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

Втора лекция по ДАА на втори поток КН
Асимптотични нотации. Сортиране. Елементарни сортиращи алгоритми.

10 март 2014

Абстракт

Извеждаме точни изрази за сложността на конкретни прости алгоритми. Показваме необходимостта от по-малко прецизна, по-груба мярка за сложността на алгоритмите. Въвеждаме петте общоприети асимптотични нотации. Разглеждаме задачата Сортиране и нейната важност за решаването на други задачи. Въвеждаме понятието *стабилност на сортиращ алгоритъм*. Разглеждаме два елементарни сортиращи алгоритъма: Insertion Sort и Selection Sort и анализираме тяхната коректност, сложност по време и памет и стабилност. Особено внимание отделяме на коректността, като въвеждаме доказване чрез инварианта на цикъл.

Съдържание

1 Точната сложност по време на SelectionSort и InsertionSort

Да разгледаме следния елементарен сортиращ алгоритъм. По-късно ще докажем неговата коректност. В момента се интересуваме само от това, колко бързо работи. Спазваме допусканията от първата лекция, че числата имат големина единица и че всяка елементарна инструкция отнема единица време.
Псевдокодът, който ще разгледаме, е по учебника [CLR09].

```
InsertionSort(A[1,2,...n]: масив от цели числа)
1  for i ← 2 to n
2      key ← A[i]
3      j ← i − 1
4      while j > 0 and A[j] > key do
5          A[j + 1] ← A[j]
6          j ← j − 1
7      A[j + 1] ← key
```

Колко стъпки отнема изпълнението на този алгоритъм върху n числа? Използваме следните съображения.

- В нашия опростен модел това, което е на ред 1—а именно инициализацията на управляващата променлива i, проверката за продължаване и инкрементирането—е една инструкция.[†] Тя се изпълнява n пъти.

  Да поясним защо ред 1 се изпълнява n пъти. По принцип, всяка инструкция от вида **for** $i \leftarrow a$ **to** b се изпълнява точно $b - a + 2$ пъти (ако $a \leqslant b$). Тук **не става дума за тялото на цикъла**, което се изпълнява точно $b - a + 1$ пъти. Редът **for** $i \leftarrow a$ **to** b се изпълнява веднъж за всяко "завъртане" на цикъла, и после още веднъж, когато условието за продължаване вече не е вярно. В този случай, $n - 2 + 2$ е точно n.

- Тялото на цикъла **for** на редове 1–7 се изпълнява, както казахме, $n - 2 + 1 = n - 1$ пъти. Следователно, всеки от редовете 2, 3 и 7 се изпълнява $n - 1$ пъти, и тъй като всеки от тях има по една инструкция, която се изпълнява за една стъпка, това са общо $3(n - 1)$ стъпки.

- Условието (ред 4) на вложения **while** цикъл (редове 4–6) се достига $n - 1$ пъти, но вложеният цикъл **не се изпълнява за една инструкция**. Освен това, точно колко пъти ще се изпълни вложеният цикъл при някакво i зависи от **конкретния вход**. Така че, за да оценим точно колко пъти ще бъдат изпълнени всеки от редовете 4–6, трябва да извършим по-подробен анализ.

  Нека k_i е броят на изпълненията на ред 4 за всяко $i \in \{2, \ldots, n\}$. Очевидно $k_i \in \{1, \ldots, i\}$, защото ред 4 се изпълнява поне веднъж и най-много i пъти. Защо най-много i пъти? – защото j може да стане най-малко 0, бивайки инициализирано със стойност $i - 1$ на ред 3.

  Всеки от редове 5 и 6 се изпълнява $k_i - 1$ пъти. Тогава ред 4 се изпълнява $\sum_{i=2}^{n} k_i$ пъти *общо за цялото изпълнение на алгоритъма*, а редове 5 и 6 се изпълняват по $\sum_{i=2}^{n}(k_i - 1)$ пъти. Общо изпълнението на **while** цикълът "струва" $\sum_{i=2}^{n} k_i + 2\sum_{i=2}^{n}(k_i - 1) = \left(3\sum_{i=2}^{n} k_i\right) - 2(n - 1)$ стъпки по време на цялото изпълнение.

И така, в нашия опростен модел, изпълнението на алгоритъма става в

$$n + 3(n - 1) + 3\left(\sum_{i=2}^{n} k_i\right) - 2(n - 1) = 2n - 1 + 3\sum_{i=2}^{n} k_i$$

[†] При програмиране на истински компютър това не е така. Следният тривиален **for** цикъл на езика C:

```
for (i=2; i <= n; i++);
```

като фрагмент от програма се компилира върху Intel iCore3 процесор, Ubuntu Linux, 64 битов gcc компилатор без оптимизация, в следния код (адресите и отместванията, естествено, варират):

```
0x4005bb <main+30> movl $0x2,-0x4(%rbp)
0x4005c2 <main+37> jmp 0x4005c8 <main+43>
0x4005c4 <main+39> addl $0x1,-0x4(%rbp)
0x4005c8 <main+43> mov -0x8(%rbp),%eax
0x4005cb <main+46> cmp %eax,-0x4(%rbp)
0x4005ce <main+49> jle 0x4005c4 <main+39>
```

Както виждаме, при истинския компютър инкрементирането с единица `addl`, проверката за край на изпълнението на цикъла `cmp`, и условният скок `jle` в изпълнението, са отделни инструкции.

стъпки. За да довършим анализа, трябва да определим минималната и максималната възможна стойност на сумата. k_i може да е най-малко 1, когато тялото на **while** цикъла не се изпълнява изобщо, и най-много i, когато тялото на **while** цикъла се изпълнява $i-1$ пъти.

Показахме, че сложността по време в най-добрия случай е

$$P(n) = 2n - 1 + 3\sum_{i=2}^{n} 1 = 2n - 1 + 3n - 3 = 5n - 4 \tag{1}$$

и сложността по време в най-лошия случай е

$$T(n) = 2n - 1 + 3\sum_{i=2}^{n} (i-1) = 2n - 1 + \frac{3n(n-1)}{2} = \frac{3n^2+n}{2} - 1 \tag{2}$$

Ще анализираме сложността на още един алгоритъм по аналогичен начин.

```
SELECTION SORT(A[1,2,...n]: масив от цели числа)
1   for i ← 1 to n − 1
2       for j ← i + 1 to n
3           if A[j] < A[i]
4               swap(A[i], A[j])
```

Ред 1 се изпълнява общо $n-1-1+2 = n$ пъти. Ред 2 се изпълнява $n-i+1$ за всяко i, общо $\sum_{i=1}^{n-1} n-i+1 = \frac{(n+2)(n-1)}{2}$. Ред 3 се изпълнява $n-i$ за всяко i, общо $\sum_{i=1}^{n-1} n-i = \frac{n(n-1)}{2}$. Ред 3 се изпълнява общо k пъти за някакво k, такова че $0 \leqslant k \leqslant \frac{n(n-1)}{2}$, в зависимост от конкретния вход. Общо сложността е

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

Тогава сложността в най-добрия случай е

$$P'(n) = n^2 + n - 1 \tag{3}$$

и сложността в най-лошия случай е

$$T'(n) = n^2 + n - 1 + \frac{n(n-1)}{2} = \frac{3n^2+n}{2} - 1 \tag{4}$$

2 Необходима ли е толкова прецизна оценка на сложността?

От изложението досега трябва да е станало ясно, че точни изрази като (1), (2), (3) и (4) не са необходими. Нашият модел на сложността е достатъчно далече от реалността, така че събираемото -1, което имаме в (4), няма практически никакво значение по отношение на каквато и да е софтуерна реализация на SELECTION SORT. Както казахме, не можем да предвидим времето за изпълнение с точност до милисекунда, така че по отношение на реално изпълнение, събираемото -1 не ни казва *нищо*. Нещо повече, цялото събираемо $\frac{1}{2}n-1$ е безсмислено. Ако разполагаме с конкретна реализация на алгоритъма и я тестваме върху различни тестови входове, за които имаме най-лоша сложност (обратно сортирани последователности) и после анализираме данните, ще стане ясно, че бързодействието се интерполира от някаква квадратична функция, и толкова. Фактори като конкретна операционна система, компилатор, архитектура и виртуална машина, както и умението на програмиста да пише бързи програми върху конкретната машина, със сигурност ще се отразят повече на реалното бързодействие.

