Κατσούπης Ευάγγελος
2017030077

$1^{η}$ Άσκηση

Έχοντας στα χέρτρια μας μια ακολουθία από αριθμούς έστω f, θέλουμε να την μοιράσουμε σε B συνεχόμενα buckets (ιστόγραμμα), όπου το καθένα από αυτά θα εμφανίζει ένα σφάλμα σύμφωνα με τα στοιχεία τα οποία έχει. Έτσι για κάθε bucket, που έχει σαν στοιχεία τα $b_i = [s_i, s_{i+1}, \cdots, e_i]$ θα εμφανίζεται ένα σφάλμα $ERR(s_i, e_i)$.

Σκοπός της ασκησης να ελαχιστοποιηθεί το συνολικό σφάλμα που εμφανίζει το ιστογραμμα ή αλλιώς να ελαχιστοποιηθεί το σφάλμα που εμφανίζει το κάθε bucket ξεχωριστά.

Κάποιες από τις παρατηρήσεις που μπορούν να γίνουν σε αυτό το στάδιο, είναι ότι στην περίπτωση που ο διαμοιρασμός γίνει για τιμή του $B = 1$ τότε το συνολικό σφάλμα θα εξαρτάτε εξ' ολοκλήρου από το σφάλμα των n στοιχείων της ακολουθίας εισαγωγής f. Στην περίπτωση που $B = 2$ τότε το συνολικό σφάλμα θα εξαρτάτε από το σφάλμα του πρώτου bucket μαζί με το σφάλμα του δευτέρου.

Αρα μπορούμε να διαπιστώσουμε ότι αν σπάσουμε το πρόβλημα σε μικρότερα υποπροβλήματα και βρούμε τις βέλτιστες λύσεις αυτών, μπορούμε να βρούμε και την συνολική βέλτιστη λύση χρησιμοποιώντας τις τιμές που έχουν ίδη υπολογιστεί-αποθηκευτεί σε ένα πίνακα.

Άρα μπορούμε να ισχυριστούμε πώς το ελάχιστο δυνατό σφάλμα για την δημιουργία ιστογράμματος μπορεί να δοθεί από την παρακάτω σχέση:

$$Hist[i, B] = \begin{cases} ERR(1, n), & \text{for } b = 1 \\ min\{Hist(s_b - 1, b - 1) + ERR(s_b, e_b)\}, & \text{for } b > 1 \end{cases}$$

Σαν $Hist$ ορίζεται ο πίνακας με τις μικρότερες τιμές σφάλματος για κάθε διαχωρισμό και αποτελείτε από $i \cdot B$ στοιχεία (δυνατοί διαχωρισμοί). Αν το B ισούται με την μονάδα, τότε το σφάλμα θα ισούται με το σφάλμα όλων των στοιχείων που έχουν μείνει και σε διαφορετική περίπτωση θα ισούται με την προηγούμενη βέλτιστη τιμή (όπου θα αποτελείται από το σφάλμα του προηγούμενου bucket), προσθέτοντάς της την βέλτιστη τιμή (ο λόγος που χρησιμοποιείται το ελάχιστο) που μας ενδιαφέρει.

Με βάση τα παραπάνω μπορούμε να θέσουμε τον παρακάτω κώδικά ώς:

```
1  bucketize(seq, n, B){
2
3      init matrix[n,B]
4
5      for i is 1 to B
6          for j is 1 to n
7
8              matrix[j,i] = + \infty
9
10             for k is 1 to j-1
11
12                 if matrix[k, i-1] + ERR( seq[k+1], seq[j] ) < matrix[j, i]
13                     matrix[j, i] = matrix[k, i-1] + ERR( seq[k+1], seq[j])
14 }
```

