Технически Университет - София

КУРСОВА РАБОТА

по Синтез и анализ на алгоритми

Тема: Игра на пермутация

Изработил: Шериф Бюлентов Пашов

Специалност: Телекомуникации

Факултетен №: 111222091 Група: 57Б

Съдържание:

1. Какво е алгоритъм?

- Произход на думата „Алгоритъм“: Според някои автори думата „алгоритъм“ произлиза от името на персийския учен Ал Хорезми, който през през 825 г. е написал научен трактат за това как да се представят (записват) числата в Десетична бройна система и как се извършват аритметичните операции с тези представяния.

- Значение на думата „Алгоритъм“ : „Алгоритъм“ е интуитивно понятие. То е основно понятие в науката и не се дефинира (както точка, права и равнина в геометрията, пространство и време във физиката и др.), но за него може да се даде интуитивна представа. Понятието „алгоритъм“ е широкообхватно. Разглеждаме алгоритмите, предназначени за изпълнение от компютър. Алгоритъм, това е последователност от стъпки (инструкции, команди, указания, оператори) за действие, която при изпълнението си реализира зададена функционална зависимост между данните и резултатите (иначе казано след изпълнението на предписанието ще бъде решена съответната задача). Ако последователността не реализира зададената функционална зависимост, то тя не е алгоритъм на дадената задача.

  Във всяка инструкция (стъпка от алгоритъма) се указват всички или някои от следните елементи:

- Основни етапи при разработката на алгоритми за решаване на практически задачи.

  1. Определяне на стратегията на алгоритьма, т.нар. "идеен проект".
  2. Формиране и описание на основните относително самостоятелни части, които могат да се обособят в задачата и алгоритьма за решението й.
  3. Определяне на връзките между обектите и процесите в модулите.
  4. Разработване на пълен блоков алгоритъм за решаване на задачата с отчитане на необходимите входни данни и очакваните изходни резултати.
  5. Тестова проверка и доказателство на верността на алгоритьма с характеристични набори от работоспособни данни.

- Свойства на (компютърните) алгоритми:

  а) **Детерминираност** (определеност). Алгоритъмът като цяло и всяка негова стъпка при едни и същи данни дават един и същ (точно определен) резултат при различни изпълнения. Това свойство се спазва при класическото понятие за алгоритъм, но при т.нар вероятностни алгоритми то се нарушава. Друго тълкувание за определеност на алгоритми: всяка стъпка и алгоритъмът като цяло трябва еднозначно да се разбират от изпълнителя.

  б) **Крайност** (финитност). Алгоритъмът и всяка негова стъпка се изпълняват за крайно време. Алгоритъмът съдържа краен брой стъпки и следователно се изпълнява за краен брой стъпки.

В математиката има безкрайни процеси. Например, изчисляването на функция чрез разлагането й в ред на Тейлор тъй като е безкраен процес, то не е алгоритъм. Но ако функцията се изчислява чрез определен брой начални членове на реда, то безкрайният процес се превръща в краен и това вече е алгоритъм.
Често поради грешки в алгоритъма (или програмата) процесът се превръща в безкраен, например при циклите ако е сгрешено условието за излизане от цикъла.
За да се докаже, че дадена последователност от инструкции е алгоритъм, винаги като доказателство се използва, че тя се изпълнява за крайно време.

в) **Дискретност**. Това свойство е свързано с обстоятелството, че описанието представено от алгоритъма се състои от краен брой елементи (декларации, обекти, инструкции и др.), а съответният алгоритмичен процес протича на отделни стъпки. Изпълнението на алгоритъма във времето се извършва на интервали, на стъпки и всяка негова стъпка се изпълняват за крайно време. Алгоритъмът съдържа краен брой стъпки и следователно се изпълнява за краен брой стъпки.
Свойството дискретност налага непрекъснатите по своята природа процеси и обекти да се моделират чрез дискретни компютърни представяния. А това води до необходимостта от допълнителни проверки във всеки конкретен случай – доколко алгоритмичният модел е съответен на (адекватен) на реалния обект/процес.

