
Задача 1: Даден е масив от цели числа $A[1\,..\,n]$, където $n \geq 2$. *Мажорант* на масива е число, такова че повече от половината елементи на масива са равни на него. Конструирайте итеративен алгоритъм с линейна сложност по време, чийто вход е масивът A, а изходът е или мажорант на A, ако такъв съществува, или съобщение, че A няма мажорант, в противен случай. Обосновете **много подробно и прецизно** коректността на Вашия алгоритъм с инвариант на цикъла.

Решение: Следният алгоритъм решава задачата.

```
Boyer-Moore(A[1 .. n]: int, n ≥ 2)
 1  c ← 0, i ← 1
 2  do
 3      if c = 0
 4          m ← A[i]
 5          c ← 1
 6      else
 7          if m = A[i]
 8              c++
 9          else
10              c--
11      i++
12  while i ≤ n
13  c ← 0
14  for k ← 1 to n
15      if A[k] = m
16          c++
17  if c ≥ ⌊n/2⌋ + 1
18      return m
19  else
20      return няма мажорант
```

Преди доказателство за коректност ще направим няколко дефиниции. Нека $S[1\,..\,n]$ е целочислен масив и $S[p\,..\,q]$ е негов непразен подмасив. Казваме, че $S[p]$ е *лидерът* на $S[p\,..\,q]$. *Дисбалансът* в $S[p\,..\,q]$ е следната разлика: броят на срещанията на лидера минус броя на останалите елементи на $S[p\,..\,q]$. Казваме, че $S[p\,..\,q]$ е *регулярен*, ако неговият дисбаланс е неотрицателен, и че $S[p\,..\,q]$ е *балансиран*, ако неговият дисбаланс е нула. Примерно, ако $S = [2, 5, 1, 3, 6, 1, 2, 2, 3]$, то $S[1\,..\,3] = [2, 5, 1]$ не е регулярен, $S[7\,..\,9] = [2, 2, 3]$ е регулярен, но не е балансиран, $S[1\,..\,1] = [2]$ също е регулярен, но не балансиран, а $S[3\,..\,6] = [1, 3, 6, 1]$ е балансиран.

Мажорантното разбиване на S ще наричаме редицата $\mathrm{maj}(S)$ от подмасиви на S:

$$\mathrm{maj}(S) \stackrel{\text{def}}{=} \big(S[1\,..\,i_1], S[i_1+1\,..\,i_2], S[i_2+1\,..\,i_3], \ldots, S[i_{t-1}+1\,..\,i_t], S[i_t+1\,..\,n]\big)$$

където $i_1, \ldots, i_t$ са уникалните индекси, такива че $1 < i_1 < i_2 < \cdots < i_{t-1} < i_t < n$, като $t \geq 0$ и освен това

- $S[1 \,..\, i_1]$ е най-късият балансиран подмасив с лидер $S[1]$,
- $S[i_1 + 1 \,..\, i_2]$ е най-късият балансиран подмасив с лидер $S[i_1 + 1]$,
- $S[i_2 + 1 \,..\, i_3]$ е най-късият балансиран подмасив с лидер $S[i_2 + 1]$,
- $\ldots$
- $S[i_{t-1} + 1 \,..\, i_t]$ е най-късият балансиран подмасив с лидер $S[i_{t-1} + 1]$,
- $S[i_t + 1 \,..\, n]$ е регулярен. Освен това, или $S[i_t + 1 \,..\, n]$ е най-късият балансиран подмасив с лидер $S[i_t + 1]$, или $S[i_t + 1]$ не е лидер балансиран подмасив.