Това, което има значение в (4) е фактът, че от трите събираеми най-бързо растящото е квадратичната функция $\frac{3}{2}n^2$. Тя задава *квадратична сложност по време*. Ако входът е достатъчно голям, дори и най-добрата конкретна реализация ще има квадратична сложност в n. Никакъв избор на конкретна операционна система, компилатор, програмистки трикове и така нататък не могат да подобрят квадратичната сложност до, да речем, *линейна* (след малко ще дефинираме линейна сложност).

И така, най-съществено за сложностите на INSERTION SORT и SELECTION SORT е, че в най-лошия случай[†] те са квадратични функции.

[†]Казахме, че сложността в най-добрия случай не е особено информативна и не се ползва, но си заслужава да се отбележи, че в най-добрия случай INSERTION SORT има сложност, която е линейна функция, без алгоритъмът да е модифициран чрез изкуствено пришити инструкции за само един най-добър случай.

В нашата поредица от допускания, които опростяват нещата и ни позволяват изобщо да правим анализ на сложността на алгоритми, следва допускането, че *степента на нарастване* на функцията на сложността е всичко, което ни интересува за нея. Защо?

Защото се интересуваме само от сложността при неограничено нарастване на големината на входа. За нас функцията на сложността е тенденцията при неограничено нарастване на големината на входа. Никаква конкретна големина на входа не ни интересува! Никаква нейна конкретна стойност, ако ще да е $10\,000\,000^{10\,000\,000^{10\,000\,000}}$, не е достатъчно голяма.

След малко ще дадем прецизно определение на "степен на нарастване", засега ще се задоволим с примера с анализа на двата алгоритъма горе. И на двата функциите на сложността в най-лошия случай са квадратични. Ще смятаме, че сложността им е еднаква: квадратична.

И така:

- ако изразът за сложността на даден алгоритъм е сума, разглеждаме най-бързо растящата функция измежду събираемите, в *асимптотичния смисъл*. А именно, ако изразът е $f_1(n) + f_2(n) + \ldots + f_k(n)$ и $\lim_{n\to\infty} \frac{f_1(n)}{f_i(n)} = \infty$ за всяко i, такова че $2 \leqslant i \leqslant k$, то $f_1(n)$ е най-бързо растящата в асимптотичния смисъл функция измежду $f_1(n)$, $\ldots$, $f_k(n)$[†]. Ще смятаме, че най-бързо растящата в този смисъл функция доминира над останалите и тях можем да пренебрегнем.

- за два различни алгоритъма, ако водещите функции в горния смисъл в изразите за сложността са $f_1(n)$ и $g_1(n)$ и $0 < \lim_{n\to\infty} \frac{f_1(n)}{g_1(n)} < \infty$, ще смятаме, че тези алгоритми работят еднакво бързо.

Дори първото допускане *не винаги е реалистично*. Смисълът да се разглежда само тенденция на нарастването, когато n клони към безкрайност, е ясен: по-големите входове са по-интересни. Но на практика това е вярно само донякъде. На практика, наистина поведението на алгоритъм за сортиране върху вход с големина $10\,000\,000$ е по-интересно от поведението му върху вход с големина 10, но едва ли можем да твърдим, че поведението му върху вход с големина $10\,000\,000^{10\,000\,000}$ е още по-интересно, а това върху вход с големина $10\,000\,000^{10\,000\,000^{10\,000\,000}}$, дори още по-интересно. Последните две числа са напълно отвъд възможностите на всеки истински компютър, сегашен и бъдещ, така че програмата, реализираща въпросния алгоритъм, никога няма да има възможност да демонстрира бързината си върху входове с такива размери. Когато се фокусираме само върху водещото събираемо, ние твърдим, че измежду два алгоритъма, единият от които има сложност $n^2 + n$, а другият, $n\sqrt{n} + 10\,000\,000^{10\,000\,000^{10\,000\,000}} n$, вторият работи по-бързо, защото неговата водеща функция $n\sqrt{n}$ расте по-бавно от n^2, водещата функция на първия. Очевидно съществува стойност на големината на входа n_0, такава че за всяко $n \geqslant n_0$, вторият алгоритъм печели като бързодействие, но това n_0 е твърде голямо. Както вече стана ясно, то е число, което никога не е големина на вход в реалния свят.

Второто допускане—игнорирането на мултипликативната константа във водещата функция—е дори по-фрапиращо. Според него, два алгоритъма, единият от които има сложност $n^2 + n$, а другият има сложност $10\,000\,000^{10\,000\,000^{10\,000\,000}} n^2$, са с една и съща сложност, а именно квадратична. "Оправданието" ни е, че алгоритмите, които се използват на практика и почиват върху прагматични идеи по правило имат сложности, в които тези мултипликативни константи не са фрапиращо големи и никога не са от сорта на $10\,000\,000^{10\,000\,000^{10\,000\,000}}$, така че опростяването не е прекалено грубо. В теорията обаче са известни линейни алгоритми, при които тази мултипликативна константа е кула от степени на двойката

$$2^{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}}$$

чиято височина е кула от степени на двойката, чиято височина е кула от степени на двойката, и така нататък, девет пъти. Очевидно такъв алгоритъм е непрактичен дори за вход с големина единица.

Да обобщим. Когато разглеждаме тенденцията на нарастването при неограничено нарастване на аргумента n, ние просто правим поредното опростяващо допускане. Причината да го правим е, че **така е по-лесно**. Освен това, практиката показва, че често—но **не винаги**—резултатите, които ни дава това опростяване, са добре корелирани с качествата на реалните програми, реализиращи съответните алгоритми.

[†]Тук предполагаме, че всички те са положителни функции.

3 Асимптотични нотации

Следните дефиниции ни дават възможност да изразим формално твърдението, че две функции имат една и съща степен на нарастване или че една расте по-бързо от другата или строго по-бързо от другата или че двете са несравними, и то в асимптотичния смисъл. Забележете, че дефинициите не използват нотацията lim.

Преди това, едно помощна дефиния: функция $f(n)$ с домейн и кодомейн $\mathbb{R}^+$ е *асимптотично положителна,* ако $\exists n_0 \in \mathbb{R}^+ : \forall n \geqslant n_0, f(n) > 0$. Отсега нататък ще ще допускаме, че функции са такива. При анализа на алгоритми сложността няма как да не е положителна функция, така че това допускане не е ограничаващо.

$$\Theta(g(n)) \stackrel{\text{def}}{=} \{f(n) \mid \exists c_1, c_2 > 0, \exists n_0 : \ \forall n \geqslant n_0, \ 0 \leqslant c_1.g(n) \leqslant f(n) \leqslant c_2.g(n)\} \tag{5}$$

Формално, $\Theta(g(n))$ е безкрайно множество от функции. Ако искаме да кажем, че $h(n)$ е една от тях, пишем $h(n) = \Theta(g(n))$, наместо формално коректното $h(n) \in \Theta(g(n))$. Четем, "$h(n)$ е Тета-голямо от $g(n)$". Смисълът е, че $g(n)$ задава безкрайно множество функции, всяка от които ще смятаме за еквивалентна на $g(n)$ като асимптотична степен на нарастване Забележете, че "$\lim_{n\to\infty} \frac{h(n)}{g(n)}$ съществува и е равно на някакво L, такова че $0 < L < \infty$" е по-силно твърдение от "$h(n) = \Theta(g(n))$". С други думи, Тета-нотацията е по-мощна от изразяването чрез граница, защото границата на отношението може да не съществува и въпреки това Тета-отношението да е изпълнено. Примерно, нека $f(n) = (2 + \sin n)n^2$ и $g(n) = n^2$. Вярно е, че $f(n) = \Theta(g(n))$. За доказателството можем да вземем константи 1 и 3. Обаче границата $\lim_{n\to\infty} \frac{f(n)}{g(n)}$ не съществува, така че във формализма на границите не бихме могли да изразим желаното твърдение.

Тривиално се доказва, че за всички асимптотично положителни функции $f(n)$, $g(n)$ и $h(n)$:

$$f(n) = \Theta(f(n))$$
$$f(n) = \Theta(g(n)) \to g(n) = \Theta(f(n))$$
$$f(n) = \Theta(g(n)) \wedge g(n) = \Theta(h(n)) \to f(n) = \Theta(h(n))$$

Това ни дава основание да твърдим, че Тета-нотацията задава релация на еквивалентност. Затова въвеждаме алтернативна нотация: наместо $f(n) = \Theta(g(n))$ можем да запишем $f(n) \asymp g(n)$. Този запис подчертава, че става дума за релация на еквивалентност.

