ДАА Семинар 9
Алгоритми върху масиви

// За задачата уникалност на елементи
↳ може да се провери в хода на изпълнение на merge
↳ В изходния (сортирания от MergeSort) масив, ако има еднакви елементи, то те са съседни, а merge в хода на работата си е сравнил всяка двойка съседни елементи в изходния масив ↳ (// в целия ход на работата си при извикване на MergeSort)

↓ преработен merge

```
merge(A, low, mid, high)
1.  left ← low
2.  right ← mid + 1
3.  B - dynArr  // може и B[high-low+1]
4.  while left ≤ mid && right ≤ high
5.          if A[left] < A[right]
6.
7.              B.push-back(A[left])
8.              left++
9.          else if A[left] == A[right]
10.                 return false
11.         else
12.             B.push-back(A[right])
13.             right+++
14. while left≤mid
15.         B.push-back(A[left])
16.         left++
17. while right≤high
18.         B.push-back(A[right])
19.         right++
20. A[low, high] ← B // θ(high-low)
21. return true
```

→ масивите A[low, mid] и A[mid+1, high] са сортирани при предишни викания на merge по време на MergeSort и в тях няма не уникални елементи, в противен случай при предишните изпълнения на merge над A[low, ..., mid] и A[mid+1, ..., high] ще е терминирал алгоритъмът с false.

Зад ① Даден е масив $A[1..n]$. $\forall i \in \{1,..,n\}: A[i] \in \{$бяло, зелено, червено$\}$.
Има поне един елемент от всеки цвят. Допустимите операции върху A са

getColor(i) - $\forall i \in \{1,..,n\}$ връща цвета на елемента $A[i]$

swap(i,j) - $\forall i,j \in \{1,..,n\}$ - разменя местата на $A[i]$ и $A[j]$ в масива

Да се състави алг. с колкото се може по-ниска сложност по време и памет, която пренарежда масива по такъв начин, че в края на алгоритъма да е изпълнено: $\exists k,l \in \{1,..,n\}$: $A[1..k]$ съдържа само бели елементи
$A[k+1..l]$ съдържа само зелени елементи
$A[l+1..n]$ съдържа само червени елементи.

Решение:

Изходът трябва да изглежда така:

б	з	ч

Знаем, че алгоритъмът partition може да разбие масива на две части, такава, че елементите в първата част са единствено бели или зелени, а елементите от втората - червени. Ще получим:

б/з	ч

Може да приложим същия алгоритъм още веднъж върху първата част, така че след приключване белите да са преди зелените:

б	з

Ще използваме модифициран partition-hoare

```
MyPartition(A[1..n], low, high, x)
1.  left ← low-1
2.  right ← high+1
3.  while True.                          → T(n) ≥ n
4.      do                                 M(n) ≥ 1
5.        ++left
6.      while A[left] != x
7.      do
8.        --right
9.      while A[right] = x
10.     if left<right
11.       swap(A[left], A[right])
12.     else
13.       return right
```

Transform(A[1...n])

1. index1 ← MyPartition(A, 1, n, "червено") // A[index1+1...n] = ["червено", ..., "червено"]
2. index2 ← MyPartition(A, 1, index1, "зелено") // A[1...index2] = ["бяло", ..., "бяло"]
   A[index2+1...index1] = ["зелено", ..., "зелено"]

↳ подаваме елемента (pivot) за сравнение отделно

// пример:

A[1...10] = ["г", "б", "з", "б", "г", "б", "з", "з", "г", "г"]

След изпълнение на ред 1 A има вида:

A[1...10] = ["з", "б", "з", "б", "з", "б", "г", "г", "г", "г"] , а index1 = 6

След изпълнение на ред 2 A има вида:

A[1...10] = ["б", "б", "б", "з", "з", "з", "г", "г", "г", "г"] , а index2 = 3

зад ② Дадена е купчина от палачинки с различни размери. Трябват да се сортират според големината им, в намаляващ ред от най-голямата към най-малката. С шпатула могат да се обръщат наведнъж избран брой палачинки от горната част на стека/купчината. Освен да сортира, алгоритъмът трябва да извежда последователните позиции, в които обръщаме с шпатулата. Палачинките са представени чрез диаметъра си в масив