Αρχικά ο αλγόριθμος αρχικοποιεί τον πίνακα που κρατά τις τιμές, στην γραμμή 3, με πολυπλοκότητα $\Theta(1)$ (ας την ορίσουμε έτσι). Στην συνέχεια, γραμμές 5 - 13, για κάθε διαθέσιμο bucket, ελέγχονται οι αντίστοιχες τιμές του πίνακα (για κάθε τιμή της συστοιχίας, γραμμές 6 και 10).
Σε περίπτωση που το σφάλμα που μας ενδιαφέρει είναι μικρότερο από την τιμή του πίνακα, τότε αποθηκεύεται σε αυτόν ώσπου φτάσουμε στο τελευταίο στοιχείο αυτού, όπου θα έχουμε και έναν πίνακα με τα ελάχιστα σφάλματα ανάλογα των αριθμό των bucket.
Οι συναρτήσεις $ERR()$ γνωρίζουμε ότι έχουν σταθερή πολυπλοκότητα και άρα μιλάμε για επαναλήψεις σταθερού αριθμού πράξεων.

Ο παραπάνω αλγόριθμος θα επαναληφθεί B φορές όπου κάθε φορά θα κάνει n (γραμμή 6) $\cdot$ n (γραμμή 10 μιας και για μεγάλα n ισχύει $j - 1 \simeq n$).

Άρα θα έχουμε μια πολυωνυμική συνολική πολυπλοκότητα:

$$T(n) = O(B \cdot n^2) \overset{\text{If B<<n}}{\Longrightarrow} T(n) = O(n^2)$$

2^{η} Άσκηση

Έστω ο αλγόριθμος $FindSubsetSum(set, n, B)$, οποίος δέχεται σαν ορίσματα ένα σύνολο από αριθμούς, set, το πλήθος των αριθμών n και ψάχνει υποσύνολά του, τα στοιχεία των οποίων αθροίζουν στον αριθμό B. Σε περίπτωση που βρεθεί ένα τέτοιο σύνολο, ο παραπάνω αλγόριθμος επιστρέφει $TRUE$.

1. **Bruet Force**

   Για την υλοποίηση με Bruet Force θα πρέπει να υπολογίσουμε με κάποιο τρόπο όλα τα πιθανά υποσύνολα και να υπολογίσουμε το άθροισμα αυτών. Αν υπάρξει έστω και ένα υποσύνολο με άθροισμα ίδιο με αυτό που θέλουμε ο αλγόριθμος θα επιστρέψει $TRUE$.

   Όλα τα πιθανά υποσύνολα τα οποία μπορούν να υπάρξουν είναι 2^n και μπορούν να γίνουν συνολικά n ελέγχοι. Το παραπάνω μπορεί να υλοποιηθεί με depth-first search σε ένα δυαδικό δένδρο όπου κάθε κόμβος έχει σαν παιδιά το σύνολο αριθμών με/ή χωρίς τον αριθμό που μας ενδιαφέρει.

   Με αυτό τον τρόπο μπορούμε να ισχυριστούμε ότι η χρονική πολυπλοκότητα είναι $O(n \cdot 2^n)$.

   Μια πιθανή υλοποίηση με χρήση αναδρομής είναι η παρακάτω:

```
1  FindSubsetSum(set, n, B) {
2
3      if B is 0
4          return TRUE
5      if n is 0
6          return FALSE
7
8      if set[n-1] > B
9         FindSubsetSum(set, n-1, B)
10
11     return FindSubsetSum(set, n - 1, B) or FindSubsetSum(set, n - 1, B - set[n-1])
12 }
```

   Παρατηρούμε ότι ο αλγόριθμος στις γραμμές 3 έως 6 ελέγχει τις περιπτώσεις όπου το άθροισμα είναι 0 και άρα υπάρχει πάντα το κενό υποσύνολο το οποίο θα αθροίζει στο B ή το σύνολο δεν περιέχει στοιχεία και άρα δεν μπορεί να αθροίζει σε κάποιο θετικό αριθμό B.