Използвайки Тета-нотацията, можем да запишем сложността в най-лошия случай $T(n)$ на Insertion Sort и $T'(n)$ на Selection Sort така:

$$T(n) = \Theta(n^2)$$
$$T'(n) = \Theta(n^2)$$

От гледна точка на Тета-асимптотиката, по-прецизен анализ на тези алгоритми няма. Те са квадратични и толкова. Линеен е всеки алгоритъм, чиято сложност в най-лошия случай е $\Theta(n)$.

Записът $\Theta(n^2) = T(n)$ **е некоректен**. Знакът за равенство в Тета-изразите е еднопосочен: Тета-нотацията се появява вдясно.

Изключение от това правило са изрази като

$$n^2 + \Theta(n) = \Theta(n^2)$$

В този случай Тета-нотациите вляво и вдясно имат съвсем различен смисъл. Тета-нотацията вдясно представлява безкрайно множество от функции, а тази вляво е *анонимна функция.* Да повторим: Тета-нотацията вляво представлява не множество функции, а само една, като всичко, което знаем за нея е, че принадлежи на множеството $\Theta(n)$. Целият израз се чете така: сумата на n^2 и коя да е функция от множеството $\Theta(n)$ е функция от множеството $\Theta(n^2)$.

Формално погледнато, дали ще пишем

$$10\sqrt{7}n^2 + 15.5n\sqrt{n} + 35n = \Theta(n^2)$$

или

$$n^2 = \Theta(10\sqrt{7}n^2 + 15.5n\sqrt{n} + 35n)$$

е без значение, и двете твърдения са верни. На практика никой не би използвал втория запис. По правило в скобите на Тета-нотацията слагаме възможно най-простия запис; тоест, функцията от класа на еквивалентност, която има най-прост запис.

Както казахме, що се отнася до асимптотиката на нарастването, Тета-нотацията дава възможно най-подробната информация. Въпреки че Тета-нотацията е доста "хлабава" в смисъл, че, примерно, $\frac{n^3}{1000}$ и $1000^{1000}n^3$ са Тета една от друга и следователно са еквивалентни, понякога не можем да намерим Тета-оценката на някаква функция. За такива случаи има други нотации, по-слаби от Тета-та, които са все пак информативни за асимптотиката на нарастването.

$$O(g(n)) \stackrel{\text{def}}{=} \{f(n) \mid \exists c > 0, \exists n_0 : \forall n \geqslant n_0, 0 \leqslant f(n) \leqslant c.g(n)\} \tag{6}$$

$$\Omega(g(n)) \stackrel{\text{def}}{=} \{f(n) \mid \exists c > 0, \exists n_0 : \forall n \geqslant n_0, 0 \leqslant c.g(n) \leqslant f(n)\} \tag{7}$$

$$o(g(n)) \stackrel{\text{def}}{=} \{f(n) \mid \forall c > 0, \exists n_0 : \forall n \geqslant n_0, 0 \leqslant f(n) < c.g(n)\} \tag{8}$$

$$\omega(g(n)) \stackrel{\text{def}}{=} \{f(n) \mid \forall c > 0, \exists n_0 : \forall n \geqslant n_0, 0 \leqslant c.g(n) < f(n)\} \tag{9}$$

Чрез тях можем да записваме съответно асимптотична нестрога горна граница, асимптотична нестрога долна граница, асимптотична строга горна граница и асимптотична строга долна граница. Ако $f(n) = O(g(n))$, то за някаква положителна константа c, $f(n)$ не надвишава $c.g(n)$, и то не непременно навсякъде, а от *някакво място нататък*, като става дума за n_0. Не знаем точно какво е n_0, но има такова. Ако $f(n) = o(g(n))$, то $f(n)$ изостава произволно много от $g(n)$ в смисъл, че колкото и голяма константа c да изберем, от едно място—някакво n_0 нататък—$f(n)$ е по-малка от $c.g(n)$. За $\Omega(\,)$ и $\omega(\,)$ има аналогични съображения в обратната посока – те са долни граници.

Наместо $f(n) = O(g(n))$, $f(n) = o(g(n))$, $f(n) = \Omega(g(n))$ и $f(n) = \omega(g(n))$ можем да пишем съответно $f(n) \preceq g(n)$, $f(n) \prec g(n)$, $f(n) \succeq g(n)$ и $f(n) \succ g(n)$. Тези записвания са подходящи, защото напомнят, че става дума за нещо като наредби. Да си припомним, че релация, която е рефлексивна и транзитивна се нарича преднаредба или квазинаредба. Очевидно $\preceq$ и $\succeq$ са релации на преднаредба понеже са рефлексивни и транзитивни. $\prec$ и $\succ$ са антирефлексивни и транзитивни. Релациите не са антисиметрични, защото може за различни функции $f(n)$ и $g(n)$ да е изпълнено $f(n) \preceq g(n)$ и $g(n) \preceq f(n)$. Следователно, в общия случай тези релации не са частични наредби, откъдето следва, че не са и тотални наредби.

Забележете, че за две функции $f(n)$ и $g(n)$ може нито едно от $f(n) \asymp g(n)$, $f(n) \preceq$, $f(n) \prec g(n)$, $f(n) \succeq g(n)$, и $f(n) \succ g(n)$ да не бъде изпълнено, примерно ако са дефинирани върху естествените числа и $f(n) = n$, а

$$g(n) = \begin{cases} n^2, & \text{четно} \\ 1, & \text{нечетно} \end{cases}$$

Очевидно $f(n) \asymp g(n) \leftrightarrow f(n) \preceq g(n) \wedge f(n) \succeq g(n)$. Също така, няма функции $f(n)$ и $g(n)$, такива че $f(n) \prec g(n)$ и $f(n) \succ g(n)$. За малкото-о и малкото-омега е вярно, че

$$f(n) \prec g(n) \leftrightarrow \lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$$

$$f(n) \succ g(n) \leftrightarrow \lim_{n\to\infty} \frac{g(n)}{f(n)} = \infty$$

Подробна дискусия на асимптотичните нотации има в [CLR09].

4 Сортиране

4.1 Обща дефиниция

Тъй като задачата Сортиране се дефинира чрез някаква наредба, първо ще кажем каква наредба имаме предвид. *Квази-наредба* или *преднаредба* (на английски *quasi-oder* или *preorder*) се нарича релация, която е рефлекивна и транзитивна[†]. Релацията в Сортиране е квази-наредба, която съдържа тотална наредба като подрелация (т.е., като поднможество). Най-общата дефиниция на Сортиране е следната. Нека $A = \{a_1, \ldots,$

[†]Лесно се вижда, че релациите на частична наредба и релациите на еквивалентност са частни случаи на квази-наредби: ако квази-наредбата е симетрична, то тя е релация на еквивалентност; а ако е антисиметрична, то тя е релация на частична наредба.

$a_n\}$ е множество, върху което е дефинирана квази-наредба $\leqslant$, която съдържа тотална наредба[†]. Освен това, допускаме, че за единица време може да правим сравнения от вида $a_i \overset{?}{\leqslant} a_j$ и за единица време можем да извършваме присвоявания с тези обекти. Не се иска обектите да са непременно различни. Подчертаваме, че примерът на задачата е *последователността* $a_1, a_2, \ldots, a_n$, а не (мулти)множеството $\{a_1, \ldots, a_n\}_M$. По отношение на крайния резултат началната подредба няма значение[‡], но по отношение на сложността по време има значение точно как са наредени елементите във входа.

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

$a_1, \ldots, a_n$ не са непременно естествени числа. Може да са реални числа (тогава е особено важно да се подчертае допълнителното допускане, че имат големина единица и че може да бъдат сравнявани и присвоявани в единица време). Може да са дори комплексни числа, като тогава има различни възможности за дефиниране на релацията $\leqslant$. Може да са някакви по-общи структури. На практика най-често възниква необходимостта да се сортират *записи*, всеки от които има *ключ* и *някаква сателитна информация*, като релацията е дефинирана само върху ключовете. Ключовете може да са повече един. В такъв случай говорим за първичен ключ, вторичен ключ и така нататък.

С учебна цел ще разглеждаме основно сортиране на цели числа.

