**Лекция 1**

Какво е Алгоритъм?

* Последователност стъпки за решаване на проблем.

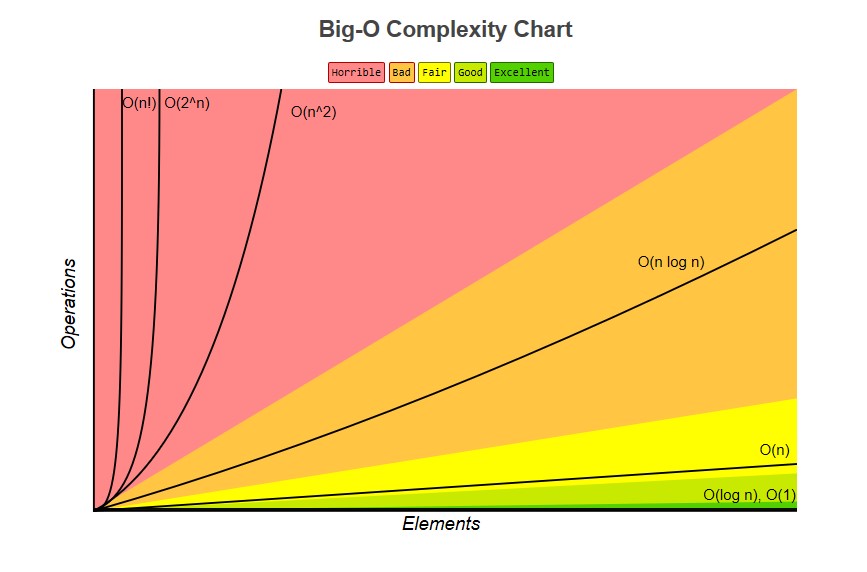
Какво е “Структура от данни”?

* Начин за организиране на данни във формат удобен за ползване.

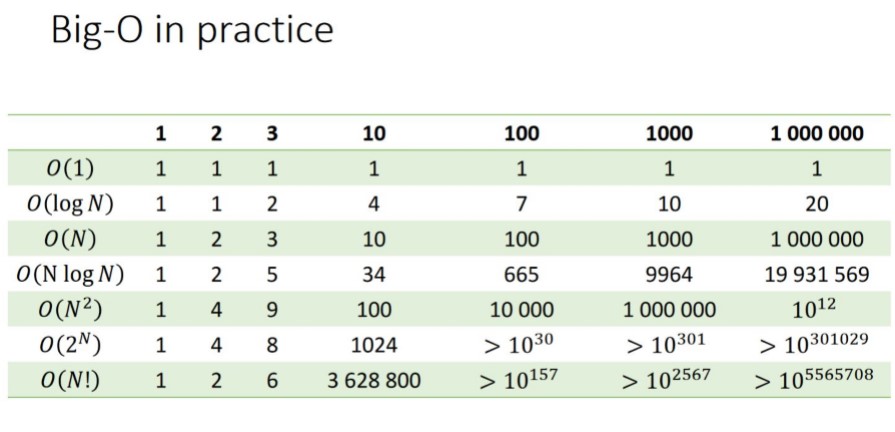
Видове оценка на сложност

* В най-лошия случай (песимистична) Какъв е максималният възможен брой операции (единици памет), които могат да са нужни на алгоритъма, за да реши задачата?
* В най-добрия случай (оптимистична) Какъв е минималният възможен брой операции (единици памет), които може да извърши (използва) алгоритъмът, за да реши задачата?
* В средния случай (средна) Ако считаме, че всеки възможен вход е равновероятен, какво е “средното аритметично” на броя операции (единици памет), които трябват при всички възможни входове?
* При многократно изпълнение (амортизирана) Ако алгоритъмът ще се извиква няколко пъти в рамките на дадена програма, колко операции (единици памет) средно ще са му необходими за едно извикване?

Big O - За големи стойности на входните данни каква функция може да се използва за да се ограничат отгоре броя на необходимите операции (пренебрегвайки константи с които може да се умножи функцията)<https://www.bigocheatsheet.com/>







**Примери**

Константна сложност - знаем точният отговор и го връщаме веднага.

Логаритмична сложност - данните, които трябва да претърсим са разделени, така че търсейки на всяка стъпка изхвърляме половината от възможностите.

Линейна сложност - обхождаме всички данни и връщаме намереният резултат.

**Как да измерим сложността на алгоритъм?**

* С оценка на сложностите на различните компоненти по-брой операции/памет
* Емпирично - Увеличаваме линейно големината на входните данни и измерваме време/памет

**Изводи**

* За да може да сравним различни решения на една задача трябва да може да оценим сложността на решението.
* Сложността на алгоритъм, най-често оценяваме сложността в найлошият случай с Big О нотацията.
* При оценяване с Big O нотацията, премахваме всякакви константи и свободни членове от функциите.
* О(1) < О(log(n)) < O(n) < O(n\*log(n)) < O(n^2) < О(2^n) < O(n!)

**Лекция 2 – Sort**

**Какво е сортиране и за какво ни е необходимо?**

* Основен клас алгоритми много често необходим за решаване на реални проблеми
* Добър пример за демонстриране на това какво е алгоритъм, как се изчислява сложност на алгоритъм, как да бъдем по-критични към това с каква сложност решаваме проблем.

**Сортиране с метод на мехурчето** - бавна

Основна идея: Започвайки последователно от началният елемент до крайният сравняваме всеки елемент със следващия като ги разменяме ако не са подредени. По този начин на всяка стъпка изкарваме най-големият в края на правилното за него място. Повтаряйки този процедура толкова пъти колкото е големината на масива постигаме правилно подреждане на целият масив.

**Сортиране с пряка селекция** - бавна

Намаля броя на разменянията в сравнение с метода на мехурчето!

Основна идея: Търсим най-големият елемент в масива и директно го поставяме на последно място. След това следващият по големина и отново го поставяме на място и т.н. Докато всички се подредят

**Сортиране с вмъкване** - бавна

Основна идея: сортиране постепенно на все по-голяма част от масива, като обхождайки несортираната част всеки един елемент го поставяме в сортираната част с намиране на правилното за него място на което сортираният масив остава сортиран.

**Сортиране чрез сливане(merge sort)** - бърза

Основна идея: Ако имаме два сортирани масива то със линейна сложност може да ги влеем в един масив. Тогава ако разделим масива който искаме да сортираме на по-малки масиви и на всяка стъпка сливаме два по-малки масива в един голям то за log(N) стъпки ще слеем всички масиви до един масив, като всяка от стъпките е била линейна.

