обработват повече, а направо се връщат чрез оператора **return**. Затова при навлизане в по-дълбоките равнища не е нужно да се пази паметта за по-горните равнища на рекурсията. Вместо това всички равнища могат да използват обща памет. Така сумирането отпада и сложността по памет се получава $M(n) = c = \Theta(1)$, като c има по-малка стойност — размера на паметта само за m , без ℓ и h . Тук няма добри и лоши входни данни: полученият извод важи за всякакви входни данни.

Всяка рекурсия е заменима с цикъл — преход към началото на алгоритъма със заделяне на памет от програмния стек за новите копия на локалните променливи и входните параметри. При опашкова рекурсия отпада заделянето на памет, остава само преходът.

При двоичното търсене замяната на рекурсията с цикъл има следния резултат:

ДВОИЧНО ТЪРСЕНЕ ($A[\ell \dots h]$, key)

- 1) **if** $\ell > h$
- 2) **return** -1 // празен масив или подмасив
- 3) $m \leftarrow \left\lfloor \frac{\ell + h}{2} \right\rfloor$
- 4) **if** $A[m] = key$
- 5) **return** m
- 6) **if** $A[m] > key$
- 7) $h \leftarrow m - 1$
- 8) **go to** 1 // преход към ред № 1
- 9) // $A[m] < key$
- 10) $\ell \leftarrow m + 1$
- 11) **go to** 1 // преход към ред № 1

Ако решим да се придържаме към стила на структурното програмиране, всеки преход назад трябва да заменим с цикъл. Проверката на ред № 1 превръщаме в проверка за край на цикъла, като заменяме неравенството с неговото отрицание. Структурираният код изглежда така:

ДВОИЧНО ТЪРСЕНЕ ($A[\ell \dots h]$, key)

- 1) **while** $\ell \leq h$ **do**
- 2) $m \leftarrow \left\lfloor \frac{\ell + h}{2} \right\rfloor$
- 3) **if** $A[m] = key$
- 4) **return** m // Ключът е намерен на място № m .
- 5) **if** $A[m] > key$
- 6) $h \leftarrow m - 1$
- 7) **else**
- 8) $\ell \leftarrow m + 1$ // $A[m] < key$
- 9) **return** -1 // Ключът не е намерен в масива.

Тази итеративна реализация е равностойна на опашковата рекурсия: двете реализации изпълняват едни и същи команди в една и съща последователност. Затова имат еднакви характеристики — коректност, сложност по време, сложност по памет. Това може да се установи и без позоваване на еквивалентността на двете реализации.

Анализ на итеративната версия:

— Нека $n = h - \ell + 1$ е дължината на входния масив $A[\ell \dots h]$, където ℓ и h означават началните стойности на променливите.

— *Полуинвариант* е дължината $h - \ell + 1$ на текущия подмасив $A[\ell \dots h]$, където ℓ и h означават текущите стойности на променливите. След всяко изпълнение на тялото на цикъла дължината на подмасива намалява строго (приблизително два пъти).

— Сложността по памет е $M(n) = \Theta(1)$, тъй като допълнителна памет се изразходва само за локалната променлива m .

— Сложността по време е $T(n) = \Theta(\log n)$ при най-лошите входни данни: когато масивът не съдържа търсената стойност. След k изпълнения на тялото на цикъла остава подмасив с дължина около $n/2^k$. За да установи, че масивът не съдържа ключа, алгоритъмът трябва да стигне до дължина 1. От равенството $n/2^k \approx 1$ следва, че тялото на цикъла се изпълнява $k \approx \log_2 n$ пъти. Следователно $T(n) = \Theta(\log n)$ в най-лошия случай.

— *Инвариант*: При всяко изпълнение на проверката за край на цикъла $\ell \leq h$ е в сила твърдението: $A[i] \neq key$ за $\forall i \in \{\ell_0; \ell_0 + 1; \ell_0 + 2; \dots; \ell - 1\} \cup \{h + 1; h + 2; h + 3; \dots; h_0\}$, където ℓ_0 и h_0 са началните стойности съответно на променливите ℓ и h .

Доказателството на инварианта се извършва с математическа индукция по поредния номер на проверката за край на цикъла. Предварително нека отбележим следното: от редове № 1 и № 2 следва, че в тялото на цикъла ℓ , h и m са цели числа и $\ell \leq m \leq h$. Оттук и от редове № 6 и № 8 следва, че ℓ може само да расте, а пък h може само да намалява. (Могат и да не се променят.) Затова $\ell \geq \ell_0$ и $h \leq h_0$. Като съединим четирите неравенства, получаваме: $\ell_0 \leq \ell \leq m \leq h \leq h_0$.

База: Първата проверка за край се извършва при влизане в цикъла. Тогава $\ell = \ell_0$, $h = h_0$ и множеството от допустими стойности на i има $[(\ell - 1) - (\ell_0 - 1)] + (h_0 - h) = 0$ елемента, тоест то е празно. Инвариантът приема вида: $A[i] \neq key$ за $\forall i \in \emptyset$, което е тривиално вярно.

Индуктивна стъпка: Нека инвариантът важи при някоя непоследна проверка за край, т.е. $A[i] \neq key$ за $\forall i \in \{\ell_0; \ell_0 + 1; \ell_0 + 2; \dots; \ell - 1\} \cup \{h + 1; h + 2; h + 3; \dots; h_0\}$, където ℓ и h са текущите стойности на променливите. Ще докажем, че инвариантът остава в сила и при следващата проверка на условието за край на цикъла, тоест при нея ще важи следното: $A[i] \neq key$ за $\forall i \in \{\ell_0; \ell_0 + 1; \ell_0 + 2; \dots; \tilde{\ell} - 1\} \cup \{\tilde{h} + 1; \tilde{h} + 2; \tilde{h} + 3; \dots; h_0\}$, където $\tilde{\ell}$ и \tilde{h} ще бъдат новите стойности на двете променливи.

Наистина, щом текущата проверка за край на цикъла не е последна, то ред № 4 от кода няма да бъде изпълнен. От ред № 3 следва, че $A[m] \neq key$. Значи, $A[m] < key$ или $A[m] > key$.

— Първи случай: $A[m] > key$. От редове № 5 и № 6 на кода следва, че $\tilde{h} = m - 1$, $\tilde{\ell} = \ell$. Ето защо $m = \tilde{h} + 1$. Понеже масивът е сортиран и $\ell_0 \leq \ell \leq m \leq h \leq h_0$, то важат неравенствата $key < A[m] \leq A[m + 1] \leq \dots \leq A[h_0]$. Заместваме $m = \tilde{h} + 1$ и неравенствата приемат формата: $key < A[\tilde{h} + 1] \leq A[\tilde{h} + 2] \leq \dots \leq A[h_0]$, т.е. $A[i] \neq key$ за $\forall i \in \{\tilde{h} + 1; \tilde{h} + 2; \tilde{h} + 3; \dots; h_0\}$. А съгласно с индуктивното предположение $A[i] \neq key$ за $\forall i \in \{\ell_0; \ell_0 + 1; \ell_0 + 2; \dots; \ell - 1\}$. Заместваме $\ell = \tilde{\ell}$ в последното твърдение: $A[i] \neq key$ за $\forall i \in \{\ell_0; \ell_0 + 1; \ell_0 + 2; \dots; \tilde{\ell} - 1\}$. Като обединим двете множества, в които се изменя променливата i , получаваме желания извод: $A[i] \neq key$ за $\forall i \in \{\ell_0; \ell_0 + 1; \ell_0 + 2; \dots; \tilde{\ell} - 1\} \cup \{\tilde{h} + 1; \tilde{h} + 2; \tilde{h} + 3; \dots; h_0\}$.

— Втори случай: $A[m] < key$. От редове № 7 и № 8 на кода следва, че $\tilde{\ell} = m + 1$, $\tilde{h} = h$. Ето защо $m = \tilde{\ell} - 1$. Понеже масивът е сортиран и $\ell_0 \leq \ell \leq m \leq h \leq h_0$, то важат неравенствата $key > A[m] \geq A[m - 1] \geq \dots \geq A[\ell_0]$. Заместваме $m = \tilde{\ell} - 1$ и неравенствата приемат формата: $key > A[\tilde{\ell} - 1] \geq A[\tilde{\ell} - 2] \geq \dots \geq A[\ell_0]$, т.е. $A[i] \neq key$ за $\forall i \in \{\ell_0; \ell_0 + 1; \ell_0 + 2; \dots; \tilde{\ell} - 1\}$. А съгласно с индуктивното предположение $A[i] \neq key$ за $\forall i \in \{h + 1; h + 2; h + 3; \dots; h_0\}$. Заместваме $h = \tilde{h}$ в последното твърдение: $A[i] \neq key$ за $\forall i \in \{\tilde{h} + 1; \tilde{h} + 2; \tilde{h} + 3; \dots; h_0\}$. Като обединим двете множества, в които се изменя променливата i , получаваме желания извод: $A[i] \neq key$ за $\forall i \in \{\ell_0; \ell_0 + 1; \ell_0 + 2; \dots; \tilde{\ell} - 1\} \cup \{\tilde{h} + 1; \tilde{h} + 2; \tilde{h} + 3; \dots; h_0\}$.

С това е доказан инвариантът на цикъла.

Коректност на двоичното търсене: Видяхме, че полуинвариант е дължината на подмасива, която намалява поне два пъти при всяко изпълнение на тялото на цикъла. Тъй като не съществува безкрайна строго намаляваща редица от цели неотрицателни числа, то алгоритъмът завършва. Остава да докажем, че върнатата стойност е коректна — тоест че е индекс на някой елемент, чиято стойност е равна на ключа, или минус единица, ако няма такъв елемент.

Стойност връщат само редове № 4 и № 9 от алгоритъма, затова разглеждаме два случая:

— Ако алгоритъмът излезе през ред № 4, то върнатата стойност m е правилна, тъй като $\ell_0 \leq m \leq h_0$ и $A[m] = key$. Неравенствата бяха доказани по-горе, а пък равенството следва от проверката на ред № 3: тя е минала успешно, иначе алгоритъмът не би излязъл през ред № 4.

— Ако алгоритъмът излезе през ред № 9, то при последната проверка за край на цикъла е било в сила неравенството $\ell > h$. Понеже ℓ и h са цели числа, то $\ell - 1 \geq h$. Следователно $\{\ell_0; \ell_0 + 1; \dots; \ell - 1\} \cup \{h + 1; h + 2; \dots; h_0\} \supseteq \{\ell_0; \ell_0 + 1; \dots; h\} \cup \{h + 1; h + 2; \dots; h_0\} = \{\ell_0; \ell_0 + 1; \dots; h_0\}$. От това включване и от доказанния инвариант стигаме до извода, че $A[i] \neq key$ за $\forall i \in \{\ell_0; \ell_0 + 1; \ell_0 + 2; \dots; h_0\}$, тоест входният масив не съдържа ключа, следователно ред № 9 правилно връща минус единица.

Названията “последователно търсене” и “двоично търсене” се употребяват понякога не само за алгоритмите, разгледани дотук, но и за други алгоритми, основани на подобни идеи. Ако например търсим грешка в дълга редица от преобразувания на някакъв алгебричен израз, не е ефективно да използваме последователно търсене: неговата времева сложност е $\Theta(n)$. Двоичното търсене е много по-бързо: изразходва време $\Theta(\log n) = o(n)$, където n е броят изрази (дължината на редицата от преобразувания).

По-конкретно, двоичното търсене може да се приложи към тази задача по следния начин. Избираме стойности на променливите, за които всички алгебрични изрази се пресмятат лесно; ако няма такива, избираме произволни стойности.

Пресмятаме стойността A на първия израз в редицата; за него по определение приемаме, че е верен (например той е даден в условието на алгебрична задача и трябва да бъде опростен).

Пресмятаме стойността на последния израз в редицата — отговора на алгебричната задача. Ако стойността се получи A , приемаме, че няма грешка (при все че това не е съвсем сигурно). Ако стойността се получи различна от A , то със сигурност има поне една грешка. Започваме да търсим мястото ѝ. (Ако има няколко грешки, алгоритъмът ще открие само една от тях.)

Делим редицата на две равни части, тоест пресмятаме стойността на средния израз. Ако тя е равна на A , със сигурност има грешка във втората половина на редицата от преобразувания. Ако стойността на средния израз е различна от A , със сигурност има грешка в първата половина на редицата от преобразувания.

Повтаряме описаното действие: след всяка проверка смаляваме редицата наполовина, докато открием два съседни изрази с различни стойности. Грешката се намира в прехода от единия към другия израз.

Търсене на екстремална стойност

Търсенето на най-малка стойност е аналогично на търсенето на най-голяма стойност, както личи от съпоставката на двата алгоритъма.

Най-голяма стойност ($A[1 \dots n]$)	Най-малка стойност ($A[1 \dots n]$)
1) $m \leftarrow A[1]$	1) $m \leftarrow A[1]$
2) for $k \leftarrow 2$ to n do	2) for $k \leftarrow 2$ to n do
3) if $A[k] > m$	3) if $A[k] < m$
4) $m \leftarrow A[k]$	4) $m \leftarrow A[k]$
5) return m	5) return m

С малко промени алгоритмите могат да се преправят така, че вместо екстремалната стойност да връщат мястото в масива, където тя се намира. Тук ще разгледаме алгоритмите във вида, в който са описани.

Анализ на алгоритмите:

— *Полуинвариант*: броячът k . Расте с единица след всяко изпълнение на тялото на цикъла. От 2 до n приема $n - 1$ стойности, тоест тялото на цикъла се изпълнява точно $n - 1$ пъти, след което цикълът завършва и съответният алгоритъм връща m .

— *Инвариант*: При всяко изпълнение на проверката за край на цикъла е в сила равенството:
 $m = \max \{ A[i] \mid 1 \leq i \leq k - 1 \}$ при търсене на най-голяма стойност;
 $m = \min \{ A[i] \mid 1 \leq i \leq k - 1 \}$ при търсене на най-малка стойност.

— Сложността по памет е $M(n) = \Theta(1)$ при произволни входни данни; допълнителна памет е нужна само за променливите k и m , а те са от примитивен тип — цели числа.

— Сложността по време е $T(n) = \Theta(n)$ при всички входни данни, понеже тялото на цикъла (и при единия, и при другия алгоритъм) се изпълнява точно $n - 1 = \Theta(n)$ пъти.

Оптимални ли са тези алгоритми?

Сложността по памет е оптимална по порядък. Да допуснем противното: че $M(n) \neq \Omega(1)$. Следователно за всяко $c > 0$ съществуват безброй много стойности на n , такива че $M(n) < c$. При $c = 1$ получаваме неравенството $M(n) < 1$, което е равносилно на равенството $M(n) = 0$, защото на практика количеството памет е дискретно: измерено в битове, то винаги е цяло число. Следователно $M(n) = 0$ за безброй много стойности на n . От друга страна, $M(n) > 0$ за всяко n , защото е нужен поне един брояч за обхождането на масива. Полученото противоречие показва, че допускането не е било вярно. Ето защо $M(n) = \Omega(1)$.

Сложността по време също е оптимална по порядък. $T(n) = \Omega(n)$, защото е необходимо да се извърши поне едно прочитане на всеки елемент на входния масив: всеки елемент би могъл да съдържа екстремалната стойност. Това разсъждение важи за повечето алгоритмични задачи и се нарича тривиална долна граница по размера на входа: $T(n) = \Omega(n)$, защото почти винаги се налага да бъде прочетен целият масив от входни данни. От това правило има изключения, например двоичното търсене, но при него входът е сортиран масив, затова имаме възможност да правим заключения за някои елементи, без да ги четем. Когато между входните данни има някаква зависимост (например когато масивът е сортиран), се казва, че те са структурирани. Тривиалната долна граница на времевата сложност не важи за структурирани входни данни.

Обаче в задачата за търсене на екстремална стойност масивът не е сортиран, поради което стойността на един елемент не дава никаква информация за стойностите на другите елементи. Ето защо се налага да прочетем всички входни данни. Следователно тривиалната долна граница важи за тази задача и въобще за всички задачи с неструктурирани входни данни.

Забележка: Съществуват и други видове тривиални долни граници:

- по размера на изхода: времето на алгоритъм е поне колкото дължината на неговия изход;
- по размера на паметта: $T(n) = \Omega(M(n))$ заради инициализирането на паметта.

Има по-точни оценки, не само на порядъка. При всяко изпълнение на тялото на цикъла всеки от алгоритмите извършва точно едно сравнение и не повече от едно присвояване. Ето защо можем да използваме броя на сравненията в ролята на мярка за времевата сложност. Така получаваме $T(n) = n - 1$, тоест извършват се $n - 1$ сравнения при всякакви входни данни, защото тялото на цикъла се изпълнява $n - 1$ пъти.

Двата алгоритъма са оптимални по броя на сравненията: екстремалният елемент не може да се намери с по-малко от $n - 1$ сравнения. Това твърдение има разнообразни доказателства. Тъй като двата алгоритъма си приличат, ще разглеждаме само търсенето на най-голяма стойност.

