Европейски съюз

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

Европейски социален фонд

Пета лекция по ДАА на втори поток КН
Сортиращи алгоритми MERGESORT и QUICKSORT

31 март 2014

Абстракт

Въвеждаме два бързи сортиращи алгоритъма, MERGESORT и QUICKSORT, и изследваме тяхната коректност, сложност по време и сложност по памет. Изследваме средната сложност по време на QUICKSORT.

Съдържание

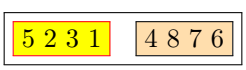
1 MERGESORT

Сортиращият алгоритъм MERGESORT е изобретен от John von Neumann през 1945 г. [Knu98]. Той е типичен пример за алгоритъм, изграден по схемата Разделяй-и-Владей. Във фазата **Разделяй** той дели входа на две равни части, всяка с размер $\frac{n}{2}$[†], без да променя относителния порядък на елементите. Във фазата **Владей** прави по едно рекурсивно викане върху всяка от двете части. Във фазата **Комбинирай** всяка от двете части вече е сортирана, така че алгоритъмът слива (откъдето идва и името, *сливам* в този смисъл е *to merge* на английски) двете сортирани части в една окончателна сортирана последователност. Естествено, всичко това става, когато входът е достатъчно голям. Ако входът е с размер единица или нула, алгоритъмът не прави нищо (защото празният масив и едноелементният масив са сортирани) – това е спирачката на рекурсията. Акцентът е върху третата фаза, там се извършва истинската работа по сортирането.

Първо ще дадем пример за сортиране с MERGESORT. Примерът нарочно използва вход с големина, която е точна степен на двойката, но кодът, който даваме нататък, работи коректно за всяка големина. И така, нека входът е:

С жълт фон и червен цвят на ограждащата кутия означаваме тази част от масива, която е вход на текущо активното рекурсивно извикване. В самото начало тази част съвпада с целия масив.

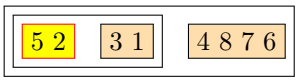
Тъй като големината на входа е повече от единица, делим входа на две равни части и правим по едно рекурсивно извикване върху всяка от тях. Тези две рекурсивни извиквания не се случват едновременно, защото нашият изчислителен модел не позволява паралелизъм. Да речем, че първо извикваме рекурсивно алгоритъма върху масива вляво.



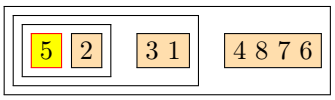
С жълт фон и червена ограждаща кутия е само тази част от началния масив, която е вход на текущото рекурсивно викане. Понеже големината на този вход е четири, което е по-голямо от едно, пак делим входа на две и

[†]В действителност разделянето е на една част с размер $\lfloor \frac{n}{2} \rfloor$ и друга част с размер $\lceil \frac{n}{2} \rceil$.

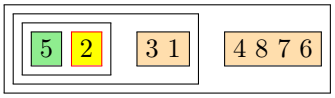
правим две рекурсивни викания, първото върху частта вляво:



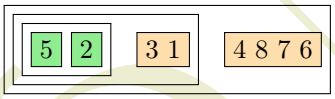
Тъй като големината на входа на текущото рекурсивно викане е две, което е по-голямо от едно, пак делим входа на две правим две рекурсивни викания, първото върху частта вляво:



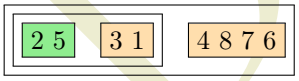
Сега вече големината на входа на текущото рекурсивно викане е едно, и алгоритъмът не прави нищо в това извикване и връща директно управлението на предното извикване, което на свой ред вика алгоритъма върху другия подмасив с големина едно:



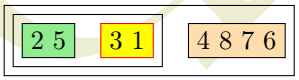
Светлозеленият фон означава, че върху този подмасив вече е достигнато дъното на рекурсията. Отново големината на текущия вход (жълт фон) е едно и алгоритъмът директно връща управлението на горното ниво, бидейки привършил и с двете викания