**Бързо сортиране ( quick sort)** - бърза

Основна идея: Ако вземем едно произволно число от масива, то с линейна сложност можем да прехвърлим всички по-малки числа от масива да са в ляво на числото, а всички по-големи в дясно. При бързото сортиране избираме число от масива прехвърляме по-малите отляво, по-големите от дясно и след това изпълняваме същата процедура за лявата и дясната половина.

По тази процедура получаваме сложност в средният случай О(n\*log(n))

Рандомизирано Бързо Сортиране - Справя се с проблема, че точно определена редица прави сложността О (n^2), като използва произволно избиране на елемент за разделяне.

Quicksort - по-бърз от mergesort за малки масиви, но mergesort е по-добър за големи масиви.

**Сортиране с броене**

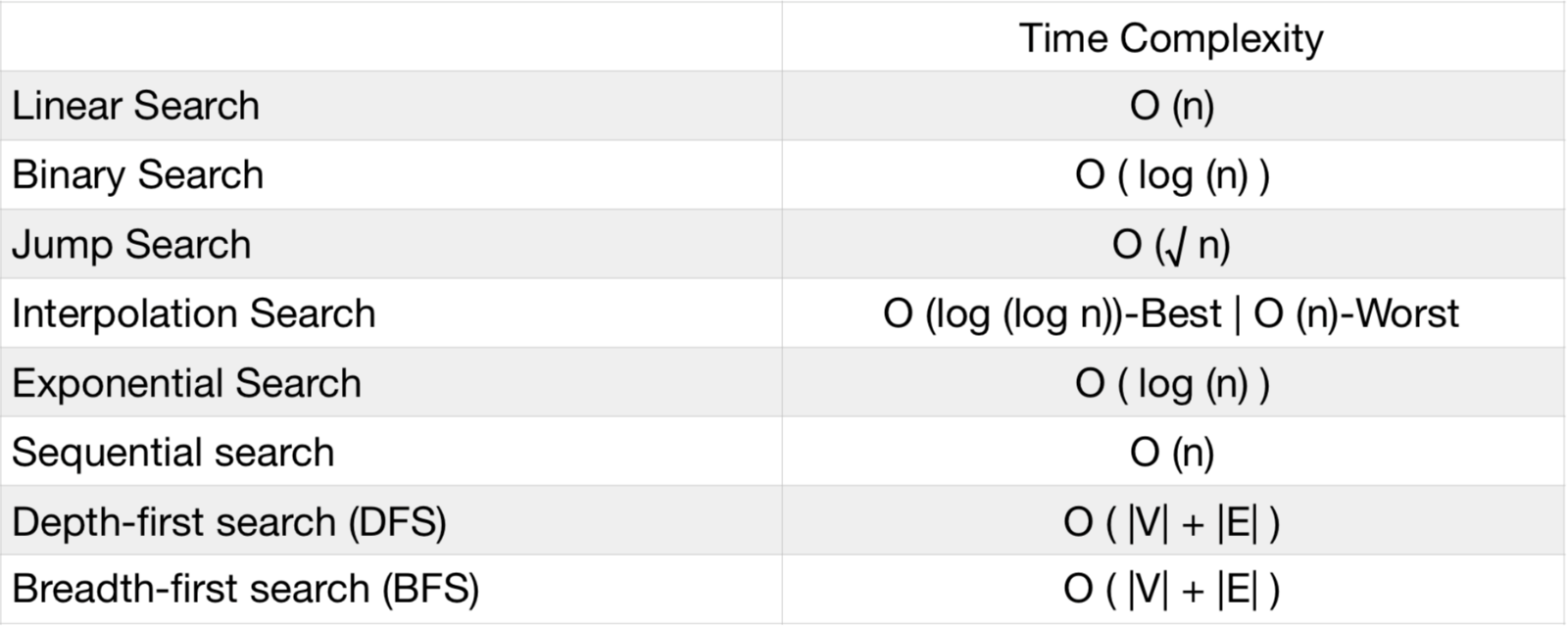
Основна идея: Понеже имаме ограничен брой различни стойности в масива то може да преброим по колко пъти се среща всяка една от тези стойности(с едно обхождане на масива) и след това със второ обхождане да наредим стойностите по техният ред. Стабилност на сортирането - ако имаме два елемента които са равни в първоначалният масив, то във финалният те се срещат във същият ред като първоначалният масив.



**Лекция 3.1 – Search**

**Двоично търсене(Binary search)**

При несортиран масив няма как да търсим със сложност по-малка от О(n), в случай, че ще извършваме много търсения върху един масив то за да ускорим търсенето можем първоначално да го сортираме ( за О(nlog(n)) ) и после да търсим със по-ниска сложност O(log(n))

****

**Минимум, Максимум, Средна точка**

Задача: Намерете най-големият(най-малкият) елемент на масив

Задача: Намерете к-тия най-голям елемент на масив

Задача: Намерете средният елемент на масив.

**Лекция 3.2 – Linked List**

|  |  |  |
| --- | --- | --- |
| Характеристики | Масив | Списък |
| + | -**Access** - O(1)  -**Insert in end** - O(1) (if not full) | - **Insert at end or beg** O(1)  - **space is proportional to number of elements**  - **Delete is easy** |
| - | -**space is a lot if empty and if it fills we have to copy it**  - **delete** O(n)  - **insert** O(n) | - **access** O(n)  - **more space- keeps the address of the next node** |

**Основни операции**

1. Traversing a linked list. – useful for printing and searching for x node
2. Create an empty list
3. Append a new node (to the end) of a list
4. Prepend a new node (to the beginning) of the list
5. Inserting a new node to a specific position on the list
6. Deleting a node from the list at position pos
7. Updating a node in the list

**Лекция 4 – Stack**

**Стек** - абстрактна структура от данни от линеен вид при която последователността на добавяне на елементи е точно обратната на последователноста на извличане на елементи от нея (FILO)

**Операции**

● Create() – O(1)

● Push() – O(1)

● Pop() - O(1)

● Peek() – O(1)

● IsEmpty() – O(1)

**1.Реализация със свързан списък**

* Push = InsertAtFront за свързан списък
* Pop = DeleteFromFront за свързан списък

**2. Реализация със фиксиран масив**

* Заделя се масив с определена големина N
* Поставя се брояч на броя на елементи в стека
* Push -> записва елемента в масива спрямо стойността на брояча и учеличава брояча с 1
* Pop -> връща елемент от масива спрямо стойността на брояча и намаля брояча с 1
* При излизане трябват проверки за броя на елементите в масива за да не се получи достъп до памет извън масива
* push() - когато масива е пълен се удвоява по размер
* pop(): - когато масива е пълен на ¼ се намалява на половина.