4.1.1 За квази-наредбите

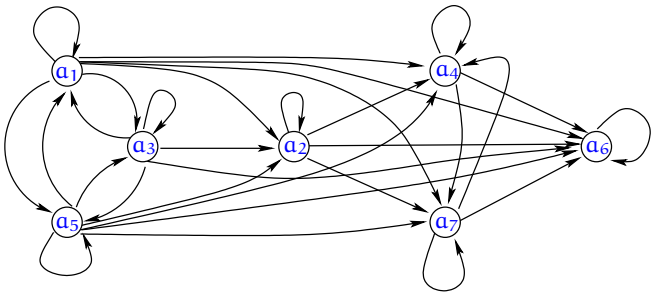
Няколко пояснения върху това, какво представлява квази-наредба. Ако $\leqslant$ е квази-наредба, в нея може да има *контури*, тоест за някакви, два по два различни, $u_1, \ldots, u_t$ за $t \geqslant 2$ да е изпълнено

$$u_1 \leqslant u_2 \leqslant \ldots \leqslant u_{t-1} \leqslant u_t \leqslant u_1$$

В релациите на частична наредба това е невъзможно. Държим релацията да е именно квази-наредба, а не тотална наредба[§], за да може във входа да има различни елементи с еднакъв ключ. По отношение на релацията $\leqslant$, всяко подмножество от елементи с еднакъв ключ индуцира подрелация, която е пълна: всеки от тях е в релация със всеки от това подмножество.

От друга страна, не искаме $\leqslant$ да е частична наредба, защото в частичните наредби има двойки несравними елементи, докато при Сортиране искаме всяка двойка елементи да са сравними. Затова искаме не произволна квази-наредба, защото частичните наредби са квази-наредби, а такава квази-наредба, която съдържа тотална наредба в себе си.

Най-общият вид на релацията, която имаме предвид при Сортиране, може лесно да се опише чрез нейния граф. Неговите силно свързани компоненти са пълни (ориентирани) графи с примки, а тези силно свързани компоненти са на свой ред подредени с тотална наредба. Лесно се вижда, че релацията, която имаме предвид, е квази-наредба, в която има верига, съдържаща всички елементи. Примерно, ако $A = \{a_1, \ldots, a_7\}_M$, където $a_1 = a_3 = a_5 = 2$, $a_2 = 4$, $a_4 = a_7 = 8$, и $a_6 = 9$, графът на релацията $\leqslant$ изглежда така:

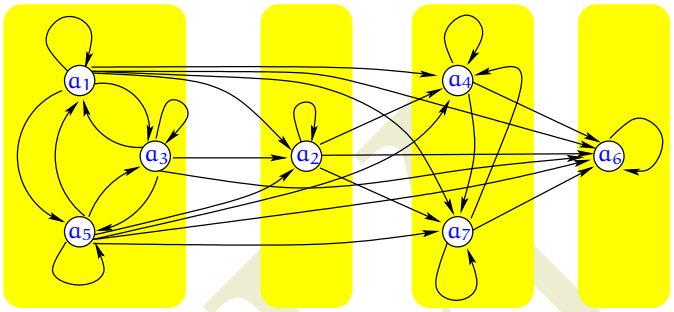


Действително силно свързаните компоненти са пълни ориентирани графи с примки:

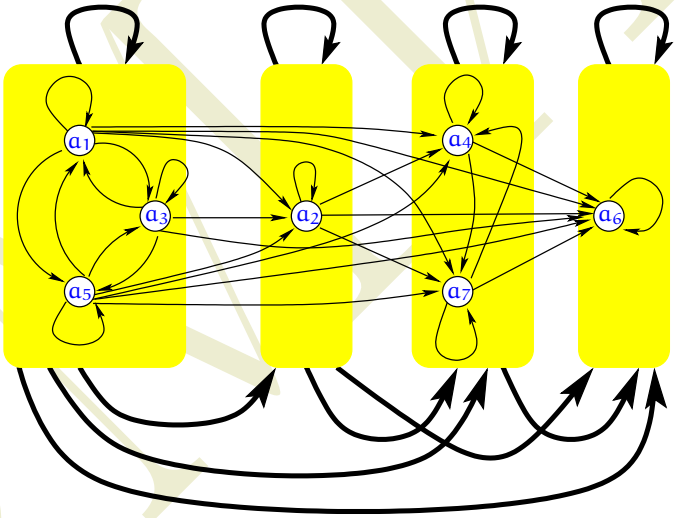
[†]Да настояваме $\leqslant$ да е тотална наредба е ограничение: това работи само ако елементите са уникални. Бихме искали да дефинираме по-общия случай, в който някои елементи или дори всички елементи може да имат еднаква стойност.

[‡]Освен ако няма елементи с еднаква стойност, които все пак някак различаваме. В такъв случай резултатът може да се различен за различни наредби на входа, ако сортиращият алгоритъм не е стабилен. За това ще говорим повече нататък.

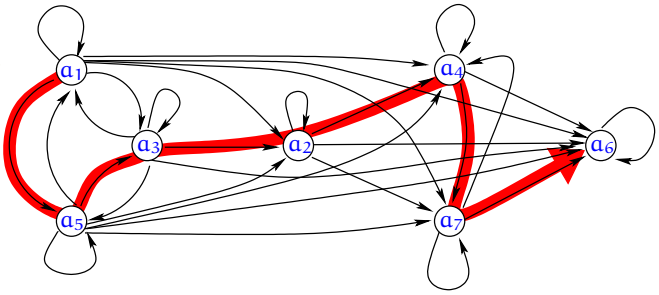
[§]Припомнете си, че тоталните наредби са специален случай на частични наредби.



и върху тях има тотална наредба:



Действително релацията съдържа тотална наредба:



4.2 Защо сортирането е важно

На практика сортирането не е самоцелно, а обикновено е първа част от решаването на някаква задача. Има доста примери за важни задачи, за които бързото решение използва сортирането като предварителна обработка. Алгоритмичният фолклор казва, че ако съставител на алгоритми е изправен пред непозната задача, едно от първите неща, които трябва да опита, за да направи ефикасен алгоритъм, е да сортира някакви данни и да види дали от това няма да произтече нещо полезно [Ski08, стр. 106].

4.2.1 Двоично търсене

В простия си вариант с еднобитов отговор, задачата Търсене се дефинира така.

Изчислителна задача Търсене
Параметри: $a_1, a_2, \ldots, a_n$, key: обекти от един и същи тип
Въпрос: Дали има елемент измежду $a_1, a_2, \ldots, a_n$, който е равен на key?

На практика по-полезна е тази версия, да я наречем оптимизационна версия на задачата.

Изчислителна задача Търсене
Общ пример: $a_1, a_2, \ldots, a_n$, key: обекти от един и същи тип
Изход: Ако има елемент измежду $a_1, a_2, \ldots, a_n$, който е равен на key, индексът на един такъв елемент. В противен случай индикация, че такъв елемент няма.

Ако входът на даден алгоритъм за Търсене е *произволна* последователност $a_1, a_2, \ldots, a_n$, то единствената възможност за действие е проверяване дали $a_i \stackrel{?}{=} \text{key}$ по всички $i \in \{1, \ldots, n\}$. Това се нарича *последователно търсене*. Най-лошият случай по отношение на сложността е, key да не е нито един от елементите $a_1, a_2, \ldots, a_n$. Тогава алгоритъмът трябва да направи n сравнения. Ако обаче

- е известно, че елементите $a_1, a_2, \ldots, a_n$ са уникални[†]
- те са сортирани

можем да приложим *двоично търсене* (на английски е *binary search*), което е много по-бързо от последователното търсене.

```
BinSearchIter(A[1,...,n], key)
 1  (* A[ ] е сортиран *)
 2  ℓ ← 1
 3  h ← n
 4  while ℓ < h do
 5      mid ← ⌊(ℓ+h)/2⌋
 6      if A[mid] = key
 7          return mid
 8      else if key < A[mid]
 9          h ← mid − 1
10      else ℓ ← mid + 1
11  return −1
```

```
BinSearchRec(A[1,...,n], key, ℓ, h)
 1  (* A[ ] е сортиран *)
 2  if ℓ > h
 3      return −1
 4  else mid ← ⌊(ℓ+h)/2⌋
 5      if A[mid] = key
 6          return mid
 7      else if key < A[mid]
 8          return BinSearchRec(A[ ], key, ℓ, mid − 1)
 9      else
10          return BinSearchRec(A[ ], key, mid + 1, h)
```

Началното викане на рекурсивната версия е BinSearchRec$(A[1, \ldots, n], \text{key}, 1, n)$. Посочените два алгоритъма решават оптимизационната версия на Търсене. Накратко, тяхната коректност се обосновава така: ако изобщо елементът key се намира в масива $A[\,]$, той се намира в подмасива $A[\ell, \ldots, h]$. Това твърдение остава вярно след промените в индексите ℓ и h само защото масивът е сортиран предварително.