Първо доказателство: с теорията на графите. Разглеждаме неориентиран граф с n върха, съответстващи на елементите на масива $A[1 \dots n]$. Отначало графът е празен: не съдържа ребра. Прилагаме произволен алгоритъм за търсене на най-голяма стойност, не непременно този, който беше описан по-горе. Когато алгоритъмът сравни два елемента, слагаме ребро между тях. Ясно е, че броят на сравненията е по-голям или равен на броя на ребрата (има строго неравенство, ако някоя двойка елементи са сравнени повече от един път). Ето защо е достатъчно да докажем, че в края на алгоритъма графът ще има поне $n - 1$ ребра.

В края на алгоритъма вече ще знаем най-големия елемент. Щом сме сигурни в отговора, значи има начин да го съпоставим с всеки от останалите елементи. Ако например сме сравнили елементите x и y , а също y и z , тогава може да не се наложи да сравняваме x и z пряко: ако $x < y$ и $y < z$, то $x < z$; успяхме да съпоставим x и z , без да ги сравняваме пряко.

Следователно от екстремалния връх до всеки друг връх на графа трябва да има път, съставен от едно, две или повече ребра. Тоест графът трябва да е свързан. Ако съдържа цикъл, можем да премахнем кое да е ребро от цикъла: това не нарушава свързаността на графа. Накрая ще остане неориентиран свързан граф без цикли, тоест дърво. Както е известно, всяко дърво с n върха има $n - 1$ ребра. Преди изтриването техният брой е бил не по-малък. Затова всеки свързан граф има поне $n - 1$ ребра. Това се отнася и за графа, получен в края на който да е алгоритъм за намиране на най-голям сред n елемента. Понеже броят на ребрата е по-малък или равен на броя на сравненията, то всеки такъв алгоритъм непременно прави поне $n - 1$ сравнения.

Второ доказателство: чрез свеждане до игра. Когато търсим най-голям елемент на масив, при всяко сравняване на два елемента отпада един — по-малкият (ако са равни, няма значение кой от тях отпада). Това прилича на турнир по системата на елиминациите: след всяка среща отпада един играч — победеният. Най-големият елемент съответства на първенеца в турнира. Това, че алгоритъмът за търсене е произволен, съответства на произволна схема на турнира, тя може да е различна от стандартната (осминафинали, четвъртфинали, полуфинали, финал). Каквато и да е схемата, щом отначало има n играчи, а накрая остава само един — първенецът, то трябва $n - 1$ играчи да бъдат елиминирани. Обаче всяка среща елиминира само един играч. Следователно трябва да се проведат поне $n - 1$ срещи, тоест алгоритъмът трябва да извърши поне $n - 1$ сравнения.

Търсене на втора екстремална стойност

Понякога се налага да търсим втория най-голям или най-малък от общо n елемента. Двете задачи са аналогични, затова ще разгледаме само търсенето на втория най-голям елемент.

Един очевиден начин за решаване на задачата е следният. Намираме най-големия елемент, но не стойността, а индекса. После обхождаме масива повторно в търсене на най-голям елемент, обаче прескачаме споменатия индекс. Нужно е да се използва индексът вместо стойността, защото може да има елементи с равни стойности.

Този подход е правилен, но не е най-бързият възможен. Да пресметнем времевата сложност. При първото обхождане се правят $n - 1$ сравнения между елементи, а при второто — само $n - 2$, защото най-големият елемент не участва. Общият брой сравнения между елементи на масива е равен на $T(n) = (n - 1) + (n - 2) = 2n - 3 = \Theta(n)$. Броят сравнения е оптимален по порядък заради тривиалната долна граница $T(n) = \Omega(n)$, обаче не е оптимален като точна стойност, тоест задачата може да се реши с по-малко сравнения на елементи.

Идеята най-лесно се формулира посредством игра — турнир по системата на елиминациите. Все едно че искаме да излъчим не само златен, но и сребърен медалист. Тъй като тук търсим конкретен алгоритъм, а не разсъждаваме за всички възможни алгоритми, ще бъде достатъчно да посочим една конкретна схема за провеждане на турнира, при която броят на срещите е по-малък от $2n - 3$. Такава е стандартната схема: играчите се разпределят по двойки, всяка двойка провежда среща, победеният отпада от състезанието, а победителят в срещата минава на следващия кръг. След всеки кръг играчите намаляват приблизително наполовина. Ако в някой кръг има нечетен брой играчи, един от тях отива на следващия кръг, без да играе.

Златен медалист (шампион на турнира) е победителят в последния кръг, който се състои от една среща — финала на първенството. Кой състезател трябва да получи сребърния медал, тоест кой играч е втори по сила? Изобщо не е задължително това да е другият финалист! Ясно е, че играчът, втори по сила, е победил всички, срещу които е играл, без шампиона, от когото е загубил. За сребърния медалист знаем само това: бил е елиминиран от шампиона. Обратно, всеки състезател, елиминиран от шампиона, би могъл да бъде сребърен медалист: тези играчи не са се срещали помежду си (иначе нямаше да бъдат елиминирани от шампиона) и са победили всичките си противници без шампиона, тоест те са не по-силни от шампиона и не по-слаби от всички останали играчи.

Накратко, за да определим сребърния медалист, трябва да организираме втори турнир, но само между състезателите, елиминирани от шампиона. Ако те са k на брой, то вторият турнир ще съдържа $k - 1$ срещи. Срещите (сравненията) стават общо $(n - 1) + (k - 1) = n + k - 2$.

Числото k можем да намерим по следния начин. По определение k е броят на играчите, елиминирани от шампиона. Тъй като във всеки кръг шампионът играе най-много една среща, той елиминира най-много един състезател. Затова k не надхвърля броя на кръговете в турнира. В най-лошия случай (когато шампионът е играл по една среща във всеки кръг на първенството) числото k е равно на броя на кръговете.

След всеки кръг броят на играчите намалява наполовина; ако се получи дробно число, закръгля се нагоре (заради състезателя, който минава служебно). След изиграване на k кръга остават $\left\lceil \frac{n}{2^k} \right\rceil$ играчи. Но k е броят на всички кръгове; след тях остава един играч — шампионът.

Тоест $\left\lceil \frac{n}{2^k} \right\rceil = 1$, откъдето следва, че $k = \lceil \log_2 n \rceil$. Заместваме k във формулата по-горе: $n + k - 2 = n + \lceil \log_2 n \rceil - 2$. Това е броят на всички срещи за излъчване на сребърния медалист. С други думи, това е броят на сравненията между елементите на входния масив с цел намиране на втория най-голям елемент.

И така, оказва се, че за определянето на втория най-голям от общо n елемента са достатъчни $n + \lceil \log_2 n \rceil - 2$ сравнения между елементите.

По порядък първото събираемо в получения израз е по-голямо от другите две събираеми, затова те могат да се пренебрегнат: броят на сравненията по тази схема е приблизително n . По предишния метод броят на сравненията беше приблизително $2n$. Тоест новият алгоритъм е около два пъти по-бърз от предишния.

Едновременно търсене на най-голяма и най-малка стойност

Ако в даден числов масив $A[1 \dots n]$ търсим както най-голяма, така и най-малка стойност, то имаме избор между различни подходи.

Можем да намерим двете стойности поотделно. Първото търсене изисква $n - 1$ сравнения. Премахваме намерената стойност, затова второто търсене прави $n - 2$ сравнения. Получават се общо $2n - 3 \sim 2n = \Theta(n)$ сравнения. Това е оптимално по порядък (тривиална долна граница по размера на входа), обаче множителят 2 не е оптимален.

Броят на сравненията между елементите може да се намали с 25% чрез умело подреждане. Нека отново представим търсенето на екстремалните стойности като провеждане на турнир: все едно че търсим и най-силния, и най-слабия играч. За целта разделяме играчите по двойки. Всяка двойка играе една среща. Оформяме две групи от състезатели — победители и победени. Най-силният играч очевидно се намира сред победителите, а най-слабият е сред победените. Оттук нататък провеждаме два турнира с елиминации, поотделно във всяка от двете групи, само че в турнира между победителите елиминираме всеки състезател, който загуби среща, а в турнира между победените елиминираме състезателите, които печелят срещи. Накрая остават най-силният играч в първата група и най-слабият играч във втората. Именно те са търсените. По този начин се провеждат $\frac{n}{2} + 2 \left(\frac{n}{2} - 1 \right) = \frac{3}{2}n - 2 \sim \frac{3}{2}n = \Theta(n)$ срещи (сравнения).

Сметките в предишния абзац важат за четно n . Ако n е нечетно, отделяме един играч, който накрая се среща с двамата излъчени кандидати (едната среща може да се окаже излишна, обаче ние разглеждаме най-лошия случай). Така броят на срещите (сравненията) излиза общо $\left(\frac{3}{2}(n - 1) - 2 \right) + 2 = \frac{3}{2}(n - 1) \sim \frac{3}{2}n = \Theta(n)$.

Този алгоритъм е по-бърз от предишния, макар и не по порядък: спестява 25% от времето (множителят $3/2$ е равен на 75% от множителя 2 пред n).

Търсене на k -ти най-малък елемент, процентил и медиана

Даден е масив $A[1 \dots n]$ от n реални числа. Дадено е и цяло число k между 1 и n вкл. Търсим k -тия най-малък елемент на A . Ако сортираме масива, това ще бъде елементът $A[k]$. Обаче няма да го сортираме, защото това изисква време $\Omega(n \log n)$ при най-лоши входни данни. Вместо това ще предложим алгоритъм, който намира k -тия най-малък елемент за време $\Theta(n)$ при всякакви входни данни. Времето е оптимално по порядък заради тривиалната долна граница по размера на входа: трябва да прочетем всички елементи на масива, защото всеки един от тях може да се окаже k -тият най-малък елемент.

Тъй като k -тият най-малък елемент съвпада с $(n - k + 1)$ -ия най-голям, то не се налага да създаваме отделен алгоритъм за дуалната задача; ще търсим само k -тия най-малък елемент.

Процентил от порядък p (или p -процентил) се нарича такъв елемент, който е по-голям от $p\%$ от останалите елементи. Това е елементът, който би притежавал индекс $k = \frac{(n-1)p}{100} + 1$, ако масивът беше сортиран. Когато стойността на този израз е дробно число, то или се закръгля, или се извършва интерполация между две стойности, съседни по големина.

Различните учебници предлагат определения, които се различават малко в подробностите. Понеже $k = \frac{(n-1)p}{100} + 1 = \frac{np}{100} + (1 - \frac{p}{100}) \approx \frac{np}{100}$ при големи n , понякога за простота се използва приближението $k \approx \frac{np}{100}$ вместо по-точния израз $k = \frac{(n-1)p}{100} + 1$. Тези разлики нямат значение, защото такива пресмятания се правят предимно в статистически изследвания, а техните данни поначало са непълни (най-често се работи с някаква извадка вместо с генералната съвкупност) и неточни (например величини, измерени с някаква грешка). От алгоритмична гледна точка посочените малки разлики също нямат значение: така или иначе се решава задачата за търсене на k -тия най-малък елемент, само че сега k не е константа, а е подходяща функция на n и p , като различните изследователи избират различни, макар и приблизително равни функции.

Особено важен и за практиката, и за теорията е 50-ият процентил. Той се нарича медиана. С други думи, медианата е средният по големина елемент, когато елементите са нечетен брой. Ако броят n на елементите на масива е четно число, тогава има не един, а два средни елемента. Маловажно е кой от тях ще изберем. Най-често се избира тяхното средно аритметично.

Важно свойство на медианата е, че тя е устойчива на влиянието на екстремални стойности, за разлика от средното аритметично. Да вземем например числовия масив $A = (1; 4; 8; 15; 16)$. Неговата медиана е 8, а средното аритметично е 8,8. Вижда се, че те не съвпадат, но са близки. Ако променим обаче най-голямото число в масива от 16 на 160, тоест ако $A = (1; 4; 8; 15; 160)$, то средното аритметично ще се увеличи от 8,8 на 37,6, а медианата ще остане 8.

Не е нужен отделен алгоритъм за търсене на медиана. Тя е частен случай на процентил, а търсенето на процентил се свежда до търсене на k -ти най-малък елемент. Остава да решим само последната задача. За нея има поне два метода — алгоритъма PICK и т. нар. бързо търсене. Бързото търсене ще бъде разгледано по-нататък, заедно с алгоритъма, наречен бързо сортиране, с който има много общо. Тук ще опишем забележителния алгоритъм PICK, съставен през 1972 г. от Мануел Блум, Роберт Флойд, Воган Прат, Роналд Ривест и Роберт Тарджан.

Алгоритъм PICK за търсене на k -тия най-малък елемент на числовия масив $A[1 \dots n]$:

- 1) Копираме елементите на масива $A[1 \dots n]$ в таблица с 21 реда и $\left\lceil \frac{n}{21} \right\rceil$ стълба.
- 2) Във всеки стълб сортираме числата така, че да растат отгоре надолу.
- 3) Рекурсивно намираме медианата M на числата от ред № 11.
- 4) Разделяме ред № 11 относно M , като разместяваме целите стълбове.
- 5) Намираме $\ell_1 = |\{1 \leq k \leq n : A[k] < M\}|$ и $\ell_2 = |\{1 \leq k \leq n : A[k] \leq M\}|$.
- 6) Ако $\ell_1 < k \leq \ell_2$, алгоритъмът приключва работа с резултат M .
- 7) Ако $k \leq \ell_1$, изтриваме елементите в IV квадрант на таблицата.
- 8) Ако $k > \ell_2$, изтриваме елементите във II квадрант и намаляваме k с техния брой.
- 9) Рекурсивно изпълняваме алгоритъма върху останалите елементи.

В това описание умишлено са пропуснати редица технически подробности, които са нужни, ако искаме да програмираме алгоритъма, но не влияят съществено на идеята.

Броят 21 на редовете може да бъде заменен (в разумни граници) с друго нечетно число. Тогава вместо ред № 11 се взима средният ред на таблицата. Тук използваме таблица с 21 реда, както е в оригиналната публикация на алгоритъма.

В нашето описание дъното на рекурсията не се съдържа в явен вид, а само се подразбира. Алгоритъмът се изпълнява по този начин само за достатъчно големи масиви, тоест за $n \geq n_0$ (където n_0 е твърдо заложено в кода на алгоритъма). Очевидно $n_0 > 21$, но точната стойност не е от значение за алгоритъма. Масивите с дължина $n < n_0$ се обработват нерекурсивно. Може например да се сортират, след което да се връща техният k -ти елемент. Точният начин не е съществен за порядъка на времевата сложност: асимптотичните оценки важат при $n \rightarrow \infty$, следователно $n \geq n_0$.

На стъпка № 1 от алгоритъма може n да не се дели на 21. Тогава броят на стълбовете се закръглява нагоре, затова елементите на масива не стигат за попълването на таблицата. Допълваме таблицата с много големи числа ($+\infty$); това не променя отговора на задачата.

На стъпка № 2 сортираме всеки стълб поотделно. Не е казано коя сортировка ползваме, но това отново няма значение за порядъка на времевата сложност (има значение единствено за константния множител пред порядъка): размерът 21 на всеки стълб не зависи от n , затова сортирането на отделен стълб изразходва време $\Theta(1)$, а общо за стълбовете времето е $\Theta(n)$.

На стъпка № 4 прилагаме операцията разделяне към средния ред (№ 11) на таблицата. Разделяме го спрямо неговата медиана M и тя става среда на реда и център на таблицата. При разделянето се разместват не само елементите на ред № 11, а стълбовете, тоест всеки стълб се движи като едно цяло. Затова е добре да представим таблицата не като масив от стълбове, а като масив от указатели към стълбове: за да не разместваме всичките 21 елемента на стълб, а само указателя към него. Така стъпка № 4 ще се ускори 21 пъти, но порядъкът на времето ще остане $\Theta(n)$, колкото поначало е времето за разделяне на масив.

След стъпка № 4 числото M се намира в центъра на таблицата. Неговият ред и стълб разделят таблицата на четири части, които в описанието на алгоритъма нарекохме квадранти по аналогия с правоъгълната координатна система.

II квадрант малки елементи	малки елементи	I квадрант
малки елементи	M	големи елементи
III квадрант	големи елементи	IV квадрант големи елементи

Заради разделянето на средния ред спрямо M , извършено на стъпка № 4, елементите му наляво от M са по-малки или равни на M , а надясно от M са по-големи или равни на M .

Заради сортирането на стълбовете, извършено на стъпка № 2, елементите в средния стълб нагоре от M са по-малки или равни на M , а надолу от M са по-големи или равни на M .

Пак заради сортирането, като сравним елементите във всеки стълб със средния му елемент, стигаме до извода, че всички елементи във II квадрант на таблицата са по-малки или равни на M , а елементите в IV квадрант са по-големи или равни на M .

Елементите от I и III квадрант не могат да бъдат сравнени с M , тоест те могат да бъдат по-големи, по-малки или равни на M .

Коректност на стъпка № 6: От определенията на ℓ_1 и ℓ_2 от стъпка № 5 при $\ell_1 < k \leq \ell_2$ следва, че k -тата най-малка стойност е M и алгоритъмът правилно връща M .