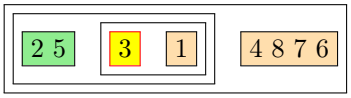


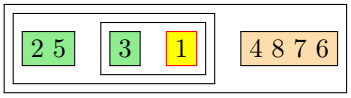
Сега алгоритъмът слива двата сортирани подмасива с големини единици в един сортиран подмасив с големина две:

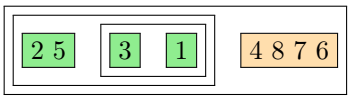


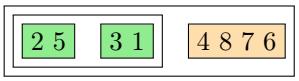
Изчислението продължава така:

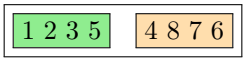




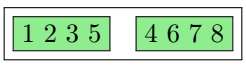








Прескачаме няколко стъпки, които съответстват на второто рекурсивно викане върху 4876, и разглеждаме масива след приключване на това второ рекурсивно викане:



Сливането на двата сортирани подмасива води до

1 2 3 4 5 6 7 8

Сега ще дадем псевдокод са функцията, която осъществява сливането. Нейната коректност е условна: тя работи коректно **при условие**, че двете половини на входния ѝ масив са сортирани. Символът ∞ означава число, по-голямо от всяко друго число, което може да се срещне. Този символ се нарича *пазач* (на английски, *sentinel*, по терминологията на [CLRS09]). Използването му не е задължително, но с него кодът става по-лесен за четене и верификация.

```
MERGE(A[1,2,...n]: array of integers; l, mid, h: positive integers, such that 1 ⩽ l < mid < h ⩽ n)
 1  (* подмасивите A[l,...,mid] и A[mid+1,...,h] са сортирани *)
 2  n1 ← mid − l + 1
 3  n2 ← h − mid
 4  създай L[1,...,n1 + 1] и R[1,...,n2 + 1]
 5  L ← A[l,...,mid]
 6  R ← A[mid + 1,...,h]
 7  L[n1 + 1] ← ∞
 8  R[n2 + 1] ← ∞
 9  i ← 1
10  j ← 1
11  for k ← l to h
12      if L[i] ⩽ R[j]
13          A[k] ← L[i]
14          i ← i + 1
15      else
16          A[k] ← R[j]
17          j ← j + 1
```

Лема 1. *При допускането, че подмасивите* $A[l,\ldots,mid]$ *и* $A[mid+1,\ldots,h]$ *са сортирани, масивът* $A[l,\ldots,h]$ *е сортиран след като* MERGE *завърши.*

Доказателство:
Следното е инварианта за **for** цикъла (редове 11–17):

> **част i** Всеки път, когато изпълнението на MERGE е на ред 11, $A[l,\ldots,k-1]$ съдържа $k-l$ най-малки елемента на $L[\,]$ и $R[\,]$, в сортиран вид
>
> **част ii** Освен това, $L[i]$ и $R[j]$ са най-малките елементи съответно в $L[\,]$ и $R[\,]$, които още не са копирани в $A[\,]$.

База. Когато изпълнението е на ред 11 за пръв път е вярно, че $k = l$. Тогава подмасивът $A[l,\ldots,k-1]$ всъщност е $A[l,\ldots,l-1]$, тоест е празен. Празният масив е сортиран и съдържа $k-l=0$ най-малки елемента на $L[\,]$ и $R[\,]$, в сортиран вид. Освен това, $L[1]$ и $R[1]$ са най-малките елементи съответно в $L[\,]$ и $R[\,]$, които все още не са копирани в $A[\,]$.

Поддръжка. Нека твърдението е в сила при някое достигане на ред 11, което не е последното. Има два алтернативни начина, по които изпълнението може да премине през тялото на цикъла. Ще ги разгледаме и двата. Преди да направим това обаче, ще докажем едно важно помощно твърдение; забележете, че и $L[\,]$, и $R[\,]$ съдържат стойност ∞, така че трябва да сме сигурни, че двете стойности ∞ не биват сравнявани[†].