Сложността по време и на двата алгоритъма е $\Theta(\lg n)$ в най-лошия случай, която обосновка остава на читателя. Нещо повече, точната сложност като брой сравнения $A[\text{mid}] \stackrel{?}{=} \text{key}$ е $\lfloor \lg n \rfloor + 1$ в най-лошия случай. Разликата между последователното търсене в линейно време и двоичното търсене в логаритмично време е огромна. Ако, примерно, $n = 7\,000\,000\,000$, с *не повече от* 33 *четения-и-сравнения на елементи от масива* можем да установим, че даден елемент не е във входа, ако ползваме двоично търсене. Сравнете 33 с 7 000 000 000!

4.2.2 Най-близки елементи

Ето два варианта на задачата.

Изчислителна задача Най-близки Елементи 1
Общ пример: $a_1, a_2, \ldots, a_n$: цели числа
Изход: $\min\{|a_i - a_j| : 1 \leqslant i < j \leqslant n\}$

Изчислителна задача Най-близки Елементи 2
Общ пример: $a_1, a_2, \ldots, a_n$: цели числа
Изход: Двойка индекси (i, j), такива че разликата $|a_i - a_j|$ е минимална.

[†]Двоичното търсене не е приложимо, ако може да има елементи с еднаква стойност, които все пак различаваме някак, и търсим не просто по стойност, а точно определен елемент измежду тези с дадена стойност. С други думи, за да работи двоичното търсене, трябва елементите да са подредени с тотална наредба.

Не е задължително елементите да са цели числа, но трябва да бъдат от множество, върху което е дефинирана тотална наредба, и трябва да е дефинирано разстояние между всеки два елемента, което да се изчислява в константно време. В горната дефиниция $|a_i - a_j|$ е разстоянието.

Наивното решение на задачата е да се тестват всички двуелементи подмножества, които са $\binom{n}{2} = \Theta(n^2)$ на брой. По-добро решение е, първо елементите да бъдат сортирани и след това с едно единствено сканиране на сортираната последователност отляво надясно най-близката двойка ще бъде намерена. Както ще видим в следваща лекция, сортирането може да бъде имплементирано със сложност $\Theta(n \lg n)$, така че бързото решение има сложност

$$\underbrace{\Theta(n \lg n)}_{\text{сортиране}} + \underbrace{\Theta(n)}_{\text{сканиране}} = \Theta(n \lg n),$$

което е много по-добре от $\Theta(n^2)$.

4.2.3 Уникалност на елементите

Изчислителна задача Уникалност на елементите
Параметри: $a_1, a_2, \ldots, a_n$: цели числа
Въпрос: Вярно ли е, че няма нито два еднакви елемента измежду дадените?

Напълно аналогично на Най-близки елементи, наивното решение се състои в тестване на всички двуелементни подмножества, което означава $\Theta(n^2)$ тестове в най-лошия случай. Ако обаче първо сортираме числата, ако има еднакви елементи, всички елементи, които са два по два еднакви, задължително ще се появят в непрекъсната последователност в сортираната наредба. С едно сканиране отляво надясно ще установим дали има еднавки елементи. Отново, сложността на подхода, използващ първо сортиране, е $\Theta(n \lg n)$.

4.2.4 Мода и медиана

Мода е най-често срещаната стойност измежду дадени стойности. *Медиана* е стойността, която разделя големините на дадени стойности наполовина. Строго погледнато, тази дефиниция на медиана е смислена само когато броят на стойностите е нечетно число, но да игнорираме тази подробност.

И модата, и медианата може да се изчисляват във време $\Theta(n \lg n)$ чрез предварително сортиране. Прочее, дефинията на медиана използва сортиране неявно: медианата е елементът, стоящ в средата на сортираната последователност на стойностите. За да бъде намерена модата е достатъчно едно сканиране на сортираните данни, медианата се намира в константно време след сортирането.

Заслужава да се отбележи, че медианата може да бъде изчислена в $\Theta(n)$ чрез алгоритъма, намиращ k-тото по големина число измежду n дадени числа в *линейно време* за произволно k [CLR09].

Модата не може да бъде изчислена във време $o(n \lg n)$. Това ще покажем в бъдеща лекция, когато разглеждаме асимптотични долни граници на задачи.

4.3 Анализ на сортиращи алгоритми

Анализът на даден сортиращ алгоритъм се състои в това, което изброихме миналата лекция за анализа на произволен алгоритъм: доказателство на коректността и анализ на сложността по време и по памет. В контекста на сортиращите алгоритми има още едно свойство, което представлява интерес. То се нарича *стабилност*. Сега ще обясним в детайли какво означава сортиращ алгоритъм да бъде стабилен с един пример[†] от [Knu98].

Задача 1 (задача 2 на стр. 5 в [Knu98])**.** *Да допуснем, че всеки запис R_j в даден файл съдържа два ключа, първичен ключ K_j и вторичен ключ k_j, като върху ключовете е дефинирана линейна наредба $<$. Дефинираме* лексикографска наредба *върху ключовете по стандартния начин:*

$$(K_i, k_i) < (K_j, k_j) \text{ ако } K_i < K_j \text{ или } K_i = K_j \text{ и } k_i < k_j$$

Alice взема файла и го сортира първо по първичните ключове, получавайки n групи от записи с еднакви първични ключове във всяка група:

$$K_{p(1)} = \ldots = K_{p(i_1)} < K_{p(i_1+1)} = \ldots = K_{p(i_2)} < \ldots < K_{p(i_{n-1}+1)} = \ldots = K_{p(i_n)},$$

[†]Кнут означава с N броя на елементите във входа, а сортираната пермутация с "$K_{p(1)} \leqslant K_{p(2)} \leqslant \ldots \leqslant K_{p(N)}$".

където $i_n = N$. *След това тя сортира всяка от* n *на брой групите по вторичния ключ.*

Bill взема оригиналния файл и първо го сортира по вторичния ключ, а след това, по първичния.

Chris взема оригиналния файл и прави едно единствено сортиране, като обаче ползва и двата ключа с лексикографската наредба.

Дали всеки от тях е получил непременно един и същи резултат?

Отговорът е, със сигурност Alice и Chris ще получат един и същи резултат. Резултатът на Bill ще е гарантирано същият само при условие, че неговото второ сортиране—това по първичния ключ—не променя относителния порядък на записи, имащи еднакъв (първичен) ключ. Ето малък пример: нека наредените двойки от ключовете са $(1,2)$, $(5,2)$, $(4,5)$, $(2,5)$, $(3,1)$, $(1,4)$, $(3,2)$. Сортираната с лексикографска наредба последователност е:

$$(1,2),(1,4),(2,5),(3,1),(3,2),(4,5),(5,2)$$

Това е резултатът, който ще получи Chris след своето сортиране. Какво точно ще получи Bill след първото сортиране зависи от сортиращия алгоритъм, който ползва. Със сигурност на първа позиция ще е наредената двойка $(3,1)$, после трите наредени двойки с втори елемент 2, после $(1,4)$, и накрая двете наредени двойки с втори елемент 5. Примерно,

$$(3,1),(5,2),(3,2),(1,2),(1,4),(4,5),(2,5)$$

Bill сортира тази наредба по първичните ключове. В получената наредба на първите две места са наредените двойки $(1,2)$ и $(1,4)$. Но забележете, че при произволно сортиране по първи ключ няма как да знаем дали $(1,2)$ ще е вляво от $(1,4)$ или не. От гледна точка на коректна лексикографска сортировка, разбира се, искаме началото да бъде

$$(1,2),(1,4),\ldots$$

но може да се получи

$$(1,4),(1,2),\ldots$$

тъй като при сортирането по първични ключове не "гледа" вторичните ключове. За сортирането само по първични ключове, $(1,2)$ и $(1,4)$ са еднакви елементи! Първата фаза—сортирането по вторични ключове—коректно е сложила $(1,2)$ преди $(1,4)$, но втората фаза може потенциално да сложи $(1,4)$ вляво от $(1,2)$, ако ги счита за еднакви.

Стабилно сортиране е такова, което не размества елементи с еднакви ключове. При стабилното сортиране, ако има два елемента с еднаква стойност, които са все пак различими, този, който е вляво от другия преди сортирането, задължително ще е вляво от него и след сортирането.

5 Елементарни Сортиращи Алгоритми

Разделянето на сортиращите алгоритми на елементарни и други, неелементарни, е доста условно. Под елементарни сортиращи алгоритми разбираме такива, които биха хрумнали бързо на човек, който за пръв път се сблъсква със Сортиране. По правило те работят във време $\Theta(n^2)$ в най-лошия случай и не използват никакви изтънчени структури данни или нетривиални идеи.

