Αν. Καθηγητής Π. Λουρίδας

Τμήμα Διοικητικής Επιστήμης και Τεχνολογίας

Οικονομικό Πανεπιστήμιο Αθηνών

Γύρω-γύρω

Αν έχουμε μια ακολουθία αριθμών που παράγονται από μία συνάρτηση, πώς μπορούμε να ξέρουμε ότι οι τιμές που παράγονται δεν επαναλαμβάνονται; Ή, αν επαναλαμβάνονται, πότε αρχίζει να συμβαίνει αυτό;

Το ερώτημα δεν είναι απλώς μια ακαδημαϊκή σπαζοκεφαλιά. Έχουμε συζητήσει τη σημασία ακολουθιών τυχαίων αριθμών· ή, δεδομένου ότι η παραγωγή πραγματικά τυχαίων αριθμών στον υπολογιστή δεν είναι απλή, καλών ψευδοτυχαίων αριθμών. Οι ψευδοτυχαίοι αυτοί αριθμοί δεν θα πρέπει να επαλαμβάνονται, ή αν επαναλαμβάνονται αυτό θα πρέπει να συμβαίνει με μία όσο το δυνατόν μεγαλύτερη περίοδο. Στην κρυπτογραφία, η ασφάλεια της μεθόδου ανταλλαγής κλειδιών Diffie-Hellmann βασίζεται στη δυσκολία επίλυσης του προβλήματος του διακριτού λογαριθμού· και αυτό με τη σειρά του βασίζεται στην ύπαρξη συναρτήσεων της μορφής $f(x) = g^x \bmod p$ όπου η συνάρτηση $f(x)$ έχει όσο το δυνατόν μεγαλύτερη περίοδο γίνεται (ιδανικά, p).

Πώς μπορούμε λοιπόν, αν έχουμε μια ακολουθία αριθμών, να βρούμε αν η ακολουθία αρχίζει να επαλαμβάνεται; Και αν ναι, πότε; Και πόσο μεγάλη είναι η περίοδός της; Αυτό είναι το πρόβλημα του *εντοπισμού κύκλου* (cycle detection).

Έχουν προταθεί διάφοροι τρόποι για την επίλυση του προβλήματος. Όπως πολλές φορές στην πληροφορική, και εδώ έχουμε ένα αντιστάθμισμα χρόνου-χώρου (space-time trade-off). Ένας απλός τρόπος θα ήταν να αποθηκεύουμε κάπου όλες τις τιμές που έχουμε δει. Όταν ξαναδούμε μια τιμή, τότε σημαίνει ότι μπήκαμε σε κύκλο. Το πλεονέκτημα αυτής της μεθόδου είναι η απλότητα και το ότι κάθε τιμή υπολογίζεται μία και μόνο φορά (οπότε και την αποθηκεύουμε), άρα δεν κάνουμε περιττούς υπολογισμούς. Το μειονέκτημα είναι ότι ο χώρος αποθήκευσης μπορεί να είναι πολύ μεγάλος, αν απαιτηθούν πολλές τιμές για να αρχίσει ο κύκλος. Και τι θα γίνει αν δεν υπάρχει κύκλος;

Ένας άλλος τρόπος, ο πιο γνωστός, είναι η μέθοδος του λαγού και της χελώνας, ο οποίος συνήθως αποδίδεται στον Robert W. Floyd. Στη μέθοδο αυτή, ξεκινάμε να περιδιαβαίνουμε την ακολουθία με δύο δείκτες. Ο ένας δείκτης, η χελώνα, προχωράει ένα βήμα τη φορά. Ο άλλος δείκτης, η χελώνα, προχωράει δύο βήματα τη φορά. Κύκλος στην ακολουθία θα υπάρχει αν και μόνο αν κάποια στιγμή η τιμή στην οποία βρίσκεται η χελώνα είναι η ίδια με την τιμή στην οποία βρίσκεται ο λαγός.

Γιατί; Διότι αφού ο λαγός προχωράει με διπλάσια βήματα, η μόνη περίπτωση για τη χελώνα να τον προλάβει είναι να υπάρχει κύκλος, να μπουν και ο λαγός και η χελώνα στον κύκλο, και έτσι η χελώνα κάποια στιγμή θα συμπέσει με το λαγό μέσα

στον κύκλο (όπου ο λαγός θα έχει κάνει περισσότερες περιστροφές από τη χελώνα).