> *В сравнението на ред 12, не може едновременно* $L[i]$ *и* $R[j]$ *да са* ∞.
>
> **Доказателство:** Да допуснем противното. Съгласно индуктивното предположение, $k-l$ на брой елементи са копирани в A от $L[\,]$ и $R[\,]$, и освен това, тялото на цикъла ще се изпълни поне още веднъж, така броят на копираните елементи е $\leqslant h-l$. Съгласно допускането, което направихме, вече са копирани n_1+n_2 елемента, които не са ∞, така че броят на копираните елементи е $\geqslant n_1+n_2 = mid-l+1+h-mid = h-l+1 > h-l$. ↯

[†] Сравнение на две ∞ стойности не е дефинирано

Да разгледаме сравнението на ред 12. Тъй като няма как и L[i], и R[j] да са ∞, резултатът от сравнието е дефиниран. Първо да допуснем, че $L[i] \leqslant R[j]$. Очевидно, $L[i] < \infty$. Съгласно **част ii** от индуктивното предположение и допускането, че $L[i] \leqslant R[j]$, заключаваме, че L[i] е най-малкият елемент в L[] и R[], който все още не е копиран. Съгласно **част i** на индуктивното предположение, L[i] не е по-малък от никой елемент на $A[l, \ldots, k\text{-}1]$. Изпълнението отива на ред 13. Забелязваме, че A[k] не е по-малък от нито един елемент на $A[l, \ldots, k\text{-}1]$ и заключаваме, че $A[l, \ldots, k]$ е сортиран и съдържа $k - l + 1 = (k + 1) - l$ на брой най-малки елемента на L[] и R[]. Но k бива инкрементирано при следващото достигане на ред 11. Спрямо новата стойност на k е вярно, че $A[l, \ldots, k\text{-}1]$ съдържа $k - l$ най-малки елемента на L[] and R[], в сортиран вид. И така, **част i** на инвариантата остава в сила.

Сега ще докажем и **част ii** на инвариантата. По допускане, L[] и R[] са сортирани. Преди присвояването на ред 13, L[i] беше най-малък елемент от L[], който все още не е копиран в A[]. След това присвояване, L[i + 1] е най-малък елемент от L[], който все още не е копиран в A[]. Но променливата i бива инкрементирана при следващото достигане на ред 14. По отношение на новата стойност на i, L[i] е най-малък елемент от L[] който все още не е копиран в A[].

Сега да допуснем, че изпълнението все още е на ред 12 и $L[i] \nleqslant R[j]$, тоест L[i] > R[j]. Доказателството е напълно аналогично на току-що направеното.

Терминация. Променливата k съдържа $h + 1$ при последното достигане на ред 11. Заместваме тази стойност в инвариантата и получаваме "подмасивът $A[l, \ldots, h - 1]$ съдържа $k - l$ най-малки елемента на L[] and R[], в сортиран вид". □

MERGESORT(A[1, 2, ..., n]: array of integers; l, h: indices in A[])
1 **if** $l < h$
2 $\quad mid \leftarrow \lfloor \frac{l+h}{2} \rfloor$
3 $\quad$ MERGE SORT(A, l, mid)
4 $\quad$ MERGE SORT($A, mid + 1, h$)
5 $\quad$ MERGE(A, l, mid, h)

Лема 2. *Алгоритъм* MERGESORT *е коректен сортиращ алгоритъм, ако началното извикване е* MERGE SORT(A, 1, n).

Доказателство:
По индукция по разликата $h - l$[†]. Смятаме за очевидно, че $h - l$ може да стане най-малко нула, но не и по-малко, и че базата на нашето доказателство е $h - l = 0$.

База. Нека $h - l = 0$, тоест h = l. Масивът A[l, ..., h] е едноелементен. От една страна, едноелементният масив $A[l]$ е тривиално сортиран. От друга страна, MERGESORT не прави нищо, когато $h = l$. Така че масивът е сортиран в края на алгоритъма.

Поддръжка. Допускаме, че MERGESORT сортира коректно подмасивите $A[l, \ldots, mid]$ и $A[mid + 1, \ldots, h]$ (редове 3 и 4) при всички рекурсивни викания, такива че $h > l$. От Лема 1 следва, че в края на работата на текущото рекурсивно извикване, целият $A[l, \ldots, h]$ е сортиран.