Коректност на стъпка № 7: Ако $k \leq \ell_1$, то k -тата най-малка стойност е по-малка от M , затова можем да изтрием всички елементи, за които знаем, че са по-големи или равни на M : елементите от IV квадрант на таблицата, включително центъра M , ред № 11 надясно от M и средния стълб надолу от M .

Коректност на стъпка № 8: Ако $k > \ell_2$, то k -тата най-малка стойност е по-голяма от M , затова можем да изтрием всички елементи, за които знаем, че са по-малки или равни на M : елементите от II квадрант на таблицата, включително центъра M , ред № 11 наляво от M и средния стълб нагоре от M . Намаляваме k с броя на изритите елементи.

Техническа подробност по стъпки № 7 и № 8: Няма как да изтриваме елементи на масив (мястото на елемент в паметта не може просто да изчезне). Изтриване тук означава следното. Преди рекурсивното обръщение на ред № 9 таблицата се копира обратно в масива $A[1 \dots n]$. Елементите, за които е казано, че са изтрити, не се копират. При рекурсивното обръщение към алгоритъма се подават оригиналният (непромененият) указател A към началото на масива и намалено цяло число n (оригиналната дължина минус броя на изритите елементи).

Анализ на времевата сложност на алгоритъма RICK: Нека $T(n)$ е времевата сложност в най-лошия случай — когато ред № 6 не се изпълнява никога и алгоритъмът връща стойност от дъното на рекурсията (тоест достига до къс масив и го обработва nereкурсивно).

Стъпки № 1, № 2, № 4 и обхождането на масива на стъпка № 5 изразходват време $\Theta(n)$. Копирането на числа от таблицата към масива преди рекурсивното обръщение на стъпка № 9 също е с линейна времева сложност. В зависимост от това, как точно е декларирана таблицата, може да се наложи да копираме ред № 11 от нея преди рекурсивното обръщение на стъпка № 3; това копиране също е с линейна времева сложност $\Theta(n)$. Проверките на редове № 6, № 7 и № 8 се изпълняват за константно време $\Theta(1)$. Окончателно, времето за едно равнище на рекурсията е линейно: $\Theta(n)$; по-точно, то има вида cn за подходяща константа $c > 0$, която е сборът от времената, нужни за всички операции с един елемент — сравняване на стойността му с M и копиране съответен брой пъти.

Рекурсията на стъпка № 3 се изпълнява върху един от всичките 21 реда на таблицата. Тоест тя засяга $\frac{1}{21}$ от данните, затова нейната времева сложност е $T\left(\frac{n}{21}\right)$.

Рекурсията на стъпка № 9 приема около $\frac{3}{4}$ от всички елементи: на стъпка № 7 или № 8 сме изтрили около $\frac{1}{4}$ от елементите. По-точно, изтрили сме 11 реда от 21 (вкл. средния ред), но не изцяло, а само около половината стълбове (разликата от половината има порядък $\Theta\left(\frac{1}{n}\right)$ и е незначителна). И така, изтрили сме $\frac{1}{2} \cdot \frac{11}{21} = \frac{11}{42}$ от елементите, следователно остават $\frac{31}{42}$ от тях. Затова рекурсията на стъпка № 9 има времева сложност $T\left(\frac{31n}{42}\right)$.

Общото време за изпълнение на алгоритъма е

$$T(n) = T\left(\frac{31n}{42}\right) + T\left(\frac{n}{21}\right) + cn, \quad \forall n \geq n_0.$$

Понеже времето е неотрицателно, то оттук следва, че

$$T(n) \geq cn, \quad \forall n \geq n_0.$$

Полагаме

$$b = \max \left\{ \frac{T(1)}{1}, \frac{T(2)}{2}, \dots, \frac{T(n_0-1)}{n_0-1}, \frac{42c}{9} \right\}.$$

Тогава е в сила неравенството

$$T(n) \leq bn, \quad \forall n \geq 1.$$

Ще го докажем с математическа индукция.

База: Нека $1 \leq n < n_0$. От определението на константата b е очевидно, че $b \geq \frac{T(n)}{n}$.

Следователно $T(n) \leq bn$, $\forall n < n_0$.

Индуктивна стъпка: Нека $n \geq n_0$ и неравенството важи за всички допустими стойности на аргумента на T , по-малки от текущия n . Ще докажем, че неравенството важи и за него:

$$T(n) = T\left(\frac{31n}{42}\right) + T\left(\frac{n}{21}\right) + cn \leq \frac{31}{42}bn + \frac{1}{21}bn + cn \leq \frac{31}{42}bn + \frac{2}{42}bn + \frac{9}{42}bn = bn,$$

което трябваше да се докаже. В хода на доказателството използвахме неравенството $c \leq \frac{9}{42}b$, което следва от определението на b .

Дотук направихме оценка отгоре и оценка отдолу на времевата сложност. От тези оценки следва двойното неравенство

$$cn \leq T(n) \leq bn, \quad \forall n \geq n_0.$$

Оттук $T(n) = \Theta(n)$ по определение. Тоест времевата сложност на алгоритъма RICK е линейна при най-лоши входни данни. Нейният порядък е оптимален заради тривиалната долна граница по размера на входа.

Появата на алгоритъма RICK е била доста изненадваща: дотогава не се е предполагало, че търсенето на k -ти най-малък елемент може да се осъществи с линейна времева сложност дори при най-лоши входни данни.

Анализ на сложността по памет на алгоритъма PICK: Нека $M(n)$ е сложността по памет. Ще видим, че нейният порядък не зависи от входните данни.

За едно равнище на рекурсията се изразходва памет $\Theta(n)$. Тя е необходима за таблицата, в която се копира входният масив $A[1 \dots n]$ на стъпка № 1, а евентуално още за копирането на средния ред на таблицата при рекурсията от стъпка № 3.

Ясно е, че толкова допълнителна памет се изразходва дори при най-добри входни данни (когато алгоритъмът приключва работа на стъпка № 6 веднага — без рекурсивни извиквания). Следователно $M(n) = \Omega(n)$ при всякакви входни данни.

Сега да разгледаме най-лошия случай — когато ред № 6 не се изпълнява и алгоритъмът връща стойност от дъното на рекурсията (т.е. достига до къс масив и го обработва нерекурсивно). Може да се наложи да сумираме допълнителната памет, изразходвана от различните равнища на рекурсията. По-просто е обаче да се позовем на тривиалната долна граница на времето по размера на паметта: $T(n) = \Omega(M(n))$. Тук паметта е долна граница за времето, а пък то е горна граница за паметта: $M(n) = O(T(n))$. Понеже $T(n) = \Theta(n)$, то $M(n) = O(n)$.

От двете асимптотични оценки $M(n) = \Omega(n)$ и $M(n) = O(n)$ следва, че $M(n) = \Theta(n)$.

С други думи, алгоритъмът PICK работи бързо, но изразходва много памет.

Този недостатък — голямото количество допълнителна памет — може да бъде преодолян чрез внимателно програмиране на алгоритъма.

Таблицата от стъпка № 1 е излишна. Тя е само “изглед” към масива. Можем да си мислим, че подмасивът $A[1 \dots 21]$ е първият стълб на таблицата, $A[22 \dots 42]$ — вторият ѝ стълб, $A[43 \dots 63]$ — третият и т.н. Тоест разцепваме входния масив на подмасиви с дължина 21 и смятаме всеки от тях за стълб на таблицата (последният подмасив може да има дължина, по-малка от 21). Така стъпка № 1 отпада.

На стъпка № 2 сортираме всеки от споменатите подмасиви.

Стъпки № 3 и № 4 привидно изискват копиране на елементи — заделяне на памет за ред № 11 от бившата таблица. В действителност няма нужда от копиране. Достатъчно е да разместим елементите на масива така, че елементите от бившия ред № 11 да дойдат в началото на масива. Тоест разместваме $A[1]$ и $A[11]$, $A[2]$ и $A[32]$, $A[3]$ и $A[53]$... По този начин пестим памет, но пък увеличаваме времето за работа на алгоритъма. Обаче увеличението е от порядък $\Theta(n)$, затова не променя нито рекурентното уравнение, нито неговото решение. Времето остава $\Theta(n)$, увеличава се само ненаписаният константен множител пред порядъка.

В стъпки № 5 и № 6 няма промяна.

На стъпки № 7 и № 8, в които се говори за изтриване на елементи, всъщност се извършваше копиране на елементи от таблицата към входния масив. Вече няма таблица, отпада и копирането. Заместваме го с разделяне на масива относно M .

На стъпка № 7 правим класическо разделяне: първо малките, после големите елементи.

На стъпка № 8 разделянето е по обратния начин: първо големите, после малките елементи (за целта е нужно да “обърнем” съответния алгоритъм за разделяне).

Кой алгоритъм за разделяне да използваме на стъпки № 7 и № 8? Тук е важна бързината, а не устойчивостта на разделянето, следователно трябва да използваме разделянето по Хоор.

След тези оптимизации става ясно, че алгоритъмът PICK може да търси на място — без да копира елементите на масива. Но това не значи, че сложността по памет е константа. Само количеството допълнителна памет за едно равнище на рекурсията е някаква константа a . Обаче всяко равнище на рекурсията притежава собствена памет.

Рекурсивното обръщение на стъпка № 9 е опашково, затова то лесно се заменя с цикъл: намаляваме n до дължината на подмасива от началните елементи (които са пред разделителя) след разделянето от стъпка № 7 или № 8; после правим преход към началото на алгоритъма и вече обработваме подмасива. Това означава, че вече изтриваме не само II или IV квадрант, а всички малки или всички големи елементи (каквито може да има също в I или III квадрант), затова на стъпка № 8 намаляваме k с броя на всички малки елементи, тоест с ℓ_2 . Тази промяна ускорява алгоритъма, но не по порядък: порядъкът на времевата сложност вече е минимален.

За съжаление, рекурсията на стъпка № 3 е неопашкова. Допълнителна памет се изисква например за дължината n на входния масив. На стъпка № 3 се подава 21 пъти по-къс масив, но новото n не може да се запише на мястото на старото, тъй като старото n ще бъде нужно и на следващите стъпки, например на стъпка № 5. Следователно всяко равнище на рекурсията пази собствена стойност на n . Общото количество допълнителна памет за целия алгоритъм е $a + a + a + \dots + a$; броят на събираемите в този сбор е равен на дълбочината на рекурсията. При всяко рекурсивно извикване дължината на масива се дели на 21, докато стигне дъното n_0 , затова дълбочината на рекурсията $\approx \log_{21} \left(\frac{n}{n_0} \right) = \log_{21} n - \log_{21} n_0$ и сложността по памет е $M(n) = a (\log_{21} n - \log_{21} n_0) = \Theta(\log n)$, което е много по-добро от предишната версия.

Окончателно, съществува реализация на алгоритъма PICK, която в най-лошия случай има времева сложност $T(n) = \Theta(n)$ и сложност по памет $M(n) = \Theta(\log n)$. Това превръща PICK в ефективен алгоритъм за търсене на k -ти най-малък елемент, процентил и медиана.

СОРТИРАНЕ

Сортирането е операция над масив или списък. Тук ще работим с числов масив $A[1 \dots n]$, но алгоритмите ще бъдат приложими за произволен масив, между чиито елементи е дефинирана релация на линейна наредба (тоест всеки два елемента са сравними). Да сортираме масив, означава да подредим елементите му в ненамаляваща редица: $A[1] \leq A[2] \leq A[3] \leq \dots \leq A[n]$. За тази цел са разработени множество алгоритми.

Елементарни сортировки

Най-простите сортиращи алгоритми са методът на мехурчето, сортирането чрез вмъкване и сортирането чрез пряк избор.

Методът на мехурчето обхожда дадения масив, сравнява всеки два последователни елемента и ако са в неправилен ред, ги разменя. Масивът се обхожда пак и пак, докато се установи, че вече е сортиран. Големите или малките елементи (според посоката на обхождане на масива) изплуват като въздушни мехурчета във вода, тоест бързо намират местата си.

Сортирането чрез вмъкване обхожда масива, като на всяка стъпка обходените елементи образуват сортиран масив. Текущият елемент се вмъква сред предишните на подходящо място — така, че заедно с тях да образува ненамаляваща редица.

Сортирането чрез пряк избор търси най-малкия елемент и го поставя на първо място, най-малкия сред останалите елементи — на второ място и тъй нататък.

И трите алгоритъма са итеративни и сортират на място, затова имат сложност по памет $M(n) = \Theta(1)$ при всякакви входни данни: локалните променливи са все от примитивен тип.

Сложността по време също е еднаква за трите алгоритъма — квадратична: $T(n) = \Theta(n^2)$ в най-лошия случай — когато $A[1] > A[2] > A[3] > \dots > A[n]$. Тези входни данни са най-лоши за трите сортировки, но анализите на времевата сложност се различават в някои подробности.

Всяка сортировка има някое предимство, което я прави подходяща в определени случаи:

— Методът на мехурчето е най-простият от всички алгоритми за сортиране. Това го прави особено удобен за запознаване на начинаещи програмисти с операцията сортиране.

— Сортирането чрез вмъкване е най-бързият алгоритъм за сортиране при къси масиви (докъм десет-двайсет елемента). Затова служи за дъно на рекурсията при някои сортировки от типа “разделяй и владей”.

— Сортирането чрез пряк избор се отличава с най-малък брой размествания на елементи (въпреки че броят на сравненията остава квадратичен). Това му дава предимство в задачи, в които сравнението е бърза операция, а разместването — бавна. Ако някакви предмети трябва да се подредят чрез механично преместване, тогава е важен именно броят на разместванията: механичното движение е бавно в сравнение с електрическите процеси.

<p>МЕТОД НА МЕХУРЧЕТО ($A[1 \dots n]$)</p> <pre> 1) $h \leftarrow n$ 2) repeat 3) $\text{sorted} \leftarrow \text{true}$ 4) $h \leftarrow h - 1$ 5) for $k \leftarrow 1$ to h do 6) if $A[k] > A[k+1]$ 7) $\text{sorted} \leftarrow \text{false}$ 8) $\text{swap}(A[k], A[k+1])$ 9) until sorted </pre>	<p>СОРТИРАНЕ ЧРЕЗ ВМЪКВАНЕ ($A[1 \dots n]$)</p> <pre> 1) for $k \leftarrow 2$ to n do 2) $r \leftarrow k$ 3) while $r > 1$ and $A[r] < A[r-1]$ do 4) $\text{swap}(A[r], A[r-1])$ 5) $r \leftarrow r - 1$ </pre>	<p>СОРТИРАНЕ ЧРЕЗ ПРЯК ИЗБОР ($A[1 \dots n]$)</p> <pre> 1) for $k \leftarrow 1$ to $n - 1$ do 2) $m \leftarrow k$ 3) for $\ell \leftarrow k + 1$ to n do 4) if $A[\ell] < A[m]$ 5) $m \leftarrow \ell$ 6) if $m > k$ 7) $\text{swap}(A[k], A[m])$ </pre>
<p>Методът на мехурчето е устойчив, тъй като разменя само <i>различни</i> съседни елементи.</p>	<p>Сортирането чрез вмъкване е устойчиво, защото разменя единствено <i>различни</i> съседни елементи.</p>	<p>Сортирането по метода на прякия избор е неустойчиво: $A[k]$ може да се размени с равен елемент.</p>
<p>Сложност по памет: $M(n) = \Theta(1)$ при всички възможни входни данни. Допълнителна памет се изразходва само за променливите h, k и sorted, а те са все от примитивни типове — целочислен и логически.</p>	<p>Сложност по памет: $M(n) = \Theta(1)$ при всички възможни входни данни. Допълнителна памет се изразходва само за променливите k и r, които са от примитивен тип — цели числа.</p>	<p>Сложност по памет: $M(n) = \Theta(1)$ при всички възможни входни данни. Допълнителна памет се изразходва само за индексите k, ℓ и m, които са от примитивен тип — цели числа.</p>
<p>Сложност по време: $T(n) = \Theta(n^2)$ при най-лоши входни данни: $A[1] > A[2] > A[3] > \dots > A[n]$.</p>	<p>Сложност по време: $T(n) = \Theta(n^2)$ при най-лоши входни данни: $A[1] > A[2] > A[3] > \dots > A[n]$.</p>	<p>Сложност по време: $T(n) = \Theta(n^2)$ при най-лоши входни данни: $A[1] > A[2] > A[3] > \dots > A[n]$.</p>
<p>Предимство: простотата на идеята.</p>	<p>Предимство: бързодействие при къси масиви.</p>	<p>Предимство: най-малко размествания.</p>

За да обосновем твърдението за квадратичната времева сложност на трите алгоритъма, ще анализираме всеки от тях подробно — брой сравнения и брой размествания поотделно.

Най-лошият случай за бързодействието на метода на мехурчето е, когато входните данни образуват строго намаляваща числова редица. Тогава проверката на ред № 6 винаги е успешна, поради което редове № 7 и № 8 се изпълняват винаги. Следователно броят на разместванията е равен на броя на сравненията, а именно

$$\sum_{h=1}^{n-1} \sum_{k=1}^h 1 = \sum_{h=1}^{n-1} h = \frac{(n-1)n}{2} = \frac{n^2 - n}{2} = \Theta(n^2).$$

Общият брой сравнения и размествания е $n^2 - n = \Theta(n^2)$. Това е времевата сложност на метода при най-лоши входни данни. Тъй като h намалява, първите събираеми в сбора са най-големи, а последните събираеми са най-малки: $(n-1) + (n-2) + (n-3) + \dots + 3 + 2 + 1 = \frac{n^2 - n}{2}$. Когато елементите на масива образуват строго намаляваща редица, в хода на сортирането първият елемент отива на последното място ($n-1$ размествания), новият първи елемент отива на предпоследното място ($n-2$ размествания) и т.н.