**Свързан списък или масив за реализация?**

* Със свързан списък за push и pull имаме гарантирана константна сложност в най-лошият случай.
* При реализацията със масив имаме амортизирана константна сложност на push and pull

**Лекция 5.1 - Queue**

**Опашка** - абстрактна структура от данни от линеен вид при която последователността на добавяне на елементи е същата като последователността на извличане на елементи от нея(FIFO)

**Операции** ● create() ● enqueue() ● dequeue() ● peek() ● isEmpty()

1. **Реализация със свързан списък**

* enqueue = InsertAtEnd за свързан списък
* dequeue = DeleteFromFront за свързан списък

**2.Реализация с фиксиран масив**

* Заделя се масив с определена големина N
* Използват се два брояча на индекси един за началото на опашката и един за края на опашката
* Enqueue -> записва елемента в масива спрямо стойността на брояча старт и учеличава брояча с 1
* Dequeue -> връща елемент от масива спрямо стойността на брояча end и намаля брояча с 1
* NB! - трябват проверки за броя на елементите в масива за да не се получи достъп до памет извън масива

**Свързан списък или масив за реализация?**

* Със свързан списък за push и pull имаме гарантирана константна сложност в най-лошият случай.
* При реализацията със масив с променлива големина имаме амортизирана константна сложност на enqueue and dequeue

Задача: Дадена е опашка q. Да се изключи от q най-малкият й елемент.

**Лекция 5.2- Tree**

**Дърво**

* Йерархична структура от данни.
* Използва се за организиране на данни в йерархия.
* Основна търсена характеристика: бързо добавяне и търсене на елемент

**Двоично дърво (Binary Tree)**

дърво в което всеки възел има максимум два наследника(ляв и десен)

Индуктивна дефиниция на двоично дърво:

* Празно дърво е двоично дърво
* Възел, който има два наследника, които са двоични дървета е двоично дърво.

**Двоично наредено дърво (Binary Search Tree)** :

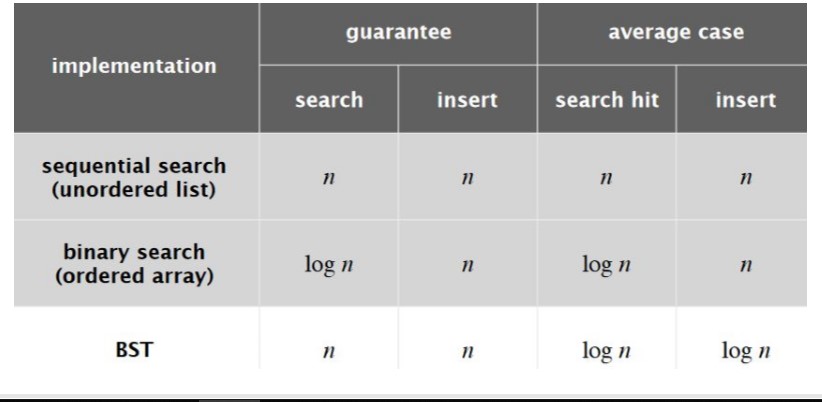
двоично дърво, в което всеки възел и наследниците му са в симетрична подредба( например: Всички леви наследници са по-малки от бащата, всички десни по-големи)

**Основни операции**

* search(X) - търси елемент в дървото
* insert(X) - добавя елемент в дървото
* remove(X) - изтрива елемент от дървото
* print() - принтира дървото в специфичен ред

**Printing Orders**

* Pre-order – Root/Left/Right, Root/Right/Left
* In-order – Left/Root/Right, Right/Root/Left
* Post-order-Left/Right/Root, Right/Left/Root



**Лекция 6 – Balanced Trees**

**-Каква структура да използваме ако само ще търсим без да променяме числата в структурата?**

● Масив - 1. Сортираме го , 2. Търсим със сложност O(log(n))

**-Каква структура да използваме ако ще търсим, но съшо така ще добавяме и изваждаме числа?**

**Балансирано дърво (Balanced Tree):** Дърво, в което за всеки възел имаме свойството, че височината на лявото му поддърво се различава от височината на дясното поддърво с максимум единица.

**Сложности в най-лошият случай:**

* Търсене О(log(n))
* Добавяне на елемент О(log(n))
* Изтриване на елемент О(log(n))

**Видове балансирани дървета**

* 2-3 дърво
* AVL дърво
* Red-Black дърво
* Splay дърво
* Treap

**AVL дърво**

**Основна идея**: Всяко поддърво T = (X, L, R) поддържа коефициент на баланс:

balance(T) = height(R) − height(L) ϵ [-1,0,1]

**Как AVL дървото запазва баланса си?**

* Дървото преди и след всяка операция задължително се намира в състояние, че за всеки възел X имаме b(X)∈[-1,0,1]
* Операциите добавяне и премахване първоначално се извършват стандартно като за BST и оттам понеже имаме промяна на броя на възлите може да имаме и промяна на балансиращият индекс за бащите на добавеният или премахнат възел. -> Промяната може да направи стойности на b от -2 или 2.
* След стандартната процедура за добавяне или премахване ако е необходимо се прилагат ротации за балансиране на дървото.

**Балансиране при включване и изключване**

Балансиране при включване

* При дъното на включването височината винаги се увеличава с 1
* Ако височината на по-ниското дете се увеличи, то височината на родителя не се променя
* При балансиране винаги компенсираме за увеличената височина на детето

Балансиране при изключване

* При дъното на изключването дъното височината винаги се намалява с 1
* Ако височината на по-високото дете се намали, то височината на родителя не се променя
* Ако след балансиране b(T) != 0, значи сме компенсирали за намалената височина на детето
* Ако след балансиране b(T) = 0, значи височината се е намалила

**Лекция 7 -Пирамида (Heap)**

Когато говорим за пирамида ще имаме предвид **двоична пирамида(Binary Heap)** (въпреки, че има и други).