// Пример

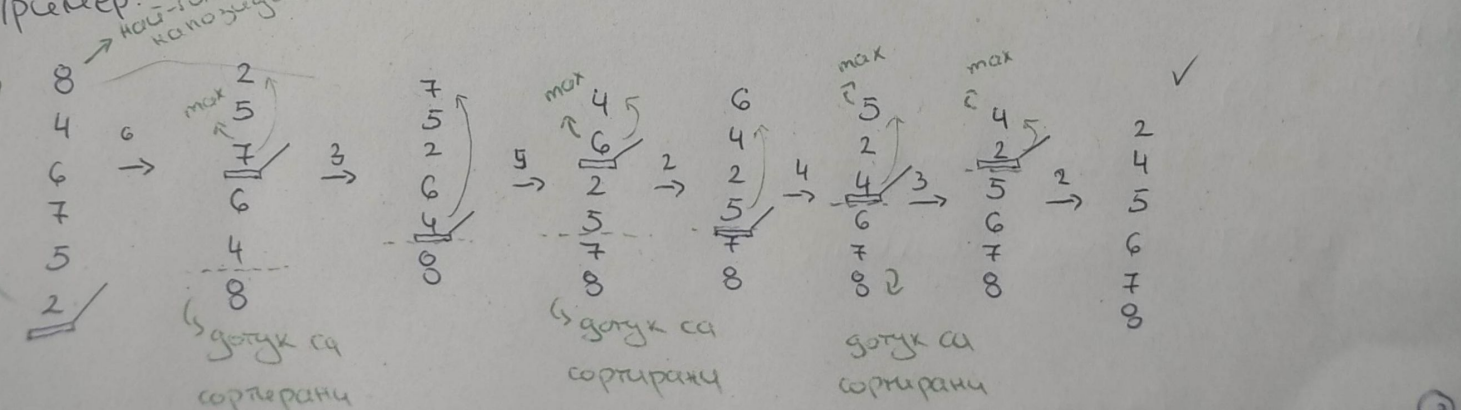
[5, 4, 3, 2, 1] (1 2 3 4 5) ⟶ с едно обръщане в позиция 5 → [1, 2, 3, 4, 5]

[5, 1, 2, 3, 4] (1 2 3 4 5) ⟶ pos: 5 → [4, 3, 2, 1, 5] ⟶ pos: 4 → [1, 2, 3, 4, 5]

Решение:

Ще сортираме купчината/масива като последователно поставяме текущия най-голям елемент на текущата последна позиция - всяко такова действие не струва най-много две операции: намиране на индекса на този най-голям елемент и с едно обръщане го "поставяме" най-отгоре на стека. След това с още едно обръщане от текущата последна позиция го поставяме на правилното му място.

// Пример:



```
SortPancakes(A[1...n])
1. end ← n // n-end са броят сортирани най-големи
2. While end > 1
3.     index ← findMaxIndex(A, end)
4.     if index != 1
5.         flip/reverse(A, index)  ← cout<<index<<" " (в if-а)
6.
7.     reverse(A, end)
8.     end ← end - 1               ← cout<<end<<" "
9.     }
```

зад ③ Да се преработи QuickSort така че да сортира елементите в ненарастващ ред.

Решение:

Идеята на QuickSort е следната:

1. Избери pivot и го сложи на правилното му място в сортирания масив
2. Постави по-малките вляво от pivot, а по-големите вдясно от pivot
3. Рекурсивно приложи същата процедура за лявата и дясната част

// Пример:

[4, 6, 2, 5, 7, 9, 1, 3] - текущо извикване

* 1. Pivot = 4 // За pivot ще избираме най-левия елемент

2. [2, 1, 3, 4, 6, 5, 7, 9] // pivot е на мястото си

(по-малки) (по-големи)

[2, 1, 3] - текущо извикване

(Δ) 1. Pivot = 2

2. [1, 2, 3]

[1] - текущо извикване

едноелементният масив е тривиално сортиран

[3] - текущо извикване

" — "

От * и Δ получаваме [1, 2, 3, 4, 6, 5, 7, 9]

[6, 5, 7, 9] - текущо извикване

1. pivot = 6

2. [5, 6, 7, 9]

[5] - текущо извикване

" — "

[7, 9] - текущо извикване

1. pivot = 7

2. [7, 9]

Получихме [1, 2, 3, 4, 5, 6, 7, 9] ✓

Стъпка 2 „Постави по-малките вляво от pivot, а по-големите вдясно" се осъществява чрез partition:

[4, 6, 2, 5, 7, 9, 1, 3]

low ↑ i ... ↑ j high

pivot = a[low] = 4

(*) местим i надясно докато A[i] <= pivot
местим j наляво докато A[j] >= pivot

(□) След това swap(A[i], A[j]) ако i<j

(*) [4, [6], 2, 5, 7, 9, 1, [3]] (i под 6, j под 3) → (□) [4, 3, 2, 5, 7, 9, 1, 6]

(*) [4, 3, 2, [5], 7, 9, [1], 6] (i под 5, j под 1) → (□) [4, 3, 2, 1, 7, 9, 5, 6]

[4, 3, 2, 1, 7, 9, 5, 6] (j под 1, i под 7) (i и j се разминаха) → тогава swap(pivot, A[j])

[1, 3, 2, 4, 7, 9, 5, 6] (j ↓ над 4)

След това трябва да се сортират [1, 3, 2] = A[low, j-1] и
[7, 9, 5, 6] = A[j+1, high]

```
QS(A, low, high)
if low < high
    pivotIndex ← partition(A, low, high)
    QS(A, low, pivotIndex-1)
    QS(A, pivotIndex+1, high)
```

Ако искаме QS да сортира в ненарастващ ред, то трябва да преработим partition, така че да поставя по-големите вляво от pivot, а по-малките вдясно

```
partition(A, low, high)
pivot ← A[low], i ← low, j ← high
while i<j
    while A[i] >= pivot && i <= high
        i ← i+1
    while A[j] < pivot && j >= low
        j ← j-1
    if i<j
        swap(A[i], A[j])
swap(A[j], A[low])
return j
```

Зад ④ Фирмен отсек се състои от списък, съдържащ общото работно време – h_b за всеки служител b на фирмата през изминалата седмица. Списъкът е сортиран спрямо позициите на служителите – от най-висока към най-ниска $(1 \dots n)$. След края на всяка седмица се определят екипи от по двама души/служители A и B, където A заема по-ниска позиция от B, но $h_A > 2 * h_B$.