Що се отнася до сортирането чрез вмъкване, най-лошият случай за бързодействието е този, в който тялото на вътрешния цикъл се изпълнява най-много пъти. За целта текущият елемент трябва да отиде на първото място, затова той трябва да е по-малък от всички предишни елементи, тоест входният масив трябва да е строго намаляваща числова редица. За да отиде k -тият елемент на първото място, са нужни $k-1$ размествания за всяко k от 2 до n включително. Този път първите събираеми са малки, а последните са големи: $1 + 2 + 3 + \dots + (n-1) = \frac{n^2 - n}{2}$. Това е броят на разместванията, а също и броят на сравненията между елементи, защото има взаимноеднозначно съответствие между операциите: всяко сравнение между елементи на ред № 3 е последвано от разместване на елементи на ред № 4 от алгоритъма. Общият брой размествания и сравнения между елементи на масива е $n^2 - n = \Theta(n^2)$. Това е също времевата сложност на сортирането чрез вмъкване при най-лоши входни данни.

При сортирането чрез пряк избор има разлика между двете бройки. Броят на сравненията между елементи на масива е

$$\sum_{k=1}^{n-1} \sum_{\ell=k+1}^n 1 = \sum_{k=1}^{n-1} (n-k) = (n-1) + (n-2) + \dots + 3 + 2 + 1 = \frac{(n-1)n}{2} = \frac{n^2 - n}{2} = \Theta(n^2)$$

при всякакви входни данни. Тъй като за всяко k от 1 до $n-1$ вкл. ред № 7 на алгоритъма се изпълнява най-много веднъж, то броят на разместванията на елементи е не по-голям от $n-1$. Този брой размествания се достига например за масива $A = (n; 1; 2; 3; \dots; n-1)$, който затова е един най-лош случай за броя на разместванията (има и други най-лоши случаи). Ето защо броят на разместванията е $n-1 = \Theta(n)$ при най-лоши входни данни, тоест по брой размествания сортирането чрез пряк избор е по-добро от другите сортировки. Но общата му времева сложност (брой на разместванията плюс брой на сравненията между елементи на масива) е равна на $\frac{n^2 - n}{2} + (n-1) = \frac{n^2 + n - 2}{2} = \Theta(n^2)$ при всякакви входни данни. Затова, ако ни интересува общото време за изпълнение, сортирането чрез пряк избор не е по-добро от другите сортировки. То ги превъзхожда само по брой размествания на елементи. Освен това, тъй като общото време на сортирането чрез пряк избор е едно и също по порядък при всевъзможни входни данни, то, формално погледнато, всички случаи са най-лоши, в това число и случаят, когато масивът е строго намаляваща редица, но по порядък на общото време този случай не е по-лош от другите.

Доказателствата на инвариантите на вложени цикли се влагат едно в друго по същия начин. За пример ще докажем коректността на сортирането чрез вмъкване. Формално погледнато, твърденията, че k и r са цели числа, би трябвало да бъдат част от инвариантите, но за простота ги споменаваме тук с очевидната обосновка, че те се инициализират с целочислени стойности и после само се увеличават или намаляват с единица. Ясно е също, че $k > 1$, защото k приема целочислените стойности от 2 до $n+1$ включително.

Инвариант на външния цикъл: Всеки път, когато се изпълнява проверката за край $k \leq n$, са в сила неравенствата $A[1] \leq A[2] \leq A[3] \leq \dots \leq A[k-1]$.

Инвариант на вътрешния цикъл: При всяка проверка за край на вътрешния цикъл важат четири групи от неравенства: $1 \leq r \leq k$, $A[1] \leq A[2] \leq A[3] \leq \dots \leq A[r-1]$ при $r > 1$, $A[r-1] \leq A[r+1]$ при $1 < r < k$ и $A[r] \leq A[r+1] \leq A[r+2] \leq \dots \leq A[k]$.

Доказателство на инварианта на външния цикъл: с помощта на математическа индукция по поредния номер на проверката за край на външния цикъл.

База: Първата проверка се изпълнява при влизане във външния цикъл. От ред № 1 следва, че в този миг $k = 2$ и инвариантът е конюнкция от $k - 2 = 0$ неравенства, т.е. празна конюнкция, а тя е тривиално вярна.

Индуктивна стъпка: Нека инвариантът на външния цикъл е в сила при някоя проверка, която не е последна: $A[1] \leq A[2] \leq A[3] \leq \dots \leq A[k-1]$. Ще докажем, че инвариантът важи и при следващата проверка за край на външния цикъл, т.е. $A[1] \leq A[2] \leq A[3] \leq \dots \leq A[\tilde{k}-1]$, където \tilde{k} е стойността на брояча при следващата проверка.

Действително, щом текущата проверка за край на външния цикъл не е последна, то $k \leq n$ и се изпълнява тялото на външния цикъл. Ред № 2 присвоява стойност k на променливата r , след което алгоритъмът преминава към ред № 3 — проверката за край на вътрешния цикъл.

Доказателството на инварианта на вътрешния цикъл се извършва чрез индукция по поредния номер на неговата проверка за край.

База: Първата проверка за край се изпълнява при влизане във вътрешния цикъл. Тогава $r = k$ заради ред № 2. Неравенството $1 \leq r \leq k$ е изпълнено, защото $r = k > 1$. В конюнкцията $A[r] \leq A[r+1] \leq A[r+2] \leq \dots \leq A[k]$ има $k - r = 0$ неравенства, тоест тя е празна, поради което е тривиално вярна. А пък редицата от неравенства $A[1] \leq A[2] \leq A[3] \leq \dots \leq A[r-1]$ при $r = k$ приема следната равносилна форма $A[1] \leq A[2] \leq A[3] \leq \dots \leq A[k-1]$, която съвпада с индуктивното предположение за инварианта на външния цикъл.

Индуктивна стъпка: Нека $1 \leq r \leq k$ и $A[r-1] \leq A[r+1]$ при $1 < r < k$, $A[1] \leq A[2] \leq A[3] \leq \dots \leq A[r-1]$ при $r > 1$ и $A[r] \leq A[r+1] \leq \dots \leq A[k]$ по време на някоя непосредна проверка за край на вътрешния цикъл. При следващата проверка ще важат неравенствата $1 \leq \tilde{r} \leq k$ и $A[\tilde{r}-1] \leq A[\tilde{r}+1]$ при $1 < \tilde{r} < k$, $A[1] \leq A[2] \leq A[3] \leq \dots \leq A[\tilde{r}-1]$ при $\tilde{r} > 1$ и $A[\tilde{r}] \leq A[\tilde{r}+1] \leq \dots \leq A[k]$; тук с \tilde{r} е означена стойността на едноименния индекс при следващата проверка.

Действително, щом текущата проверка не е последна, то $r > 1$ и $A[r] < A[r-1]$. Оттук и от индуктивното предположение за инварианта на вътрешния цикъл следват групите неравенства $2 \leq r \leq k$, $A[r] \leq A[r-1] \leq A[r+1] \leq \dots \leq A[k]$ при $r < k$ и $A[1] \leq A[2] \leq \dots \leq A[r-2] \leq A[r-1]$. Последната група е непразна при $r > 2$; отделяме последното неравенство от нея: $A[r-2] \leq A[r-1]$; така остава редицата $A[1] \leq A[2] \leq \dots \leq A[r-2]$ при $r > 2$. След като ред № 4 от кода на алгоритъма размени стойностите на елементите $A[r]$ и $A[r-1]$, четирите групи от неравенства приемат вида: $2 \leq r \leq k$, $A[r-1] \leq A[r] \leq A[r+1] \leq \dots \leq A[k]$ (вкл. при $r = k$), $A[1] \leq A[2] \leq \dots \leq A[r-2]$ при $r > 2$ и $A[r-2] \leq A[r]$ при $r > 2$. Ред № 5 от кода намалява r с единица: $\tilde{r} = r - 1 < k$. Заместваме $r = \tilde{r} + 1$ в неравенствата и те приемат следната форма: $1 \leq \tilde{r} \leq k - 1 < k$, $A[\tilde{r}-1] \leq A[\tilde{r}+1]$ при $\tilde{r} > 1$, $A[1] \leq A[2] \leq A[3] \leq \dots \leq A[\tilde{r}-1]$ при $\tilde{r} > 1$ и $A[\tilde{r}] \leq A[\tilde{r}+1] \leq \dots \leq A[k]$. Това приключва доказателството на инварианта на вътрешния цикъл.

Продължаваме разсъжденията от индуктивната стъпка на инварианта на външния цикъл. Вече имаме право да използваме доказани инвариант на вътрешния цикъл. Ще го приложим към последната проверка за край на вътрешния цикъл, но първо ще докажем, че тя съществува.

Полуинвариант на вътрешния цикъл: променливата r . Тя намалява строго (с единица) след всяко изпълнение на тялото на вътрешния цикъл. Започва от k и намалява най-много до 1, затова вътрешният цикъл се изпълнява не повече от $k - 1$ пъти, т.е. завършва рано или късно.

Да разгледаме последната проверка за край на вътрешния цикъл. Понеже той не съдържа команди за преход навън (return, break и go to), то логическият израз на ред № 3 от алгоритъма има стойност “лъжа” при последната проверка. Следователно $r \leq 1$ или $A[r] \geq A[r - 1]$.

Първи случай: Нека $r \leq 1$. От доказани инвариант на вътрешния цикъл знаем, че $r \geq 1$. Следователно $r = 1$. От същия инвариант знаем, че $A[r] \leq A[r + 1] \leq A[r + 2] \leq \dots \leq A[k]$. Заместваме $r = 1$ и редицата от неравенства приема вида: $A[1] \leq A[2] \leq A[3] \leq \dots \leq A[k]$.

Втори случай: Нека $r > 1$ и $A[r] \geq A[r - 1]$. Второто от тези неравенства съединяваме с двете редици неравенства от инварианта на вътрешния цикъл; получаваме следната верига: $A[1] \leq A[2] \leq A[3] \leq \dots \leq A[r - 1] \leq A[r] \leq A[r + 1] \leq A[r + 2] \leq \dots \leq A[k]$.

Както видяхме, и в двата случая важат неравенствата $A[1] \leq A[2] \leq A[3] \leq \dots \leq A[k]$. Със завършването на вътрешния цикъл приключва изпълнението на тялото на външния цикъл и алгоритъмът преминава към ред № 1 — увеличава брояча k на външния цикъл с единица и проверява условието за край на външния цикъл. Новата стойност на брояча е $\tilde{k} = k + 1$. В доказаната редица от неравенства $A[1] \leq A[2] \leq A[3] \leq \dots \leq A[k]$ заместваме $k = \tilde{k} - 1$, след което тя приема желаните запис: $A[1] \leq A[2] \leq A[3] \leq \dots \leq A[\tilde{k} - 1]$.

Дотук доказахме инварианта на външния цикъл. Предстои да приложим инварианта към последната проверка за край на цикъла. Но първо трябва да се убедим, че той завършва.

Полуинвариант на външния цикъл: броячът k . Нараста с единица след всяко изпълнение на тялото на цикъла. Приема целочислени стойности от 2 до $n + 1$ включително, следователно тялото на външния цикъл се изпълнява точно $n - 1$ пъти.

При последната проверка за край на външния цикъл броячът $k = n + 1$ и освен това важи инвариантът: $A[1] \leq A[2] \leq A[3] \leq \dots \leq A[k - 1]$. Заместваме $k = n + 1$ в инварианта и правим извод, че $A[1] \leq A[2] \leq A[3] \leq \dots \leq A[n]$, т.е. масивът $A[1 \dots n]$ е сортиран. После алгоритъмът приключва работа, така че резултатът е сортираният масив. Така се убеждаваме, че сортирането чрез вмъкване е коректен алгоритъм.

Всъщност тук има тънкости, срещани и при други алгоритми. За да е пълно доказателството, трябва да установим и това, че изходът е пермутация на входа, а не просто сортиран масив с някакви числови стойности, взети незнайно откъде. Що се отнася до трите алгоритъма по-горе, това твърдение е очевидно: те не присвояват стойности на елементите на масива направо, а само ги разместват; ясно е, че така се получават само пермутации на входните данни. Формално погледнато, това допълнение трябваше да бъде част от инвариантите на циклите, но предпочетохме да не усложняваме доказателствата, които и без това имат сложна структура.

Ефективни сортировки

Ефективността при алгоритмите означава бързина или икономично използване на паметта. Тъй като паметта често е евтин ресурс, ще съсредоточим усилията си върху бързодействието.

Елементарните сортировки са сравнително бавни: те имат квадратична времева сложност. Съществуват по-бързи сортировки — с време $T(n) = \Theta(n \log n)$ при най-лоши входни данни. Такива са например сортирането чрез сливане, пирамидалното сортиране и др.

Сортирането чрез сливане е от тип “разделяй и владей”: дели масива на две равни части, сортира всяка от тях рекурсивно и ги слива в един сортиран масив.

Пирамидалното сортиране разчита на специална структура от данни — т. нар. пирамида. Елементите на масива се добавят в пирамидата един по един. После се извличат от нея пак един по един, всеки път — най-големият елемент. Времето на двете операции е $\Theta(\log n)$.

Ще разгледаме подробно тези две сортировки.

СЛИВАНЕ ($A_1[1 \dots n_1]$, $A_2[1 \dots n_2]$, $A[1 \dots n]$)

- 1) // Вход: A_1 и A_2 — предварително сортирани числови масиви.
- 2) // Изход: A — масив, в който елементите на A_1 и A_2 се записват в ненамаляващ ред.
- 3) // При равенство между елемент на A_1 и елемент на A_2
- 4) // елементът на A_1 се записва първи в изхода.
- 5) // Изискване: $n_1 + n_2 = n$.
- 6) $k \leftarrow 1$
- 7) $k_1 \leftarrow 1$
- 8) $k_2 \leftarrow 1$
- 9) **while** $k_1 \leq n_1$ **or** $k_2 \leq n_2$ **do**
- 10) **if** $k_2 > n_2$
- 11) $A[k] \leftarrow A_1[k_1]$
- 12) $k_1 \leftarrow k_1 + 1$
- 13) **else if** $k_1 > n_1$
- 14) $A[k] \leftarrow A_2[k_2]$
- 15) $k_2 \leftarrow k_2 + 1$
- 16) **else if** $A_1[k_1] \leq A_2[k_2]$
- 17) $A[k] \leftarrow A_1[k_1]$
- 18) $k_1 \leftarrow k_1 + 1$
- 19) **else**
- 20) $A[k] \leftarrow A_2[k_2]$
- 21) $k_2 \leftarrow k_2 + 1$
- 22) $k \leftarrow k + 1$

Полуинвариант: сборът $k_1 + k_2$. Той нараства с една единица при всяко изпълнение на тялото на цикъла, затова цикълът завършва след точно $n_1 + n_2 = n$ изпълнения на тялото и времевата сложност $T(n) = \Theta(n)$ при всякакви входни данни; тя е оптимална по порядък (това следва от тривиалната долна граница по размера на изхода).

Индексът k също е полуинвариант на цикъла, но за да докажем, че цикълът завършва, е нужна оценка отгоре за k , тоест така усложняваме доказателството. По-просто е да използваме посочения сбор като полуинвариант.

Инвариант: При всяко достигане до ред № 9 от алгоритъма са едновременно изпълнени твърденията: k , k_1 и k_2 са цели числа; $1 \leq k_1 \leq n_1 + 1$; $1 \leq k_2 \leq n_2 + 1$; $k = k_1 + k_2 - 1$; $A[1 \dots k - 1] = A_1[1 \dots k_1 - 1] \cup A_2[1 \dots k_2 - 1]$, като масивите се смятат за мултимножества, тоест обединението отчита кратностите на елементите (коя стойност колко пъти се повтаря); ако $k \geq 3$, то $A[1] \leq A[2] \leq \dots \leq A[k - 1]$, тоест попълнената част от изхода е сортирана; ако $k \geq 2$ и $k_1 \leq n_1$, то $A[k - 1] \leq A_1[k_1]$; ако $k \geq 2$ и $k_2 \leq n_2$, то $A[k - 1] \leq A_2[k_2]$.

Както обикновено, инвариантът се доказва с математическа индукция по поредния номер на проверката за край на цикъла.

От съотношенията на индексите следва, че $1 \leq k \leq n + 1$, така получаваме необходимата горна граница за k , която позволява на k да играе ролята на полуинвариант.