5.1 Insertion Sort

Insertion Sort сортира по начин, който далечно напомня на начина, по който картоиграч сортира картите, които държи в ръка (които са в произволна наредба в началото) – плъзгайки поглед отляво надясно, той или тя разглежда последователно картите. При това всяка карта, която е обект на внимание в момента, бива сложена на правилното ѝ място в подпоследователността от картите вляво. Има една важна разлика между този начин на нареждане и работата на алгоритъма Insertion Sort: карта може да бъде пъхната между другите карти мигновено, докато ако искаме да "пъхнем" дадено a_i някъде вляво, първо трябва да му направим място, за да не бъде затрит елементът a_j, който вече е там.
Да си припомним псевдокода на Insertion Sort

```
Insertion Sort(A[1,2,...n]: array of integers)
1   for i ← 2 to n
2       key ← A[i]
```

```
3       j ← i − 1
4       while j > 0 and A[j] > key do
5           A[j + 1] ← A[j]
6           j ← j − 1
7       A[j + 1] ← key
```

5.1.1 Коректност на Insertion Sort

Доказателството за коректност ще направим чрез *инварианта на цикъла*. Тъй като сега за първи път ще демонстрираме тази техника, ще обясним подробно как става доказателство чрез инварианта. Инвариантата е *едноместен предикат*, свързан с цикъла, и по-точно с реда, в който е условието за край на цикъла. В случая това е ред 1, ако става дума за външния цикъл, и ред 4, ако става дума за вътрешния цикъл. Този предикат трябва да е

- верен първия път, когато изпълнението на алгоритъма стигне до въпросния ред, и
- верността му трябва да се запазва по време на всяко изпълнение, и
- в момента на напускане на цикъла (такъв момент настъпва неизбежно, щом алгоритъмът завършва) неговата верност да влече директно това, което искаме да докажем за алгоритъма.

Очевидно става дума за вид доказателство по индукция, само че върху краен домейн—а именно, множеството $\{1, 2, , \ldots, m\}$, където m е броят на достиганията на реда, съдържащ условието за край на цикъла. Най-общо, предикатът се формулира така:

> При k-тото достигане на ред ℓ на алгоритъм Някакъв Алгоритъм е изпълнено *«някакво-твърдение-формулирано-чрез-*k*»*.

Ако предикатът е $P(k)$, то доказателството следва позната схема на доказателства по индукция

$$P(1) \wedge \forall k_{1 \leqslant k < m}(P(k) \to P(k+1)) \vdash \forall k_{1 \leqslant k \leqslant m}(P(k))$$

Съществува възможност предикатът да се дефинира по-просто, не чрез номера на текущото достигане на реда с условието за край, а направо чрез управляващата променлива на цикъла. Примерно, ако цикълът е **for**-цикъл с инкрементиране на управляващата променлива с единица

for $i \leftarrow n_1$ **to** n_2
«тяло-на-цикъла»

за някакви фиксирани n_1 и n_2, то може да вземем за домейн $\{n_1, \ldots, n_2 + 1\}$[†]. В този случай дефинираме предиката чрез името на индексната променлива. Щом тя е i, предикатът е $P(i)$. Но при по-сложни **while**-цикли, в които управляващата променлива се мени в тялото на цикъла по някакъв сложен начин, по-удачно е да формулираме предиката върху номера на достигането на реда с условието.

Да се върнем на доказателството на коректността на Insertion Sort. Целта е да докажем, че Insertion Sort сортира всеки свой вход. Алгоритъмът има два вложени цикъла. Много подробно доказателство трябва да има две отделни инварианти.

- Инварианта за вътрешния, **while**, цикъл. Тази инварианта ще формулираме спрямо j, управляващата променлива на вътрешния цикъл.
- Инварианта за външния, **for**, цикъл. Нея ще формулираме спрямо i.

Доказателствата за коректност на алгоритми са тежки, защото обектите, за които трябва да изказваме и доказваме твърдения, са *динамични*. Данните се менят с работата на алгоритъма заради присвояванията. Примерно, Insertion Sort мести присвоява на елементи на $A[\,]$ други елементи от $A[\,]$. Споменавайки $A[i]$ в Лема 1, какво имаме предвид? Елементът на A на позиция i **преди** вътрешният цикъл да премести подмасива

[†]Това е множеството от стойностите, които индексната променлива взема. Забележете, че най-голямата от тях е $n_2 + 1$, а не n_2, понеже при последното достигане на реда с условието—когато условието вече не е изпълнено и тялото на цикъла няма да се изпълнява повече—управляващата променлива е именно $n_2 + 1$.

"нагоре" с една позиция? Или **след** това? Позиция i на $A[]$ може да бъде засегната от това преместване, така че в общия случай това са **различни** елементи. За да избягваме подобни неясноти ще даваме различни имена на масива, или на негови подмасиви, в различни моменти от изпълнението.

Навсякъде в доказателствата долу допускаме, че $n > 1$.

Лема 1. *Да разгледаме* Insertion Sort. *Спрямо дадено изпълнение на* ***for*** *цикъла (редове 1–7), нека наричаме подмасива* $A[1, \ldots, i]$ *непосредствено преди изпълнението на* ***while*** *цикъла (редове 4–6) с името* $A'[1, \ldots, i]$. *Да допуснем, че* $A'[1, \ldots, i-1]$ *е сортиран. Тогава* ***while*** *цикълът има следните ефекти:*

- j *получава стойността на най-голямото число* $p \in \{1, 2, \ldots, i-1\}$, *такова че* $A'[p] \leqslant$ ***key***, *ако има такова число; ако няма такова число, тоест, ако всяко число от* $\{1, 2, \ldots, i-1\}$ *е по-голямо от* ***key***, *то* j *получава стойност* 0.
- *спрямо тази стойност на* j, *масивът* $A'[j+1, \ldots, i-1]$ *бива преместен с една позиция нагоре.*

Доказателство: Следното твърдение е инварианта за **while** цикъла:

> Всеки път, когато изпълнението стигне до ред 4:
>
> - за всеки елемент x от текущия подмасив $A[j+2, \ldots, i]$, $x >$ ***key***, и освен това
> - текущият подмасив $A[j+2, \ldots, i]$ е същият като $A'[j+1, \ldots, i-1]$.

Както казахме вече, доказателството на инвариантата е доказателство по индукция. Започва с ***база****, в която верността на предиката просто се проверява. В обикновените доказателства по индукция следващите две фази са, индуктивно предположение и индуктивна стъпка. Тук ги сливаме в една фаза от доказателството, която наричаме* ***поддръжка****. И накрая има последна фаза* ***терминация****, която няма еквивалент при обикновените доказателства по индукция. Терминацията се отнася до последното достигане на условието на цикъла – когато тялото няма да се изпълни нито веднъж повече. Когато заместим стойността на управляващата променлива от този момент в инвариантата, трябва да получим точно това твърдение, което ни трябва за доказателство на коректността.*

База. Когато изпълнението достигне до ред 4 за първи път е вярно, че $j = i - 1$. Следователно, текущият подмасив $A[j+2, \ldots, i]$ всъщност е $A[i+1, \ldots, i]$. Тъй като това е празен подмасив, първата част от инвариантата е изпълнена. Втората част е изпълнена, защото и двата подмасива, за които тя се отнася, се празни.

Поддръжка. Да допуснем, че твърдението е в сила в даден момент t, в който изпълнението е на ред 4 и **while** ще бъде изпълнен поне още веднъж. Последното означава, че $j > 0$ и $A[j] >$ ***key***. След ред 5 е вярно, че $A[j+1] >$ ***key*** и това е спрямо стойността на j, с която е започнала текущата итерация. Съгласно първата част на инвариантата, за всеки елемент x от текущия подмасив $A[j+2, \ldots, i]$, $x >$ ***key***. Тогава за всеки елемент x от текущия подмасив $A[j+1, \ldots, i]$, $x >$ ***key***. На ред 6 j намалява с единица. Спрямо новата стойност на j току-що направения извод звучи така: за всеки елемент x от текущия подмасив $A[j+2, \ldots, i]$, $x >$ ***key***. Доказахме първата част на инвариантата.

Да разгледаме отново изпълнението в момент t. Да разгледаме втората част на инвариантата. Според нея, текущият подмасив $A[j+2, \ldots, i]$ е същият като $A'[j+1, \ldots, i-1]$. Започва изпълнението на тялото на цикъла. На ред 5, стойнстта на текущото $A[j]$ се записва в $A[j+1]$. Но елемент $A[j]$ не е модифициран от **while** цикъла досега[†]—факт, който е напълно очевиден и няма нужда да бъде включван в инвариантата—така че $A[j]$ всъщност е $A'[j]$. Следва, че текущият $A[j+1, \ldots, i]$ е същият като $A'[j, \ldots, i-1]$ след записването на стойността на ред 5. Тогава j намалява с единица (ред 6). Когато изпълнението дойде на ред 4 отново, по отношение на новата стойност на j е вярно, че $A[j+2, \ldots, i]$ същият като $A'[j+1, \ldots, i-1]$. Така доказахме и втората част от инвариантата.

