Αλγόριθμοι και Πολυπλοκότητα

2η Σειρά Γραπτών Ασκήσεων

Ονοματεπώνυμο: Λιαροκάπης Αλέξανδρος
Αριθμός Μητρώου: 03114860

Άσκηση 1

Ο αλγόριθμος θα παίρνει ως είσοδο ένα ζεύγος συνόλων κλειδιών και κλειδαριων και θα δίνει ως έξοδο μία λίστα ζευγαριών κλειδιών-κλειδαριών αυξανόμενου μεγέθους.

```
sortKeysLocks :: (Set Key, Set Lock) -> [(Key, Lock)]
sortKeysLocks (Empty, Empty) = []
sortKeysLocks data =
    let
       pivot = choosePivot data
       lessThanPivot = keepSmaller pivot data
       greaterThanPivot = keepLarger pivot data
    in
       sortKeysLocks lessThanPivot ++ [pivot] ++ sortKeysLocks greaterThanPivot

choosePivot :: Set Key -> Set Lock -> (Key, Lock)
keepSmaller :: (Key, Lock) -> (Set Key, Set Lock) -> (Set Key, Set Lock)
keepLarger :: (Key, Lock) -> (Set Key, Set Lock) -> (Set Key, Set Lock)
```

Εξετάζοντας τον αλγόριθμο βλέπουμε πως έχει την ίδια μορφή με τον quicksort. Αρκεί να αποδείξουμε πως οι συναρτήσεις choosePivot, keepSmaller και keepLarger έχουν πολυπλοκότητα $O(n)$ και η ανάλυση θα ανάγεται στην ανάλυση του quicksort που έχει γνωστή πολυπλοκότητα $O(nlogn)$ στη μέση περίπτωση.

Η choosePivot μπορεί να διαλέξει τυχαία ένα κλειδί και μετά να το δοκιμάσει σε κάθε κλειδαριά έτσι ώστε να βρεί την αντίστοιχη κλειδαριά. Έχει προφανώς πολυπλοκότητα $O(n)$.

Η keepSmaller φιλτράρει δύο φορές, μία φορά για τα κλειδιά και μία για τα λουκέτα. Αυτο για τα κλειδιά το επιτυγχάνει βρίσκοντας ποια χωράνε στην pivot κλειδαριά, ενώ για τα λουκέτα δοκιμάζει σε ποια δε χωράει το pivot κλειδί. Η πολυπλοκότητα της παραπάνω διαδικασίας είναι $O(n)$ αφού χρειάζεται να διασχισει γραμμικά μία φορά κάθε σύνολο n στοιχείων.

Η keepLarger λειτουργεί με τρόπο αντίστοιχο της keepSmaller και επομένως έχει επίσης πολυπλοκότητα $O(n)$.

Άσκηση 2

Ο Αλγόριθμος θα παίρνει ως είσοδο ένα puzzle, τα όρια της περιοχής που μας ενδιαφέρει και την τοποθεσία του απαγορευτικού τετραγνώνου και θα γεμίζει την περιοχή με κατάλληλα κομμάτια.

```
//checks if limits describe the base case
isTwoByTwo :: Limits -> Bool

//puts puzzle piece on given adjacent points
fillPoints :: Puzzle -> (Point, Point, Point) -> Puzzle

//splits limits to four quadrants
splitToQuadrants :: Limits -> (Limits, Limits, Limits, Limits)

//returns fake forbiddens for all quadtrants, first limits is
//given forbidden's quadrant. The forbiddens are adjacent.
getMockForbiddens :: Point -> (Limits, Limits, Limits, Limits)
                          -> (Limits, (Limits,Point), (Limits,Point), (Limits,Point))

fillPuzzle :: Puzzle -> Limits -> Point -> Puzzle
fillPuzzle puzzle limits forbidden
    | isTwoByTwo limits = .. //base case
    | otherwise =
        let
          quadrants = splitToQuadrants limits forbidden
          (limitsOfForbidden,
            (limitsa, forbiddena),
            (limitsb, forbiddenb),
            (limitsc, forbiddenc)) = getMockForbiddens forbidden quadrants
          filledOfForbbiden = fillPuzzle puzzle limitsOfForbidden forbidden
          andFilledA = fillPuzzle filledOfForbbiden limitsa forbiddena
          andFilledB = fillPuzzle andFilledA limitsb forbiddenb
          andFilledC = fillPuzzle andFilledB limitsc forbiddenc
        in
          putPiece andFilledC (forbiddena, forbiddenb, forbiddenc)
```