* Пирамида е почти пълно двоично дърво, като само най-дълбокото ниво не е пълно и то последователно
* За пирамидата имаме свойството, че за всеки възел бащата е по-голям и от двете си деца (при максимална пирамида)

**Операции -** getMax(X), extractMax(X), insert(X), delete(X)

**Приоритетна опашка- priority\_queue<int,vector<int>,greater<int>> == max\_heap**

* Опашка, в която се вкарват последователно елементи, а при изкарване елементите ги получаваме наредени по приоритет (например големина)
* Ако всички са с еднакъв приоритет ще работи както обикновена опашка

**Реализации на пирамида**

* Свързано представяне стандартно за двоично дърво
* Представяне с масив (space efficient!)

**Представяне с масив**

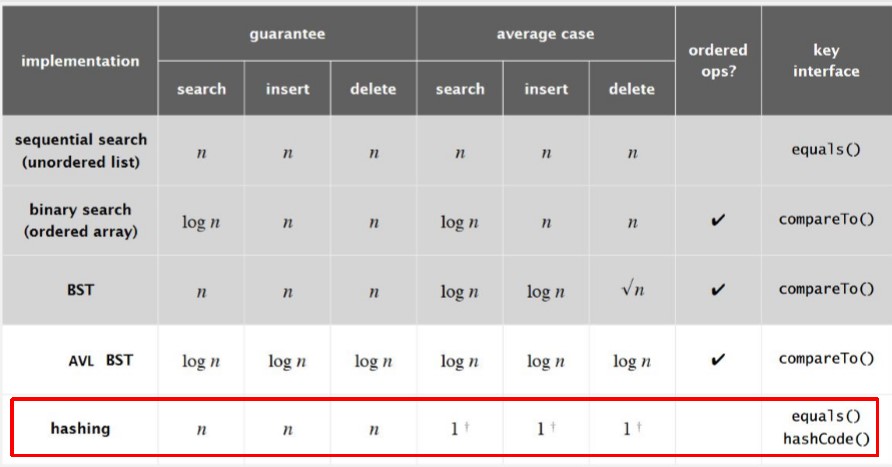
* Удобно поради последователното запълване на пирамидата
* Корена е на позиция 0, а наследниците му са на позиция 1 и 2
* За всяка позиция от масива i наследниците са 2\*i+1 и 2\*i+2 (ако съществуват)

**Пирамидално сортиране(Heap Sort)**

Основна идея: Преобразува масива в пирамида(операцията heapify n пъти- O(n), след което вади един по един елементите от върха на пирамидата(O(n\*logn)) и ги поставя сортирани последователно в края на масива.

Сортиране със сложност O(n\*log(n)) и константна допълнителна памет!

**Лекция 8 – Геометрично търсене. КД дърво**

**Лекция 9 – Hash Table**

**Хеш функция**

* Функция, която преобразува обект в число в определени граници
* Свойства:

Хеш функцията трябва да е бърза.

Трябва да връща винаги един и същи резултат за един и същ обект

**Хеш множество**

Ненаредена структура от данни, която може да запомня обекти, като позволява търсенето за това дали обект е вече добавен да става с сложност О(1) в средният случай.

**Хеш таблица**

Структура от данни, която съдържа двойки (ключ, стойност), която позволява добавяне и изваждане на нови двойки и търсене по ключ със сложности 0(1) в средният случай.

-**Основна идея:** Ако имаме хешираща функция, която да хешира ключовете в интервала 0..К бихме могли да използваме масив от К елемента за запомняне на стойностите, като от хеширащата функция ще разберем в коя клетка да пишем.

**- Колизии** - два обекта имат една и съща хеш стойност h(k1)=h(k2); Двата обекта са различни, но поради ограничението в размера на пространството за хеширане, техните стойности съвпадат.

**Какво правим ако има колизия?-**основни стратегии:

1. Запазване на няколко стойности в една клетка (separate chaining)- Вместо една стойност пазим списък със всички добавени двойки. Като при търсене търсим в списъка.

* Insert - изчисляване на хеш функцията, която връща число от 0 до n, добавяне в списъка, който е на позиция n
* Търсене - изчисляване на хеш функцията, която връща число от 0 до n, търсене в списъка, който е на позиция n
* При добра хешираща функция и броя на числата ако са около ¼ от големината на хеш таблицата - константна сложност(в средният случай)
* Сложността в най-лошият случай: О(n) - ако всички добавени елементи са с един и същ хеш код и всички числа са застанали в списък
* Увеличаване на големината на таблицата

Цел: Търси се съотношението на броя на елементите към броя на клетките в таблицата да е константа. Примерна стратегия:

* Удвояваме големината на таблицата ако броят на елементите стане 8 пъти по голям от броя на клетките в таблицата
* Намаляме големината на таблицата на две ако броя на елементите стане само 2 пъти по-малък от броя на клетките в таблицата.
* След промяна на големината на таблицата всички елементи се пре хешират!

1. Търсене на следваща празна клетка (Linear probing)- ако клетката е заета търсим следваща свободна клетка и разполагаме двойката там(ключа и стойността)

* Clustering - при образуване на групи от елементи в масива с ключове става все по вероятно някоя от стойностите да бъда получена като хеш от нов елемент и оттам да се търси следваща стойност (което пък от своя страна разширява и групата)

Хеш таблица с Linear Probing

* За да е оптимално търсенето - трябва да се поддържа N/M < ½ За това следваме следната процедура:
* Ако N/M > ½ удвояваме големината на таблицата
* Ако N/M < ⅛ намаляме на половина големината на таблизата
* При намаляване или увеличаване на големината на таблицата за всички елементи до момента се преизчислява хеш функцията
* Изтриване в хеш таблица с Linear Probing
  + - Не може просто да изтрием ключ, която е между други ключове понеже ще направи дупка и след това ще провали евентуално търсене на ключ.
    - Решение: Вместо изтриване отбелязваме за изтрито.

1. Двойно хеширане- Имаме две(или повече) хеш функции и когато се получи колизия със първата се изчислява хеп от втората хеш функция и се търси да се добави стойността на място хеш1+хеш2, ако и то е заето се търси място на хеш1+2\*хеш2 и т.н. Докато се намери място.

**Каква хеш функция да използваме?**

Цел: стойностите, които хешираме да се разположат максимално еднакво в целият интервал. Може да ползваме различни специфични хеш функции за нашият конкретен случай, като ако нямаме конкретна функция предвид, то може да ползваме универсална хешираща функция.

**Универсално хеширане:** h(x) = ((a\*x+b) mod p) mod n

Където n e големината на таблицата. p e просто число по-голямо от n а и b са произволни числа по-малки от p

**Как да хешираме масиви?**

**“31x + y rule”:** Обхождаме масива и последователно акумулираме стойностите му като умножаваме сумата до сега с 31 и добавяме новото число.