Инвариантът важи и при последното преминаване през ред № 9 от алгоритъма. Тогава $k_1 = n_1 + 1$, $k_2 = n_2 + 1$, $k = k_1 + k_2 - 1 = n_1 + n_2 + 1 = n + 1$. Заместваем k , k_1 и k_2 в някои други съотношения от инварианта:

- $A[1] \leq A[2] \leq \dots \leq A[n]$, тоест изходът е сортиран масив;
- $A[1 \dots n] = A_1[1 \dots n_1] \cup A_2[1 \dots n_2]$, тоест изходът се състои именно от входните данни, а не от никакви други стойности.

Алгоритъмът за сливане на сортирани масиви притежава сложност по памет $M(n) = \Theta(1)$ при всякакви входни данни, защото локалните променливи, изразходващи допълнителна памет, а именно k , k_1 и k_2 , са от примитивен тип — цели числа. Паметта за входа и изхода не се брои, защото тя е заделена от извикващия код.

По принцип паметта за входа не се брои, защото се заделя от извикващия код винаги — не само при конкретния алгоритъм.

Въпросът с изхода е по-сложен. В практическото програмиране паметта за изхода се заделя понякога от самия алгоритъм, понякога — от извикващия код. В теорията на алгоритмите обаче тази разлика е маловажна. По-важно е как алгоритъмът използва изхода.

Ако алгоритъмът не само пише в изхода, но и чете от него, тогава изходът играе ролята на локална променлива, затова паметта за изхода се брои като допълнителна памет.

Обратно, ако изходът се използва от алгоритъма само за запис, но не и за четене, то паметта за изхода не се брои, тъй като изходът не се използва като локална променлива. В този случай изпращането на данни към изхода често се оформя като редица от команди за печат към някакво изходно устройство. Това обаче е само удобен начин за оформяне на псевдокода; в истинска компютърна програма алгоритъмът би изпращал съобщения към извикващия код.

В конкретния алгоритъм по-горе е налице вторият случай: изходът се ползва само за запис, но не и за четене, затова паметта за изхода не се брои при изчисляване на сложността по памет на алгоритъма за сливане. (Тази памет ще се брои при анализ на кода, който я заделя.) Без значение е, че вместо команди за печат (print) са използвани оператори за присвояване: индексването на изхода като масив е само удобно средство за посочване на реда на числата.

СОРТИРАНЕ ЧРЕЗ СЛИВАНЕ ($A[1 \dots n]$)

- 1) // Първа версия на алгоритъма — неоптимизирана.
- 2) **if** $n < 2$
- 3) **return**
- 4) $n_1 \leftarrow \left\lfloor \frac{n}{2} \right\rfloor$
- 5) $n_2 \leftarrow \left\lceil \frac{n}{2} \right\rceil$
- 6) $A_1[1 \dots n_1], A_2[1 \dots n_2]$: числови масиви
- 7) **for** $k \leftarrow 1$ **to** n_1 **do**
- 8) $A_1[k] \leftarrow A[k]$
- 9) $r \leftarrow n_1$
- 10) **for** $k \leftarrow 1$ **to** n_2 **do**
- 11) $r \leftarrow r + 1$
- 12) $A_2[k] \leftarrow A[r]$
- 13) СОРТИРАНЕ ЧРЕЗ СЛИВАНЕ ($A_1[1 \dots n_1]$)
- 14) СОРТИРАНЕ ЧРЕЗ СЛИВАНЕ ($A_2[1 \dots n_2]$)
- 15) СЛИВАНЕ ($A_1[1 \dots n_1], A_2[1 \dots n_2], A[1 \dots n]$)

Полуинвариант: дължината n на входния масив. Тя е цяло число, което намалява два пъти при всяко рекурсивно обръщение. Тъй като не съществува безкрайна строго намаляваща редица от цели числа, по-големи или равни на 2, то рано или късно дължината n ще стане по-малка от 2 и рекурсията ще стигне до своето дъно — ред № 3 от кода.

Коректността на двата цикъла се доказва с подходящи инварианти. Формалната обосновка пропускаме, тъй като циклите са достатъчно прости: те копират половинките на входния масив в два отделни масива. Това е нужно за операцията сливане, защото в нейния код се предполага, че някои два масива нямат общи клетки в паметта.

Коректността на сортировката като цяло се доказва със силна индукция по дължината n на входния масив.

База: $n < 2$. Алгоритъмът излиза незабавно през ред № 3, тоест не променя нищо по масива, което е правилно, защото всички масиви с по-малко от два елемента са сортирани поначало.

Индуктивна стъпка: Нека $n \geq 2$ и нека сортировката работи правилно при всички масиви с дължини, по-малки от текущата стойност на n . Ще докажем, че сортирането чрез сливане работи правилно и върху масивите с дължина n .

Доказателството се извършва чрез проследяване на работата на алгоритъма. Двата цикъла копират половинките на масива в отделни масиви с дължини $n_1 = \lfloor n/2 \rfloor < n$ и $n_2 = \lceil n/2 \rceil < n$. Затова към редове № 13 и № 14 от кода имаме право да приложим индуктивното предположение: те правилно сортират масивите. Операцията сливане се извършва над допустими входни данни: първите ѝ два аргумента са сортирани масиви, никои два масива нямат общи клетки в паметта и $n_1 + n_2 = \lfloor n/2 \rfloor + \lceil n/2 \rceil = n$. Както доказахме по-горе, сливането на масиви работи правилно, затова след изпълнението на ред № 15 масивът A ще има вид на ненамаляваща числова редица, образувана от обединението на масивите A_1 и A_2 . Те са двете половинки на входния масив, следователно изходният масив A (споменатата ненамаляваща числова редица) ще бъде също пермутация на входния масив A . Това означава, че описаният алгоритъм правилно сортира масивите с дължина n .

Анализ на времевата сложност: Нека $T(n)$ е времето, нужно за сортиране чрез сливане. То не зависи от входните данни и удовлетворява рекурентното уравнение

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n).$$

Множителят 2 съответства на двете рекурсивни обръщения — на редове № 13 и № 14 от кода. Аргументът на функцията се дели на 2, защото всеки подмасив е половина от входния масив. Събираемост $\Theta(n)$ е времето за копиране на данните и за сливане на масивите.

Полученото рекурентно уравнение се решава чрез мастър-теоремата: $T(n) = \Theta(n \log n)$. Това е времевата сложност на сортирането чрез сливане при всякакви входни данни. Вижда се, че този алгоритъм е много по-бърз от всички елементарни сортировки (поне при дълги масиви). Може да се докаже, че неговата времева сложност притежава най-малкия възможен порядък. Но сортирането чрез сливане не е единствената бърза сортировка: има и други сортировки с времева сложност $\Theta(n \log n)$.

Анализ на сложността по памет: Първо ще пресметнем количеството допълнителна памет, изразходвано от всяко отделно равнище на рекурсията. Масивите $A_1[1 \dots n_1]$ и $A_2[1 \dots n_2]$ притежават общо $n_1 + n_2 = \lfloor n/2 \rfloor + \lceil n/2 \rceil = n$ елемента, затова заедно изразходват памет с размер cn , където c е количеството памет за един елемент. Някакво количество памет b се използва за дължините на масивите и за указателите към първите елементи на масивите. За индексите k и r и за операцията сливане е нужна още памет a . Общото количество памет за едно равнище на рекурсията излиза $cn + b + a$.

Някои от тези количества памет могат да бъдат споделени между равнищата на рекурсията: индексите са нужни само преди, а паметта за сливането — само след рекурсивните извиквания. Затова равнищата могат да споделят както паметта за сливането, така и паметта за индексите (все едно че индексите са декларирани като глобални променливи). Ето защо събираемост a не се натрупва с нарастване на дълбочината на рекурсията.

При другите събираеми това не е тъй: елементите, указателите и дължините на масивите се използват и преди, и след рекурсивните обръщения, затова съответната памет се заделя преди, а се освобождава след двете рекурсивни извиквания. Ето защо тя не може да бъде споделена между равнищата на рекурсията. (Рекурсивни извиквания от едно равнище ползват обща памет, тъй като едното завършва преди началото на другото.) Затова количеството $cn + b$ се сумира по всички равнища на рекурсията. От едно равнище към следващото n намалява двойно, докато достигне стойност 2, затова дълбочината на рекурсията е около $\log_2 n$.

Сложността по памет на целия алгоритъм се получава така:

$$\begin{aligned} M(n) &= a + (cn + b) + \left(\frac{cn}{2} + b\right) + \left(\frac{cn}{4} + b\right) + \left(\frac{cn}{8} + b\right) + \left(\frac{cn}{16} + b\right) + \left(\frac{cn}{32} + b\right) + \left(\frac{cn}{64} + b\right) + \dots = \\ &= a + cn \left(1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \dots\right) + (b + b + b + \dots + b) \sim 2cn + b \log_2 n + a \sim 2cn = \Theta(n). \end{aligned}$$

Тоест $M(n) \sim 2cn = \Theta(n)$.

Ще намалим количеството допълнителна памет, като накараме рекурсивните обръщения да се изпълняват на място (върху оригиналния масив), а копирането на двете му половинки ще става непосредствено преди сливането:

СОРТИРАНЕ ЧРЕЗ СЛИВАНЕ ($A[1 \dots n]$)

- 1) // Втора версия на алгоритъма — оптимизирана.
- 2) **if** $n < 2$
- 3) **return**
- 4) $n_1 \leftarrow \left\lfloor \frac{n}{2} \right\rfloor$
- 5) $n_2 \leftarrow \left\lceil \frac{n}{2} \right\rceil$
- 6) СОРТИРАНЕ ЧРЕЗ СЛИВАНЕ ($A[1 \dots n_1]$)
- 7) СОРТИРАНЕ ЧРЕЗ СЛИВАНЕ ($A[n_1 + 1 \dots n]$)
- 8) $A_1[1 \dots n_1], A_2[1 \dots n_2]$: числови масиви
- 9) **for** $k \leftarrow 1$ **to** n_1 **do**
- 10) $A_1[k] \leftarrow A[k]$
- 11) $r \leftarrow n_1$
- 12) **for** $k \leftarrow 1$ **to** n_2 **do**
- 13) $r \leftarrow r + 1$
- 14) $A_2[k] \leftarrow A[r]$
- 15) СЛИВАНЕ ($A_1[1 \dots n_1], A_2[1 \dots n_2], A[1 \dots n]$)

По този начин паметта за копиране на масива също се споделя между равнищата на рекурсията и събираемостта cn не се натрупва. Единствено паметта за указателите и дължините на масивите остава несподелена и се натрупва. Затова сложността по памет на оптимизираната версия е

$$M(n) = a + cn + (b + b + b + \dots + b) \sim cn + b \log_2 n + a \sim cn = \Theta(n).$$

Въпреки че няма разлика в порядъка, има разлика в константния множител пред порядъка: количеството допълнителна памет е намаляло наполовина — от $2cn$ на cn .

Сортирането чрез сливане е устойчива сортировка, тоест не размества равни елементи. Това се доказва със силна индукция по n . Базата е при $n < 2$: тогава няма какво да се размества. За по-големи n важи индуктивната стъпка: редове № 6 и № 7 не разместват равни елементи съгласно с индуктивното предположение, сливането на ред № 15 от сортировката също копира елементите от всяка половина в реда, в който са били подадени на входа на алгоритъма. Остава опасността да бъдат разместени равни елементи от различни половинки. Тази опасност е предотвратена от ред № 16 на алгоритъма за сливане на масиви: към изхода се подава първо елементът, който е бил първи във входните данни.

И така, сортирането чрез сливане е бързо и устойчиво, но изразходва твърде много памет. Този недостатък е принудил изследователите да потърсят други бързи алгоритми за сортиране, които да използват паметта по-икономично.

Пирамидалното сортиране работи бързо и на място, обаче не е устойчива сортировка. Идеята на този метод е да се използва специален тип данни, наречен приоритетна опашка. Това е абстрактен тип данни, тоест спецификацията му съдържа операции, за които е уточнено само какво правят, но не и как точно го правят. По-конкретно, приоритетната опашка поддържа три операции: добавяне на елемент, извличане на екстремален елемент и промяна на ключ. Последната операция не е нужна за пирамидалното сортиране. Извличането на елемент включва както получаването на елемента, така и изтриването му от приоритетната опашка. “Екстремален” означава “най-малък” или “най-голям”. Една от двете възможности се избира при създаване на приоритетната опашка; изборът не може да се променя по време на работа. Пирамидалното сортиране използва приоритетна опашка с извличане на най-голям елемент и се състои от два етапа:

- Попълване на приоритетната опашка: елементите на входния масив $A[1 \dots n]$ се добавят към приоритетната опашка един по един, като прочитането им може да стане в произволен ред.
- Опръзване на приоритетната опашка: елементите ѝ се извличат един по един; всеки път се извлича най-големият елемент и се поставя на последното свободно място в масива. Тоест първият извлечен елемент се записва в $A[n]$, вторият — в $A[n-1]$, третият — в $A[n-2]$ и т.н. Последният извлечен елемент се записва в $A[1]$ и масивът е сортиран.

Приоритетната опашка може да се реализира по различни начини, но не всички са бързи.

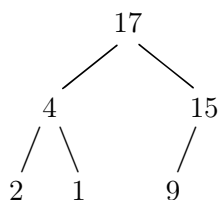
Ако използваме несортиран списък, добавянето на елемент ще бъде бързо — за време $\Theta(1)$; извличането на екстремален елемент ще бъде бавно — за време $\Theta(k)$, k е дължината на списъка. По-точно, изтриването на елемента е бързо, а намирането му е бавно — с последователно търсене. Попълването на списъка ще се извършва за време $\Theta(n)$, но опръзването му ще изисква време $n + (n-1) + (n-2) + \dots + 3 + 2 + 1 = \Theta(n^2)$, т.е. получава се квадратична времева сложност.

Ако пък използваме сортиран списък, то извличането на екстремален елемент ще бъде бърза операция — с време $\Theta(1)$; добавянето на елемент ще бъде бавно — за време $\Theta(k)$. По-точно, вмъкването на елемента е бързо, намирането на мястото му е бавно — с последователно търсене, защото нямаме достъп до елемент по индекс. Опръзването на списъка ще става за време $\Theta(n)$, но попълването му ще изисква време $1 + 2 + 3 + \dots + (n-2) + (n-1) + n = \Theta(n^2)$, т.е. отново се получава квадратична времева сложност в най-лошия случай.

Ако предпочетем сортиран масив пред сортиран списък, направеният анализ остава в сила с тази разлика, че добавянето на елемент е бавно по друга причина: намирането на мястото му е бързо (двоично търсене), обаче самото вмъкване е бавно — новият елемент трябва да избута някои стари елементи. По същество се получава сортиране чрез вмъкване.

Тези наивни реализации на приоритетна опашка страдат от едно и също недоглеждане: няма полза едната операция да е бърза, ако другата е бавна. Времето на сортировката се определя от времето на по-бавната операция. За предпочитане е да използваме такава структура от данни, в която двете операции имат времева сложност между $\Theta(1)$ и $\Theta(k)$. На това изискване отговарят структури от данни, наречени пирамиди — двоичната пирамида, пирамидата на Фибоначи и др. И двете водят до пирамидално сортиране с времева сложност $\Theta(n \log n)$ в най-лошия случай. Ще използваме по-простата от тях — двоичната пирамида.

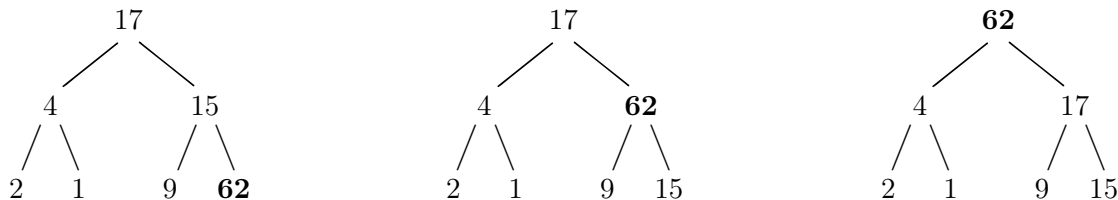
Двоична пирамида се нарича двоично дърво, в което ключът на всеки връх (без корена) не надвишава ключа на неговия родител и всички етажи са запълнени с евентуално изключение на последния етаж, чиито елементи са разположени плътно вляво. Пример за двоична пирамида:



В показания пример последният ред не е запълнен: има място за още един елемент.

От това определение става ясно, че в корена на дървото стои елемент с най-голям ключ. Това важи и за поддърветата: всеки елемент е не по-малък от наследниците си — преки и косвени.

Добавяне на елемент към пирамида: елементът се поставя на най-лявото свободно място на последния етаж, а ако той е запълнен вече, елементът се добавя на нов етаж, най-отляво. След това, докато ключът на добавения елемент е по-голям от ключа на неговия родител, елементът се разменя с родителя си (в краен случай елементът отива в корена на дървото). На следващия чертеж са показани последователно (отляво надясно) промените на пирамидата от предишната страница, ако към нея се добави елемент с ключ 62.



Бързодействие на първия етап от пирамидалното сортиране — погълването на пирамидата: Разглеждаме най-лошия случай: когато всеки добавен елемент се придвижва от листо до корена, т.е. по-голям е от всички предишни. С други думи, двоичната пирамида се погълва най-бавно, ако на входа на алгоритъма се подаде предварително сортиран масив.