Терминация. Когато доказваме коректност на рекурсивни алгоритми по индукция, стъпката **Терминация** се отнася до приключването на изпълнението на началното викане. В този случай, началното извикване е MERGESORT(A[1, ..., n]). С MERGESORT, тази стъпка е тривиална: просто забелязваме, че алгоритъмът приключи работата си, $A[1, \ldots, n]$ е сортиран. □

Сложността по време на MERGESORT се определя с рекурентното отношение

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

Решението, съгласно втория случай на Мастър теоремата, е $T(n) \asymp n \lg n$. Сложността по памет е $\Theta(n)$, ако алгоритъмът се имплементира грамотно (имплементация, която буквално следва псевдокода горе би имала линейна сложност по време, но на практика би била твърде бавна заради честото алокиране и деалокиране на памет). MERGESORT е стабилен сортиращ алгоритъм, защото на ред 12 във функцията MERGE има нестрого неравенство. Поради това, ако биват сравнявани два елемента (от L[] и от R[]) с равни ключове, елементът от L[] ще окаже вляво от този от R[] в окончателната подредба.

[†]Забележете, че това не е доказателство с инвариант на цикъла, понеже MERGESORT не е итеративен, а е рекурсивен, алгоритъм.

2 Quicksort

2.1 Функцията Partition

Quicksort е изобретен от Hoare [Hoa62] в началото на 1960-те. На практика това се оказва най-бързият сортиращ алгоритъм, оттам и името, и това е стандартният сортиращ алгоритъм в библиотечните имплементации, като реалните имплементации са по-сложни от псевдокода, който даваме тук.

Quicksort е типичен за схемата Разделяй-и-Владей алгоритъм, макар да е много различен от Mergesort. Най-общо казано, идеята е да бъде избрана някаква стойност, която се нарича *pivot*, и спрямо нея масивът да бъде пренареден така, че в лявата част (може и да е празна, това зависи от pivot) да са само елементи, по-малки или равни на pivot, а вдясно от тях, само елементи, по-големи от pivot. Какъв е относителният порядък в лявата и дясната част няма значение – не се иска при това пренареждане да се постигне сортиран масив, това би било прекалено силно изискване. Сортиране ще се получи чак след края на рекурсивните викания. На този етап искаме просто вляво да са "малките" елементи, а вдясно от тях, "големите". Това пренареждане е фазата **Разделяй** на алгоритъма. След това алгоритъмът бива викан рекурсивно върху всяка от тези части, ако тя е достатъчно голяма. Това е фазата **Владей**. Фазата **Комбинирай** е празна: след приключването на извикванията, масивът е сортиран.

Изборът на pivot, както ще видим, е от огромно значение за сложността по време. Не е задължително pivot да е елемент от масива, макар че простите имплементации избират за pivot произволен елемент, примерно най-левия или най-десния. В идеалния случай стойността на pivot е такава, че спрямо него масивът се разделя на равни части. С други думи, в идеалния случай pivot е медиана. Добре известно е, че медиана може да се намери в линейно време [CLRS09, стр. 189], само че на практика забавянето, до което води търсенето на медиана, прави алгоритъмът напълно непрактичен[†]. На практика изборът на pivot става за константно време и няма гаранция, че разделянето става на две равни части – разликата между големината на лявата и на дясната част може да е произволна.

Виждаме една основна разлика между Quicksort и Mergesort. При Mergesort фазата **Разделяй** е тривиална и същината на алгоритъма е във фазата **Комбинирай**. При Quicksort фазата **Разделяй** е същината на алгоритъма, докато фазата **Комбинирай** е празна.