   Στις γραμμές 8 με 9 ελέγχεται αν ο τελευταίος αριθμός του συνόλου είναι μεγαλύτερος του B και στην περίπτωση που ισχύει, καλείται αναδρομικά η συνάρτηση για τα υπόλοιπα στοιχεία.

   Τέλος, γραμμή 11 γίνονται αναδρομικές κλίσεις για τις περιπτώσεις όπου ελέγχεται το σύνολο, χωρίς το τελευταίο στοιχείο να επηρεάσει το άθροισμα αλλά και για περιπτώσεις όπου από το άθροισμα, μειώνεται σύμφωνα με αυτό (έλεγχος αθροίσματος).

   Η συνάρτηση θα επιστρέψει TRUE ή FALSE

   Μπορούμε να διακρίνουμε ότι η πολυπλοκότητα είναι $O(n2^n)$.

2. **Dynamic Programming**

   Για την επίλυση του προβλήματος με δυναμικό προγραμματισμό μπορούμε να κάνουμε χρήση της παραπάνω μεθοδολογίας αλλά αποθηκεύοντας το αποτέλεσμα της κάθε αναδρομής σε ένα πίνακα με $n \cdot B$ στοιχεία, ώστε να χρησιμοποιηθούν σε περίπτωση που χρειαστεί να υπολογιστούν σε μελλοντικές κλίσεις.

Ένας αλγόριθμος για την παραπάνω υλοποίηση είναι ο εξής:

```
1  FindSubsetSum(set, B) {
2
3    init matrix[]
4
5    for i is 1 to n
6      for j is 1 to sum
7          matrix[i,j] = NULL
8
9    return recursion(set, n, B, matrix)
10 }
```

Ο παραπάνω δεν κάνει τίποτα άλλο από το να δημιουργεί έναν πίνακα με τα απαιτούμενα στοιχεία και να αρχικοποιεί το κάθε στοιχείο με την τιμή $NULL$ (παραλείπονται οι αρχικοποιήσεις για B = 0 και για μηδενικό πλήθος στοιχείων στο σύνολο μιας και θα γίνει άμεση επιστροφή) Για τις παραπάνω γραμμές (3 - 7) συνολική πολυπλοκότητα προσεγγίζει την $O(n \cdot B)$.

Στην συνέχεια (γραμμή 9) καλείται η αναδρομική συνάρτηση recursion(set , n, B, matrix) με επιπλέον όρισμα τον πίνακα ο οποίος δημιουργήθηκε και επιστρέφει το θεμιτό αποτέλεσμα.

Η συνάρτηση recursion(set , n, B, matrix) είναι η εξής:

```
1  recursion(set, n, B, matrix){
2    if B is 0
3          return TRUE
4      if n is 0
5          return FALSE
6
7    if matrix[n-1,B] is not NULL
8          return matrix[n - 1,B]
9
10   if set[n-1] > B
11     matrix[n - 1,B] = recursion(set, n - 1, B, matrix)
12     return matrix[n - 1,B]
13
14   if recursion(set, n - 1, B, matrix) is not FALSE    OR    recursion(set, n - 1, B - set
       [n - 1], matrix) is not FALSE
15     matrix[n - 1,B] = TRUE
16     return matrix[n - 1,B]
17   else
18     matrix[n - 1,B] = FALSE
19     return matrix[n - 1,B]
20 }
```

Όμοια με την περίπτωση 1, αρχικά γίνονται οι βασικοί ελέγχοι (γραμμές 2-5) με πολυπλοκότητα $\Theta(1)$.
Στην συνέχεια ελέγχετε (γραμμές 7 - 8) αν το αποτέλεσμα το οποίο 'ψάχνεται' έχει ίδη υπολογιστεί και αν ναι επιστρέφεται η τιμή του (επίδραση DP και πολυπλοκότητα $\Theta(1)$).