Примерно, ако $S = [2, 5, 1, 3, 6, 1, 2, 2, 3]$, то $\mathrm{maj}(S) = \big([2, 5], [1, 3], [6, 1], [2, 2, 3]\big)$, като $[2, 5]$, $[1, 3]$ и $[6, 1]$ са балансирани, а $[2, 2, 3]$ не е балансиран. Като друг пример, ако $S' = [2, 2, 5, 1, 3, 3, 6, 3, 1, 1, 2, 2, 3, 4]$, то $\mathrm{maj}(S') = \big([2, 2, 5, 1], [3, 3, 6, 3, 1, 1], [2, 2, 3, 4]\big)$, като и трите подмасива са балансирани. Като трети пример, ако $S'' = [2, 2, 2, 3, 4]$, то $\mathrm{maj}(S'') = \big([2, 2, 2, 3, 4]\big)$, като масивът не е балансиран; в този случай $t = 0$ и $i_t = 0$.

Наблюдение 1 *В контекста на горните дефиниции, нека $A[p \,..\, q]$ е елемент на $maj(S)$. Тогава за всяко $\ell \in \{p, \ldots, q\}$, подмасивът $A[p \,..\, \ell]$ е регулярен.*

Лема 1 *При последното достигане на ред 12, променливата m съдържа лидера на последния масив от $maj(S)$.*

Доказателство. Следното твърдение е инвариант на **do-while** цикъла на редове 2–12. По отношение на произволно достигане на ред 12 дефинираме, че *текущият подмасив* е този елемент на $\mathrm{maj}(A)$, който съдържа $A[i - 1]$.

> *За всяко достигане на ред 12, ако текущият подмасив е $A[p \,..\, q]$, то m съдържа лидера му, а c съдържа дисбаланса на $A[p \,..\, i - 1]$.*

Доказателство.

База. В началото на алгоритъма, c получава стойност 0 и i получава стойност 1 на ред 1. Условието на ред 3 е истина, така че m получава стойност $A[1]$ на ред 4, а c получава стойност 1 на ред 5. После i се инкрементира на ред 11 и става 2. Разглеждаме първото достигане на ред 12. Тъй като $i = 2$, текущият подмасив $A[p \,..\, q]$ е този елемент на $\mathrm{maj}(S)$, който съдържа $A[1]$. Тогава $A[p \,..\, q]$ е $A[1 \,..\, q]$, за някакво q. Подмасивът $A[p \,..\, i - 1]$, за който говори инварианта, е $A[1 \,..\, 1]$.

От една страна, лидерът на текущия подмасив е $A[1]$. От друга страна, $m = A[1]$. От една страна, дисбалансът на $A[p \,..\, i-1]$ е 1. От друга страна, $c = 1$. Заключаваме, че инвариантът е в сила при първото достигане на ред 12. ✓

Поддръжка. Разглеждаме някое достигане на ред 12, което не е последното. Нека t е моментът от дискретното време на алгоритъма, в който става това достигане.

Нека t' е моментът на следващото достигане на ред 12; такъв момент очевидно съществува. Допускаме, че инвариантът е в сила в момента t. Съгласно Наблюдение 1 и допускането, c се явява дисбаланс на регулярен масив, така че $c \geq 0$. Тогава следните случаи са взаимно изключващи се и изчерпателни.

- $c = 0$. От допускането следва, че $A[p \,..\, i-1]$ е балансиран, от което следва, че $A[p \,..\, i-1]$ е масив-елемент на $\mathrm{maj}(S)$, от което пък следва, че $A[i]$ е първият елемент от следващия масив от $\mathrm{maj}(S)$. Нека този следващ масив е $A[p' \,..\, q']$, където $p' = i$.

  Разглеждаме изпълнението на тялото на **do–while** цикъла от момента t до момента t'. Условието на ред 3 е истина, така че на ред 4, m получава стойност $A[i]$, който е лидерът на $A[p' \,..\, q']$, а на ред 5, c получава стойност 1, която е дисбалансът на масива $A[p' \,..\, i]$. След това i се инкрементира на ред 11. Спрямо новото i е вярно, че c съдържа дисбаланса на $A[p' \,..\, i-1]$.