Фазата **Разделяй** на Quicksort очевидно може да се имплементира в линейно време, като това е и долна граница (защото всеки елемент от масива трябва да бъде "прегледан" при това пренареждане). Ключовото наблюдение е, че тя може да бъде реализирана само с константна допълнителна памет, тоест in-place. Ето псевдокодът на разделянето според Hoare.

```
Partition–Hoare(A[1,2,...,n]: array of integers; l, h: indices in A[])
 1  pivot ← A[l]
 2  i ← l − 1
 3  j = h + 1
 4  while True do
 5      repeat
 6          j ← j − 1
 7      until A[j] ⩽ pivot
 8      repeat
 9          i ← i + 1
10      until A[i] ⩾ pivot
11      if i < j
12          swap(A[i], A[j])
13      else
14          return j
```

Псевдокодът е почти буквално по [CLRS09][‡]. Да си припомним, че **repeat ... until** конструкцията изпълнява

[†] Heapsort и Mergesort са $\Theta(n \lg n)$ сортиращи алгоритми. Както ще видим в следваща лекция, това е асимптотично оптимално, по-бърз алгоритъм в асимптотичния смисъл за Сортиране не е възможен. Предимството на Quicksort е, че е около два пъти по-бърз на практика от тези алгоритми, въпреки че, както ще видим, в най-лошия случай той е $\Theta(n^2)$. Ако се опитаме да "подобрим" Quicksort, настоявайки pivot да е непременно медианата, в най-лошия случай той ще стане $\Theta(n \lg n)$, но на практика ще е много по-бавен от Heapsort и Mergesort и никой няма да го използва реално.

[‡] Този псевдокод размества ненужно еднакви елементи, но, от друга страна, той е максимално опростен.

тялото на цикъла поне веднъж, а изпълнението на цикъла се прекратява тогава и само тогава, когато условието след **until** не е вярно.

Грубо казано, `i` и `j` са индекси, които "вървят" един срещу друг, съответно отляво надясно и отдясно наляво, докато не "открият" елементи от масива, които са "неправилно разположени" спрямо избрания pivot, след което тези елементи се разменят.

По-прецизно казано, по време на работата на алгоритъма, масивът е разбит на три зони (някои, но не всички, от които може да са празни):

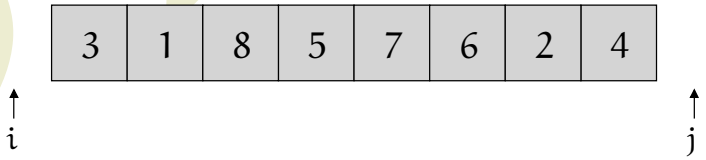
- **Зелена зона** $A[l, \ldots, i]$. Там са елементи, по-малки или равни на `pivot`.
- **Сива зона** $A[i+1, \ldots, j-1]$. Това е неизследваната част от масива.
- **Жълта зона** $A[j, \ldots, h]$. Там са елементи, по-големи или равни на `pivot`.

Изборът на имена за зоните е произволен. С оцветяване лесно се илюстрира работата на алгоритъма върху конкретен пример. Когато и двата **repeat ... until** цикъла приключат, и $A[i] \geqslant$ `pivot`, и $A[j] \leqslant$ `pivot`. В случай, че $i < j$, изпълнението отива на ред 12; след разместването, следващите увеличаване на `i` (ред 9) и намаляване на `j` (ред 6) увеличават съответно зелената и жълтата зона с по една клетка. Ако $i \not< j$, няма какво повече да се прави.

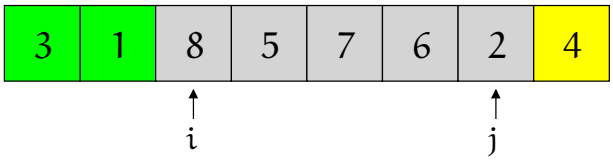
Това не е истинско формално доказателство за коректност с инварианта на цикъла. Нека читателите опитат да направят формално доказателство за коректността на Partition–Hoare. Сега ще демонстрираме работата на тази функция с пример. Нека

$$A = \boxed{3\ 1\ 8\ 5\ 7\ 6\ 2\ 4}$$

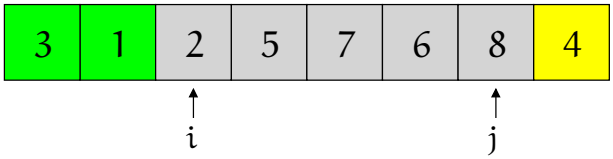
като l е 1 и h е 8. В началото `pivot` е $A[l] = 3$. После `i` и `j` "застават" извън границите на масива, така че зелената и жълтата зона са празни:



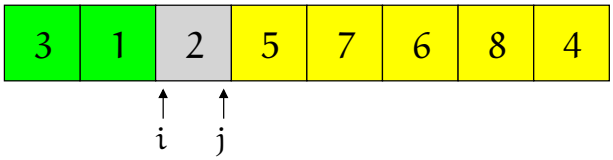
После `j` "слиза" надолу и става 7 ($A[7] = 2$, така че $A[2] \leqslant$ `pivot`), а `i` се "качва" нагоре и става 3 ($A[3] = 1$, така че $A[3] \geqslant$ `pivot`):



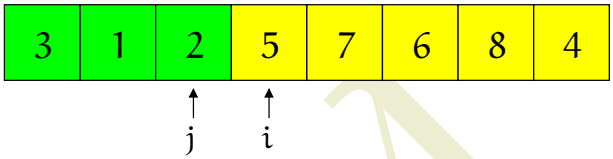
Изпълнението достига ред 11. Понеже $i < j$, изпълняваме размяната:



После `j` "слиза" надолу и става 3, колкото е `i`, защото $A[3] = 2$ е най-десният елемент, по-малък или равен на `pivot`:



После j се "качва" нагоре, ставайки 4:



Изпълнението отново е на ред 11. Сега вече $i \not< j$, така че изпълнението преминава към ред 14, като алгоритъмът връща стойност 3. И наистина, $A[1,\ldots,3]$ са точно тези елементи, които са не по-големи от `pivot`.

Сега само ще покажем алтернативен псевдокод на функция, която пренарежда масива по желания начин. Идеята за този код е на Nico Lomuto, около 1984 г., и е публикувана в статията *Programming Pearls: Little Languages* в списанието Journal of the ACM [Ben86].

```
PARTITION–LOMUTO(A[1,2,...,n]: array of integers; l, h: indices in A[])
 1  pivot ← A[h]
 2  pp ← l
 3  for i ← l to h − 1
 4      if A[i] < pivot
 5          swap(A[i], A[pp])
 6          pp ← pp + 1
 7  swap(A[pp], A[h])
 8  return pp
```

Формалното доказателство за коректност и този път остава за читателя. Тук само отбелязваме, че и при PARTITION–LOMUTO можем да дефинираме зеленн, жълт и сив район, само че взаимното им разположение е различно, сега те са зелен, жълт и сив, отляво надясно. С други думи, PARTITION–LOMUTO "търкаля" жълтия район надясно. Името на променливата `pp` идва от "pivot position".

Виждаме още една разлика между QUICKSORT и MERGESORT. Фазата **Разделяй** на QUICKSORT връща стойност, а именно индексът, който определя разделянето на масива на две части за рекурсивните викания. Това се налага съвсем естествено: за разлика от MERGESORT, сега не можем да изчислим индекса, определящ разделянето, без да сме прегледали масива.

Псевдокодът на самия QUICKSORT е съвсем кратък (функцията PARTITION е или PARTITION–LOMUTO, или PARTITION–HOARE):

```
QUICKSORT([A1,2,...,n]: array of integers; l, h: indices in A[])
1  if l < h
2      mid ← PARTITION(A, l, h)
3      QUICK SORT(A, l, mid − 1)
4      QUICK SORT(A, mid + 1, h)
```

Доказателството за коректност е напълно аналогично на доказателството за коректност на MERGESORT. QUICKSORT не е стабилен сортиращ алгоритъм. Лесно е да се намери пример, в който PARTITION разменя относителната подредба на два еднакви елемента. Примерът за PARTITION–LOMUTO може да се различава от примера за PARTITION–HOARE, но и за двата варианта има примери.