Στις γραμμές 14-19 γίνεται το ζητούμενο της άσκησης, δηλαδή με αναδρομικό τρόπο ελέγχονται οι αντίστοιχες περιπτώσεις με την υλοποίηση της σκέτης αναδρομής. Ποιο συγκεκριμένα Αν μια από τις δύο αναδρομές επιστρέψει αποτέλεσμα διαφορετικό του FALSE θα σημαίνει ότι θα υπάρχει έστω και ένα υποσύνολο το οποίο αθροίζει στο B. Αν βρεθεί τότε, το $TRUE$ αποθηκεύεται στην αντίστοιχη θέση στον πίνακα και επιστρέφεται η τιμή του.
Αξίζει να σημειωθεί ότι μας αρκεί να βρεθεί έστω και ένα διαφορετικό στοιχείο του πίνακα διαφορετικό του $TRUE$ μιας και με τις αναδρομές, θα αποθηκεύεται πάντα μια τιμή διαφορετική του $NULL$. Σε αντίθετη περίπτωση, δηλαδή όλες οι τιμές, που κάθε φορά ψάχνουμε, έχουν αποθηκευμένη τιμή $FALSE$ επιστέφεται, και αποθηκεύεται η τιμή $FALSE$.

Οι παραπάνω γραμμές είναι έχουν χρονική πολυπλοκότητα $O(B \cdot n)$.

Άρα η συνολική πολυπλοκότητα του αλγορίθμου θα είναι:

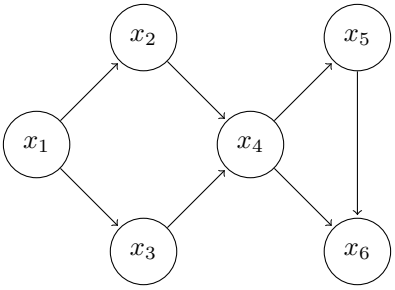
$$T(n) = \underbrace{O(B \cdot n)}_{matrix\ initialisation} + \underbrace{O(B \cdot n)}_{recursive\ DP} + \underbrace{\Theta(1)}_{constant\ operations} = O(B \cdot n)$$

3^{η} Άσκηση

Έχουμε στην διάθεσή μας ένα σύνολο από εργασίες, n, με κάθε μια από αυτές να απαιτεί χρόνο t_i, με $1 < i < n$. Για να ξεκινήσει μια εργασία, έστω n_i θα πρέπει να έχουν ολοκληρωθεί ή να περιμένει να ολοκληρωθούν, οι προαπαιτούμενες εργασίες που χρειάζεται. Αξίζει να σημειωθεί ότι οι εργασίες μπορούν να τρέχουν παράλληλα.

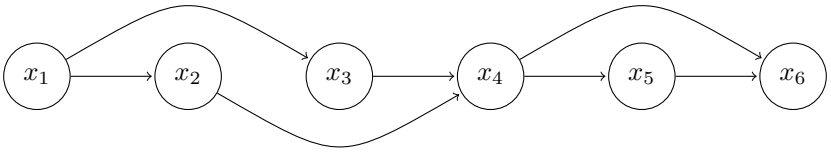
Για την επίλυση της άσκησης θα γίνει χρήση γράφου, στον οποίο θα μεταφραστεί το παραπάνω πρόβλημα.

Έστω ο παρακάτω γράφος όπου το κάθε node είναι μια διεργασία, έστω x_i με $i \in [1, n]$ από το σύνολο των διεργασιών. Λόγου του ότι κάθε εργασία για να ξεκινήσει χρειάζεται να έχουν ολοκληρωθεί όλες οι προαπαιτούμενες διεργασίες του. Αρά ο γράφος θα είναι κατευθυντικός με κάθε διεργασία να λαμβάνει ακμή από αυτή/ές που θα πρέπει να έχουν ολοκληρωθεί.



Άρα έχουμε στα χέρια μας έναν Κατευθυντικό Γράφο ο οποίος θα πρέπει να είναι και Άκυκλος, μιας και δεν θα πρέπει να υπάρχουν αλληλοεξαρτήσεις-κύκλοι μεταξύ των διεργασιών, μιας και σε αυτή την περίπτωση θα δημιουργείτε deadlock.