  Изпълнението отива на ред 12. Това е моментът t'. Ясно е, че текущият подмасив в този момент е $A[p' \,..\, q']$. Заключаваме, че в момента t', m съдържа лидера на текущия подмасив $A[p' \,..\, q']$, а c съдържа дисбаланса на $A[p' \,..\, i-1]$. Инвариантът е в сила. ✓

- $c > 0$ и $A[i]$ е равен на лидера на текущия подмасив. Щом $c > 0$, от допускането следва, че $A[p \,..\, i-1]$ не е масив-елемент на $\mathrm{maj}(S)$, от което пък следва, че $A[i]$ също е елемент от $A[p \,..\, q]$.

  Разглеждаме изпълнението на тялото на **do–while** цикъла от момента t до момента t'. Условието на ред 3 е лъжа, така че изпълнението отива на ред 6. Условието на ред 7 е истина, защото по допускане, m съдържа лидера на $A[p \,..\, q]$. На ред 8, c се инкрементира. Предвид допускането, c вече съдържа дисбаланса на $A[p \,..\, i]$. На ред 11, i се инкрементира. Спрямо новото i е вярно, че c съдържа дисбаланса на $A[p \,..\, i-1]$.

  Изпълнението отива на ред 12. Това е моментът t'. Ясно е, че текущият подмасив в този момент е пак $A[p \,..\, q]$ и m, която не беше променяна от момента t до момента t', съдържа лидера на текущия подмасив, а c съдържа дисбаланса на $A[p' \,..\, i-1]$. ✓

- $c > 0$ и $A[i]$ не е равен на лидера на текущия подмасив. Щом $c > 0$, от допускането следва, че $A[p \,..\, i-1]$ не е масив-елемент на $\mathrm{maj}(S)$, от което пък следва, че $A[i]$ също е елемент от $A[p \,..\, q]$.

  Разглеждаме изпълнението на тялото на **do–while** цикъла от момента t до момента t'. Условието на ред 3 е лъжа, така че изпълнението отива на ред 6. Условието на ред 7 е лъжа, защото по допускане, m съдържа лидера на $A[p \,..\, q]$. На ред 10, c се декрементира. Предвид допускането, c вече съдържа дисбаланса на $A[p \,..\, i]$. На ред 11, i се инкрементира. Спрямо новото i е вярно, че c съдържа дисбаланса на $A[p \,..\, i-1]$.

  Изпълнението отива на ред 12. Това е моментът t'. Ясно е, че текущият подмасив в този момент е пак $A[p \,..\, q]$ и m, която не беше променяна от момента t до момента t', съдържа лидера на текущия подмасив, а c съдържа дисбаланса на $A[p' \,..\, i-1]$. ✓

Терминация. Разглеждаме последното достигане на ред 12, което става в някакъв момент $\tilde{t}$. В този момент очевидно $i = n + 1$. Тогава текущият подмасив е този елемент на $\mathrm{maj}(S)$, който съдържа $A[n]$. Но това очевидно е последният масив от $\mathrm{maj}(S)$. От инварианта знаем, че m съдържа лидера му. Следователно, в момента $\tilde{t}$, променливата m съдържа лидера на последния масив от $\mathrm{maj}(S)$. □

Лема 2 *Разглеждаме входния масив A на* Boyer-Moore. *Нека x е лидерът на последния масив от $maj(A)$. За всеки y от A, такъв че $y \neq x$, е вярно, че y не е мажорант на A.*

Доказателство. Ключовото наблюдение е, че нито един масив от $\mathrm{maj}(A)$, който не е последният, сам по себе си няма мажорант, защото е балансиран; балансиран масив очевидно не може да има мажорант.

Нека "конкатенацията" на масивите от $\mathrm{maj}(A)$ без последния се нарича B. Нека последният масив от $\mathrm{maj}(A)$ се нарича D. Нека $b = |B|$ и $d = |D|$. Забележете, че b е четно, понеже B е конкатенация от балансирани масиви, а всеки балансиран масив има четна големина.

