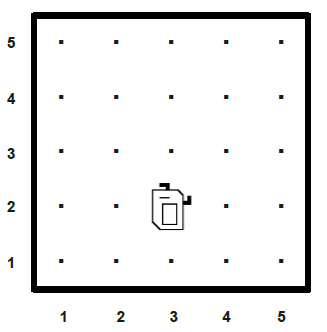
**დავალება 3**

დავალება იწყება ორი გასახურებელი ამოცანით, დავალების არქივს მოყვება მათი ამოხსნებიც, თუმცა ჩვენ გირჩევთ დამოუკიდებლად ამოხსნათ ორივე ამოცანა, რაც დამატებითი გავარჯიშება იქნება. Warm-up ამონცანების დასაბმიტება google classroom ზე არ არის საჭირო.

**Warm-up Problem 0A: Karel Goes Home**

როგორც იცით რობოტი კარელის მართკუთხა სამყარო შედგება ქუჩებისა და გამზირებისგან:



დავუშვათ კარელი დგას მეორე ქუჩისა და მესამე გამზირის გადაკვეთაზე და სურს რომ დაბრუნდეს სათავეში პირველი ქუჩისა და პირველი გამზირის გადაკვეთაზე.

სათავეზე დასაბრუნებლად არსებობს რამდენიმე თანაბრად მოკლე მარშრუტი. მაგალითად:

* მარცხნივ, მარცხნივ, ქვემოთ.
* მარცხნივ, ქვემოთ, მარცხნივ.
* ქვემოთ, მარცხნივ, მარცხნივ.

თქვენი დავალებაა დაწეროთ რეკურსიული ფუნქცია

**int numPathsHome(int street, int avenue)**

რომელიც აბრუნებს რიცხვს თუ რამდენნაირად შეუძლია კარელს სათავეზე დაბრუნება განსაზღვრული საწყისი პოზიციიდან. რა საკვირველია, კარელს არ სურს ზედმეტი ნაბიჯების გადადგმა, აქედან გამომდინარე მოძრაობს მხოლოდ დასავლეთითა და სამხრეთით (მარცხნივ და ქვემოთ დიაგრამის მიხედვით)

**Warm-up Problem 0B: Random Subsets**

როგორც ლექციის მასალებიდან ვნახეთ n ელემენტისთვის შესაძლო ქვესიმრავლეების რაოდენობა არის 2n, რაც პატარა n ებისთვისაც საკმაოდ დიდი რიცხვია.

რა მოხდება თუ ყველა შესაძლო ქვესიმრავლის დაგენერირების მაგივრად, მხოლოდ რამდენიმე შემთხვევით ქვესიმრავლეს დავაგენერირებთ? ეს მოგვცემს საშუალებას დავინახოთ რა ტიპის სიმრავლეებთან გვაქვს საქმე.

სიმრავლიდან შემთხვევითი ქვესიმრავლის დაგენერირება მარტივად შეიძლება, თუ საწყისი სიმრავლის თითოეული ელემენტისთვის მონეტის აგდებით გადავწყვიტავთ კონკრეტული ელემენტი შედის თუ არა საბოლოო ქვესიმრავლეში.

დავწეროთ ფუნქცია

**Set<int> randomSubsetOf(Set<int>& s);**

რომელიც იღებს ინფუთს **Set<int>** და აბრუნებს შემთხვევით ქვესიმრავლეს. თქვენი ალგორითმი უნდა იყოს რეკურსიული და არ იყენებდეს არცერთ ციკლს (for, while, და ა.შ.)