Άρα μιλάμε για τοπολογική ταξινόμηση του παραπάνω ΚΑΓ (διδάχθηκε στο μάθημα).



- Κάθε διεργασία x_i χρειάζεται χρόνο t_i για να ολοκληρωθεί.
- Για να ξεκινήσει να εκτελείται μια διεργασία θα πρέπει να έχουν ολοκληρωθεί οι προαπαιτούμενες διεργασίες. Άρα ο χρόνος που θα πρέπει να περιμένουμε για να ξεκινήσει μια διεργασία, έστω t_s, είναι ο χρόνος που χρειάζεται για να ολοκληρωθεί η πιό 'αργή', προαπαιτούμενή της μιας και οι διεργασίες μπορούν να εκτελεστούν παράλληλα.

Με βάση τα παραπάνω μπορούμε να ισχυριστούμε ότι η διεργασία x_i θα ολοκληρωθεί σε χρόνο:

$$t_{e_i} = t_i + t_{s_i}, \quad t_{s_i} = max\{t_{e_k}\} \text{ και } k = \text{προαπαιτούμενες διεργασίες της } i$$

Επομένως, μπορεί να οριστεί η συνάρτηση $findMinTime(list, S, t)$ με ορίσματα:

- Την λίστα με τις διεργασίες, G
- Το υποσύνολο των προαπαιτούμενων διεργασιών, S
- Το σύνολο με τους χρόνους που απαιτεί για να διεκπεραιωθεί η κάθε διεργασία. (θα μπορούσε να μην χρησιμοποιηθεί ανάλογα με τον τρόπο αποθήκευσης των διεργασιών στον σύνολο G).

Έτσι έχουμε τον ψευδοκώδικα:

```
1  findMinTime(G, S ,t)
2
3  initialize end_time[]
4
5  list = TopSort(G)
6
7  for i in list
8      for k in S(i)
9
10         if (end_time[i] < end_time[k] + t(v))
11             end_time[i] = end_time[k] + t(v)
12
13 return end_time
```

Στην γραμμή 3, γίνεται η αρχικοποίηση του πίνακα, ο οποίος θα κρατά τους χρόνους περάτωσης των διεργασιών.Κάθε θέση του θα μπορούσε να αρχικοποιηθεί με τον απαιτούμενο χρόνο της αντίστοιχης διεργασίας μιας και θα χρειαστεί σίγουρα. Στην γραμμή 5, γίνεται η τοπολογική ταξινόμηση, με χρήση της συνάρτησης $TopSort()$, όπως αυτή διδάχθηκε στο μάθημα και επιστέφεται η νέα λίστα. Για κάθε κόμβο της λίστας (γραμμή 7), ελέγχεται και αποθηκεύεται, αν χρειαστεί ο ελάχιστος χρόνος τερματισμού της εκάστοτε διεργασίας, σύμφωνα με τους χρόνους περάτωσης των προαπαιτούμενών της (γραμμές 8-11). Τέλος επιστρέφεται ο πίνακας που περιέχει τις ελάχιστες τιμές χρόνων τις κάθε διεργασίας.

Για τον υπολογισμό της πολυπλοκότητας, παρατηρείται πως απαιτείται $\Theta(n+m)$ πολυπλοκότητα για την τοπολογική ταξινόμηση (γραμμή 5). Στην συνέχεια, στις γραμμές 10-11 γίνονται πράξεις με πολυπλοκότητα $\Theta(1)$. Αυτές πραγματοποιούνται $n \cdot m$ $(m \leq n)$ φορές (ν εργασίες επί τις προαπαιτούμενες).
Άρα ο συνολικός χρόνος θα είναι:

$$T(n) = O(n+m)$$

4^{η} Άσκηση

Συμφωνα με τον ορισμό ενός ΕΣΔ έστω Τ, ως βάρος $w(T)$ θεωρείται το άθροισμα των βαρών των ακμών του ή $w(T) = \sum_{e \in T} w(e)$.