Разглеждаме произволен елемент y от A, такъв че $y \neq x$. Но y се среща най-много $\frac{b}{2}$ пъти в B, защото B е конкатенация от балансирани масиви, и y се среща най-много $\left\lfloor \frac{d}{2} \right\rfloor$ пъти в D, защото D е регулярен и лидерът му е x, а $x \neq y$ по дефиниция. Заключаваме, че y се среща не повече от $\left\lfloor \frac{n}{2} \right\rfloor$ в A. За да бъде мажорант на A, елемент трябва да се среща поне $\left\lfloor \frac{n}{2} \right\rfloor + 1$ в A. □

Лема 3 *Променливата c на ред 17 съдържа броя на появите на m в A.*

Доказателство. Следното твърдение е инвариант за **for**-цикъла на редове 14–16.

> *За всяко достигане на ред 14, c съдържа броя на срещанията на m в $A[1 \,..\, k-1]$.*

База. $k = 1$, $c = 0$ при първото достигане. Наистина, m се среща 0 пъти в празния подмасив $A[1 \,..\, 0]$.

Поддръжка. Разглеждаме някое достигане на ред 14, което не е последното. Допускаме, че твърдението е вярно. Разглеждаме два случая.

- $A[k] = m$. Тогава, от една страна, m се среща $c + 1$ пъти в $A[1 \,..\, k]$. От друга страна, условието на ред 15 е истина, c се инкрементира и спрямо новото c е вярно, че m се среща c пъти в $A[1 \,..\, k]$. След инкрементирането на k на ред 14, спрямо новото k, пак е вярно, че m се среща c пъти в $A[1 \,..\, k-1]$.

- $A[k] \neq m$. Тогава, от една страна, m се среща c пъти в $A[1 \,..\, k]$. От друга страна, условието на ред 15 е лъжа, така че c не се инкрементира и остава вярно, че m се среща c пъти в $A[1 \,..\, k]$. След инкрементирането на k на ред 14, спрямо новото k, пак е вярно, че m се среща c пъти в $A[1 \,..\, k-1]$.

Терминация. При последното достигане на ред 14, $k = n + 1$. Заместваме в инварианта и получаваме "c съдържа броя на срещанията на m в $A[1 \,..\, n]$". □

Теорема 1 BOYER-MOORE($A[1 \,..\, n]$) *връща или стойността на мажорант на A, ако такъв съществува, или съобщение, че няма мажорант, в противен случай.*

Доказателство. От Лема 1 знаем, че на ред 12, променливата m съдържа лидера на последния масив от $\mathrm{maj}(S)$. От Лема 2, че ако A има мажорант, това е лидерът на последния масив от $\mathrm{maj}(S)$. Това означава, че когато изпълнението е на ред 14, единственият възможен мажорант на A е m. От Лема 3 знаем, че на ред 17, променливата c съдържа броя на появите на m в A. Заключаваме, че ако c на ред 17 е поне $\left\lfloor \frac{n}{2} \right\rfloor + 1$, то A има мажорант и това е m, а в противен случай A няма мажорант. От друга страна, ако на ред 17, $c \geq \left\lfloor \frac{n}{2} \right\rfloor + 1$, алгоритъмът връща m, в противен случай връща съобщение, че няма мажорант. □

Задача 2: Професор Дълбоков предлага следния сортиращ алгоритъм, написан на C. Функцията `swap` е помощна функция, която извършва точно това, което името ѝ казва, а именно `swap(array, i, j)` разменя елементи `i` и `j` на масива `array`.

```
1  int Dsort (int *array, int i) {
2    int j;
3
4    if (i == N-1) {
5      for (j = 1; j < N; j ++) {
6        if (array[j-1] > array[j]) {
7          return 0;
8        }
9      }
10     return 1;
11   }
12   else {
13     for (j = i; j < N; j ++) {
14       swap(array, i, j);
15       if (Dsort(array, i+1)) {
16         return 1;
17       }
18       else {
19         swap(array, i, j);
20       }
21     }
22     return 0;
23   }
24 }
```

foo1.c

Началното викане е `Dsort(array, 0);`, като `array` е масив от `N` цели числа, а `N` е глобална променлива.