г) **Масовост**. Това свойство отразява възможността при изпълнението на алгоритъма за всеки начален елемент (от допустимото множество входни данни) да се получава търсеният резултат. С други думи: алгоритъмът да може да се прилага не само при решаването на една конкретна задача, а на цял клас от еднотипни задачи.

д) **Резултатност**. Това свойство означава, че завършването изпълнението на един алгоритъм е осигурено (за произволни начални данни) след краен брой елементарни операции. Празно множество резултати е недопустимо!
Резултатността на алгоритъма се третира като насоченост на алгоритъма – след краен брой стъпки трябва да се получи или решението на поставената задача или отговор за неприложимостта на този алгоритъм към конкретно избраното множество от началните данни (което също е резултат). В общия случай е възможно изпълнението на определения от алгоритъма процес да не завършва (нарушена е крайността) или да прекъсва на някоя стъпка с резултат „няма решение”.

е) **Цикличност**. Това свойство е присъщо за повечето съвременни алгоритми. Алгоритъмът съдържа определена последователност от стъпки, която се повтаря определен брой пъти.

ж) **Формалност**. Не е необходимо изпълнителят да има представа за решаваната задача и естеството на получаваните резултати – достатъчно е той да изпълнява една след друга предписаните му елементарни операции (команди). Свойството формалност е от съществено значение, защото позволява изпълнителят на един алгоритъм да бъде и автомат.

з) **Изпълнимост**. „Силно” изискване, което се поставя пред компютърните алгоритми да се състоят от „изпълними стъпки”. [5]

и) **Ефективност**. „Алгоритмичният процес е ефективен, ако приключва в „реално” време и всички присъщи му резултати се получават след „приемлив” брой стъпки”. Алгоритми, които не решават задачата в ”разумен” срок от време или при изпълнението им се налага съхраняване на твърде много начални или междинни резултати, не са ефективни алгоритми в информатиката [6]

- Видове алгоритми:
    - Линейни
    - Алгоритъм за груба сила(Brute Force Algorithm)
    - Разклонени
    - Циклични
    - Рекурсивни (Recursive Algorithm)
    - Алгоритъмза обратно проследяване (Backtracking Algorithm)
    - Алгоритъм за търсене(Searching Algorithm)
    - Алгоритъм за сортиране (Sorting Algorithm)
    - Алгоритъм за хеширане (Hashing Algorithm)
    - Алгоритъм „Разделяй и владей"(Divide and Conquer Algorithm)
    - Евристични
    - "Алчни" алгоритми (Greedy Algorithm)
    - Вероятностни
    - Паралелни
    - Алгоритъм за динамично програмиране(Dynamic Programming Algorithm)
    - Рандомизиран алгоритъм (Randomized Algorithm)

- Предимства на алгоритмите
    - Лесно е за разбиране.
    - Алгоритъмът е поетапно представяне на решение на даден проблем.
    - В алгоритъма проблемът е разбит на по-малки части или стъпки, следователно за програмиста е по-лесно да го преобразува в действителна програма.

- Недостатъци на алгоритмите
    - Писането на алгоритъм отнема много време.
    - Разбирането на сложна логика чрез алгоритми може да бъде много трудно.
    - Изявленията за разклоняване и цикъл са трудни за показване в Algorithms(imp).

2. Видове алгоритми за сортиране

- Алгоритъмът за сортиране е алгоритъм, съставен от поредица от инструкции, които приемат масив като вход и извеждат сортиран масив.
- Има много алгоритми за сортиране, като например:

  Selection Sort, Bubble Sort, Insertion Sort, Merge Sort,

  Heap Sort, QuickSort, Radix Sort, Counting Sort, Bucket Sort, ShellSort, Comb Sort, Pigeonhole Sort, Cycle Sort