Η μέθοδος του Floyd απαιτεί να υπολογίζουμε κάθε τιμή της ακολουθίας μία φορά (για τη χελώνα) και τις μισές τιμές της ακολουθίας ακόμα μία φορά (για το λαγό). Εμείς λοιπόν θα δούμε τώρα μια άλλη μέθοδο, η οποία απαιτεί λιγότερους υπολογισμούς της τιμής της ακολουθίας, χρησιμοποιώντας ένα σταθερό κομμάτι μνήμης.

Η βασική ιδέα στον αλγόριθμο είναι ότι αντί να εξετάζουμε όλες τις τιμές της ακολουθίας, εξετάζουμε μόνο b τιμές, η οποίες βρίσκονται σε διάστημα gb μεταξύ τους, όπου b και g είναι παράμετροι που εμείς επιλέγουμε. Τις τιμές αυτές τις αποθηκεύουμε σε έναν πίνακα *table*, ο οποίος περιέχει ζευγάρια (y, j), όπου y είναι ένα στοιχείο της ακολουθίας και i είναι μια θέση στην οποία το συναντούμε. Για κάθε y, ο πίνακας μπορεί να περιέχει περισσότερα από ένα ζευγάρι (με διαφορετικές τιμές των j). Όπως θα δούμε, στον πίνακα αυτό θα πρέπει να μπορούμε να κάνουμε αναζητήσεις για ζευγάρια τόσο με βάση το y όσο και με βάση το j.

Θεωρούμε ότι για κάθε τιμή της ακολουθίας, η επόμενη προκύπτει καλώντας μια συνάρτηση f με παράμετρο την εν λόγω τιμή. Για παράδειγμα, αν x είναι η πρώτη τιμή, $f(x)$ είναι η δεύτερη τιμή, $f(f(x))$ είναι η τρίτη τιμή, κ.ο.κ. Τότε ο αλγόριθμος έχει την εξής μορφή:

`DetectCyclePrelim(`x, f`)`$\rightarrow (y, i, j)$

Input: x, the first element of the sequence
f, a function taking an element of the sequence and producing the next element
Data: *table*, a table of values
b, a positive integer
g, a positive integer
Output: y, a repeated element
i, the current occurence of y
j, the previous occurence of y stored in *table*

```
1  y ← x
2  i ← 0
3  while TRUE do
4      if i mod b = 0 then
5          InsertInTable(table, y, i)
6      y ← f(y)
7      i ← i + 1
8      if i mod gb < b then
9          j ← SearchTableY(table, y)
10         if j ≠ −1 then
11             return (y, i, j)
```

Στον αλγόριθμο αυτό, η συνάρτηση `SearchTableY(`*table*, y`)` αναζητά στον πίνακα *table* το στοιχείο y και επιστρέφει τη μικρότερη θέση του που έχουμε αποθηκεύσει

στον πίνακα , ή -1 αν δεν το έχουμε αποθηκεύσει στον πίνακα. Η συνάρτηση `InsertInTable(`*table*$, y, i$`)` καταχωρεί στον πίνακα *table* το στοιχείο y και τη θέση i στην οποία το συναντούμε.

Ο αλγόριθμος αυτός αποθηκεύει στοιχεία στον πίνακα, ώστε να μην χρειάζεται να ξαναυπολογίσουμε τις τιμές της ακολουθίας, πλην όμως η ακολουθία μπορεί να είναι οσοδήποτε μεγάλη, πράγμα το οποίο σημαίνει ότι ο πίνακας μπορεί να μεγαλώσει χωρίς όριο. Αυτό φυσικά δεν είναι δυνατόν. Για να το αποφύγουμε, θα εξελίξουμε τον αλγόριθμο ώστε ο πίνακας να μπορεί να χωρέσει το πολύ μέχρι *max_size* ζευγάρια (y, j), όπου $1 \leq g <$ *max_size*. Όταν ο πίνακας γεμίζει, τότε εμείς θα του αδειάζουμε όλα τα ζευγάρια (y, j) για τα οποία ισχύει $j \bmod 2b \neq 0$ και θα διπλασιάζουμε την παράμετρο b. Αυτό είναι ισοδύναμο με το να τρέχαμε από την αρχή τον αλγόριθμο με τη νέα διπλάσια τιμή του b μέχρι το τρέχον στοιχείο i της ακολουθίας, και να συνεχίσουμε από εκεί. Ο αλγόριθμός μας λοιπόν γίνεται:

`DetectCycle(`x, f`)`$\rightarrow (y, i, j)$

Input: x, the first element of the sequence
f, a function taking an element of the sequence and producing the next element
Data: *table*, a table of values
max_size, the maximum size of *table*
b, a positive integer
g, a positive integer
Output: y, a repeated element
i, the current occurence of y
j, the previous occurence of y stored in *table*

```
1  y ← x
2  i ← 0
3  m ← 0
4  while TRUE do
5      if i mod b = 0 and m = max_size then
6          b ← 2b
7          Purge(table, b)
8          m ← ⌈m/2⌉;
9      if i mod b = 0 then
10         InsertInTable(table, y, i)
11         m ← m + 1
12     y ← f(y)
13     i ← i + 1
14     if i mod gb < b then
15         j ← SearchTableY(table, y)
16         if j ≠ −1 then
17             return (y, i, j)
```

Η συνάρτηση `Purge(`*table*$, b$`)` είναι υπεύθυνη για την εκκαθάριση που αναφέραμε.

Τώρα ο πίνακας δεν θα έχει παραπάνω από *max_size* θέσεις. Αν το σκεφτείτε, ο αλγόριθμός προσαρμόζεται καθώς εξελίσσεται, όσο περισσότερο προχωράμε χωρίς να έχουμε βρει κύκλο τόσο αραιώνει τα διαστήματα b από τα οποία βάζουμε τιμές στον πίνακα.

Αν διαπιστώσουμε ότι πραγματικά η ακολουθία οδηγεί σε κύκλο, το επόμενο βήμα είναι να βρούμε πού ακριβώς ξεκινάει αυτός ο κύκλος και πόσο μεγάλος είναι· ή, με άλλα λόγια, σε ποιο σημείο, το οποίο ονομάζουμε *οδηγό* (leader), ξεκινάει η περίοδος της συνάρτησης και πόσο μεγάλη είναι η περίοδος αυτή. Αυτό μπορούμε να το κάνουμε με τον παρακάτω αλγόριθμο:

`RecoverCycle`$(f, y, i, j) \rightarrow (l, c)$

Input: f, a function taking an element of the sequence and producing the next element
y, a duplicated item in the sequence
i, the current occurence of y
j, the previous occurence of y stored in *table*

Data: *table*, a table of values
b, a positive integer
g, a positive integer

Output: l, the leader
c, the cycle

1 $c \leftarrow 1$
2 $found\_c \leftarrow$ FALSE
3 $y_c \leftarrow y$
4 **while** $c \leq (g+1) \times b$ **and** $found\_c =$ FALSE **do**
5 $y_c \leftarrow f(y_c)$
6 **if** $y = y_c$ **then**
7 $found\_c \leftarrow$ TRUE
8 **else**
9 $c \leftarrow c + 1$
10 **if** $found\_c =$ FALSE **then**
11 $c \leftarrow i - j$
12 $block\_length \leftarrow g \times b$
13 $final\_block \leftarrow block\_length \times \lfloor i / block\_length \rfloor$
14 $previous\_block \leftarrow final\_block - block\_length$
15 $i' \leftarrow$ `Max`$(c, previous\_block)$
16 $j' \leftarrow i' - c$
17 $l \leftarrow j' + 1$
18 **while** $f^l(\cdot) \neq f^{l+c}(\cdot)$ **do**
19 $l \leftarrow l + 1$
20 **return** l, c

Στον αλγόριθμο αυτό συμβολίζουμε με $f^k(\cdot)$ την j στη σειρά τιμή της ακολουθίας ξεκινώντας από την πρώτη τιμή, έστω x: δηλαδή, $f^2(x) = f(f(x))$, $f^3(x) =$