30 т. • Докажете коректността на алгоритъма.

10 т. • Изследвайте сложността на алгоритъма по време.

Решение: Идеята на алгоритъма би трябвало да е очевидна: той генерира систематично пермутациите на входния масив и за всяка генерирана пермутация проверява чрез `for`-цикъла на редове 5–8 дали тя е сортирана; ако се установи, че е сортирана, връща `1` на ред 10 от последното рекурсивно викане, което води до връщане на `1` през всички рекурсивни викания чак до първоначалното рекурсивно викане. Това е и краят на изпълнението. Тъй като поне една пермутация на входните елементи е сортирана (може и да са повече, защото може да има повторения на елементи), алгоритъмът терминира. Сложността на алгоритъма е ужасна: в най-лошия случай, чак последната генерирана пермутация се оказва сортираната, така че сложността е $\Omega(N!)$.

Генерирането на пермутациите става чрез суопове, които при пермутациите се наричат *транспозиции*. Фактът, че от дадена пермутация на дадени елементи може да се генерира всяка пермутация на тези елементи чрез серия от транспозиции е добре известен от Дискретната математика. В случая с `Dsort`, аргументът `i` е индекс в масива; ако $0 \leq$ `i` $\leq$ `N-2`, изпълнява се `for`-цикълът на ред 13, в който се прави всяка възможна транспозиция (ред 14) на елемент `i` със следващ елемент, последвана от рекурсивно викане на ред 15 с втори аргумент `i+1` и възстановяване от транспозицията на ред 16. Управляващият параметър на рекурсията е разликата `N-1` $-$ `i`; тстк тя стане `0`, условието на ред 4 става истина и се прави проверка за сортираност.

Сега ще направим по-формално доказателство за коректност. Да разгледаме програмата `genperm.c`.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int N;
5
6  void printit (int *A) {
7    int i;
8
9    for (i = 0; i < N; i ++) {
10     printf(" %d", A[i]);
11   }
12
13   printf("\n");
14 }
15
16 void swap (int *A, int i, int j) {
17   int tmp;
18   tmp = A[i];
19   A[i] = A[j];
20   A[j] = tmp;
21 }
22
23 void genperm (int *A, int i) {
24   int j;
25
26   if (i == N-1) {
27     printit(A);
28   }
29   else {
30     for (j = i; j < N; j ++) {
31       swap(A, i, j);
32       genperm(A, i+1);
33       swap(A, i, j);
34     }
35   }
```

```
}

int main (int argc, char* argv[]) {
  int i;
  int A[argc - 1];

  N = argc - 1;

  for (i = 1; i < argc; i ++) {
    A[i-1] = atoi(argv[i]);
  }

  genperm(A, 0);

  return 0;
}
```

genperm.c

Тя наистина генерира всички пермутации на входа, примерно `./genperm 3 5 7 2` връща