Нека имаме двоична пирамида с височина h и със запълнен последен ред. По определение височина на кореново дърво се нарича дължината на най-дълъг път от листата до корена. Да номерираме етажите на дървото от корена към листата: коренът самичък заема етаж № 0, преките наследници на корена заемат етаж № 1, техните преки наследници — етаж № 2 и т.н. Двоичното дърво се състои от $h + 1$ етажа, номерирани с целите числа от 0 до h включително, като етаж № k съдържа 2^k елемента, а цялото дърво — общо n елемента и

$$n = \sum_{k=0}^h 2^k = 2^{h+1} - 1, \text{ откъдето } h = \log_2(n + 1) - 1.$$

При добавяне към двоичната пирамида всеки елемент от етаж № k извършва k размени, докато стигне до корена. Броят на размените (а също и броят на сравненията между елементи) е равен на

$$\begin{aligned} \sum_{k=0}^h k \cdot 2^k &= \sum_{k=0}^h k(2^{k+1} - 2^k) = \sum_{k=0}^h k \cdot 2^{k+1} - \sum_{k=0}^h k \cdot 2^k = \sum_{k=1}^{h+1} (k-1) \cdot 2^k - \sum_{k=0}^h k \cdot 2^k = \\ &= h \cdot 2^{h+1} + \sum_{k=1}^h (k-1) \cdot 2^k - \sum_{k=1}^h k \cdot 2^k = h \cdot 2^{h+1} + \sum_{k=1}^h ((k-1) - k) \cdot 2^k = h \cdot 2^{h+1} - \sum_{k=1}^h 2^k \\ &= h \cdot 2^{h+1} - (2^{h+1} - 2) = (h-1) \cdot 2^{h+1} + 2 = (n+1) \cdot (\log_2(n+1) - 2) = \Theta(n \log n). \end{aligned}$$

Сега нека допуснем възможността последният етаж на дървото да не е запълнен. Тогава $2^h \leq n \leq 2^{h+1} - 1$, откъдето $\log_2(n+1) - 1 \leq h \leq \log_2 n$. Понеже логаритъмът с основа 2 е строго растяща функция, то $(\log_2 n) - 1 < h \leq \log_2 n$, тоест $h = \lfloor \log_2 n \rfloor$. Тази формула ни позволява да намираме височината на двоична пирамида по даден брой елементи.

От последното двойно неравенство следва двустранното ограничение

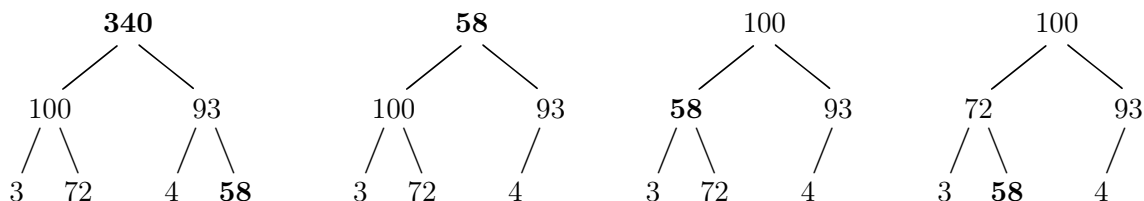
$$(n \log_2 n) - 2n + 2 < (h-1) \cdot 2^{h+1} + 2 \leq (2n \log_2 n) - 2n + 2.$$

Горната и долната граница, а значи и ограниченият израз имат порядък $\Theta(n \log n)$:

$$(h-1) \cdot 2^{h+1} + 2 = \Theta(n \log n).$$

Изразът в лявата страна е броят размествания, а също и броят сравнения между елементи, тоест този израз представлява времевата сложност на погълването на двоичната пирамида при най-лоши входни данни. Току-що се убедихме, че въпросната времева сложност притежава асимптотичен порядък $\Theta(n \log n)$.

Коренът на двоичната пирамида има най-голям ключ. Изтриването на корена става така: най-десният елемент от етажа с най-голям номер се поставя на мястото на корена, след което се разменя с най-големия си пряк наследник, докато стане листо или се окаже не по-малък от двата си преки наследника. Пример:



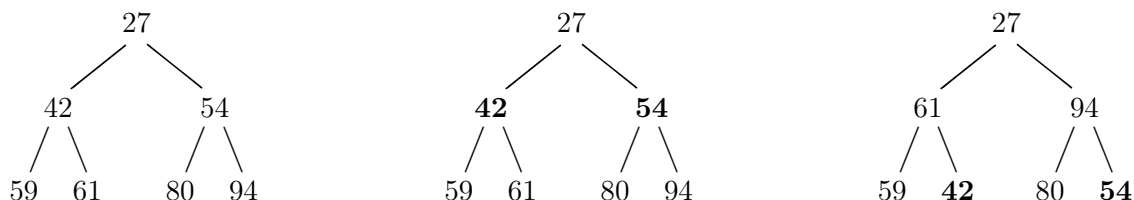
Бързодействие на втория етап от пирамидалното сортиране — опразването на пирамидата: Разглеждаме най-лошия случай: когато всеки от елементите (без първия изтрят) се придвижва от корена до някое листо. В сравнение с първия етап има разлика в посоката на придвижване, но не и в дължините на изминатите пътища. Затова се получават същите аритметични изрази, същият брой размествания, същият брой сравнения между елементи и същата времева сложност, само че n се заменя с $n - 1$ заради първия изтрят елемент, който не пътува между етажите. Тази промяна не влияе на асимптотичния порядък: $(n - 1) \log (n - 1) = \Theta(n \log n)$.

Времевата сложност на пирамидалното сортиране е сборът от времената на двата етапа. Следователно $T(n) = \Theta(n \log n)$ при най-лоши входни данни.

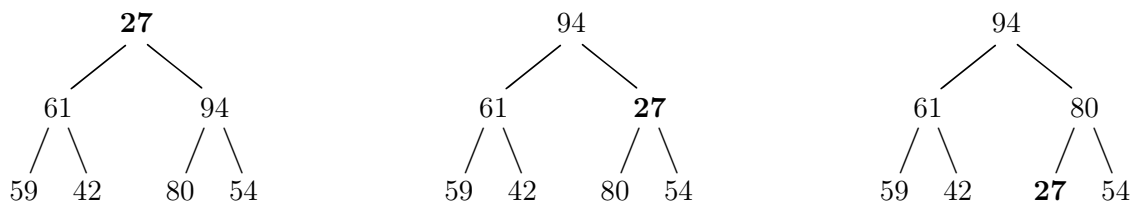
Двете описани процедури за придвижване на елемент от листо към корена и обратно се използват за реализиране на третата операция с приоритетна опашка — промяна на ключ. При намаляване на ключ елементът се премества към листата, а при увеличаване — към корена.

Първият етап от пирамидалното сортиране може да се ускори. Попълването на пирамидата се извършва по-бързо, ако елементите се придвижват към листата, не към корена. Най-напред всички елементи на входния масив се разполагат в пирамидата, после дефектите се отстраняват един по един. Вървим от етаж № $h - 1$ към етаж № 0. Елементите от всеки етаж се разглеждат като корени на своите поддървета и се преместват към листата, ако е нужно.

Да разгледаме най-лошия случай: когато входните данни образуват строго растяща редица. Например нека това е редицата $A = (27; 42; 54; 59; 61; 80; 94)$. Според казаното по-горе първо се придвижват към листата елементите от предпоследния етаж на двоичната пирамида:



След това се придвижват елементите от етажите с по-малки номера, а най-накрая — коренът:



На етаж № k има 2^k елемента. Всеки от тях извършва $h - k$ сравнения и толкова размествания в най-лошия случай. Следователно броят на разместванията, както и броят на сравненията, е

$$\sum_{k=0}^h (h - k) \cdot 2^k = h \cdot \sum_{k=0}^h 2^k - \sum_{k=0}^h k \cdot 2^k = h \cdot (2^{h+1} - 1) - ((h - 1) \cdot 2^{h+1} + 2) = 2^{h+1} - h - 2.$$

Последният израз е от асимптотичен порядък $2^{h+1} = \Theta(n)$, понеже $n < 2^{h+1} \leq 2n$. Ето защо този алгоритъм за попълването на двоичната пирамида има линейна времева сложност.

Времовата сложност на пирамидалното сортиране $T(n) = \Theta(n \log n)$ в най-лошия случай; тя се определя от времето за втория етап от алгоритъма — опразването на двоичната пирамида. Този етап не може да се ускори: всяко сортиране чрез сравняване изисква $\Omega(n \log n)$ сравнения.

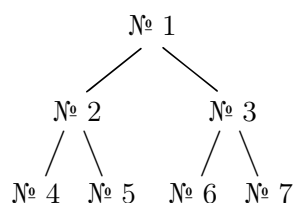
Всички реализации на пирамидалното сортиране са неустойчиви. Това се доказва с примери.

Сложността по памет зависи от физическите типове данни, използвани в реализацията. Различаваме абстрактни, логически и физически типове данни. Пример за абстрактен тип е приоритетната опашка: за нейните операции е казано само какво правят, а не как го правят. Един и същи абстрактен тип може да бъде реализиран чрез разнообразни логически типове, които може да имат различно бързодействие. По-горе бяха разгледани няколко реализации на приоритетна опашка — списъци, масиви и пирамиди. Всички те са логически типове данни. (Разбира се, списъците и масивите са също и физически типове.) Да се внимава с омонимията: “логически тип” тук се противопоставя на “физически тип”, но другаде означава “булев тип”.

Двоичната пирамида е логически, но не и физически тип: изискванията към данните и операциите върху структурата са достатъчно конкретизирани за анализ на бързодействието, но все пак пирамидата съществува само идейно, не е уточнено представянето ѝ в компютъра, затова не можем да определим сложността по памет.

Двоичната пирамида може да се представи физически като съвкупност от записи — по един запис за всеки елемент на пирамидата. Освен полета за данните на съответния елемент всеки запис съдържа и три служебни полета с указатели — към левия и десния наследник и към родителя. Това физическо представяне предполага копиране на данните от входния масив, следователно изразходва памет $\Theta(n)$.

Ще спестим памет, използвайки масив за физическата реализация на двоичната пирамида. Индексираме елементите ѝ с естествените числа от 1 до n вкл. Движим се от корена към листата, а всеки етаж обхождаме отляво надясно:



Въвеждаме следните три обозначения: нека $left(k)$, $right(k)$ и $parent(k)$ са индексите съответно на левия наследник, десния наследник и родителя на елемент № k от пирамидата. Не е трудно да забележим, че

$$left(k) = 2k, \quad right(k) = 2k + 1, \quad parent(k) = \left\lfloor \frac{k}{2} \right\rfloor.$$

В истински програмен код (например на Си) е най-добре тези изрази да бъдат декларирани като макроси, а не като функции; побитовите операции са за предпочитане пред аритметичните.

При всяко добавяне на елемент от входния масив на алгоритъма към двоичната пирамида входният масив се скъсява, а масивът, представящ пирамидата, се удължава с един елемент, така че общата им дължина остава постоянна. При всяко извличане на елемент от пирамидата изходът на алгоритъма се удължава с един елемент, а пирамидата се скъсява пак с толкова, следователно общата им дължина остава постоянна. Става ясно, че един масив може да изиграе и трите роли — вход, изход и двоична пирамида. Нужен е само индекс, който да сочи границата между частите на физическия масив — пирамидата и частта от масива, използвана за вход-изход. Ако попълваме пирамидата по бързия начин (с линейна времева сложност), такъв индекс е нужен само за втория етап — опразването на пирамидата.

Тази реализация сортира на място, поради което нейната сложност по памет е $M(n) = \Theta(1)$ при всякакви входни данни. Тя е бърза: $T(n) = \Theta(n \log n)$ в най-лошия случай.

Недостатъци на пирамидалното сортиране: то е неустойчиво, а при памет с много равнища то често е по-бавно от останалите ефективни сортировки, тъй като обработва един след друг далечни вместо близко разположени елементи на масива. Това му пречи да запише подмасив в по-бързите равнища на паметта, тоест не може да оползотвори възможностите на хардуера. Разбира се, това не променя порядъка на времевата сложност, а само константния множител.

HEAPIFY ($A[1 \dots n]$, k)

- 1) // Отстранява дефект в двоична пирамида, придвижвайки елемент към листата.
- 2) $b \leftarrow k$
- 3) $q \leftarrow 2k$ // $q = \text{left}(k)$
- 4) **if** $q \leq n$
- 5) $m \leftarrow A[k]$
- 6) $v \leftarrow A[q]$
- 7) **if** $v > m$
- 8) $b \leftarrow q$
- 9) $m \leftarrow v$
- 10) $q \leftarrow q + 1$ // $q = \text{right}(k)$
- 11) **if** $q \leq n$
- 12) $v \leftarrow A[q]$
- 13) **if** $v > m$
- 14) $b \leftarrow q$
- 15) $m \leftarrow v$
- 16) **if** $b = k$
- 17) **return**
- 18) swap($A[k]$, $A[b]$) // Разместване на два елемента.
- 19) $k \leftarrow b$
- 20) **go to** 3

ПИРАМИДАЛНО СОРТИРАНЕ ($A[1 \dots n]$)

- 1) **for** $k \leftarrow \left\lfloor \frac{n}{2} \right\rfloor$ **downto** 1 **do**
- 2) HEAPIFY ($A[1 \dots n]$, k)
- 3) $k \leftarrow n$
- 4) **while** $k \geq 2$ **do**
- 5) swap($A[1]$, $A[k]$) // Разместване на два елемента.
- 6) $k \leftarrow k - 1$
- 7) HEAPIFY ($A[1 \dots k]$, 1)

Първият цикъл попълва двоичната пирамида. Този етап беше обяснен неформално (с пример), затова тук няма да се занимаваме повече с попълването. Вторият цикъл опразва пирамидата и притежава следния инвариант: Всеки път, когато пирамидалното сортиране достигне ред № 4, са изпълнени едновременно следните три твърдения:

- подмасивът $A[1 \dots k]$ е двоична пирамида;
- подмасивът $A[k + 1 \dots n]$ е сортиран;
- всеки елемент на $A[1 \dots k]$ е по-малък или равен на всеки елемент на $A[k + 1 \dots n]$.

Алгоритмите *бързо сортиране* и *бързо търсене*

Тези два алгоритъма имат много общо помежду си, но силно се различават от другите: и двата са съставени по схемата “разделяй и владей”, като основната им идея е една и съща; въпреки че се наричат бързи, в наивните си реализации те са бавни (с квадратична сложност); обаче недостатъците им могат да бъдат преодоляни до голяма степен, в резултат на което тези два алгоритъма са едни от най-често използваните на практика.

Бързото търсене намира k -ти най-малък елемент, в това число процентил и медиана; така то се явява алтернатива на алгоритъма PICK.

Бързото търсене и бързото сортиране разделят масива относно подходящ разделител, след което обработват съответно едната или двете части на масива поотделно.

БЪРЗО СОРТИРАНЕ ($A[1 \dots n]$)

- 1) **if** $n < 2$
- 2) **return**
- 3) $m \leftarrow \text{РАЗДЕЛЯНЕ}(A[1 \dots n])$
- 4) БЪРЗО СОРТИРАНЕ ($A[1 \dots m-1]$)
- 5) БЪРЗО СОРТИРАНЕ ($A[m+1 \dots n]$)

БЪРЗО ТЪРСЕНЕ ($A[1 \dots n], k$)

- 1) $m \leftarrow \text{РАЗДЕЛЯНЕ}(A[1 \dots n])$
- 2) **if** $k = m$
- 3) **return** $A[m]$
- 4) **else if** $k < m$
- 5) **return** БЪРЗО ТЪРСЕНЕ ($A[1 \dots m-1], k$)
- 6) **else**
- 7) **return** БЪРЗО ТЪРСЕНЕ ($A[m+1 \dots n], k-m$)

Двата алгоритъма са рекурсивни. Коректността им се доказва със силна математическа индукция по дължината n на входния масив. В оригиналната версия на алгоритъма за разделител служи последният елемент на масива. Самото разделяне може да се извърши по различни начини — по Ломуто, по Хоор, устойчиво. Тъй като устойчивостта няма никакво значение за търсенето и най-често не е от значение при сортирането, обикновено се използва разделянето по Хоор: въпреки че е неустойчиво, то пести памет, защото разделя масива на място. При рекурсията не копираме масива, а само предаваме указател към него и индексите на новия подмасив. Затова количеството допълнителна памет, нужно за едно равнище на рекурсията, е някаква константа, а общо за всички равнища е правопрпорционално на броя им, т.е. на дълбочината на рекурсията. От тази гледна точка най-лош е случаят, когато редицата, образувана от елементите на масива, е строго растяща или строго намаляваща. След разделянето разделителят се оказва винаги в един от краищата на масива и на всяка стъпка дължината му намалява само с една единица, затова дълбочината на рекурсията е $\Theta(n)$. Толкова е и сложността по памет на двата алгоритъма: $M(n) = \Theta(n)$ при най-лоши входни данни. Локалните променливи и формалните параметри (m и границите на подмасива) се пазят в програмния стек, за който предварително се заделя сравнително малка област от паметта. Дълбоката рекурсия може да препълни стека.