$f(f(f(x)))$, κ.ο.κ. Πώς όμως μπορούμε να υπολογίσουμε μια τιμή $f^k(\cdot)$, χωρίς να χρειαστεί να ξαναυπολογίσουμε όλες τις k διαφορετικές τιμές από την αρχή; Για να το πετύχουμε αυτό, αρκεί να σκεφτούμε ότι έχουμε ήδη αποθηκεύσει τιμές της $f(\cdot)$ στον πίνακα *table*. Συγκεκριμένα, η τιμή $b \times \lfloor k/b \rfloor$ είναι η κοντινότερη τιμή που έχει αποθηκευτεί στον πίνακα *table*. Επομένως, αρκεί να αναζητήσουμε στον πίνακα *table* το στοιχείο που το δεύτερο μέρος του είναι $b \times \lfloor k/b \rfloor$. Αν k_b είναι το πρώτο μέρος του στοιχείου αυτού, για να υπολογίσουμε το $f^k(\cdot)$ αρκεί να εφαρμόσουμε διαδοχικά $k \bmod b$ φορές τη συνάρτηση f στο k_b. Για να υλοποιήσουμε το μηχανισμό αυτό θα πρέπει να έχουμε στη διάθεσή μας μια συνάρτηση `SearchTableJ(`*table*, j`)` η οποία ψάχνει στον πίνακα *table* το στοιχείο (y, j), δηλαδή η αναζήτηση γίνεται με βάση το δεύτερο μέρος του στοιχείου του πίνακα αντί για το πρώτο που χρησιμοποιεί ως κλειδί η συνάρτηση `SearchTableY(`*table*, y`)`.

Απαιτήσεις Προγράμματος

Κάθε φοιτητής θα εργαστεί σε αποθετήριο στο GitHub. Για να αξιολογηθεί μια εργασία θα πρέπει να πληροί τις παρακάτω προϋποθέσεις:

- Για την υποβολή της εργασίας θα χρησιμοποιηθεί το ιδιωτικό αποθετήριο του φοιτητή που δημιουργήθηκε για τις ανάγκες του μαθήματος και του έχει αποδοθεί. Το αποθετήριο αυτό έχει όνομα του τύπου `username-algo-assignments`, όπου `username` είναι το όνομα του φοιτητή στο GitHub. Για παράδειγμα, το σχετικό αποθετήριο του διδάσκοντα θα ονομαζόταν `louridas-algo-assignments` και θα ήταν προσβάσιμο στο https://github.com/dmst-algorithms-course/louridas-algo-assignments. *Τυχόν άλλα αποθετήρια απλώς θα αγνοηθούν.*
- Μέσα στο αποθετήριο αυτό θα πρέπει να δημιουργηθεί ένας κατάλογος `assignment-2022-3`.
- Μέσα στον παραπάνω κατάλογο το πρόγραμμα θα πρέπει να αποθηκευτεί με το όνομα `cycle_detection.py`.
- Δεν επιτρέπεται η χρήση έτοιμων βιβλιοθηκών γράφων ή τυχόν έτοιμων υλοποιήσεων των αλγορίθμων, ή τμημάτων αυτών, εκτός αν αναφέρεται ρητά ότι επιτρέπεται.
- Επιτρέπεται η χρήση δομών δεδομένων της Python όπως στοίβες, λεξικά, σύνολα, κ.λπ.
- Επιτρέπεται η χρήση των παρακάτω βιβλιοθηκών ή τμημάτων τους όπως ορίζεται:
  - `sys.argv`
  - `argparse`
  - `collections.defaultdict`.
- Το πρόγραμμα θα πρέπει να είναι γραμμένο σε Python 3.

Το πρόγραμμα θα καλείται ως εξής (όπου `python` η κατάλληλη εντολή στο εκάστοτε σύστημα):

`python cycle_detection [-t] b g table_size input_sequence`

Η σημασία των παραμέτρων είναι η εξής:

- Η παράμετρος `b` δίνει την τιμή του b στον αλγόριθμο.
- Η παράμετρος `g` δίνει την τιμή του g στον αλγόριθμο.
- Η παράμετρος `table_size` δίνει την τιμή του *table_size* στον αλγόριθμο.
- Η παράμετρος `input_sequence` δίνει την όνομα του αρχείου που περιέχει τις τιμές της ακολουθίας.
- Η παράμετρος `-t`, αν δίνεται, σημαίνει ότι το πρόγραμμα στην έξοδό του θα τυπώσει τα τελικά περιεχόμενα του πίνακα *table*.