**ამოცანა 1: ქვემიმდევრობები.**

S სტრინგი არის T სტრინგის ქვემიმდევრობა, თუ S ის ყველა სიმბოლო იგივე მიმდევრობით გვხვდება T ში. მაგალითად, სტრინგი pin ქვემომდევრობაა სტრინგისა programming, და სტრინგი singe ქვემიმდევრობაა springtime. ამასთანავე, steal არ არის ქვემიმდევრობა least-ისა, რადგან ასოები არასწორი წყობითაა და i არ არის team-ის ქვემიმდევრობა, რადგან სიტყვა team არ შეიცავს ასოს i. ცარიელი სტრინგი ყოველი სტრინგის ქვემიმდევრობაა, რადგან ცარიელი სტრინგის 0 ასოები, ნებისმიერი სტრინგის მიმდევრობას აკმაყოფილებს.

დავწეროთ ფუნქცია

**bool isSubsequence(string text, string subseq)**

რომლსაც გადაეცემა ორი სტრინგი და აბრუნებს არის თუ არა მეორე სტრინგი პირველის ქვემიმდევრობა. თქვენი ამოხსნა უნდა იყოს რეკურსიული და არ იყენებდეს ციკლებს (for, while, და ა.შ.) ამ ამოცანას ლამაზი რეკურსიული დეკომპოზიცია აქვს. მინიშნებისთვის ყურადრება გაამახვილეთ ორ საკითხზე:

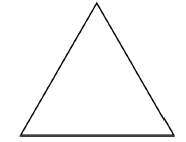
* რომელი სტრინგებია ცარიელი სტრინგების ქვემიმდევრობები?
* რა ხდება მაშინ, როცა ქვემომიმდევრობის პირველი ასო ემთხვევა ტექსტის პირველ ასოს? და რა ხდება თუ ასე არაა?

ჩათვალეთ რომ სტრინგები დიდი და პატარა ასოების მიმართ მგრძნობიარენი არიან, ანუ AGREE არ არის ქვემიმდევრობა სტრინგისა agreeable.

**ამოცანა 2: The Sierpinski Triangle**

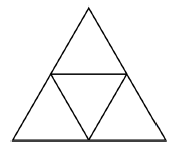
ფრაქტალების დიზაინის ერთერთი მაგალითია სერპინსკის სამკუთხედი, შექმნილი პოლონელი მათემატიკოსის ვაცლავ სერპინსკის (1882-1969) მიერ.

ნულოვანი რიგის სერპინსკის სამკუთხედი არის ტოლგვერდა სამკუთხედი.



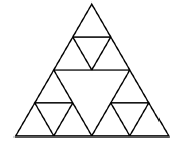
იმისათვის რომ შევქმნათ N რიგის სერპინსკის სამკუთხედი, უნდა დავხატოთ სამი N-1 რიგის სამკუთხედი, ისე რომ თითოეულ მათგანის გვერდი იყოს თავდაპირველი სამკუთხედის გვერდის ნახევრის ტოლი, ამავდროულად ეს 3 სამკუთხედი, დიდი სამკუთხედის კუთხეებში უნდა იყოს განლაგებული.

განმარტების თანახმად პირველი რიგის სერპინსკის სამკუთხედი, გამოიყურება ასე:

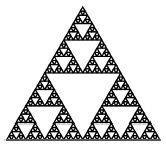


წვერით ქვემოთ მიმართული სამკუთხედი, არ არის ჩვენ მიერ დახატული, თუმცა წარმოქმნილია 3 სამკუთხედის გვერდების მიერ.

ამგვარად, მეორე რიგის სერპინსკის სამკუთხედს აქვს შუაში იგივე ღია სივრცე:



თუ ამ პროცესს გააგრძელებთ დამატებით 3 რეკურსიული დონით, მიიღებთ მეხუთე რიგის სერპინსკის სამკუთხედს, რომელიც ასე გამოიყურება:



დაწერეთ პროგრამა რომელიც ითხოვს კონსოლიდან გვერდის სიგრძისა და ფრაქტალური რიგირს შეყვანას და შედეგად ხატავს სერპინსკის სამკუთხედს, გამოტანილი ფანჯრის შუაში. აქ გამოგადგებათ GWindows-ს ფუნქცია: drawPolarLine და გაეცნოთ ფაქტს რომ drawPolarLine აბრუნებს GPoint, რომელიც ხაზის დასასრულს აღნიშნავს.