// Приемаме, че във фирмата работят точно n служители и всеки от тях е на различна позиция.

Да се напише алгоритъм, който по зададен фирмен отсек намира броя на възможните екипи // Ако $\exists A, B, C$: A е на по-ниска позиция от B и от C и $h_A > 2 * h_B \wedge h_A > 2 * h_C$, то (A, B), (A, C) се броят като две възможни двойки.

Решение:

Списъкът изглежда така $\begin{vmatrix} 1. & h_1 \\ 2. & h_2 \\ \vdots & \vdots \\ n. & h_n \end{vmatrix}$, където $1, \dots n$ са позициите на служителите в сортиран вид.

Може да го представим като масив $[h_1, h_2, \dots, h_n]$ (1 2 n), където индексите отговарят на позициите.

Задачата се свежда до търсене на броя наредени двойки $(h_i, h_j) : i < j \wedge h_i > 2 * h_j$

1н) Brute Force

За всеки един служител (i) търсим броя на екипите, в които може да участва

```
BruteForce (Report[1..n])
1. count ← 0
2. for i ← 1 to n-1
3.     for j ← i+1 to n
4.         if Report[i] > 2*Report[j]
5.             count ← count + 1
6. return count
```

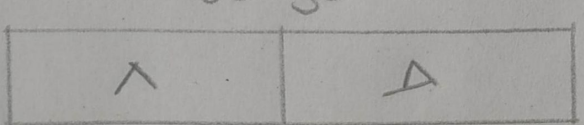
$T(n) \simeq n^2$

$M(n) \simeq 1$

Искане да подобрим бързодействието:

Ще използване логиката на MergeSort и на кания merge

общо



Броят двойки ще се образува като сборем броя на възможните двойки в лявата част, броя „—" в дясната част + броя възможни

двойки с ляв елемент от лявата част и десен елемент от дясната част

// Пример - ще намерим броя двойки с ляв елемент от A и десен от B

A: [6, 13, 21, 25]
↑ left, ↑ mid

B: [1, 2, 3, 4, 4, 5, 9, 11, 13]
↑ right, ↑ high

↳ // В хода на алгоритъма за всеки два такива A и B, в които търсим ще е изпълнено, че A, B са подмасиви на входния и mid < right

Ако A[left] > 2 * B[right] // тъй като ∀i ∈ {left+1 .. mid}: A[i] >= A[left] => A[i] > 2 * B[right]

count ← count + mid - left + 1

right ← right + 1

Ако A[left] <= 2 * B[right] // тъй като ∀i ∈ {right+1 .. high}: [i] >= [right] ∧ A[left] <= 2 * B[right] => A[left] <= 2 * B[i]

left ← left + 1

// Примерна работа на нашия алгоритъм върху конкретен вход:

count ← 0

[40, 25, 19, 12, 9, 6, 2] → 2 6 9 12 19 25 40

12: count += 4
19: count += 3
19: count += 3
count = 3

[40, 25, 19, 12] → 12 19 25 40

25 > 2 * 12
↳ count += 2

40 > 2 * 19
↳ count += 1

[40, 25] → 25 40

[40] [25]

↳ НЕ увеличаваме counter, тъй като 40 <= 2 * 25 = 50

[19, 12] → 12 19

[19] [12] — " —

[9, 6, 2] → 2 6 9

count = 2

[9, 6] → 6 9 ; [2]

count = 2

6 > 2 * 2 = 4
count += 2

[9] [6]

Общо, в края е вярно, че count = 15

→ Да се помисли как броенето може да се осъществи по време на изпълнение на самия merge-ред 9.

Псевдокод

TeamsCount(A, L, h, count)

1. if L < h
2. mid ← $\lfloor \frac{L+h}{2} \rfloor$
3. TeamsCount(A, L, mid, count)
4. TeamsCount(A, mid+1, h, count)
5. FindPairs(A, L, mid, h, count)

FindPairs(A, L, mid, h, count)

1. left ← low
2. right ← high
3. while left <= mid && right <= high
4. if A[left] > 2 * A[right]
5. count ← count + mid - left + 1
6. right ← right + 1
7. else
8. left ← left + 1
9. merge(A, L, mid, h)

$T(n) \asymp 2T(\frac{n}{2}) + 2n$ // merge-ред 9: n и редове 1-8: n

$n^{\log_2 2}$ vs $2n$

n vs $2n$

$n \asymp 2n \overset{2n}{\Rightarrow} T(n) \asymp n\lg n$