Παραδείγματα

Παράδειγμα 1

Αν ο χρήστης του προγράμματος δώσει:

```
python cycle_detection.py 3 2 3 seq_1.txt
```

τότε το πρόγραμμα θα διαβάσει την ακολουθία από το αρχείο `seq_1.txt` και στην έξοδο θα εμφανίσει:

```
cycle 6 leader 4
```

Αυτό σημαίνει ότι εντόπισε κύκλο μήκους έξι ο οποίος ξεκινάει από το τέταρτο στοιχείο της ακολουθίας.

Παράδειγμα 2

Αν ο χρήστης του προγράμματος δώσει:

```
python cycle_detection.py -t 3 2 3 seq_1.txt
```

τότε το πρόγραμμα θα διαβάσει την ακολουθία από το αρχείο `seq_1.txt` και στην έξοδο θα εμφανίσει:

```
cycle 6 leader 4
0 6
7 0
```

Στην έξοδο τώρα έχουν προστεθεί τα στοιχεία του πίνακα στο τέλος της εκτέλεσης του αλγόριθμου. Προσέξτε ότι τα στοιχεία του πίνακα δίνονται ταξινομημένα με αύξουσα σειρά.

Παράδειγμα 3

Αν ο χρήστης του προγράμματος δώσει:

```
python cycle_detection.py 3 4 5 seq_2.txt
```

τότε το πρόγραμμα θα διαβάσει την ακολουθία από το αρχείο `seq_2.txt` και στην έξοδο θα εμφανίσει:

```
cycle 1000 leader 1000
```

Παράδειγμα 4

Αν ο χρήστης του προγράμματος δώσει:

```
python cycle_detection.py 1000 2 1000 seq_3.txt
```

τότε το πρόγραμμα θα διαβάσει την ακολουθία από το αρχείο `seq_3.txt` και στην έξοδο θα εμφανίσει:

```
cycle 65536 leader 196607
```

Περισσότερες Πληροφορίες

Η εύρεση κύκλου με το λαγό και τη χελώνα αποδίδεται μεν στον Robert W. Floyd, αλλά δεν είναι εντελώς βέβαιο ότι αυτός ήταν ο εφευρέτης. Η μέθοδος που είναι το αντικείμενο της παρούσας άσκησης είναι δημιούργημα των Robert Sedgewick, Thomas G. Szymanski, Andrew C. Yao (1982). Αποδεικνύεται ότι ο αλγόριθμος απαιτεί βήματα για την εκτέλεσή του. Ένα ενδιαφέρον χαρακτηριστικό του είναι ότι, εάν μας ενδιαφέρει μόνο να διαπιστώσουμε την ύπαρξη κύκλου, αλλά όχι το πόσο μεγάλος είναι και το σημείο που ξεκινάει, τότε ο αλγόριθμος μπορεί να λειτουργήσει και με δεδομένα που μας τροφοδοτούνται δυναμικά και χωρίς να χρειάζεται να επισκεφτούμε στοιχεία της ακολουθίας που έχουμε ήδη δει. Λειτουργεί δηλαδή ως online αλγόριθμος. Αν ο υπολογισμός της f απαιτεί τ βήματα και ο έλεγχος αν κάποιο στοιχείο y της ακολουθίας υπάρχει στον πίνακα απαιτεί σ βήματα, τότε το g μπορεί να επιλεγεί ώστε ο συνολικός χρόνος που απαιτεί ο αλγόριθμος να είναι $(\mu + \lambda)(\tau + O(\frac{\sigma\tau}{m})^{1/2})$, όπου μ είναι το σημείο έναρξης του κύκλου και λ η περίοδος.

Sedgewick, Robert, Thomas G. Szymanski, and Andrew C. Yao. 1982. "The Complexity of Finding Cycles in Periodic Functions." *SIAM Journal on Computing* 11 (2): 376–90. https://doi.org/10.1137/0211030.

Καλή Επιτυχία!