**ამოცანა სამი: Inverse Genetics**

არსებობს სამი ძირითად მაკრომოლეკულა:

1. **დნმ** - იგივე დეზოქსირიბონუკლეინის მჟავა
2. **ცილა**
3. **რნმ** - იგივე რიბონუკლეინის მჟავა

**რნმ** წარმოადგენს ოთხი განსხვავებული **ნუკლეოდიტისგან** აგებულ მიმდევრობას, ნუკლეოდიტები ავღნიშნოთ შემდეგი სიმბოლოებით: A, C, U და G. მიმდევრობით სამი ნუკლეოდიტი აყალიბებს **კოდონს**(გენეტიკური კოდის ერთეული, ნუკლეოტიდური ნაშთების სამეული), რომელიც აკოდირებს **ამინო მჟავას**. ზოგიერთი ამინო მჟავების მიმდევრობა წარმოქმნის სხვადასხვა ცილას.

**ნუკლეოდიტების** მსგავსად **ამინო მჟავებიც** აღინიშნება სიმბოლოების საშუალებით, N აღნიშნავს ამინო მჟავა asparagine-ს, Q აღნიშნავს glutamine-ს, I აღნიშნავს isoleucine-ს და ა.შ.

ყოველი **კოდონი** აკოდირებს რაიმე ამინო ამინო მჟავას, მაგრამ ზოგიერთი კოდონი აკოდირებს ერთი და იგივე მჟავას. მაგალითად კოდონები UUA, UUG, CUU, CUC, CUA, და CUG ყველა ერთი და იგივე მჟავას leucin-ს აკოდირებს.

ეს ინფორმაცია შეგვიძლია წარმოვადგინოთ **Map<char, Set<string> >** სახით, სადაც გასაღები არის ამინო მჟავის შესაბამისი სიმბოლო, ხოლო მნიშვნელობა კოდონების სიმრავლე, რომელებიც კოდირდებიან მითითებულ მჟავაში.

L { UUA, UUG, CUU, CUC, CUA, CUG }

S { AGC, AGU, UCA, UCC, UCG, UCU }

K { AAA, AAG }

W { UGG }

როგორც ზემოთ ვთქვით ცილა მიიღება ამინო მჟავების კომბინაციით, ერთი და იგივე ამინო მჟავა შეიძლება მივიღოთ კოდონების სხვადასხვა კომბინაციით, შესაბამისად ცილაც შეიძლება მივიღოთ სხვადასხვა კოდონების მიმდევრობით.

მაგალითად ცილა რომელიც არის KWS ამინომჟავების მიმდევრობა, შეიძლება მივიღოთ კოდონების შემდეგი კომბინაციებით:

AAAUGGAGU AAAUGGUCC AAGUGGAGC AAGUGGUCC

AAAUGGAGC AAAUGGUCG AAGUGGAGU AAGUGGUCG

AAAUGGUCA AAAUGGUCU AAGUGGUCA AAGUGGUCU

თქვენი ამოცანაა დაწეროთ **listAllRNAStrandsFor** ფუნქცია, რომელიც ბეჭდავს გადაცემული ცილისთვის ყველა შესაძლო კომბინაციას

**void listAllRNAStrandsFor(string protein, Map<char, Set<string> >& codons)**

**protein** - ცილა, ამინო მჟავების შესამაბისი სიმბოლოების ერთობლიობა

**codons** - მეფი, რომელშიც ამინო მჟავას შეესაბამება კოდონების სიმრავლე

შეგიძლიათ ჩათვალოთ, რომ ყველა ასო ცილაში არის სწორი ამინო მჟავა, ასე რომ, თქვენ ვერ ნახავთ ასოს ცილაში, რომელიც არ არის map-ში. კომბინაციების დაბეჭდვის მიმდევრობას არ აქვს მნიშვნელობა, მთავარია ერთი კომბინაცია მხოლოდ ერთხელ დაბეჭდოთ.