Για puzzle Μεγέθους 2×2 ο αλγόριθμος προφανώς δουλεύει. Αν δουλεύει για μεγέθους $2^i \times 2^i$ ο αλγόριθμος θα δουλεύει για $2^{i+1} \times 2^{i+1}$ αφού το puzzle σπάει σε 4 κομμάτια μεγέθους $2^i \times 2^i$ τα οποία γεμίζονται αναδρομικά. Το κομμάτι που περιέχει το απαγορευμένο κομμάτι θα γεμίσει πλήρως εκτός απο αυτό, ενώ στα άλλα τρία κομμάτια βάζουμε ψεύτικα απαγορευτικά κομμάτια στην κοινή τους γωνία. Στο τέλος και τα άλλα τρία θα γεμίσουν πλήρως εκτός απο τα κομμάτια της κοινής τους γωνίας τα οποία μπορούμε και να γεμίσουμε με ένα κομμάτι του puzzle.

Άσκηση 3

Ο Αλγόριθμος θα παίρνει ως είσοδο ένα sorted set δορυφόρων με βάση ταξινόμησης την ενέργειά τους, ένα sorted set πλανητών με βάση ταξινόμησης την ασπίδα τους και θα βγάζει την αντιστοιχία τους.

```
maximizeAssignment :: Int n -> SortedSet Satellite -> SortedSet Planet
                           -> Map Satellite Planet
maximizeAssignment satellites planets =
    let
        planetWithMaxValue = getMax planets
        satelliteJustAboveShield = getJustAbove (shield planetWithMaxValue) satellite
    in
        case satelliteJustAboveShield of
            Just satellite -> { satellite => planetWithMaxValue } U
                    maximizeAssignment (satellites - satellite)
                                       (planets - planetWithMaxValue)
            Nothing -> let
                           satelliteWithMinPower = getMin satellites
                       in
                           maximizeAssignment (satellites - satelliteWithMinPower)
                                              (planets - planetWithMaxValue)
```

Έστω,

$$\mathbb{P} : \text{σύνολο συνόλων πλανητών}$$

$$\mathbb{D} : \text{σύνολο συνόλων death stars}$$

$$(\forall P \in \mathbb{P}), v : P \Rightarrow \mathbb{Z}, \text{τιμή πλανήτη}$$

$$(\forall P \in \mathbb{P}), s : P \Rightarrow \mathbb{Z}, \text{ασπίδα πλανήτη}$$

$$(\forall D \in \mathbb{D}), f : D \Rightarrow \mathbb{Z}, \text{δύναμη ακτίνας}$$

$$m : (\mathbb{P}, \mathbb{D}) \Rightarrow \mathbb{Z}, \text{συνολικό κέρδος αντιστοίχισης}$$

$$\text{Έστω } D \in \mathbb{D}, P \in \mathbb{P}$$

$$\text{Συμβολίζουμε } d_i \in D \text{ έτσι ώστε } i < j \Rightarrow f(d_i) \leq f(d_j)$$

(Θεώρημα 1)

Προφανώς άμα πρέπει να χάσουμε απο τη διάθεση μας ένα death star θα βελτιστοποιήσουμε το συνολικό κέρδος άμα χάσουμε το πιο αδύναμο. Άρα,

$$\max\{m(P, D - d_i), i \in [a, b]\} = m(P, D - d_a)$$

Έστω $p_{max} \in P : v(p_{max}) = \max\{v(p), p \in P\}$

(Θεώρημα 2)

Έστω πως $\exists d_i : f(d_i) \geq s(p_{max})$ τότε ο p_{max} πάντα είναι βέλτιστο να καταστραφεί απο κάποιο death star. Διαφορετικά μπορούμε να ανταλλάξουμε τον πλανήτη που αντιστοιχείται με τον d_i με τον p_{max} και να έχουμε καλύτερη αντιστοίχηση.