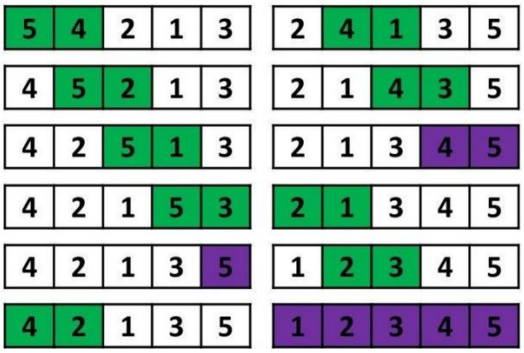
1. Bubble Sort

- **Bubble Sort** е най-простият алгоритъм за сортиране, който работи чрез многократна размяна на съседните елементи, ако са в грешен ред.

- Алгоритъм
  - ✓ Стъпка 1: Сравнете всяка двойка съседни елементи в списъка
  - ✓ Стъпка 2: Разменете два елемента, ако е необходимо
  - ✓ Стъпка 3: Повторете този процес за всички елементи, докато целият масив бъде сортиран

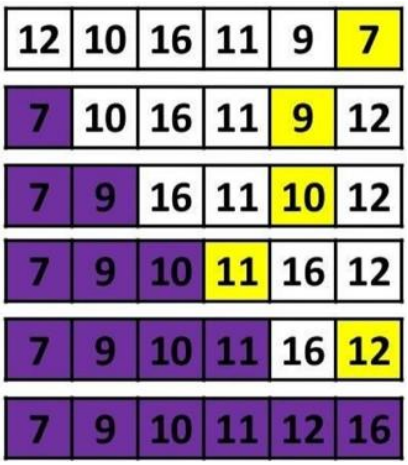
- Времева сложност: $O(n^2)$, тъй като има два вложени цикъла

```
def BubbleSort(arr):
    for i in range(len(arr)-1):
        for j in range(len(arr)-i-1):
            if arr[j] > arr[j+1] :
                arr[j], arr[j+1] = arr[j+1], arr[j]
    return arr
```



2. Selection Sort

- **Selection Sort** е алгоритъмът, който сортира масив, като многократно елемент (като се има предвид възходящ ред) от несортираната част и го поставя в началото. за сортиране намира на минималния



- Алгоритъм:
  - ✓ Стъпка 1: Намерете минималната стойност в списъка
  - ✓ Стъпка 2: Разменете я със стойността в текущата позиция
  - ✓ Стъпка 3: Повторете този процес за всички елементи, докато целият масив бъде сортиран

- Времева сложност: $O(n^2)$, тъй като има два вложени цикъла

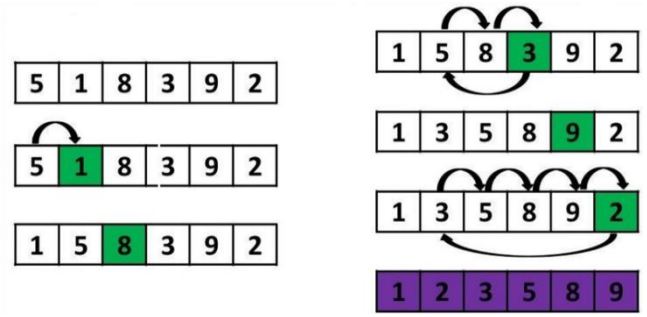
```
def SelectionSort(A):
    for i in range(len(A)):
        minind = i
        for j in range(i+1, len(A)):
            if A[minind] > A[j]:
                minind = j
        A[i], A[minind] = A[minind], A[i]
    return A
```

3. Insertion Sort

- **Insertion sort** е прост алгоритъм за сортиране, който работи по начина, по който сортираме картите за игра в ръцете си.