Можем да намалим дълбочината на рекурсията, като заменим опашковата рекурсия с цикъл. В алгоритъма *бързо търсене* опашкови са и двете рекурсивни обръщения — редове № 5 и № 7. В алгоритъма *бързо сортиране* опашкова е само рекурсията на ред № 5, но не и тази на ред № 4. Последните два реда на сортировката могат да се разместят. По-изгодно е да заменим с цикъл рекурсивната обработка на по-дългия подмасив; така дължината му ще спада поне два пъти при всяко рекурсивно обръщение.

БЪРЗО СОРТИРАНЕ ($A[\ell \dots h]$)

- 1) **while** $\ell < h$ **do**
- 2) $m \leftarrow \text{РАЗДЕЛЯНЕ}(A[\ell \dots h])$
- 3) **if** $m < \frac{\ell + h}{2}$
- 4) БЪРЗО СОРТИРАНЕ ($A[\ell \dots m-1]$)
- 5) $\ell \leftarrow m+1$
- 6) **else**
- 7) БЪРЗО СОРТИРАНЕ ($A[m+1 \dots h]$)
- 8) $h \leftarrow m-1$

БЪРЗО ТЪРСЕНЕ ($A[\ell \dots h], k$)

- 1) $m \leftarrow \text{РАЗДЕЛЯНЕ}(A[\ell \dots h])$
- 2) **if** $k = m - \ell + 1$
- 3) **return** $A[m]$
- 4) **else if** $k < m - \ell + 1$
- 5) $h \leftarrow m - 1$
- 6) **else**
- 7) $k \leftarrow k - m + \ell - 1$
- 8) $\ell \leftarrow m + 1$
- 9) **go to** 1

Тук ℓ и h са границите на обработвания подмасив. Извикваме алгоритмите с $\ell = 1$ и $h = n$, после ℓ и h се променят автоматично в хода на обработката. Ако пък приемем ℓ и h за водещи, то дължината n на подмасива се пресмята по формулата $n = h - \ell + 1$. За бързото търсене (и в итеративната, и в рекурсивната версия) се предполага двойното неравенство $1 \leq k \leq n$, откъдето следва, че $n \geq 1$, тоест $h \geq \ell$. С други думи, не се допуска масивът да бъде празен. За бързото сортиране няма такова ограничение.

Итеративният вариант на бързото търсене притежава сложност по памет $M(n) = \Theta(1)$ при произволни входни данни, стига да се използва разделянето по Хоор или по Ломуто. Устойчивото разделяне изразходва много памет, а търсенето няма нужда от устойчивост.

Последната версия на бързото сортиране има сложност по памет $M(n) = \Theta(\log n)$, защото дълбочината на рекурсията е около $\log_2 n$. Причината е, че при всяко рекурсивно обръщение дължината на масива намалява поне наполовина, а точно наполовина — в най-лошия случай: когато последният елемент на масива се окаже негова медиана. Тази ниска сложност по памет е постижима, ако използваме неустойчиво разделяне — по Хоор или по Ломуто.

Ако държим сортировката да е устойчива, можем да използваме устойчивото разделяне, но то изразходва допълнителна памет $\Theta(n)$. Тя се заделя и освобождава преди рекурсията, затова може да бъде споделена между различните равнища на рекурсията, но въпреки това сложността по памет на бързото сортиране нараства до $M(n) = \Theta(n)$ при всички входни данни. Все пак новата версия е по-добра от първоначалната, защото сега от програмния стек (който е сравнително малък) се изразходва само количество памет $\Theta(\log n)$ — за рекурсията; а допълнителният масив с размер $\Theta(n)$ за копиране на данните при устойчивото разделяне може да бъде заделен от динамичната памет (която е доста по-голяма от стека).

И тъй, сложността по памет и устойчивостта на бързото сортиране зависят от реализацията.

Преминаваме към анализ на времевата сложност на двата алгоритъма. За нея няма значение дали разглеждаме рекурсивната, или итеративната версия на съответния алгоритъм, затова ще анализираме рекурсивните версии: те са по-къси. Няма значение и методът за разделяне: разделянето по Хоор е най-бързо, но ако гледаме само порядъка, трите метода за разделяне са еднакво бързи (имат времева сложност $\Theta(n)$ при всякакви входни данни).

За бързото търсене най-добрият случай е, когато $k = m$: тогава алгоритъмът приключва без рекурсивни обръщения, най-много време ($\Theta(n)$) се изразходва за разделянето на масива, затова и бързото търсене има времева сложност $T(n) = \Theta(n)$ при най-добри входни данни — когато разделителят (последният елемент на масива) е търсеният k -ти най-малък елемент.

За бързото сортиране най-добрият случай е, когато масивът се дели на две равни части, тоест когато разделителят се оказва медиана на масива на всяко равнище на рекурсията. Тогава времевата сложност удовлетворява рекурентното уравнение

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n).$$

Множителят 2 идва от двете рекурсивни обръщения, а делителят 2 — от дължината на масива, намалена наполовина. Събираемост $\Theta(n)$ е времето за всички останали действия на алгоритъма, предимно за разделянето на масива. Чрез мастер-теоремата получаваме $T(n) = \Theta(n \log n)$. Това е времевата сложност на бързото сортиране при най-добри входни данни.

Получените времеви сложности са твърде оптимистични: отнасят се за най-добрия случай. В практиката имат значение най-лошият случай и средната сложност.

Най-лошите случаи и за двата алгоритъма са, когато елементите на масива образуват строго растяща или строго намаляваща редица (а за бързото търсене има допълнително изискване: $k = 1$ или $k = n$ съответно). Сега от масива отпада само един елемент (разделителят) и времевата сложност удовлетворява рекурентното уравнение

$$T(n) = T(n - 1) + \Theta(n).$$

Както и преди, второто събираемо представя времето за нерекурсивната част от работата, най-вече за разделянето на масива. Първото събираемо е времето за рекурсивната обработка на масива, от който е премахнат разделителят. (При бързото сортиране трябва да се отчете и времето за рекурсивна обработка на празния масив, обаче то е пренебрежимо малко: $\Theta(1)$.) Полученото уравнение е линейно-рекурентно и може да се реши с характеристично уравнение или с развиване. За времевата сложност на бързото сортиране и на бързото търсене получаваме: $T(n) = \Theta(n^2)$ при най-лоши входни данни. В този случай алгоритмите не оправдават името си: те не са никак бързи.

Названието им произлиза от тяхната средна сложност: тези алгоритми се оказват бързи за повечето входни данни. По това те се различават от елементарните сортировки, чието време е квадратично не само в най-лошия случай, но и средно взето.

Да кажем, сортирането чрез вмъкване, когато търси място за k -тия пореден елемент, прави $k - 1$ размествания в най-лошия случай: когато новият елемент е по-малък от всички предишни. Броят на разместванията може да е всяко от числата $0, 1, 2, 3, \dots, k - 1$, средно $\frac{k - 1}{2}$. Ето защо средната сложност е половината на максималната, което няма значение за порядъка. Средната времева сложност на сортирането чрез вмъкване има същия асимптотичен порядък като максималната сложност — и двете са квадратични.

Положението е различно при бързото сортиране и търсене: средните им времеви сложности съвпадат с минималните, не с максималните. Това се доказва от следния анализ. Както видяхме, и при бързото сортиране, и при бързото търсене времевата сложност за нерекурсивната им част е линейна при всякакви входни данни и се определя главно от разделянето на масива. Затова средната времева сложност на нерекурсивната част на двата алгоритъма може да се запише във вида cn . По-нататък анализът на двата алгоритъма протича по различен начин, защото бързото сортиране извършва две рекурсивни обръщения, а бързото търсене — само едно.

Времевата сложност на бързото сортиране, усреднена по стойностите на разделителя, удовлетворява рекурентното уравнение

$$T(n) = cn + \frac{1}{n} \sum_{m=1}^n (T(m-1) + T(n-m)).$$

Двете събираеми в голямата сума изразяват времената на двете рекурсивни извиквания; аргументите им са дължините на масивите. Самата сума и делителят отпред пресмятат средноаритметичното време за рекурсията. Събираемото пред сумата прибавя времето за другите действия (разделянето на масива). Ако от голямата сума образуваме две суми, ще видим, че те са равни, защото се състоят от едни и същи събираеми в различен ред. Следователно

$$T(n) = cn + \frac{2}{n} (T(0) + T(1) + \dots + T(n-1)).$$

При бързото сортиране се изпълняват и двете рекурсивни извиквания (затова голямата сума съдържа сбора от времената им). Бързото търсене изпълнява само едно рекурсивно извикване; кое точно — зависи от стойността на входния параметър k , затова събираемите на тази сума са усреднени не само по стойностите на мястото m на разделителя (общо n на брой), но също и по двете възможности за това, кое рекурсивно извикване се изпълнява; на това се дължи делението на две в голямата сума на бързото търсене. Знаменателят 2 съкращава числителя 2, който се получава от равенството на сумите, затова последните уравнения в двете колонки съдържат числител 2 при бързото сортиране, но числител 1 при бързото търсене.

По-нататък анализът продължава общо за двата алгоритъма. Уравнението е едно и също:

$$T(n) = cn + \frac{a}{n} (T(0) + T(1) + \dots + T(n-1)),$$

само че $a = 1$ за бързото търсене и $a = 2$ за бързото сортиране. Умножаваме уравнението по n :

$$n \cdot T(n) = cn^2 + a(T(0) + T(1) + \dots + T(n-1)).$$

Заместваме n с $n + 1$:

$$(n+1) \cdot T(n+1) = c(n+1)^2 + a(T(0) + T(1) + \dots + T(n-1) + T(n)).$$

От последното уравнение изваждаме предпоследното:

$$(n+1) \cdot T(n+1) = (n+a) \cdot T(n) + c(2n+1).$$

Времевата сложност на бързото търсене, усреднена по стойностите на разделителя, удовлетворява рекурентното уравнение

$$T(n) = cn + \frac{1}{n} \sum_{m=1}^n \frac{T(m-1) + T(n-m)}{2}.$$

Двете събираеми в голямата сума изразяват времената на двете рекурсивни извиквания; аргументите им са дължините на масивите. Самата сума и делителят отпред пресмятат средноаритметичното време за рекурсията. Събираемото пред сумата прибавя времето за другите действия (разделянето на масива). Ако от голямата сума образуваме две суми, ще видим, че те са равни, защото се състоят от едни и същи събираеми в различен ред. Следователно

$$T(n) = cn + \frac{1}{n} (T(0) + T(1) + \dots + T(n-1)).$$

По-нататък анализът на средната времева сложност на двата алгоритъма се провежда поотделно в зависимост от стойността на параметъра a .

За бързото сортиране: $a = 2$, тоест

$$(n+1) \cdot T(n+1) = (n+2) \cdot T(n) + c(2n+1).$$

Делим на $(n+2)(n+1)$:

$$\frac{T(n+1)}{n+2} = \frac{T(n)}{n+1} + \frac{c(2n+1)}{(n+2)(n+1)}.$$

Полагаме

$$F(n) = \frac{T(n)}{n+1}$$

и рекурентното уравнение приема вида:

$$F(n+1) = F(n) + \frac{c(2n+1)}{(n+2)(n+1)}.$$

Заместваме n с $n-1$:

$$F(n) = F(n-1) + \frac{c(2n-1)}{(n+1)n}.$$

Развиваме уравнението:

$$F(n) = F(0) + \sum_{k=1}^n \frac{c(2k-1)}{(k+1)k}.$$

Разбиваме сумата на две суми:

$$F(n) = F(0) + \sum_{k=1}^n \frac{2c}{k+1} - \sum_{k=1}^n \frac{c}{(k+1)k}.$$

Първата сума е разходяща от порядък $\log n$, а втората сума е сходяща, поради което може да се пренебрегне заедно с $F(0)$:

$$F(n) = \Theta(\log n).$$

Обръщаме полагането:

$$\frac{T(n)}{n+1} = \Theta(\log n)$$

и така намираме средната времева сложност на бързото сортиране:

$$T(n) = \Theta(n \log n).$$

За бързото търсене: $a = 1$, тоест

$$(n+1) \cdot T(n+1) = (n+1) \cdot T(n) + c(2n+1).$$

Делим на $(n+1)$:

$$T(n+1) = T(n) + \frac{c(2n+1)}{n+1}.$$

Заместваме n с $n-1$:

$$T(n) = T(n-1) + \frac{c(2n-1)}{n}.$$

Развиваме уравнението:

$$T(n) = T(0) + \sum_{k=1}^n \frac{c(2k-1)}{k}.$$

Разбиваме сумата на две суми:

$$T(n) = T(0) + \sum_{k=1}^n 2c - \sum_{k=1}^n \frac{c}{k}.$$

Оценяваме събираемите по порядък:

$$T(n) = \Theta(1) + \Theta(n) - \Theta(\log n).$$

В сбора запазваме само най-високия порядък и така намираме средната времева сложност на бързото търсене:

$$T(n) = \Theta(n).$$

Общо за двата алгоритъма: средните им времеви сложности са оптимални по порядък, което значи, че повечето входни данни се обработват бързо. Това заедно с простотата на кода прави тези два алгоритъма предпочитани пред техните конкуренти. В програмистката практика бързото търсене се използва по-често от алгоритъма РИСК, а бързото сортиране — по-често от другите сортировки. Така се печели простота на програмния код, обаче възниква опасност програмата да попадне на лоши входни данни, които да забавят нейната работа. Ако програмата трябва да бъде бърза при всякакви входни данни, тогава се предпочитат другите алгоритми: пирамидалното сортиране, сортирането чрез сливане и алгоритъмът РИСК.

Понеже бързото сортиране и бързото търсене са удобни практически, били са положени доста усилия за преодоляване на техния недостатък — бавната обработка на някои входни данни. За целта са били направени редица подобрения:

- Тъй като при къси масиви сортирането чрез вмъкване е по-бързо от другите сортировки, добре е да се повдигне дъното на рекурсията: когато дължината на подмасива спадне до десетина елемента, да се използва сортиране чрез вмъкване вместо рекурсия.
- С помощта на два допълнителни параметъра — дължината на първоначалния масив и текущото равнище на рекурсията (или текущия брой изпълнения на тялото на цикъла, ако се използва итеративната версия) алгоритъмът може да прецени по някакъв критерий дали дължината на текущия подмасив е станала достатъчно малка. В противен случай е основателно предположението, че алгоритъмът е попаднал на лоши входни данни; тогава вместо рекурсия алгоритъмът се обръща към някоя друга бърза сортировка, най-често към пирамидалното сортиране.
- За бързото сортиране и бързото търсене най-лоши входни данни са сортираните масиви. Дори да променим начина за избиране на разделител, все пак някой злонамерен потребител, запознат с подробностите на алгоритъма, може да подаде лоши входни данни. Тази опасност се предотвратява чрез избиране на случаен разделител или случайно разбъркване на масива в началото на алгоритъма. Така не намаляваме вероятността за попадане в лош случай, обаче лошият случай вече не е предварително определен, а зависи от късмета, затова е неизвестен за потребителя. Такава защита е свойствена за рандомизираните алгоритми. Недостатък на тази стратегия е, че се губи свойството определеност: при еднакъв вход алгоритъмът може да работи по различен начин (въпреки че изходът няма да се промени).
- Понеже най-добрият разделител е медианата, а за бързината на алгоритъма не е съществено точното ѝ определяне (приблизителна стойност също върши работа), можем да се опитаме да намерим медианата от извадка. Избираме три, пет или седем елемента (нечетен брой) случайно или с предварително определени индекси. Сред тях намираме средния по големина и го използваме за разделител. Този подход обикновено помага, но не дава сигурна защита срещу лоши входни данни.

Последното подобрение ни навежда на мисълта да намерим медианата с алгоритъма PICK и да я използваме за разделител. Включването на алгоритъма PICK удължава програмния код, но ускорява работата му. Времовата сложност $T(n)$ на тази реализация на бързото сортиране удовлетворява рекурентното уравнение

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n),$$

чието решение е $T(n) = \Theta(n \log n)$. В другите разгледани версии това беше времовата сложност при най-добри, а сега — при всякакви входни данни: поради това, че медианата е разделител, масивът винаги се дели наполовина.

Тази идея по принцип е приложима и към бързото търсене, но там е лишена от смисъл: алгоритъмът PICK и бързото търсене решават една и съща задача, поради което не си струва бързото търсене да извиква PICK. По-добре да се откажем от бързото търсене в полза на PICK. Именно така се постъпва, когато е важно задачата да се реши бързо при всякакви входни данни.

Сортиране чрез сравняване

Всички разгледани дотук сортировки и много други работят чрез сравняване на елементите. Затова те са универсални: работят върху всякакви данни, стига да е дефинирана линейна наредба в множеството от допустими стойности. За всички тези сортировки е доказана долна граница на времовата сложност в най-лошия случай: $T(n) = \Omega(n \log n)$, където n е броят на елементите, подлежащи на сортиране. Долната граница се отнася за броя на сравненията между елементи; броят на разместванията може да бъде по-малък.