(Θεώρημα 3)

Έστω πως $\nexists d_i : f(d_i) \geq s(p_{max})$ τότε ο p_{max} δεν θα συνεισφέρει και θα έχουμε $m(P, D) = m(P - p_{max}, D - d_j)$ για κάποιο $d_j \in D$ που αντιστοιχείται με τον p_{max}. Όμως απο το Θεώρημα 1 ξέρουμε πως για να μεγιστοποιηθεί το $m(P, D)$ τότε $d_j = d_{min}$ όπου $f(d_{min}) = \min\{f(d), d \in D\}$

(Θεώρημα 4)

Έστω πως $\exists d_i : f(d_i) \geq s(p_{max})$ τότε σύμφωνα με το Θεώρημα 2 θα πρέπει να αντιστοιχήσουμε στον p_{max} ένα death star $d_j, j \in [a, b]$. Έτσι θα εχουμε $m(P, D) = v(p_{max}) + m(P - p_{max}, D - d_j)$ το οποίο σύμφωνα με το Θεώρημα 1 μεγιστοποιείται για $d_j = d_a$.

Σύμφωνα με τα Θεωρήματα 3 και 4 έχουμε

$$m(P, D) = \left\{ \begin{array}{ll} v(p_{max}) + m(P - p_{max}, D - d_m), & \exists d_m : m = \min\{i, f(d_i) \geq s(p_{max})\} \\ m(P - p_{max}, D - d_{min}), & \text{διαφορετικά} \end{array} \right\}$$

Ο αλγόριθμος που παρατίθεται στην αρχή είναι μία απλή τροποποίηση της παραπάνω σχέσης έτσι ώστε να διατηρούνται οι αντιστοιχήσεις. Με τις κατάλληλες δομές (πχ std::set) όλες οι αναζητήσεις μπορούν να γίνουν σε $O(logn)$ χρόνο. Έτσι ο αλγόριθμος έχει πολυπλοκότητα $O(nlogn)$.

Άσκηση 4

1. Ο αλγόριθμος είναι greedy φύσεως, παίρνει ως είσοδο μία λίστα απο x-συντεταγμένες σπιτιών και την ακτίνα των κεραιών και επιστρέφει μία λίστα απο x-συντεταγμένες κεραιών:

```
antennaLocs :: [Int] -> Int -> [Int]
antennaLocs = antennaLocs' (-2*radious-1)
    where antennaLocs' :: Int -> [Int] -> Int -> [Int]
          antennaLocs _ [] _ = []
          antennaLocs leftestCoord (hc : rest) radius
            | hc - leftestCoord > 2 * radius = (hc + radius) : antennaLocs hc rest radius
            | otherwise = antennaLocs leftestCoord rest radius
```

Ο παραπάνω αλγόριθμος έχει πολυπλοκότητα $O(n)$.

2. Ο αλγόριθμος του πρώτου μέρους βασίζεται στο ότι υπάρχει μία καθορισμένη αρχή στην διάταξη των σπιτιών. Σε έναν κύκλο ωστόσο μπορεί να υπάρχει επικάλυψη της τελικής και της αρχικής κεραίας και έτσι δεν υπάρχει εύκολος τρόπος επιλογής αρχής. Ένας τρόπος επίλυσης είναι να διαλέξουμε κάθε σπίτι ως αρχή και να διαλέξουμε αυτην για την οποία ελαχιστοποιείται ο αριθμός των κεραιών σύμφωνα με τον παραπάνω αλγόριθμο. Αυτός ο αλγόριθμος έχει πολυπλοκότητα $O(n^2)$.