- Алгоритъм:
  - ✓ Стъпка 1: Сравняваме всяка двойка съседни елементи в списъка
  - ✓ Стъпка 2: Вмъкваме елемент в сортирания списък, докато заеме правилното място
  - ✓ Стъпка 3: Разменяме два елемента, ако е необходимо
  - ✓ Стъпка 4: Повтаряме този процес за всички елементи, докато целият масив бъде сортиран

- Времева сложност: $O(n^2)$,
  тъй като има два вложени цикъла

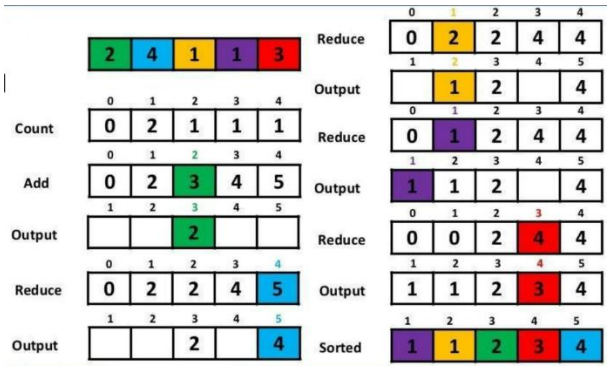


```
def InsertionSort(arr):
    for i in range(1, len(arr)):
        store = arr[i]
        j = i-1
        while j >=0 and store < arr[j] :
                arr[j+1] = arr[j]
                j -= 1
        arr[j+1] = store
    return arr
```

4. Counting Sort

- **Counting sort** е техника за сортиране, базирана на ключове между конкретен диапазон. Работи, като преброява броя на обектите, които имат различни ключови стойности (вид хеширане). След това изчислява позицията на всеки обект в изходната последователност.

- Алгоритъм
  - ✓ Стъпка 1: Създайте масив за преброяване, за да съхранявате броя на всеки уникален обект
  - ✓ Стъпка 2: Променете масива за броене, като добавите предишното Число.
  - ✓ Стъпка 3: Създайте изходен масив чрез намаляване на масива за броене




```
def countSort(arr):
    outputarr = [0 for i in range(127)]
    countarr = [0 for i in range(127)]

    for i in arr:
        countarr[ord(i)] += 1

    for i in range(127):
        countarr[i] += countarr[i-1]

    for i in range(len(arr)):
        outputarr[countarr[ord(arr[i])]-1] = arr[i]
        countarr[ord(arr[i])] -= 1

    return outputarr[0:len(arr)]
```

- Времева сложност: 0(n+k),
  където "n" е броят на елементите във входния масив, а "k" е диапазонът на входа.

5. Radix Sort

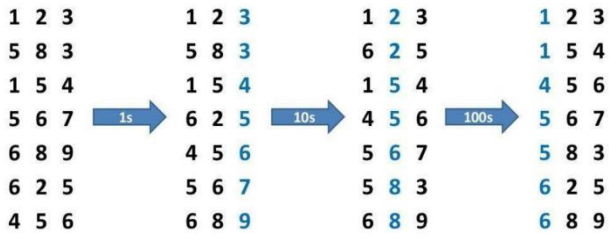
- **Radix sort** е алгоритъм, който сортира числа чрез обработка на цифри на всяко число, започвайки от най-малката цифра (least significant digit -LSD) или започвайки от най-

значимата цифра (most significant digit - (MSD). Идеята на Radix Sort е да се извършва сортиране цифра по цифра, като се започне от най-малката цифра до най-значимата цифра. Radix Sort използва сортиране с броене като подпрограма за сортиране.

- Алгоритъм
  - ✓ Стъпка 1: Вземете най-малката цифра от всеки елемент
  - ✓ Стъпка 2: Сортирайте списъка с елементи въз основа на тази цифра
  - ✓ Стъпка 3: Повторете сортирането с всяка по-значима цифра