Защо 31? - Просто, нечетно число. - 31 \* i == (i << 5) - i (лесно за изчисление)

**Как да хешираме разнообразни типове данни?**

* Double - може да го разгледаме като два int
* Long - и той може да се разглежда като два int
* String - масив от char
* Обект - множество от член данните си(всяка от която вече знаем как да хешираме). Множеството може да го разглеждаме като масив от разнородни елементи.

**Лекция 10 – Graph**

**Какво е граф? -**G(E-ребра,V-върхове) - Нелинейна структура от данни съдържаща обекти-върхове свързани помежду си с връзки - ребра.

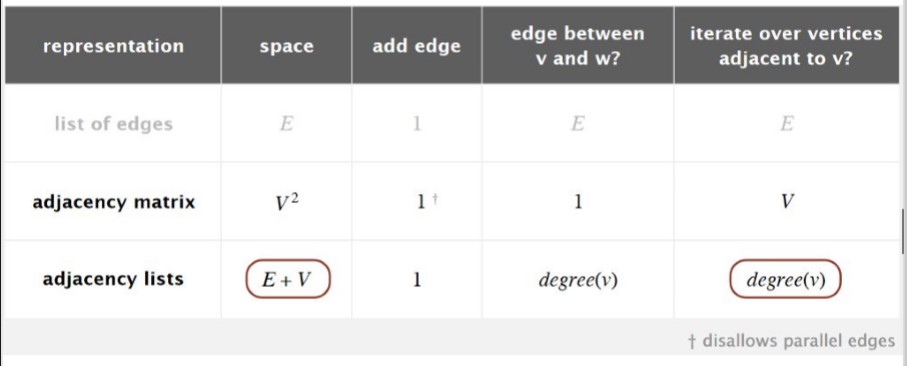
**Път в граф:** Последователност от върхове в граф, свързани с ребро, без да се повтаря ребро.

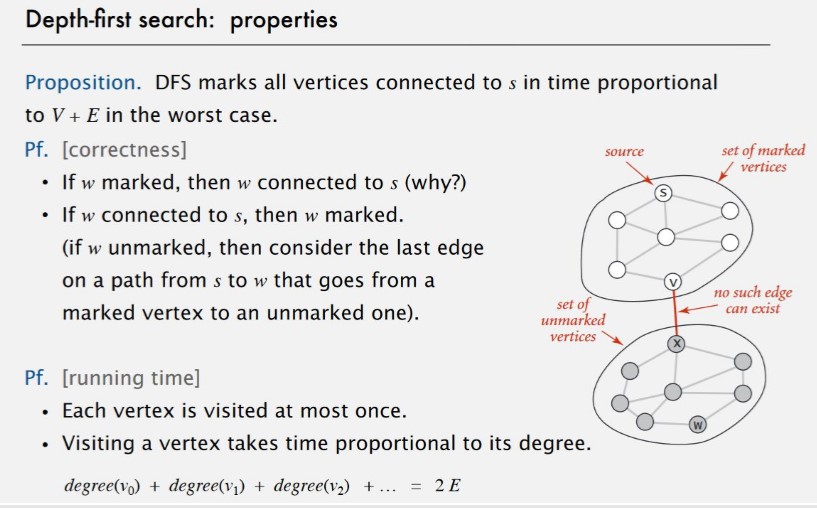
**Свързаност:** Два върха са свързани ако съществува път между тях

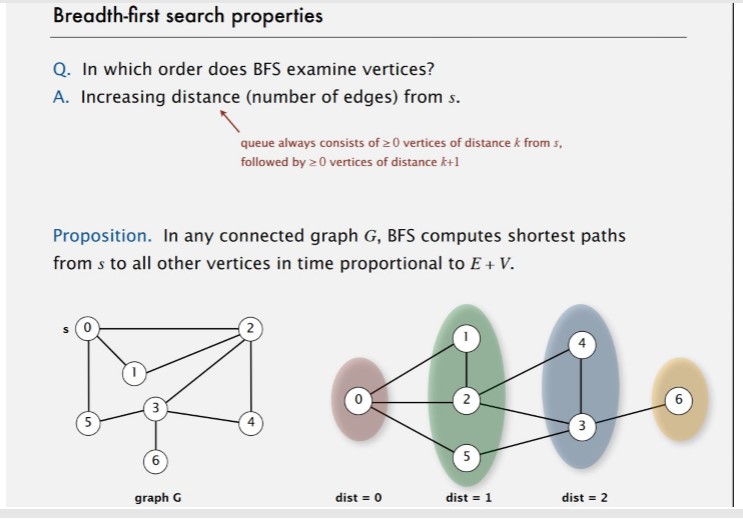
**Цикъл :** път с дължина повече от 1, който започва и свършва с един и същи възел