Άσκηση 5

Θεωρούμε $\alpha(k,n)$ η οποία επιστρέφει τα λιγότερα βήματα στη χειρότερη περίπτωση. Προφανώς $\alpha(1,n) = n-1$. Έστω πως υπάρχει ύψος $h(k,n)$ απο το οποίο παίρνουμε τον βέλτιστο αριθμό βημάτων. Τότε θα πρέπει να ισχύει:

$$\alpha(k,n) = 1 + \max(\alpha(k-1, h(k,n)), \alpha(k, l - h(k,n)))$$

(Θέωρημα 1)

Έστω h τυχαίο βήμα. Αν το αυξήσω κατα ένα αυξάνω το πολύ κατα 1 τα βήματα της περίπτωσης που το ποτήρι σπάει. Αν το μειώσω κατα 1, μειώνω το πολύ κατα 1 τα βήματα της περίπτωσης που το ποτήρι δεν σπάει.

(Θεώρημα 2)

Οι δύο παράμετροι του max θα πρέπει είτε να είναι ίσες είτε να διαφέρουν κατα 1. Έστω πως δεν ισχύει και $\alpha(k-1, h(k,n)) = \alpha_1$ και $\alpha(k-1, n-h(k,n)) = \alpha_2$ με $|a_1 - a_2| \geq 1$. Τότε μπορώ να διαλέξω $h'(k,n) = h(k,n) \pm 1$ έτσι ώστε να μειωθούν κατα 1 το πολύ τα βήματα της μέγιστης περίπτωσης και να αυξηθούν κατα 1 το πολύ τα βήματα της ελάχιστης. Αν μειωθεί έστω και μία απο τις παραμέρους, και αφού έχουμε θεωρήσει πως απέχουν τουλάχιστον κατα 2, η max τιμή θα μειωθεί σίγουρα και επομένως θα έχουμε βρει καλύτερο βέλτιστο βήμα το οποίο είναι άτοπο. Αν δεν μειωθεί μπορούμε να συνεχίσουμε να αλλάζουμε το βήμα μέχρι να βρεθούμε στην προηγούμενη περίπτωση.

(Θεώρημα 3)

Θα υπάρχει $h_1(k,n)$ τέτοιο ώστε $\alpha(k,n) = 1 + \alpha(k-1, h_1(k,n))$ και $h_2(,n) : \alpha(k,n) = 1 + \alpha(k, n - h_2(k,n))$. Απο θεώρημα 2 ξέρουμε πως οι διαφορά των δύο παραμέτρον για κάθε $h(k,n)$ θα είναι είτε 0 είτε 1. Άμα είναι 0 το παραπάνω είναι αυταπόδεκτο. Για την περίπτωση που η διαφορά είναι 1 τότε μπορώ να διαλέγω κατάλληλα διαδοχικά εναλλακτικά βήματα και η διαφορά θα μειώνεται κατα το πολύ 2. Άμα μειωθεί κατα 0 τότε διαλέγω άλλο βήμα, άμα μειωθεί κατα 1 τότε οι δύο παράμετροι θα είναι ίδιοι και επομένως θα ισχύει το θεώρημα. Άμα μειωθεί κατα 2 τότε οι δύο παράμετροι θα αλλάξουν τιμές και θα έχω βρει κατάλληλα βήματα.

1. Έστω πως x ο βέλτιστος αριθμός βημάτων. Τότε απο θεώρημα 3 θα υπάρχουν $h_1(2,n), h_2(2,n)$:

$$\alpha(2,n) = 1 + \alpha(1, h_1(2,n)) = h_1(2,n) = x$$
$$\alpha(2,n) = 1 + \alpha(2, n - h_2(2,n)) = x$$

Έστω πως $h_1(2,n) \neq h_2(2,n)$, τότε $h_2(2,n) < h_1(2,n)$. Έστω πως δοκιμάζοντας στη θέση $h_1(2,n)$, το ποτήρι δεν σπάει, τότε σύμφωνα με θεώρημα 2 θα πρέπει $\alpha(2, n - h_1(2,n)) = x - 1$ ή $x - 2$. Έστω πως δοκιμάζοντας στη θέση $h_2(2,n)$ το ποτήρι δεν σπάει. Τότε υποχρεωτικά θα πρέπει $\alpha(2, n - h_2(2,n)) = x - 1$ διαφορετικά οι βέλτιστες κινήσεις δεν θα ήταν x. Όμως $\alpha(2, n - h_2(2,n)) \geq \alpha(2, n - h_1(2,n))$ αφού $h_2(2,n) < h_1(2,n)$ και επομένως $\alpha(2, n - h_1(2,n)) = x - 1$.