**ამოცანა ოთხი: Universal Health Coverage**

ჯანდაცვის სამინისტროს აქვს სურვილი ააშენოს საავადმყოფოები ისე რომ ყველა ქალაქი უზრუნველყოფილი იყოს მაღალი ხარისხის საავადმყოფოთი. თუმცა როგორც სამინისტროს არ გააჩნია დიდი ბიუჯეტი ამიტომ დაწესებულია შეზღუდვა საავადმყოფოების რაოდენობაზე.

მოცემული გვაქვს შემდეგი ინფორმაცია:

* ქალაქების სია
* თითოეული საავადმყოფოსთვის ქალაქების სიმრავლე რომელთა მომსახურებასაც შეძლებს აშენების შემთხვევაში
* შეზღუდვა, მაქსიმუმ რამდენი საავადმყოფოს აშენება შეგვიძლია

თქვენი ამოცანაა დაეხმაროთ სამიანისტროს გაარკვიოს შესაძლებელია თუ არა დაშვებული რაოდენობის საავადმყოფოების აშენება ისე რომ ყველა ქალაქი დაიფაროს, და თუ შესაძლებელია იპოვოთ შესამაბისი სია. ამისათვის თქვენ უნდა დაწეროთ **canOfferUniversalCoverage** ფუნქცია:

**bool canOfferUniversalCoverage(Set<string>& cities, Vector< Set<string> >& locations,**

**int numHospitals, Vector< Set<string> >& result)**

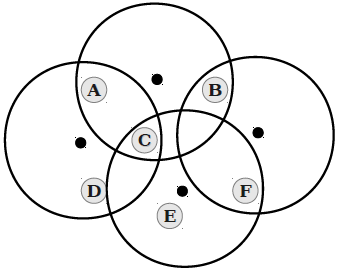
**cities** - ქალაქების სია, სიმრავლის სახით

**locations** - საავადმყოფოებისთვის, ქალაქების სიმრავლე რომელსაც ფარავს

**numHospitals** - მაქსიმალური დასაშვები რაოდენობა

**result** - თუ შესაძლებელია ყველა ქალაქის დაფარვა, **result** ვექტორში უნდა ჩაიწეროს შერჩეული საავადმყოფოების შესაბამისი სიმრავლეები **cities** ვექტორიდან. თუ შეუძლებელია დატოვეთ **result** ცარიელი.

მაგალითისთვის, სურათზე ასოებით აღნიშნულია ქალაქები(A, B, C, D, E, F). ხოლო შავი წერტილებით აღნიშნულია საავადმყოფოების სავარაუდო ადგილმდებარეობები:



ამ სურათზე, ყველა საავადმყოფოს შეუძლია უზრუნველყოს ქალაქები, რომლებიც მოექცნენ მის გარშემო შემოხაზული წრის შიგნით.

ამ შემთხვევისთვის თქვენს ფუნქციას გადმოეცემა შემდეგი ინფორმაცია:

* **cities = { "A", "B", "C", "D", "E", "F" }**
* **locations = ⟨ {"A", "B", "C"}, {"A", "C", "D"}, {"B", "F"}, {"C", "E", "F"} ⟩**

სულ ზევით არსებულ საავადმყოფოს, შეუძლია მოემსახუროს A, B და C ქალაქებს. მარცხენა საავადმყოფო ფარავს A, C და D ქალაქებს. მარჯვენა საავადმყოფო ფარავს B და F-ს, ხოლო ქვედა - C, E და F-ს.