2.2 Сложност по време на QUICKSORT

Сложността на QUICKSORT е изключително чувствителна към изборите на pivot. Казваме "изборите" в множествено число, защото става дума за изборите във всички рекурсивни викания. Ако във всяко отделно викане се оказва, че pivot дели съответния входен (под)масив на две равни части, сложността са описва от рекурентното отношение

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

което, както вече видяхме при MERGESORT, има решение $T(n) \asymp n \lg n$. Ако обаче всеки избор на pivot е такъв, че този елемент дели масива на подмасив с един елемент (самият той) и друг подмасив с $n-1$ елемента, сложността се описва от рекурентното отношение

$$T(n) = T(n-1) + 1$$

С метода с характеристичното уравнение е лесно да се покаже, че това рекурентно отношение има решение $T(n) \asymp n^2$.

Следователно, в най-лошия случай QUICKSORT е квадратичен сортиращ алгоритъм, с линейна сложност по памет. Ако се задоволяваме само с анализ на сложността в най-лошия случай, това е всичко, което имаме да кажем за сложността на QUICKSORT. На практика обаче QUICKSORT е по-бърз от всеки друг известен сортиращ алгоритъм, говорейки осреднено. Причината е, че лошите избори на pivot са твърде малко вероятни, ако pivot е случаен елемент от масива и големините на елементите от масива са униформно разпределени. Затова, за пръв и последен път в този курс, ще направим анализ на средната сложност на QUICKSORT – иначе би трябвало да класифицираме най-бързия на практика сортиращ алгоритъм като бавен.

И така, анализът на средната сложност се прави с рекурентно отношение, което обаче отразява именно средния случай. Както се убедихме, това на какви части бива разделен входът се определя от pivot елемента. Ако pivot е случаен елемент от масива, то всяко разделяне на входа на две части е равновероятно. Отчитаме само броят на сравненията – неговата асимптотика е асимптотиката на самия алгоритъм. Следното рекурентно отношение моделира казаното дотук:

$$T(n) = \frac{1}{n} \sum_{k=1}^{n} (T(k-1) + T(n-k)) + (n-1) \tag{1}$$

Променливата k е мястото (по големина) на pivot във входа; pivot може да е най-малък елемент ($k = 1$) или втори по големина ($k = 2$) или ... или най-голям ($k = n$). Това рекурентно отношение се чете така: работата на QUICKSORT е сумата от

- сумата по всички възможности за pivot ($k = 1 \ldots n$) от сумата от работата върху двата подмасива (единият е с големина $k-1$, другият с големина $n-k$, оттам и $T(k-1)+T(n-k)$), но умножена с $\frac{1}{n}$. Този множител отразява факта, че всички n възможности за относителната големина на pivot са равновероятни.
- и $n-1$, което е броят на сравненията за определяне на мястото на pivot. Очевидно pivot трябва да бъде сравнен с всеки друг елемент и това е достатъчно.

Сега ще решим (1).

$$\begin{aligned} T(n) &= \frac{1}{n} \sum_{k=1}^{n} (T(k-1) + T(n-k)) + (n-1) \\ &= \frac{1}{n} \left(\sum_{k=1}^{n} T(k-1) + \sum_{k=1}^{n} T(n-k) \right) + (n-1) \\ &= \frac{1}{n} \left(\underbrace{T(0) + T(1) + \ldots + T(n-1)}_{\sum_{k=1}^{n} T(k-1)} + \underbrace{T(n-1) + T(n-2) + \ldots + T(0)}_{\sum_{k=1}^{n} T(n-k)} \right) + (n-1) \end{aligned} \tag{2}$$

Очевидно двете подчертани суми в (2) са равни, така че

$$T(n) = \frac{2}{n}\big(T(0) + T(1) + \ldots + T(n-1)\big) + (n-1) \tag{3}$$

Умножаваме двете страни на (3) по n и получаваме

$$nT(n) = 2\big(T(0) + T(1) + \ldots + T(n-1)\big) + n(n-1) \tag{4}$$

Но тогава, ако n е достатъчно голямо,

$$(n-1)T(n-1) = 2\big(T(0) + T(1) + \ldots + T(n-2)\big) + (n-1)(n-2) \tag{5}$$

Изваждаме (5) от (4) и получаваме

$$\begin{aligned} nT(n) - (n-1)T(n-1) &= 2T(n-1) + n(n-1) - (n-1)(n-2) \\ &= 2T(n-1) + 2(n-1) \quad \leftrightarrow \\ nT(n) &= (n+1)T(n-1) + 2(n-1) \end{aligned} \tag{6}$$