Απο την τελευταία σχέση αποκτάμε αλυσίδα βημάτων: $x, (x-1), (x-2), .., 2, 1$ η οποία είναι x στο πλήθος $(\alpha(2,n) = x)$. Έτσι εξασφαλίζοντας πως όλες οι θέσεις που παίρνουμε με τα διαδοχικά βήματα είναι μέσα στο εύρος μας, έχουμε:

$$x + (x-1) + (x-2) + .. + 1 \leq n$$
$$\frac{x(x-1)}{2} \leq n$$
$$x = \left\lceil \frac{-1 + \sqrt{1 + 8n}}{2} \right\rceil$$

Για $n = 100$ έχουμε $x = 14$.

2. Σαν μία απλή λύση μπορούμε να υλοποιήσουμε απευθείας την αναδρομή της $\alpha(k,n)$ χρησιμοποιόντας δυναμικό προγραμματισμό για να μην υπολογίζουμε όλα τα υποπροβλήματα. Στην γενική περίπτωση είναι δύσκολο να υπολογίσουμε το βέλτιστο βήμα οπότε καταφεύγουμε σε εξαντλητική γραμμική αναζήτηση του βέλτιστου βήματος:

```
solveA : Int -> Int -> Int (Cached)
solveA (1, n) = (n,1)
solveA (k, 1) = (1,1)
solveA (k, 0) = (0,0)
solveA (k, n) = minimum [ max(1 + solveA(k-1, s), 1 + solveA(k, n - s)), s <- [1 .. n] ]
```

Κάθε υποπρόβλημα θα υπολογιστεί μόνο μία φορά ενώ θα χρειαστεί n επαναλήψεις για να βρεί το κατάλληλο βέλτιστο βήμα. Ο αλγόριθμος μπορεί να τροποποιηθεί έτσι ώστε να αποθηκεύεται και το βέλτιστο βήμα. Έτσι θα έχει πολυπλοκότητα $O(kn^2)$.

2. Έστω $\beta(t,k)$ συνάρτηση που επιστρέφει το μέγιστο εύρος που μπορεί να καλυφθεί με k ποτήρια και t προσπάθειες. Ο παρακάτω αναδρομικός τύπος εκφράζει το γεγονός πως το μέγιστο εύρος θα είναι το άθροισμα των μέγιστων ευρών στις δύο περιπτώσεις θράυσης των ποτηριών.

$$\beta(t,k) = \beta(t-1,k-1) + 1 + \beta(t-1,k)$$

Χρησιμοποιούμε μία βοηθητική συνάρτηση γ:

$$\begin{aligned}\gamma(t,k) &= \beta(t,k+1) - \beta(t,k) \\ &= (\beta(t-1,k+1) - \beta(t-1,k)) - (\beta(t-1,k) - \beta(t-1,k-1)) \\ &= \gamma(t,k) - \gamma(t,k-1)\end{aligned}$$

Μαζί με τις αρχικές της συνθήκες παρατηρούμε πως

$$\gamma(t,k) = \binom{t}{k+1}$$

Τώρα μπορούμε να εκφράσουμε την β ώς άθροισμα της γ:

$$\begin{aligned}\beta(t,k) &= \sum_{i=1}^{k} (\beta(t,i) - \beta(t,i-1) \\ \beta(t,k) &= \sum_{i=1}^{k} \gamma(t,i-1) \\ \beta(t,k) &= \sum_{i=1}^{k} \binom{t}{i}\end{aligned}$$

Έτσι μπορούμε υπολογίζοντας το $\beta(k,k) - \beta(k-1,k)$ να υπολογίσουμε το ύψος απο το οποίο μπορούν να αρχίσουν οι συγκρίσεις. Αν υποθέσουμε $O(k)$ αλγόριθμο για τον υπολογισμού του διονυμικού, θα έχουμε $\frac{k(k-1)}{2} = O(k^2)$ πολυπλοκότητα.