Терминация. Да разгледаме момента, в който изпълнението е на ред 4 и условието там е **Лъжа**. Тоест, $j \leqslant 0$ или $A[j] \leqslant$ ***key***.

Случай i. Да допуснем, че $j \leqslant 0$. Тъй като j намалява с единица, то не може да стане отрицателно, следователно $j = 0$. Заместваме j с 0 в инвариантата и получаваме:

[†]Помним, че сега разглеждаме само изпълненията на **while** цикъла спрямо едно изпълнение на външния **for** цикъл, а не всички изпълнения на **while** цикъла от старта на алгоритъма.

- за всеки елемент x от текущия подмасив $A[2,\ldots,i]$, $x > \textit{key}$, и освен това
- текущият подмасив $A[2,\ldots,i]$ е същият като $A'[1,\ldots,i-1]$.

Първата част на инвариантата казва, че нито едно число j от $\{1,2,\ldots,i-1\}$ не е такова, че $A'[j] \leqslant \textit{key}$. Но j има стойност 0, така че първото твърдение на лемата е в сила. Втората част на инвариантата казва, че спрямо стойността 0 на j, оригиналният подмасив $A[j+1,\ldots,i-1]$ е бил преместен с една позиция нагоре. Следователно твърденията на лемата са в сила, когато $j = 0$.

Случай ii. Сега да допуснем, че $j > 0$ и $A[j] \leqslant \textit{key}$. Но $A[1,\ldots,j]$ не е бил модифициран от **while** цикъла. Следователно, $A'[j] \leqslant \textit{key}$. Съгласно допускането в условието на лемата, $A'[1,\ldots,i-1]$ е сортиран, така че в частност $A'[1,\ldots,j]$ е сортиран и е вярно, че

$$A'[1] \leqslant \textit{key}, A'[2] \leqslant \textit{key}, \ldots, A'[j] \leqslant \textit{key} \tag{10}$$

Според инвариантата, от една страна $A[j+2] > \textit{key}$, $A[j+3] > \textit{key}$, и така нататък, $A[i] > \textit{key}$, а от друга страна, $A[j+2] = A'[j+1]$, $A[j+3] = A'[j+2]$, и така нататък, $A[i] = A'[i-1]$. Следователно,

$$A'[j+1] > \textit{key}, A'[j+2] > \textit{key}, \ldots, A'[i-1] > \textit{key} \tag{11}$$

От двата факта (10) и (11) следва, че наистина j получава стойността на най-голямото число измежду $\{1,2,\ldots,i-1\}$, такова че $A'[j] \leqslant \textit{key}$. Следователно, първото твърдение на лемата е в сила. Второто твърдение на лемата буквално същото като второто твърдение от инвариантата. С това доказателството на лемата приключва. □

Лема 2. Insertion Sort *е сортиращ алгоритъм.*

Доказателство:
Да наречем оригиналния масив $A'[1,\ldots,n]$. Следното е инвалианта за **for** цикъла:

> Всеки път, когато изпълнението на Insertion Sort е на ред 1, текущият подмасив $A[1,\ldots,i-1]$ се състои точно от същите елементи като $A'[1,\ldots,i-1]$, но в сортиран вид.

База. Първият път, когато изпълнение достигне ред 1 е вярно, че $i = 2$. Подмасивът $A[1,\ldots,1]$ се състои от един единствен елемент и очевидно е същият като $A'[1]$, така че е тривиално сортиран.

Поддръжка. Да допуснем, че твърдението е изпълнено при някое достигане на ред 1 и **for** цикълът ще бъде изпълнен още един път. Нека $\tilde{A}[1,\ldots,i]$ е името на $A[\,]$ при започването на изпълнението на **for** цикъла.

Изпълняват се редове 2, 3 и 4. С други думи, $\tilde{A}[i]$ се съхранява в $\textit{key}$, j съдържа $i-1$ и вътрешният **while** се изпълнява. Съгласно Лема 1, **while** цикълът има следните ефекти:

- j съдържа най-голямото число от $\{1,2,\ldots,i-1\}$, такова че $\tilde{A}[j] \leqslant \textit{key}$, ако има такова число, или съдържа 0, ако няма такова число.
- по отношение на тази стойност на j, подмасивът $\tilde{A}[j+1,\ldots,i-1]$ бива преместен с една позиция нагоре.

Ако има елементи $\tilde{A}[1,\ldots,i-1]$, които са по-големи от $\textit{key} = \tilde{A}[i]$, то те са в непрекъсната сортирана последователност преди започването на **while**-цикъла; това следва от допускането, което вече направихме в **Поддръжка**. Лема 1 казва, че индексът на най-малкия от тези елементи е $j+1$. Лема 1 освен това казва, че $\tilde{A}[j+1,\ldots,i-1]$ бива преместен върху текущия $A[j+2,\ldots,i]$. Тогава в текущия $A[\,]$ е вярно, че $A[j+1] = A[j+2]$, така че присвояването на ред 7 записва върху стойност, която вече е била копирана другаде. Очевидно, след изпълнението на ред 7 е вярно, че $A[1,\ldots,i]$ се състои точно от същите елементи като $\tilde{A}[1,\ldots,i]$, но в сортиран вид.

Остава да разгледаме случая, в който нито един елемент на $\tilde{A}[1,\ldots,i-1]$ не е по-голям от $\textit{key} = \tilde{A}[i]$. Съгласно Лема 1, j е равно на $i-1$ в края на **while** цикъла и нищо не е било местено нагоре в масива, така че присвояването на ред 7 записва върху i-ия елемент на A същата стойност, която той е имал в началото на изпълнението на **for** цикъла, а именно $\tilde{A}[i]$. Очевидно, в края на изпълнението на **for** цикъла, $A[1,\ldots,i]$ се състои точно от същите елементи като $\tilde{A}[1,\ldots,i]$, но в сортиран вид.

Терминация. Да разгледаме момента, в който изпълнението е на ред 1 за последен път. Очевидно i е равно на $n+1$. Заместваме стойността $n+1$ в i в инвариантата и получаваме "текущият подмасив $A[1,\ldots,(n+1)-1]$ се състои от същите елементи като $A'[1,\ldots,(n+1)-1]$, но в сортиран вид". □

5.1.2 Сложност по време на Insertion Sort

Вече показахме в (2) на стр. 3, че сложността по време в най-лошия случай на Insertion Sort е $\Theta(n^2)$.

5.1.3 Сложност по памет на Insertion Sort

Insertion Sort очевидно ползва $\Theta(1)$ допълнителна памет.

5.1.4 Стабилност на Insertion Sort

Insertion Sort е стабилен сортиращ алгоритъм. Прецизното доказателство остава за упражнение. Тук само ще спомена, че алгоритъмът е стабилен заради използването на строго неравенство $>$ на ред 4; **while** цикълът престава да се изпълнява за първата (максималната) стойност на j, за която $A[j]$ става равен на ***key*** и заради това всички двойки равни елементи остават в същия относителен ред, в какъвто са били в началото. Ако наместо строго равенство ползвахме $\geqslant$ на това място, алгоритъмът нямаше да е стабилен.

5.2 Selection Sort

Да си припомним псевдокода на SelectionSort.

```
Selection Sort(A[1,2,...n]: array of integers)
1   for i ← 1 to n − 1
2       for j ← i + 1 to n
3           if A[j] < A[i]
4               swap(A[i], A[j])
```

5.2.1 Коректност на Selection Sort

Следващите две леми обосновават коректността на Selection Sort. Това, че Selection Sort само размества елементите на входа и в края на алгоритъма елементите в масива са точно тези, които са били в началото, е очевидно: единствените промени в $A[\,]$ стават на ред 4 чрез размени. Това отличава тази реализация на Selection Sort от реализацията на Insertion Sort, която вече разгледахме. В последната не беше очевидно, че елементите в края са точно тези, които са били в началото, защото реализацията на Insertion Sort ползва не размени (swaps), а присвоявания. Поради това при анализа на Insertion Sort се наложи да обосновем защо нито един от оригиналните елементи не се "губи", бивайки презаписан с нещо друго. При анализа на Selection Sort нямаме такива притеснения.