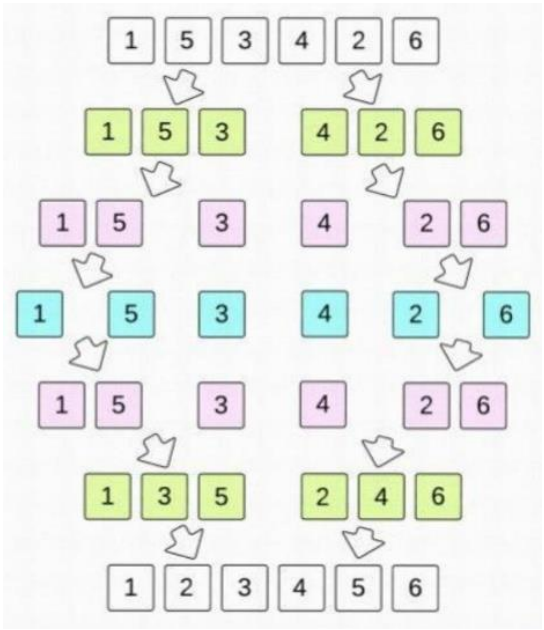
```
def countingSort(arr, count1):
    n = len(arr)
    output = [0] * (n)
    count = [0] * (10)
    for i in range(0, n):
        index = (arr[i]/count1)
        count[ int((index)%10) ] += 1

    for i in range(1,10):
        count[i] += count[i-1]

    i = n-1
    while i>=0:
        index = (arr[i]/count1)
        output[ count[ int((index)%10) ] - 1] = arr[i]
        count[ int((index)%10) ] -= 1
        i -= 1
    return output
```

- Времева сложност: 0(n+k/d), където "n" е броят на елементите във входния масив, "k" е диапазонът на входа и "d" е броят на цифрите.



6. Merge Sort

- **Merge Sort** е алгоритъм за разделяне и владеене (Divide and Conquer). Той разделя входния масив на две половини, извиква себе си за двете половини и след това обединява двете сортирани половини.
- Алгоритъм:
  - ✓ Стъпка 1: Разделете списъка рекурсивно на две половини, докато вече не може да бъде разделен
  - ✓ Стъпка 2: Обединете (завладейте) по-малките списъци в нов списък в сортиран ред



- Времева сложност: 0(n * log(n))

```
def merge(arr, l, m, r):
    n1 = m - l + 1
    n2 = r- m
    L = [0] * (n1)
    R = [0] * (n2)
    for i in range(0 , n1):
        L[i] = arr[l + i]
    for j in range(0 , n2):
        R[j] = arr[m + 1 + j]
    i = 0     # Initial index of first subarray
    j = 0     # Initial index of second subarray
    k = l     # Initial index of merged subarray
    while i < n1 and j < n2 :
        if L[i] <= R[j]:
            arr[k] = L[i]
            i += 1
        else:
            arr[k] = R[j]
            j += 1
        k += 1
    while i < n1: # Copy the remaining elements of L[]
        arr[k] = L[i]
        i += 1
        k += 1
    while j < n2: # Copy the remaining elements of R[]
        arr[k] = R[j]
        j += 1
        k += 1
```

Правила: Игра на Пермутация

Гошо и Пешо играят следната игра:

1. Те избират пермутация на първите ***N*** числа за начало.
2. Те играят последователно и Гошо играе първи.
3. В ход те могат да премахнат всяко едно останало число от пермутацията.
4. Играта приключва, когато останалите числа образуват нарастваща последователност. Човекът, който е играл последния ход (след което последователността се увеличава) печели играта.

Ако приемем, че и двамата играят оптимално, кой печели играта?

Input Format

Първият ред съдържа броя на тестовите случаи ***T***. Следват ***T*** тестови случаи. Всеки случай съдържа цяло число ***N*** на първия ред, последвано от пермутация на целите числа ***1..N*** на втория ред.

Constraints

- $1 \leq T \leq 100$
- $2 \leq N \leq 15$
- Първоначално пермутацията няма да бъде нарастваща последователност.

Output Format

Изходни редове по един за всеки тест, съдържащи "Gosho", ако Гошо спечели играта и "Pesho" в противен случай.

Sample Input

```
2
3
1 3 2
5
5 3 2 1 4
```

Sample Output

```
Gosho
Pesho
```

Explanation

За първия пример Гошо може да премахне 3-ката или 2-ката, за да направи последователността нарастваща и да спечели играта.