როგორც ხედავთ მხოლოდ ორი საავადმყოფოს აშენებით ყველა ქალაქის დაფარვა შეუძლებელია, თუმცა თუ შეზღუდვა იქნება 3 საავადმყოფო, მაშინ არსებობს ამის გზა – ააშენეთ ზედა საავადმყოფო, რათა დაფაროს A, B და C ქალაქები, ქვედა C, E დაF-ის დასაფარად, და მარცხენა A, C, და D-ს დასაფარად. შესაბამისად ფუნქციამ **result** უნდა შეავსოს შემდეგნაირად:

**result = {{"A", "B", "C"}, {"C", "E", "F"}, {"A", "C", "D"}}**

**General notes**

* თქვენ მოცემული გაქვთ ფუნქციების პროტოტიპები ოთხივე ამოცანისთვის. ფუნქციებს, რომლებსაც დაზერთ უნდა ჰქონდეთ *ზუსტად –* იგივე სახელები, არგუმენტები, დასაბრუნებლი ტიპები. თქვენი ფუნქცია ასევე აუცილებლად უნდა იყენებდეს რეკურსიას, მიუხედავად იმისა გაქვთ თუ არა ალტერნატიული ამოხსნის გზა რეკურსიის გარეშე.
* ყველა ამოცანისთვის, კარგად გატესტეთ თქვენი ფუნქცია, რომ დარწმუნდეთ ყველა შესაძლო გადმოცემული პარამეტრებისთვის სწორად მუშაობს. მაგალითად, Karel warm-up-ის შემთხვევაში, შეიძლება დაწეროთ პროგრამა, რომელიც იძახებს **numPathsHome** ფუნქციას ისეთი პარამეტრებისთვის, რომელთა პასუხიც წინასწარ იცით, ან ისეთი პროგრამა, რომელიც მომხმარებელს საშუალებას აძლევს თავისით შეიყვანოს მათი მნიშვნელობები. კოდის მსგავსი ტესტვა საჭიროა იმისათვის, რომ დარწმუნდეთ გათვალისწინებული გაქვთ ყველა შემთხვევა. თქვენ შეგიძლიათ დავალების ფაილში დატოვოთ „გამტესტავი“ კოდი, მათი წაშლა არ არის საჭირო.
* რეკურსია საკმაოდ რთული თემაა, ასე რომ არ შეშინდეთ, თუ დაჯდომისთანავე ვერ ამოხსნით ამ ამოცანებს. დახარჯეთ დრო იმისთვის, რომ ჩამოაყალიბოთ თუ როგორ არის ბუნებით რეკურსიული თითოეული ამოცანა და როგორ შეგიძლიათ ამოხსნათ ამოცან თუ მოცემული გაქვთ უფრო პატარა, ქვეამოცანის ამოხსნა. ყურადღება მიაქციეთ base case-ს რათა არ ჩაიციკლოთ რეკურსიაში.
* მას შემდეგ რაც ისწავლით რეკურსიულად ფიქრს, რეკურსიული ამონახსნები გახდება ძალიან ინტუიტიური და პირდაპირი. დახარჯეთ დრო ამ ამოცანებზე და თქვენ იქნებით ბევრად უკეთ მომზადებული შემდეგი დავალებისთვის, სადაც თქვენ დააიმპლემენტირებთ სხვა რეკურსიულ ალგორითმებს.

**Possible Extensions**

Want more to explore? Here are a few suggestions to help you get started:

**Subsequences**: An interesting variation on this problem is the following: given a collection of strings, what is the smallest string S such that every string in the collection is a subsequence of S? Try seeing how you might solve this problem.

**Sierpinski Triangle:** Although this assignment is all about recursion, if you're up for a challenge, try replacing the recursive version of your code to draw an order-n Sierpinski triangle with an iterative function that draws an order-n Sierpinski triangle.

**Inverse Genetics:** Suppose you have an RNA strand that encodes a particular protein. A point mutation is a chance to an RNA strand where one letter is replaced with a different letter. Write a program that computes the probability that a single point mutation to that strand will cause the strand to encode a different protein.

**Universal Health Coverage:** Suppose your objective now is to provide coverage to the maximum number of people. Update your function so that each city is annotated with a total population, then have your function find the maximum number of people that can be covered.