**Видове граф -** Ненасочен(undirected) и Насочен(directed)

**Степен на връх** – degree(V)= |edges from V| + |edges to V|

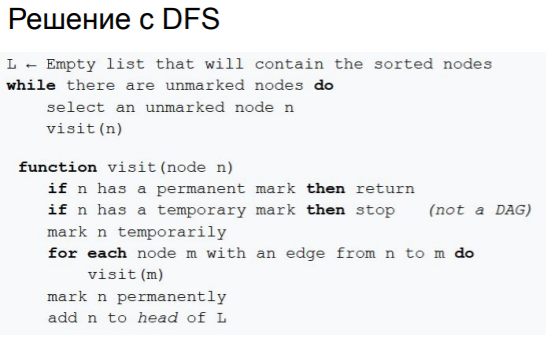


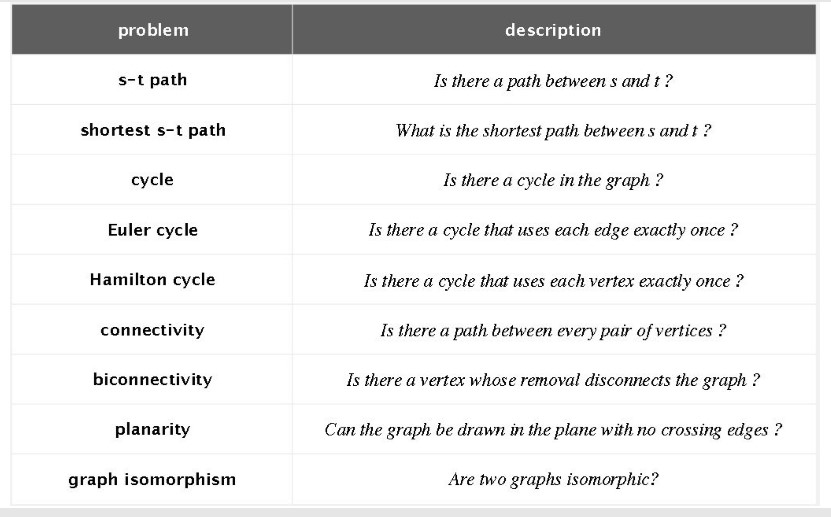


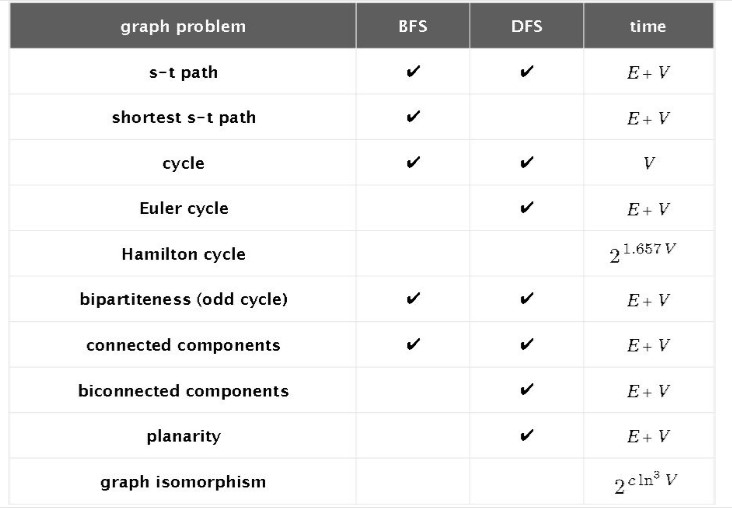


**Топологична наредба на граф -** Задача за насочен ацикличен граф

Топологична наредба: Да се наредят възлите на графа така, така че всяко насочено ребро да започва от възел по-напред в редицата и да отива във възел, който е поназад в редицата.

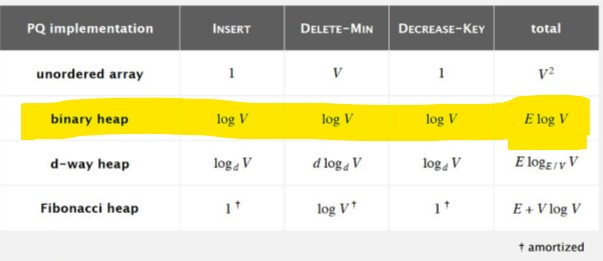






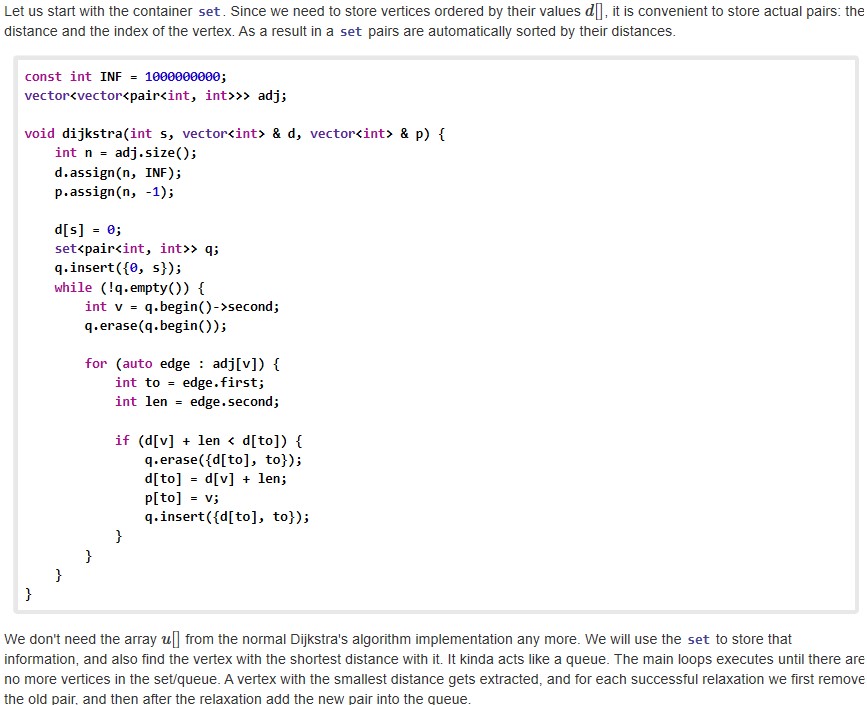
**Лекция 11 -Dijkstra’s Algorithm**

**Основна идея:** Подобен на търсене в ширина, но вместо с опашка с приоритетна опашка.



Set- red-black trees, priority queue- heap

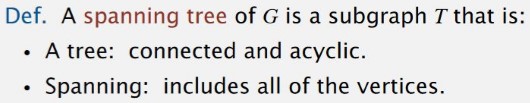




* You can improve the performance a little bit more if you don't store pairs in the containers, but only the vertex indices. In this case we must overload the comparison operator: it must compare two vertices using the distances stored in d [].
* As a result of the relaxation, the distance of some vertices will change. However, the data structure will not resort itself automatically. In fact, changing distances of vertices in the queue, might destroy the data structure. As before, we need to remove the vertex before we relax it, and then insert it again afterwards.
* Since we only can remove from set, this optimization is only applicable for the set method, and doesn't work with priority\_queue implementation. In practice this significantly increases the performance, especially when larger data types are used to store distances, like long long or double.

**Лекция 12 – Minimum Spanning Tree**

**Проблем:** Даден граф да се трансформира към дърво(да няма цикли в графа) оставяйки в графа такива ребра, че сумата им да е минимална.