Делим двете страни на (6) на $n(n+1)$ и получаваме

$$\frac{T(n)}{n+1} = \frac{T(n-1)}{n} + \frac{2(n-1)}{n(n+1)} \tag{7}$$

Удобно е да направим следното полагане, за да се освободим от нотация със знаменатели: $\frac{T(n)}{n+1}$ ще бъде наричано $S(n)$. Тогава очевидно $\frac{T(n-1)}{n} = S(n-1)$ и заместваики в (7), получаваме следното рекурентно отношение:

$$S(n) = S(n-1) + \frac{2(n-1)}{n(n+1)} \tag{8}$$

То не може да бъде решено чрез метода с характеристичното уравнение, защото нехомогенната част не е от вида, полином на n по константа на n-та степен, така че ще използваме универсалното средство – развиване.

$$\begin{aligned} S(n) &= S(n-1) + \frac{2(n-1)}{n(n+1)} \\ &= S(n-2) + \frac{2(n-2)}{(n-1)n} + \frac{2(n-1)}{n(n+1)} \\ &= S(n-3) + \frac{2(n-3)}{(n-2)(n-1)} + \frac{2(n-2)}{(n-1)n} + \frac{2(n-1)}{n(n+1)} \\ &\dots \\ &= S(1) + \frac{2\cdot 1}{2\cdot 3} + \frac{2\cdot 2}{3\cdot 4} + \frac{2\cdot 3}{4\cdot 5} + \dots + \frac{2(n-3)}{(n-2)(n-1)} + \frac{2(n-2)}{(n-1)n} + \frac{2(n-1)}{n(n+1)} \\ &= S(1) + 2\sum_{k=2}^{n} \frac{k-1}{k(k+1)} \end{aligned} \tag{9}$$

Да разгледаме (9). $S(1)$ е константа, това е някакво начално условие, и асимптотиката на $S(n)$ се определя от $\sum_{k=2}^{n} \frac{k-1}{k(k+1)}$. Но

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

Добре известно е[†], че $\sum_{k=1}^{n} \frac{1}{k} \asymp \lg n$, откъдето веднага следва, че $\sum_{k=2}^{n} \frac{1}{k+1} \asymp \lg n$. От друга страна, $\sum_{k=2}^{n} \frac{1}{k(k+1)}$ е ограничена от константа, защото редът $\sum_{k=1}^{\infty} \frac{1}{k^2}$ е сходящ със сума $\frac{\pi}{6}$. Следва, че $\sum_{k=2}^{n} \frac{k-1}{k(k+1)} \asymp \lg n$. Тоест, $S(n) \asymp \lg n$. Връщаме се към нотацията $T(n)$ и получаваме, че

$$T(n) \asymp n \lg n$$

Доказахме, че QUICKSORT има средна сложност по време $\Theta(n \lg n)$. Оттук и средната сложност по памет е логаритмична (а не линейна като в най-лошия случай), защото средната дълбочина на стека на рекурсията е $\Theta(\lg n)$, а на всяко ниво на рекурсията се ползва само константна допълнителна памет от функцията PARTITION.

Литература

[Ben86] Jon Bentley. Programming pearls: Little languages. *Commun. ACM*, 29(8):711–721, August 1986.

[†] Виж [CLRS09, стр. 44].

[CLRS09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[Hoa62] C. A. R. Hoare. Quicksort. *The Computer Journal*, 5(1):10–16, 1962.

[Knu98] Donald E. Knuth. *The Art of Computer Programming, Volume 3: (2Nd Ed.) Sorting and Searching*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1998.