За втория пример, ако 4 бъде премахнато, тогава единственият начин да имате нарастваща последователност е да имате само 1 останало число, което ще отнеме общо 4 хода, като по този начин ще позволи на Пешо да спечели. На първия ход, ако Гошо премахва 4, ще са необходими още 3 хода, за да се създаде нарастваща последователност, така че Пешо печели. Ако Гошо не премахне 4, тогава Пешо може да го премахне на следващия си ход, тъй като Гошо не може да спечели с един ход.

```python
def countMovesSorted(inputArr):
    # Функция, която сортира входния масив и връща броя на инверсиите (размени)

    lenghtArr = len(inputArr)
    cauntMoves = 0
    i = 0

    while i < lenghtArr:

        for j in range(0, lenghtArr-i-1):
            # Създаване на флаг дали има размяна
            flagMove = False

            if inputArr[j] > inputArr[j+1]:

                 # Ако текущият елемент е по-голям от следващия, ги разменяме
                inputArr[j], inputArr[j+1] = inputArr[j+1], inputArr[j]
                # Казваме, че се е случила промяна
                flagMove = True

        if flagMove:

             # Ако има размяна, премахваме последния елемент (най-големият)
            inputArr.pop()
             # Добавяме ход
            cauntMoves += 1
             # Оразмеряване на масива
            lenghtArr -= 1

            if lenghtArr == 1:

                # Ако остава само един елемент, завършваме
                return cauntMoves

        else:

            i += 1


    return cauntMoves


def permutationGame(N, permutation):

    for i in range(N):

        countMove = countMovesSorted(permutation)
        # Проверка: Кой е победил, като Гошо винаги е първи
        if countMove % 2 == 1:
            winner = "Гошо"
        else:
            winner = "Пешо"

        return winner

# Четене на броя на тестовите случаи
gameCount = int(input("Въведете броя на игри: "))

for n in range(gameCount):
    # Четене на броя на елементите и пермутацията
    elementsNumber = int(input(f"Брой елементи за игра №{n+1}: "))

    # Четене на елементите на играта (пермутацията)
    permutation = list(map(int, input().split()))

    # Определяне на победителя и извеждане на резултата
    winner = permutationGame(elementsNumber, permutation)

    # Принтиране на победителя в дадената игра
    print(f"Игра №{n+1}.  Победител е: {winner}")
```

Примерни Входни и Изходни данни:

```
Въведете броя на игри: 2
Брой елементи за игра №1: 3
1 3 2
Игра №1. Победител е: Гошо
Брой елементи за игра №2: 5
5 3 2 1 4
Игра №2. Победител е: Гошо
```

```
Въведете броя на игри: 2
Брой елементи за игра №1: 7
10 12 4 9 25 33 2
Игра №1. Победител е: Пешо
Брой елементи за игра №2: 5
5 4 3 2 1
Игра №2. Победител е: Пешо
```

```
Въведете броя на игри: 3
Брой елементи за игра №1: 4
4 3 2 1
Игра №1. Победител е: Гошо
Брой елементи за игра №2: 4
10 30 3 6
Игра №2. Победител е: Пешо
Брой елементи за игра №3: 5
4 5 1 2 3
Игра №3. Победител е: Пешо
```

```
Въведете броя на игри: 3
Брой елементи за игра №1: 5
5 4 3 1 2
Игра №1. Победител е: Гошо
Брой елементи за игра №2: 6
4 5 6 7 1 2
Игра №2. Победител е: Пешо
Брой елементи за игра №3: 5
7 6 5 3 4
Игра №3. Победител е: Гошо
```

- Във функцията countMovesSorted си използвал алгоритъма "Bubble Sort" за сортиране на входния масив. В този алгоритъм, елементите на масива се сравняват последователно и се разменят, ако не са в правилния ред. Процесът се повтаря до момента, в който няма повече размени.
- Така, като се използва алгоритъма на "Bubble Sort",

  функцията сортира масива и връща броя на инверсиите (размени), които са направени по време на сортирането.