```
3 5 7 2
3 5 2 7
3 7 5 2
3 7 2 5
3 2 7 5
3 2 5 7
5 3 7 2
5 3 2 7
5 7 3 2
5 7 2 3
5 2 7 3
5 2 3 7
7 5 3 2
7 5 2 3
7 3 5 2
7 3 2 5
7 2 3 5
7 2 5 3
2 5 7 3
2 5 3 7
2 7 5 3
2 7 3 5
2 3 7 5
2 3 5 7
```

Тази програма е почти същата като `Dsort`, с тази разлика, че `genperm` не прави проверка за сортираност и съответно не прекратява изпълнението си при установяване

на сортираност, а винаги генерира всички пермутации.

Лема 4 *Функцията* `genperm` *с вход* (A, k) *връща чрез* `printit` *точно тези пермутации на* A*, в които подмасивът* $A[0\,..\,k-1]$ *е фиксиран; тоест, същият като във входа.*

"Връща чрез `printit` всички пермутации" означава, че за всяка от тези пермутации има съответно изпълнение на ред 27 и, обратно, при всяко достигане на ред 27, точно една, различна от останалите, пермутация от въпросните е съдържанието на масива.

Доказателство. Това е доказателство по индукция, но не индукция по N, а индукция по $i = N-1, N-2, \ldots, 0$, по отношение на едно **фиксирано** N.

Базата е $i = N-1$. От една страна, има точно една пермутация, в която подмасивът $A[0\,..\,N-2]$ е фиксиран, а именно самият $A[0\,..\,N-1]$ (щом всички елементи без един са фиксирани, за последния елемент има само един избор). От друга страна, в `genperm`$(A, N-1)$, условието на ред 26 е истина и програмата принтира еднократно $A[0\,..\,N-1]$ на ред 27. ✓

Индуктивното допускане е, че твърдението е в сила за някое $i+1$, такова че $i+1 > 0$. Ограничението $i+1 > 0$ се налага, защото доказваме върху **крайно** наредено множество.

В индуктивната стъпка разглеждаме `genperm`(A, i). Ясно е, че $i \geq 0$, така че i е валиден индекс.

От една страна, за всяка π пермутация на входния A (входния масив във викането `genperm`(A, i)), в която подмасивът $A[0\,..\,i-1]$ е фиксиран, е вярно, че π започва с $A[0\,..\,i-1]$, следва елемент $A[j]$ от $A[i\,..\,N-1]$ и след това има пермутация на елементите на входния $A[i\,..\,N-1]$ без $A[j]$.

От друга страна, условието на ред 26 е лъжа и изпълнението отива на ред 29, като във `for`-цикъла на редове 30–34, програмата последователно разменя $A[i]$ с $A[j]$, вика себе си с `genperm`$(A, i+1)$ и възстановява масива чрез повторна размяна на $A[i]$ с $A[j]$, за $i \leq j \leq N-1$. Използваме индуктивното допускане, че викането на ред 32 принтира всички пермутации, в които $A[0\,..\,i]$ е фиксиран, където обаче i-ият елемент не е непременно i-ият елемент от викането `genperm`(A, i), а е резултат от някоя транспозиция, извършена на ред 31. □

Теорема 2 *Началното викане* `genperm`$(A, 0)$ *на ред 48 принтира всички пермутации на* A*.*

Стриктно говорейки, това не е точно така: ако има повторения на елементи, програмата не отчита това, така че може да принтира многократно едно и също нещо. Примерно, при вход `1 1 1 1` тя принтира 24 пъти `1 1 1 1`. Вярното твърдение е, че се принтират всички пермутации на клетки на масива, без оглед на съдържанието им.

Доказателство. Следва тривиално от Лема 4, полагайки $k = 0$. □

Следствие 1 `Dsort(array, 0);` *сортира входния масив* `array`*.*

Доказателство. Следва тривиално от Теорема 2, тъй като `Dsort` прави същото генериране на пермутациите на входа, но без да ги принтира, и освен това терминира, когато установи сортираност чрез `for`-цикъла на редове 5–8. Тъй като поне една пермутация на входните елементи е сортирана, алгоритъмът терминира със сортиран масив. □

Сложността по време в най-лошия случай **не може** да се опише адекватно от рекурентно уравнение като тези, които сме разглеждали на лекции.