- Αν ώς βάρος οριστεί το μέγιστο από τα βάρη των ακμών του ή $w(T) = max_{e \in T}\{w(e)\}$, μπορούν να εφαρμοστούν οι αλγόριθμοι Prim, Krusal. Να ξεκινήσουμε λέγοντας ότι οι παραπάνω αλγόριθμοι για να εφαρμοστούν, θα πρέπει το δένδρο το οποίο μελετάμε να είναι Minimum Bottleneck Spanning Tree (MBST). Με βάση τα παραπάνω θα πρέπει το δένδρο να μεταφράζεται σε ένα μη κατευθυνόμενο, συνδεδεμένο και βεβαρυμένο γράφο, όπου η ποιο 'βαριά' ακμή έχει ελάχιστο δυνατό κόστος.

  Άρα αν αποδειχθεί ότι κάθε ΕΣΔ είναι και MSBT, θα μπορέσει να αποδειχθεί και ότι οι παραπάνω αλγόριθμοι μπορούν να εφαρμοστούν.

  Έστω ότι έχουμε έναν γράφο G ο οποίο απεικονίζεται από ένα ΕΣΔ Τ και ένα $MSBT$ T'. Αν μοιράσουμε το Τ σε δύο υποδένδρα, τότε θα υπάρχει ακμή έστω ακμή με βάρος $w(e) > w(e')$ η οποία θα συνδέει τα δύο υποδένδρα μέ τρόπο τέτοιον ώστε $T = T_1 \cup T_2 \cup e$ (θα σχηματίζεται το αρχικό δένδρο).
  Αντίστοιχα, θα ισχύει το ίδιο και για το $MSBT$ του G, T', έτσι ώστε $T' = T'_1 \cup T'_2 \cup e'$.

  Οι παραπάνω σχέσεις μπορούν να αναπαρασταθούν με τα αντίστοιχα βάρη των επιμέρους δένδρων:

  $$w(T) = w(T_1) + w(T_2) + w(e) \quad \text{και} \quad w(T') = w(T'_1) + w(T'_2) + w(e')$$

  Αφαιρώντας κατά μέλη, φτάνουμε στην σχέση:

  $$w(T) - w(T') = w(e) - w(e') \geq 0$$

  Επομένως βλέπουμε ότι το Τ δεν μπορεί να είναι ΕΣΔ, το οποίο είναι άτοπο, σε σχέση με την αρχική υπόθεση. Άρα το Δένδρο είναι $MSBT$ και μπορούν να εφαρμοστούν σε αυτό οι παραπάνω αλγόριθμοι.

- Αν ώς βάρος οριστεί το γινόμενο των βαρών,$w(T) = \Pi_{e \in T} w(e)$ θα μιλάμε για ένα Minimum Product Spanning Tree (MPST), περίπτωση για την οποία δεν θα εφαρμόζονται οι αλγόριθμοι, γενικά μιας και μπορεί να συναντούμε αρνητικά βάρη.

  Στην περίπτωση που γνωρίζαμε πώς τα βάρη είναι μόνο θετικά, μπορούμε να λογαριθμίσουμε το συνολικό βάρος του δέντρου. Με αυτό τον τρόπο και γνωρίζοντας ότι το μέγιστο βάρος είναι το προιών από τα επιμέρους βάρη θα έχουμε στα χέρια μας ένα πρόβλημα την μορφής $log(a \cdot b \cdot \cdots \cdot i) = log(a) + log(b) + \cdots + log(i)$.

  Με το παραπάνω τρόπο μπορούμε να βρούμε το ελάχιστο συνολικό κόστος των βαρών με χρήση των αλγορίθμων που μας ενδιαφέρουν.