Сортиране чрез трансформация. Сортиране с линейна времева сложност

Названието на този клас от сортировки е доста условно: тук попадат разнородни алгоритми, между които единственото общо е, че не сравняват елементи. Затова алгоритмите от този вид не са универсални, тоест всеки от тях се прилага само към данни с подходящи характеристики. Ефективността зависи не само от количеството n на входните данни, но и от други параметри — разни числови характеристики на ключовете. Оценката на този тип алгоритми е нееднозначна: изключително бавни са при някои входни данни (дори по-бавни от елементарните сортировки), а друг път са много бързи. При определени условия е постижима дори линейна времева сложност! Тук не важи долната граница $\Omega(n \log n)$, защото тези алгоритми не сортират чрез сравняване.

Ще разгледаме два от най-важните представители на сортирането чрез трансформация: сортирането чрез броене и метода на бройните системи.

Сортирането чрез броене прави точно това, което казва името му: обхожда входните данни, преброява повторенията на ключовете, след което ги изпраща към изхода в нарастващ ред със съответния брой повторения на всеки ключ — колкото пъти се среща във входа.

Пример: Да се сортира масив от изпитни оценки — цели числа от 2 до 6 включително. Описаната идея може да се приложи по следния начин.

СОРТИРАНЕ ЧРЕЗ БРОЕНЕ ($A[1 \dots n]$: масив от изпитни оценки — цели числа от 2 до 6 вкл.)

- 1) $C[2 \dots 6]$: масив от цели числа — по един брояч за всяка възможна оценка.
- 2) **for** $i \leftarrow 2$ **to** 6 **do**
- 3) $C[i] \leftarrow 0$
- 4) **for** $j \leftarrow 1$ **to** n **do**
- 5) $C[A[j]] \leftarrow C[A[j]] + 1$
- 6) $j \leftarrow 1$
- 7) **for** $i \leftarrow 2$ **to** 6 **do**
- 8) **while** $C[i] > 0$ **do**
- 9) $A[j] \leftarrow i$
- 10) $j \leftarrow j + 1$
- 11) $C[i] \leftarrow C[i] - 1$

Анализ на алгоритъма: Локални променливи са броячите на циклите — двете променливи i и j , а също и петте брояча на оценки от масива C . Като добавим и указателя към началото на C , стават общо осем променливи от примитивен тип. Затова сложността по памет е $M(n) = \Theta(1)$. Ред № 3 се изпълнява пет пъти, ред № 5 се изпълнява n пъти, а всеки от редовете № 9, № 10 и № 11 се изпълнява общо $C[2] + C[3] + C[4] + C[5] + C[6] = n$ пъти, така че времето $T(n) = \Theta(n)$, тоест сортировката в тази задача е с линейна времева сложност. Двете сложности се отнасят за всякакви входни данни.

В общия случай масивът C е индексiran с допустимите стойности и съдържа r елемента, където r е увеличената с единица разлика на най-голямата и най-малката допустима стойност. Например в задачата с изпитните оценки $r = 6 - 2 + 1 = 5$, тоест има пет различни оценки.

При сортирането чрез броене r е допълнителен параметър, от който зависи ефективността. Параметърът r може да зависи от n , но може и да не зависи. Тъй като локалният масив C съдържа r елемента, то сложността по памет е $M(n, r) = \Theta(r)$; оказва се, че тя не зависи от n . Инициализацията на масива C изразходва време $\Theta(r)$, броенето на входните данни — време $\Theta(n)$, а изпращането на данни към изхода отнема време $\Theta(n + r)$: в изхода се записват n елемента и броячът на външния цикъл пробягва r стойности. Така времевата сложност на целия алгоритъм в общия случай е $T(n, r) = \Theta(n + r)$. И тези две сложности важат при всякакви входни данни.

Тази сортировка е бавна и изразходва много памет при голямо r . Обратно, ако r е малко, сортировката е ефективна. Сортирането чрез броене е с линейна времева сложност при $r = O(n)$.






Разгледаният вариант сортира само масиви с примитивен базов тип, например целочислен. При такива масиви е безсмислено да питаеме за устойчивостта на сортирането чрез броене.

Входният масив може да се състои от записи с няколко полета, едно от които е ключ. Сортирането чрез броене работи и в този случай, само че масивът C вместо броячи ще съдържа подходящи структури от данни. Всяка структура ще пази елементите с равни ключове.






Вместо цели числа ключовете могат да са от всеки тип с краен брой допустими стойности, но такъв тип обикновено се заменя с целочислен (допустимите стойности се номерират).

Пример: Всеки елемент на входния масив $A[1 \dots n]$ е запис с две полета — име на растение и цвят от някаква палитра. Цветът е ключ. Нека палитрата се състои от следните пет цвята: червено, оранжево, жълто, синьо и бяло (в този ред).

Входни данни:

A:					
	нарцис	роза	кокиче	лале	синчец

В първата си фаза сортирането чрез броене обхожда масива A и добавя всеки негов елемент към специалната структура под съответния ключ в масива C :

C:	червени	оранжеви	жълти	сини	бели
	↓	↓	↓	↓	↓
		нулев указател			
	роза		нарцис	синчец	кокиче
					

Във втората си фаза сортирането обхожда масива C и елементите от всяка структура добавя подред към масива A .

Изход на алгоритъма:

A:					
	роза	лале	нарцис	синчец	кокиче

Сортирането е устойчиво, когато елементите се изваждат от структурите на масива C в реда, в който са били добавени (“първи влязъл — първи излязъл”), т.е. структурите са опашки.

Този вариант на сортирането чрез броене има същата времева сложност като предишния: $T(n, r) = \Theta(n + r)$ при всякакви входни данни, ако се използва бърза реализация на опашка: добавянето и премахването на елемент да стават за константно време. Също като предишния, новият вариант е бавен при големи r и бърз при малки r , а при $r = O(n)$ времевата му сложност е линейна при всякакви входни данни.

Обаче между двата варианта има разлика в сложността по памет. За новия вариант тя е $M(n, r) = \Theta(n + r)$ при всякакви входни данни. Причината: сега е нужна не само памет $\Theta(r)$ за указателите от масива C , но и памет $\Theta(n)$ за всички опашки общо. Копирането на елементите от входния масив към опашките изразходва много памет: $\Omega(n)$ при всякакви r .

Методът на бройните системи сортира само цели неотрицателни числа, записани в някаква позиционна бройна система. Нека r е основата на използваната бройна система, например $r = 10$. Методът работи така: сортира числата първо по цифрата на единиците, после по десетиците, после по стотиците и т.н. След като ги сортира и по старшата цифра, алгоритъмът завършва. За правилната работа на метода е нужна устойчива сортировка по всеки разред (без единиците). Ако и сортировката по единиците е устойчива, то целият метод на бройните системи е устойчив.

Основата r на бройната система често е малко число, например $r = 2$ в програмирането. В такъв случай броят на допустимите стойности на ключовете (цифрите) е малък, поради което за всеки отделен разред се използва сортиране чрез броене с разход на време и памет $\Theta(n + r)$. Този порядък може да се запише и като $\Theta(n)$, защото r е малко число, тоест $r = O(n)$.

Времето $\Theta(n)$ за един разред умножаваме по броя k на разредите, за да пресметнем времето на целия алгоритъм: $T(n, k) = \Theta(kn)$ е времевата сложност на метода на бройните системи за произволни k -цифрени цели неотрицателни числа. Времевата сложност е линейна, тоест $\Theta(n)$, когато $k = \Theta(1)$. От практическа гледна точка това означава, че методът на бройните системи е бърз, когато сортира къси числа, но е бавен за дълги числа.

Отделните стъпки могат да споделят допълнителната памет, затова паметта за един разред не се умножава по броя на разредите. Така стигаме до извода, че методът на бройните системи има сложност по памет $M(n, k) = \Theta(n)$ за произволни k -цифрени цели неотрицателни числа. Тоест от гледна точка на количеството допълнителна памет методът не е ефективен.

ЗАДАЧИ

Операциите сортиране и търсене се използват често при решаване на алгоритмични задачи. От гледна точка на въздействието на сортирането върху алгоритъма се очертават четири случая:

- 1) В някои задачи сортирането само по себе си води до най-бърз алгоритъм.
- 2) Понякога сортирането води до най-бърз алгоритъм не само, а в съчетание с друга идея.
- 3) Има задачи, в които сортирането се проявява двойко: ускорява наивния алгоритъм, обаче възпрепятства създаването на най-бърз алгоритъм.
- 4) Сортирането може да бъде вредно: да забавя, вместо да ускорява алгоритъма.

Задача 1. Да се провери има ли повторения в даден масив с n елемента.

Решение: Сортирането чрез трансформация не би постигнало линейна времева сложност, защото в условието на задачата няма никакви ограничения за ключовете. Затова се налага да използваме сортиране чрез сравняване: то работи върху всякакви наредени типове данни. Сортираме масива за време $\Theta(n \log n)$ с някой бърз метод, например пирамидално сортиране. После за време $\Theta(n)$ сравняваме съседните елементи. Това е достатъчно, защото след сортирането равните елементи се намират един до друг. Времевата сложност: $\Theta(n \log n) + \Theta(n) = \Theta(n \log n)$ при най-лоши входни данни. Може да се докаже, че този порядък е най-малък.

Не можем да сортираме масива, ако базовият му тип не е нареден. Ако няма повторения, единственият начин да установим това е чрез сравняване на всеки два елемента за време $\Theta(n^2)$. Става ясно колко важно е да умеем да дефинираме пълна наредба върху базовия тип.

Задача 2. Дадени са n реални числа. Да се провери дали сред тях има противоположни.

Решение: Сортираме числата за време $\Theta(n \log n)$, например с алгоритъма *бързо сортиране*. Обаче това само по себе си не е достатъчно: местата на противоположните числа не се намират в някакво очевидно съотношение (например едно до друго). Имаме нужда от още някаква идея. Лесно е да се сетим, че щом масивът вече е сортиран, можем да използваме двоично търсене: за всяко число търсим противоположното. Всяко отделно търсене изразходва време $\Theta(\log n)$, а всичките n търсения — общо $\Theta(n \log n)$. Времева сложност на целия алгоритъм е сборът от времената на двата етапа, тоест $\Theta(n \log n)$ при най-лоши входни данни. Може да се докаже, че тази сложност е най-малка по порядък.

Задача 3. Имаме n реални положителни числа — оценки на n спортисти по някаква скала. Да се разделят играчите на два отбора с равен брой състезатели и с най-голяма разлика в силите. Сила на отбор наричаме сбора от оценките на неговите състезатели.

Решение: Предполагаме, че n е четно число, иначе задачата няма решение.

Първи начин: Наивният алгоритъм е пълното изчерпване. Броят на всички разбивания на множество от n спортисти на две равномошни множества е равен на $\frac{1}{2} C_n^{n/2} = \frac{n!}{2((n/2)!)^2}$.

За всяко разбиване е нужно време $\Theta(n)$ за събирането на оценките на играчите. Следователно времева сложност на пълното изчерпване е

$$n \cdot \frac{n!}{2((n/2)!)^2} \asymp \frac{n^{n+3/2} \cdot e^{-n}}{2^{n+1} \cdot e^{-n}/2^{n+1}} \asymp 2^n \cdot \sqrt{n}.$$

Тази сложност е експоненциална, тоест пълното изчерпване е прекалено бавно.

Втори начин: със сортиране. Тъй като търсим възможно най-неравносечно разделяне, то единият отбор трябва да се състои от най-силните играчи, а другият — от най-слабите. За време $\Theta(n \log n)$ подреждаме състезателите по сила, например с пирамидално сортиране. Първата половина на сортирания масив е слабият отбор, а втората — силният.

Този алгоритъм има полиномиална времева сложност, затова е много по-бърз от наивния, но все пак не е най-бързият възможен алгоритъм: има решение с линейна времева сложност. Но то не може да се получи с помощта на сортиране поради известната долна граница $\Omega(n \log n)$ за времева сложност при най-лоши входни данни на сортирането, основано на сравняване. (Сортиране чрез трансформация е неприложимо: в условието няма ограничения за ключовете.)

Трети начин: без сортиране. Разделяме играчите на силни и слаби спрямо медианата, а нея намираме с алгоритъма PICK. Времева сложност е $\Theta(n)$ при всякакви входни данни. Тя е оптимална заради тривиалната долна граница по размера на входа, тоест този алгоритъм е най-бърз по порядък.

Задача 4. Търсене по ключ в несортиран масив с n реални числа.

Решение: Тази задача сама по себе си е тривиална. Добавена е единствено за да стане ясно, че понякога сортирането вреди на бързината. Задачата се решава с последователно търсене за време $\Theta(n)$ при всякакви входни данни. Тази сложност е очевидно оптимална по порядък. Ако първо подредим масива с някоя бърза сортировка, а после приложим двоично търсене, ще получим алгоритъм с време $\Theta(n \log n) + \Theta(\log n) = \Theta(n \log n)$ при най-лоши входни данни. Сортиране чрез трансформация не можем да приложим, а всяко сортиране чрез сравняване изисква време $\Omega(n \log n)$, така че сортирането само забавя алгоритъма.

Последният извод трябва да се приема внимателно. Ако някой масив се променя рядко, но често търсим в него (например дължините на шосетата в пътната мрежа на някоя държава), тогава си струва да сортираме масива еднократно. Времето на една сортировка е незначително спрямо времето на многобройните заявки за търсене.

Алгоритъмът PICK може да се прилага в съчетание с двоичното търсене. Тук имаме предвид идеята на двоичното търсене, а не конкретния алгоритъм, разгледан по-рано. Пример за това дава следващата задача.

Задача 5. Даден е масив $A[1 \dots n]$ от реални положителни числа — цени на коледни сувенири. Дадено е също така едно положително число K — парична сума, която имаме на разположение. Искаме да зарадваме колкото може повече от близките си хора, като им купим коледни подаръци. Да се състави бърз алгоритъм за закупуване на възможно най-много сувенири.

Решение: Тази задача е подобна на задача 3, защото може да се реши с пълно изчерпване за експоненциално време, може да се реши с някоя ефективна сортировка за време $\Theta(n \log n)$, но най-бърз алгоритъм се получава, ако изобщо не използваме сортиране, а разделим сувенирите на евтини и скъпи. Ясно е, че за да купим най-много сувенири, трябва да изберем най-евтините. За разлика от задача 3, сега не знаем ранга на разделителя.

Именно тук може да ни помогне двоичното търсене. Пробваме разделители с различен ранг, докато уцелим нужния. Най-напред опитваме медианата на масива. Ако тя не свърши работа, опитваме 25-ия или 75-ия перцентил (т.е. медианата на една от двете половини на масива) и т.н.

По-конкретно, бързият алгоритъм изглежда така:

- 1) Намираме медианата на цените $A[1 \dots n]$ на сувенирите с помощта на алгоритъма PICK.
- 2) Разделяме сувенирите на скъпи и евтини относно медианата.
- 3) Пресмятаме сбора S от цените на евтините сувенири.
- 4) Ако $K = S$, купуваме всички евтини сувенири и само тях; край на алгоритъма.
- 5) Ако $K < S$, парите не ни стигат дори за евтините сувенири.

Затова се отказваме от всички скъпи сувенири и изпълняваме алгоритъма рекурсивно върху евтините сувенири, като използваме същата стойност на K .

- 6) Ако $K > S$, купуваме всички евтини сувенири, а за да купим колкото може повече и от скъпите сувенири, изпълняваме алгоритъма рекурсивно върху скъпите сувенири, като използваме $K - S$ вместо K , защото вече сме похарчили част от парите.

Дъното на рекурсията не е описано явно, а само се подразбира:

- ако масивът е празен, не купуваме нищо;
- ако има само един сувенир, купуваме го, когато цената му не надвишава K .

Мястото на тези проверки е преди стъпка № 1 от алгоритъма. Тя и всички следващи стъпки се изпълняват само при $n > 1$.

Анализ на времевата сложност: Изпълнението е най-бавно, когато проверката на стъпка № 4 не се удовлетворява никога. Тогавя алгоритъмът е принуден да стигне до дъното на рекурсията, описано в предишния абзац. Първите три стъпки имат линейна времева сложност. Същото важи за отбелязването на сувенирите като отхвърлени или закупени на стъпки № 5 и № 6. Ето защо

$$T(n) = T\left(\frac{n}{2}\right) + \Theta(n).$$

Аргументът на функцията е разделен на 2, защото при разделяне на масива относно медианата дължината му намалява наполовина. Коефициентът пред функцията в дясната страна е 1, защото се изпълнява едно от рекурсивните извиквания — или стъпка № 5, или стъпка № 6. От мастер-теоремата намираме $T(n) = \Theta(n)$. Това е времевата сложност на нашия алгоритъм при най-лоши входни данни. От друга страна, дори при най-добрите възможни входни данни (когато проверката на стъпка № 4 се удовлетворява още на най-горното равнище на рекурсията) времето е $\Theta(n)$ заради първите три стъпки. Окончателно, времевата сложност $T(n) = \Theta(n)$ при всякакви входни данни.

Тази задача е типова: като нея се решават много други задачи. Няколко примера:

- Да се запишат възможно най-много филми с дадени дължини върху флашка с капацитет K .
- Да се натоварят възможно най-много стоки с дадени тегла на камион, побиращ K тона.