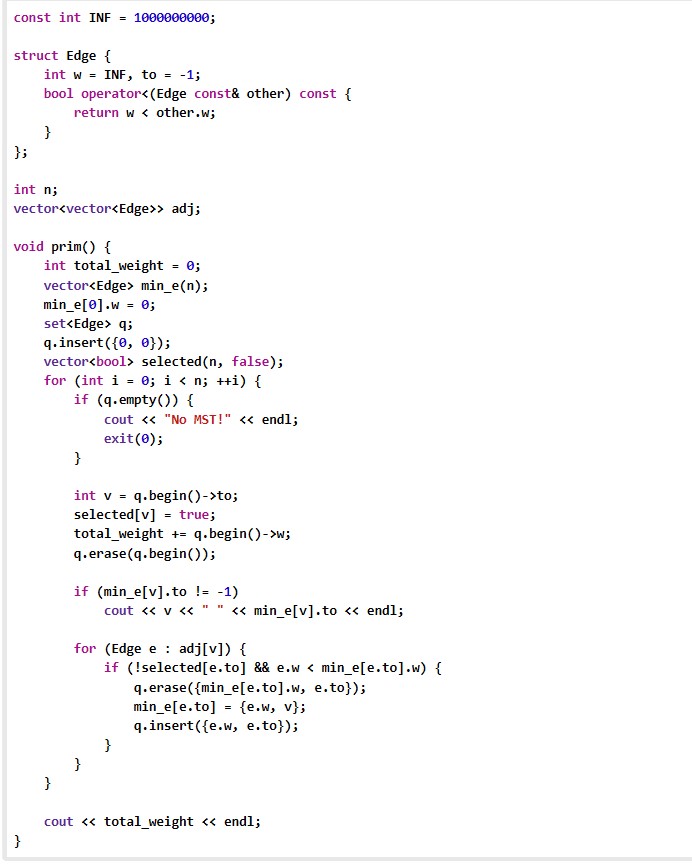
****

**Алгоритъм на Крускал – O(M\*logN),** weighted undirected graph

* Сортираме всички ребра в нарастващ ред.
* Докато покриващото дърво няма V-1 ребра
  + - * Избираме следващо по-големина ребро от графа
      * Ако то не създава цикъл със вече избраните ребра за покриващото дърво, включваме реброто като част от покриващото дърво

**Алгоритъм на Прим – O(M\*logN),** weighted undirected graph

* The minimum spanning tree is built gradually by adding edges one at a time.
* At first the spanning tree consists only of a single vertex (chosen arbitrarily).
* Then the minimum weight edge outgoing from this vertex is selected and added to the spanning tree.
* After that the spanning tree already consists of two vertices.
* Now select and add the edge with the minimum weight that has one end in an already selected vertex (i.e. a vertex that is already part of the spanning tree), and the other end in an unselected vertex.
* And so on, i.e. every time we select and add the edge with minimal weight that connects one selected vertex with one unselected vertex. The process is repeated until the spanning tree contains all vertices (or equivalently until we have *n*−1 edges).
* In the end the constructed spanning tree will be minimal. If the graph was originally not connected, then there doesn't exist a spanning tree, so the number of selected edges will be less than *n*−1.

****

**Лекция 13 – Ойлер и Хамилтон**

**Ойлеров път в граф:** път в граф, който използва всяко ребро в графа точно веднъж.

**Ойлеров цикъл:** цикъл в граф, който използва всяко ребро в графа точно веднъж.

**Ойлеров път -> Ойлеров Цикъл -** Добавяме ребро в графа от началото на Ойлеровият път до края.

**Проверка за съществуване на Ойлеров цикъл**

* Проверяваме, че графа е свързан
* Проверяваме броя на ребрата излизащи от всеки граф.Съществува ойлеров път ако точно 2 върха имат нечетен брой ребра.

**Как да намерим Ойлеров път?** - Алгоритъм на Хелхолзер – O(E)

**Хамилтонов път в граф:** път в ненасочен граф, който включва всички възли от графа, посещавайки всеки от тях точно един път. – с backtracking(O(V!)) (Рекурсивно от всеки връх пробваме всеки следващ възможен и т.н. докато се изчерпат.)– не е открито по-бързо

**P->NP->NP Complete->NP Hard**

**The class P –** decision problems (with yes or no answer), whose answer can be found in O(N^k) time

**The class NP –** decision problems (with yes or no answer), whose answer can be found in O(N^k) time. More precisely, we can verify a specific example of a “yes” answer in O(N^k) time.

**-NP problem:** Проблем за който може да проверим дали дадено решение е вярно за полиномиално време.

**-!!!** **Problems in the complexity class P are generally regarded as “easy”. For typical N, k, can complete execution within a human lifetime.**

**NP Complete Problems-**

We’ll define a problem π as NP-Complete if:

* π is a member of NP.
* π cracks every other problem in NP.

**-NP проблем:** Проблем за който може да проверим дали дадено решение е вярно за полиномиално време.

**-NP Complete problem**: Задача за която ако се намери решение за полиномиално време, то това ще доведе до автоматично намиране на решение за за всички NP проблеми.

**-NP Hard problem:** Проблеми, който са поне толкова трудни колкото най-тежките проблеми в NP (т.е. дори и да се намери полиномиално решение за NP complete проблем, то за тях не означава автоматично, че ще има полиномиално решение за този проблем).

**СДА Конспект(обобщение)**

1. Сложност на алгоритми.Оценка на сложност на алгоритъм. Big O нотацията. NP Complete проблеми.
2. Алгоритми за Сортиране. Бавни алгоритми: Метод на мехурчето, на пряка селекция, сортиране с вмъкване, Бързи алгоритми: сортиране със сливане, бързо сортиране, Алгоритми със ограничения: сортиране с броене
3. Алгоритми за търсене - линейно, двоично
4. Списък, Стек, Опашка - реализация, основни операции и сложности
5. Дървета, Двоично дърво за търсене, Балансирани дървета, Пирамида
6. Хештаблица/хешсет
7. Граф. Обхождане в дълбочина, ширина, Минимално покриващо дърво, алгоритъм на Дейкстра

**Линкове от домашни:**

<https://www.hackerrank.com/contests/practice-1-sda/challenges>

<https://www.hackerrank.com/contests/practice-2-sda/challenges>

<https://www.hackerrank.com/contests/practice-3-sda/challenges>

<https://www.hackerrank.com/contests/practice-4-sda/challenges>

<https://www.hackerrank.com/contests/practice-5-sda/challenges>

<https://www.hackerrank.com/contests/practice-6-sda/challenges>

<https://www.hackerrank.com/contests/practice-7-sda/challenges>

<https://www.hackerrank.com/contests/practice-8-sda/challenges>

<https://www.hackerrank.com/contests/practice-9-sda/challenges>

**Линкове от контролни:**

<https://www.hackerrank.com/contests/sda-2019-2020-test1/challenges/challenge-2212>

<https://www.hackerrank.com/contests/sda-test2/challenges>

<https://www.hackerrank.com/contests/sda-2019-2020-test3/challenges>

<https://www.hackerrank.com/contests/sda-2019-2020-test4/challenges/challenge-2273>