Лема 3. *По отношение на едно изпълнение на външния* ***for*** *цикъл (редове 1–4) на* Selection Sort*, изпълнението на вътрешния* ***for*** *цикъл (редове 2–4) има следния ефект:* $A[i]$ *е най-малък елемент в* $A[i,\ldots,n]$.

Доказателство:
По отношение на едно изпълнение на външния **for** цикъл, следното твърдение е инвариантa за вътрешния **for** цикъл:

> Всеки път, когато изпълнението достигне ред 2, текущият $A[i]$ е минимален елемент в $A[i,\ldots,j-1]$.

База. Първият път, когато изпълнението на вътрешния **for** цикъл е на ред 2 е вярно, че $j = i+1$. Тогава $A[i]$ е тривиално най-малък елемент в $A[i,\ldots,(i+1)-1]$.

Поддръжка. Да допуснем, че твърдението е вярно в някой момент, когато изпълнението е на ред 2 и предстои вътрешният **for** цикъл да бъде изпълнен още веднъж. Следните два случая са изчерпателни.

Случай i. $A[j] < A[i]$. Условието на ред 3 е Истина и размяната на ред 4 се случва. Съгласно допускането, $A[i]$ е минимален елемент в $A[i,\ldots,j-1]$. Тъй като $A[j] < A[i]$, съгласно транзитивността на релацията $<$, $A[j]$ е минималният елемент в подмасива $A[i,\ldots,j]$ преди размяната. Тогава $A[i]$ е минималният елемент в $A[i,\ldots,j]$ след размяната . След размяната j нараства с единица и изпълнението отива на ред 2. По отношение на новата стойност на j е вярно, че $A[i]$ е минималният елемент в $A[i,\ldots,j-1]$.

Случай ii. $A[j] \nless A[i]$. Тогава условието на ред 3 е Лъжа и размяната на ред 4 не се случва. Съгласно предположението, $A[i]$ е минимален елемент в $A[i,\ldots,j-1]$. Тъй като $A[i] \leqslant A[j]$, очевидно $A[i]$ е минимален

елемент в $A[i, \ldots, j]$. После j нараства с единица и изпълнението отива на ред 2. По отношение на новата стойност на j е вярно, че $A[i]$ е минимален елемент в $A[i, \ldots, j-1]$.

Терминация. Да разгледаме момента, в който изпълнението е на ред 2 з последен път. Очевидно j е равно на $n+1$. Заместваме $n+1$ на мястото на j в инвариантата и получаваме "текущият $A[i]$ е минимален елемент в $A[i, \ldots, (n+1)-1]$". □

Лема 4. Selection Sort *е сортиращ алгоритъм.*

Доказателство:
На наречем оригиналния масив $A'[1, \ldots, n]$. Следното е инвариантa за външния **for** цикъл (редове 1–4):

> Всеки път, кгато изпълнението на Selection Sort е на ред 1, текущият подмасив $A[1, \ldots, i-1]$ се състои от $i-1$ на брой най-малки елементи на $A'[1, \ldots, n]$ в сортиран вид.

База. При първото достигане на ред 1 е вярно, че $i = 1$. Текущият подмасив $A[1, \ldots, i-1]$ е празен и се състои от нула на брой най-малки елементи от $A'[1, \ldots, n]$ в сортиран вид.

Поддръжка. Да допуснем, че твърдението е в сила при някакво достигане на ред 1, което не е последното. Нека наричаме масива $A[\,]$ в този момент с името $A''[\,]$. Съгласно Лема 3, ефектът на вътрешния **for** цикъл е, че поставя на $i^{\text{-тата}}$ позиция най-малка стойност от $A''[i, \ldots, n]$. От друга страна, съгласно индуктивното допускане, $A''[1, \ldots, i-1]$ се състои от $i-1$ на брой най-малки елементи на $A'[1, \ldots, n]$ в сортиран вид. Заключаваме, че в края на изпълнението на външния **for** цикъл, текущият $A[1, \ldots, i]$ се състои от i на брой най-малки елемента от $A'[1, \ldots, n]$ в сортиран вид. После i нараства с единица и изпълнението отива на ред 1. По отношение на новата стойност на i е вярно, че текущият $A[1, \ldots, i-1]$ съдържа $i-1$ на брой най-малки елемента от $A'[1, \ldots, n]$ в сортиран вид.

Терминация. Да разгледаме момента, в който изпълнението е на ред 1 за последен път. Очевидно i е равно на n. Заместваме n на мястото на i в инвариантата и получаваме "текущият подмасив $A[1, \ldots, n-1]$ се състои от $n-1$ на брой най-малки елемента на $A'[1, \ldots, n]$ в сортиран вид". Но тогава текущият $A[n]$ е максимален елемент от $A'[1, \ldots, n]$. С това доказателството за коректност на Selection Sort приключва. □

5.2.2 Сложност по време на Selection Sort

Вече показахме в (4) на стр. 3, че сложността по време в най-лошия случай на Selection Sort е $\Theta(n^2)$.

5.2.3 Сложност по памет на Selection Sort

Selection Sort очевидно ползва $\Theta(1)$ допълнителна памет.

5.2.4 Стабилност на Selection Sort

В този си вид Selection Sort не е стабилен. За да се убедим в това, да разгледаме малък подходящ пример. Нека входът е

$$2,\ 3,\ 4,\ 2,\ 1$$

За да различаваме двойките, да ги означим така:

$$2',\ 3,\ 4,\ 2'',\ 1$$

Очевидно след първото изпълнение на външния **for** цикъл единицата ще отиде на най-лява позиция, бивайки разменена с $2'$:

$$1,\ 3,\ 4,\ 2'',\ 2'$$

и крайният резултат ще е

$$1,\ 2'',\ 2',\ 3,\ 4$$

Добре известно е, че всеки нестабилен сортиращ алгоритъм може да бъде направен стабилен, ако се добави първоначалната позиция на всеки елемент като вторичен ключ (това е в случай, че поначало има само един ключ; ако поначало има k ключа, добавяме още един, $k+1$-ви, най-младши ключ с първоначалните позиции). Ако направим това обаче, влошаваме сложността по памет, понеже $\Theta(n)$ допълнителна памет ще се използва от новите ключове, използвани за стабилизиране. Ако преди това изкуствено стабилизиране сложността е била $\Theta(1)$, ще я влошим до $\Theta(n)$.

5.3 Върху доказването на коректност чрез инварианти

В заключение ще отбележим нещо важно във връзка с доказателствата чрез инварианти. Инвариантата трябва не просто да бъде вярна, а освен това да бъде полезна. Не всяко вярно твърдение, чиято вярност се запазва при изпълненията на даден цикъл, е подходящо за инварианта. Примерно, по отношение на **Insertion Sort**, да разгледаме следната инварианта като евентуално средство за доказване на Лема 2:

> Всеки път, когато изпълнението на Insertion Sort е на ред 1, текущият подмасив $A[1,\ldots,i-1]$ е сортиран.

Това твърдение е истина и е инварианта на външния цикъл, защото верността му се запазва от кое да е изпълнение на следващото. И въпреки това, тази инварианта **не доказва Лема 2**. Поради простата причина, че сортиращият алгоритъм трябва не просто да създава сортиран изход, а такъв сортиран изход, който се състои **точно от оригиналните елементи**. Инвариантата, която написахме току-що, никъде не казва нищо за това дали подмасивът $A[1,\ldots,i-1]$ се състои от елементи на входа или не. За да стане съвсем ясно защо тази инварианта е безполезна, да разгледаме следния безсмислен алгоритъм.

No Sort($A[1,2,\ldots n]$: масив от цели числа)
1 **for** $i \leftarrow 1$ **to** n
2 $\quad A[i] \leftarrow i$

Без съмнение, съдържанието на $A[\,]$ след привършването му е в сортиран вид, понеже очевидно $A[i] = i$ за $1 \leqslant i \leqslant n$. Тъй като оригиналният масив бива унищожен, това не може да е сортиращ алгоритъм. Но забележете, че последната инварианта е истина за No Sort! Наистина подмасивът $A[1,\ldots,i-1]$ е сортиран при всяко достигане на цикъла[†].

Извод: не всяка вярна инварианта е полезна за това, което искаме да докажем.

Литература

[CLR09] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms, Third Edition.* The MIT Press, 3rd edition, 2009.

[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.

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

[†]За да докажем безполезната, но вярна инварианта ни трябва едно помощно твърдение, а именно че $A[i-1]$ е равно на $i-1$. Това е без значение, понеже тази инварианта е безполезна така или иначе.