Примерно, да разгледаме рекурентното уравнение $T(N) = NT(N-1)$. "Обосновката" е, че при вход с големина N се правят N рекурсивни викания върху входове с размер $N-1$. Наистина е така, но въпреки това, рекурентното уравнение не е адекватно! То предполага неявно начално условие $T(1) = \Theta(1)$, докато при `Dsort`, дъното на рекурсията извършва $\Theta(N)$ работа, проверявайки целия масив за сортираност.

Можем да намерим сложността в най-лошия случай чрез дървото на рекурсията. Коренът има N деца, всяко от тях има $N-1$ деца, което означава $N(N-1)$ върхове на ниво две, всеки от тези върхове има $N-2$ деца, което означава $N(N-1)(N-2)$ върхове на ниво три, и така нататък. Дървото е съвършено в смисъл, че всички листа са на едно и също разстояние $N-1$ от корена. Листата са $N!$ на брой и във всяко от тях се върши $\Theta(N)$ работа, което означава общо $\Theta(N \cdot N!)$ работа в листата. Вътрешните върхове са $O(N!)$ и във всеки от тях се върши $\Theta(1)$ работа, което дава общо $O(N!)$ работа във вътрешните върхове. Като цяло, сложността се описва от

$$O(N!) + \Theta(N \cdot N!) = \Theta(N \cdot N!)$$

Една дребна забележка: в най-добрия случай, този алгоритъм е линеен. Ако входът е сортиран, още след първата проверка за сортираност, алгоритъмът терминира.

Задача 3: Нека $f(n) : \mathbb{R}^+ \to \mathbb{R}^+$ и $g(n) : \mathbb{R}^+ \to \mathbb{R}^+$. Известно е, че $f(n)$ и $g(n)$ са монотонни растящи. Известно е, че съществува естествено $d \geq 2$, такова че за всяко цяло положително k, ако $n = d^k$, то $f(n) \leq g(n)$. Известно е, че $g(n) = O\left(g\left(\frac{n}{d}\right)\right)$. Докажете, че $f(n) = O(g(n))$.

Решение: Дадени са три ограничения за f и g. Първото е, че са монотонно растящи. Функция $\phi(n)$ е монотонно растяща тстк

$$n_1 \leq n_2 \ \to \phi(n_1) \leq \phi(n_2)$$

Второто ограничение казва следното. Разглеждаме числата, които са точни степени на d. Ограничението казва, че

$$\begin{aligned} f(d) &\leq g(d) \\ f(d^2) &\leq g(d^2) \\ f(d^3) &\leq g(d^3) \\ f(d^4) &\leq g(d^4) \\ &\dots \end{aligned}$$

Третото ограничение казва, че съществува положителна константа c, такава че за всички достатъчно големи n е изпълнено

$$g(n) \leq c \cdot g\left(\frac{n}{d}\right)$$

В частност, ако $n = d^{k+1}$, то

$$g(d^{k+1}) \leq c \cdot g\left(d^k\right)$$

Лесно се вижда, че всъщност $c \geq 1$ заради монотонното нарастване на g; ако допуснем, примерно, $c = \frac{1}{2}$, би следвало

$$g(d^2) \leq \frac{1}{2} g(d) \ \leftrightarrow \ g(d) \geq 2g(d^2)$$

което е несъвместимо с монотонното нарастване на g и факта, че $d < d^2$.

Иска се да докажем, че съществува положителна константа c_1 и положителна стойност n_0 на аргумента, такива че за $n \geq n_0$ е изпълнено

$$f(n) \leq c_1 \cdot g(n) \tag{1}$$

Разглеждаме два случая.

- n е точна степен на d. По условие имаме $f(n) \leq 1 \cdot g(n)$.
- n не е точна степен на d. Очевидно съществува цяло положително k, такова че

  $$d^k < n < d^{k+1}$$

Тъй като f е монотонно растяща, в сила е

$$f(n) \leq f(d^{k+1})$$

Но $f(d^{k+1}) \leq g(d^{k+1})$ по условие. Тогава

$$f(n) \leq g(d^{k+1})$$

Както вече знаем, $g(d^{k+1}) \leq c \cdot g(d^k)$. Тогава

$$f(n) \leq c \cdot g(d^k)$$

Тъй като g е монотонно растяща, в сила е

$$g(d^k) \leq g(n)$$

Заключаваме, че

$$f(n) \leq c \cdot g(n)$$

С което доказахме (1). Избираме $n_0 = d$ и $c_1 = c$. Тъй като $c \geq 1$, изборът $c_1 = c$ "покрива" и двата случая, които разгледахме поотделно.