<https://www.hackerrank.com/contests/sda-2019-2020-test-5/challenges/challenge-2298>

<https://www.hackerrank.com/contests/test6sda-renfuvidbviw/challenges/challenge-2326>

**Миналогодишни:**

Домашни работи:  
  
[https://www.hackerrank.com/practice-1-si](https://www.hackerrank.com/practice-1-si?fbclid=IwAR3P4URqv8d-65-RG-D6I9CYlVmimuf6p_TCIstJwt-S52IhWMEunAIBWSg)  
[https://www.hackerrank.com/si-practice-2](https://www.hackerrank.com/si-practice-2?fbclid=IwAR2pO3lERKG2uyznndFAjzFV8iJFtXK_Ihl_JH4F2FFhqJiAahuYCYehOqM)  
[https://www.hackerrank.com/practice-3-1](https://www.hackerrank.com/practice-3-1?fbclid=IwAR2qAlGkC0AbFatoykQoXup18gWA_4NZOu9ZaSWkm_QaGKPFjE3UtxmfocI)  
[https://www.hackerrank.com/si-practice-4](https://www.hackerrank.com/si-practice-4?fbclid=IwAR02ilNcveyW1N8GUtS2LYiiqmOllHerzU2Pp01jCAdnMesP52Gz3dWSsjU)  
[https://www.hackerrank.com/si-practice-5](https://www.hackerrank.com/si-practice-5?fbclid=IwAR2JyMpSBYRe9J78dWjuq9GnpKw4MkGSvbb_gGBAbrj2txQxrU7lLW9Agr0)  
[https://www.hackerrank.com/sda-hw-6](https://www.hackerrank.com/sda-hw-6?fbclid=IwAR2fNOnHEk2T6MVPMTACn1vg4h1qn6HI3gK69-y9TD-VmzePoCWe9wsOQfI)  
[https://www.hackerrank.com/si-practice-7](https://www.hackerrank.com/si-practice-7?fbclid=IwAR1lI-D-uXKR4zcpn9pexSbfDMS0yCxKQbA_s_8BK-bsRluWJgehe27EYJE)  
[https://www.hackerrank.com/sda-hw-8](https://www.hackerrank.com/sda-hw-8?fbclid=IwAR3-k9LVk3wVJGljheU80qdYdPyuuo-y6Y0PY6y8zt5zUogucZbjYgn4s6k)  
[https://www.hackerrank.com/sda-hw-9](https://www.hackerrank.com/sda-hw-9?fbclid=IwAR3Q2LLkbcuxwCN2QghLYDqnLPg4-WYHCGx8tgliqUm8Fcjq4UOSJyYq7Fs)  
[https://www.hackerrank.com/sda-hw-10](https://l.facebook.com/l.php?u=https%3A%2F%2Fwww.hackerrank.com%2Fsda-hw-10%3Ffbclid%3DIwAR22eLwxcMUEVJoMx0VNJcjE9Jx85fn64XUYVun3kmymJPBf8DDl0f34Du8&h=AT08pV0qyMnLMY7iVYe3OwZBP0miPyTbqFfqFxYm6JZ21gaGRM8RhyYANT5vEF03gutHo0lPX4mgeWEL1Q3l4P0MJLQm4WT_QXdMT0IMj9_9A7qE4BELP5WvOro-JbJmhGPFng)  
[https://www.hackerrank.com/sda-hw-11](https://www.hackerrank.com/sda-hw-11?fbclid=IwAR1X36QjycU0DgijF_PmhLUnMJ_W3j3ou-ivEWSKBInM2mr2lguhS3iqeCc)  
  
Контролни работи:  
  
[https://www.hackerrank.com/sda-test](https://www.hackerrank.com/sda-test?fbclid=IwAR1mRJr8nFqlsy9DmzRp8n0duuPsl-OkBKmn53RU-lufaBpBkUHPMsQAooI)  
[https://www.hackerrank.com/sda-ttest-2](https://www.hackerrank.com/sda-ttest-2?fbclid=IwAR3M-wW0leiCAI1PpEzWpB8g-HVat7KRiwFlIRVacLsZD5ZIrVRLKqo2CeM)  
[https://www.hackerrank.com/sdatest3](https://www.hackerrank.com/sdatest3?fbclid=IwAR2JRwCEA9MzpndE_h5NJjj043IPkL1UaDpKtWeO9lMnvqIllB7ZZPWYSfc)  
[https://www.hackerrank.com/test4-sda-](https://www.hackerrank.com/test4-sda-?fbclid=IwAR3M-wW0leiCAI1PpEzWpB8g-HVat7KRiwFlIRVacLsZD5ZIrVRLKqo2CeM)  
[https://www.hackerrank.com/sda-test-5](https://www.hackerrank.com/sda-test-5?fbclid=IwAR2qAlGkC0AbFatoykQoXup18gWA_4NZOu9ZaSWkm_QaGKPFjE3UtxmfocI)  
[https://www.hackerrank.com/test6-sda](https://www.hackerrank.com/test6-sda?fbclid=IwAR2JyMpSBYRe9J78dWjuq9GnpKw4MkGSvbb_gGBAbrj2txQxrU7lLW9Agr0)  
[https://www.hackerrank.com/test7-sda-soft-eng](https://www.hackerrank.com/test7-sda-soft-eng?fbclid=IwAR0KDaOWTv8ucGMZJu060CNrEVPcFDjh3qX9L0dLgTFBn8qvzOA_x9M0pzs)  
  
Подготовка за изпит:  
  
[https://www.hackerrank.com/prepareforfinalexam](https://l.facebook.com/l.php?u=https%3A%2F%2Fwww.hackerrank.com%2Fprepareforfinalexam%3Ffbclid%3DIwAR19FAZBHy9HOOHzNGX4tHcLLDP8C3MLvbczggToi4Ncu_qAGxmVsouzKfk&h=AT08pV0qyMnLMY7iVYe3OwZBP0miPyTbqFfqFxYm6JZ21gaGRM8RhyYANT5vEF03gutHo0lPX4mgeWEL1Q3l4P0MJLQm4WT_QXdMT0IMj9_9A7qE4BELP5WvOro-JbJmhGPFng)  
  
Изпит:  
  
[https://www.hackerrank.com/sda-exam-27-01-19-](https://www.hackerrank.com/sda-exam-27-01-19-?fbclid=IwAR3CGN8o_0UhpGos6NnvhQ3Larz2SrP47NKgkZ6AvFQ0ae9JpvxBcCG88So)