Dynamic Programming

Αδάμος Τοφαρή

Περιεχόμενα

- Εισαγωγή
- Κέρματα
  - Αναδρομική Συνάρτηση
  - Memoization
  - Καταμετρώντας το πλήθος Λύσεων
- Longest Increasing Subsequence
- Paths in a grid
- Knapsack
- Edit distance - Levenshtein distance
- Counting tilings

Εισαγωγη

Ο Δυναμικός Προγραμματισμός είναι μία τεχνική που συνδυάζει την ορθότητα της πλήρης αναζήτησης (complete search) και την αποτελεσματικότητα των άπληστων (greedy) αλγόριθμων. Η τεχνική μπορεί να εφαρμοστεί όταν τα υποπροβλήματα δεν είναι ανεξάρτητα (overlapping) και μπορούν να επιλυθούν ανεξάρτητα.

Ο Δυναμικός Προγραμματισμός έχει δύο χρήσεις:

- Εύρεση βέλτιστης λύσης: Όταν θέλουμε να μεγιστοποιήσουμε ή να ελαχιστοποιήσουμε το αποτέλεσμα.
- Καταμέτρηση αποτελεσμάτων: Όταν θέλουμε να βρούμε το πλήθος των πιθανών λύσεων.

Θα δούμε πρώτα πως μπορούμε να χρησιμοποιήσουμε Δυναμικό Προγραμματισμό για να βρούμε μια βέλτιστη λύση και μετά θα χρησιμοποιήσουμε αυτό τον τρόπο για να καταμετρήσουμε τις πιθανές λύσεις. Η κατανόηση του Δυναμικού Προγραμματισμού είναι σημείο αναφοράς στην εξέλιξη σε κάθε ανταγωνιστικού προγραμματιστή. Ενώ η βασική ιδέα είναι απλή, η δυσκολία εντοπίζεται στην εφαρμογή της τεχνικής σε διαφορετικά προβλήματα.

Κερματα

Θα δούμε αρχικά ένα πρόβλημα που συναντήσαμε προηγουμένως: Αν σας δοθούν συγκεκριμένες αξίες κερμάτων coins = $\{c_1, c_2, \dots, c_k\}$ και ένα ποσό n το οποίο πρέπει να συγκεντρωθεί, να βρείτε με ποιο τρόπο μπορεί να γίνει αυτό χρησιμοποιώντας τον ελάχιστο αριθμό κερμάτων. Προηγουμένως για την επίλυση του προβλήματος χρησιμοποιήσαμε ένα άπληστο αλγόριθμο που διάλεγε πάντοτε την μεγαλύτερη αξία κέρματος. Ο άπληστος αλγόριθμός δουλεύει όταν έχουμε συγκεκριμένες αξίες κερμάτων αλλά, γενικά δεν δίνει πάντοτε μια βέλτιστη λύση.

Θα επιλύσουμε το πρόβλημα κάνοντας χρήση Δυναμικού Προγραμματισμού, έτσι ώστε ο αλγόριθμος να βρίσκει μια βέλτιστη λύση με οποιεσδήποτε αξίες κερμάτων. Ο αλγόριθμος βασίζεται σε μία αναδρομική συνάρτηση που εξετάζει όλες τις περιπτώσεις που υπολογίζουν το ποσό, όπως ένας αλγόριθμος διεξοδικής αναζήτησης. Όμως, ο αλγόριθμος του Δυναμικού Προγραμματισμού είναι αποτελεσματικός γιατί υπολογίζει και αποθηκεύει (memoization) τα αποτελέσματα για κάθε υποπρόβλημα, μόνο μία φορά.

Αναδρομική συνάρτηση

Η βασική ιδέα είναι να επιλύσουμε το πρόβλημα αναδρομικά έτσι ώστε η λύση του προβλήματος να μπορεί να υπολογιστεί από λύσεις μικρότερων υποπροβλημάτων. Στο πρόβλημα με τα κέρματα η αναδρομική συνάρτηση μπορεί να σχηματιστεί ως εξής: Ποιος είναι ο ελάχιστος αριθμός κερμάτων που μπορεί να μας δώσει το ποσό x;

Ας ορίσουμε τη συνάρτηση solve(x) η οποία υπολογίζει τον ελάχιστο αριθμό κερμάτων που μας δίνουν το ποσό x. Οι τιμές της συνάρτησης βασίζονται στις αξίες των κερμάτων.

Για παράδειγμα, αν οι αξίες των κερμάτων είναι coins = {1,3,4}, οι τιμές της συνάρτησης θα είναι οι εξής:

solve(0) -> 0
solve(1) -> 1
solve(2) -> 2
solve(3) -> 1
solve(4) -> 1
solve(5) -> 2
solve(6) -> 2
solve(7) -> 2
solve(8) -> 2
solve(9) -> 3
solve(10) -> 3

Για παράδειγμα, η solve(10) = 3, επειδή χρειάζονται τουλάχιστον 3 κέρματα για να μας δώσουν το ποσό ίσο με 10. Η βέλτιστη λύση είναι 3+3+4 = 10.

Η ουσιαστική ιδιότητα της solve είναι ότι οι τιμές της μπορούν να υπολογιστούν αναδρομικά από τις μικρότερες τιμές. Η ιδέα είναι να επικεντρωθούμε στο πρώτο κέρμα που διαλέγουμε, το οποίο, στο πιο πάνω παράδειγμα, μπορεί να είναι το 1, 3 ή 4. Αν διαλέξουμε το 1, τότε πρέπει να υπολογίσουμε τον ελάχιστο αριθμό κερμάτων για να συγκεντρώσουμε το ποσό που απομένει, δηλαδή το 9. Αυτό είναι υποπρόβλημα του αρχικού προβλήματος. Οπόταν υλοποιούμε την ακόλουθη αναδρομική συνάρτηση:

solve(x) = min(solve(x-1)+1, solve(x-3)+1, solve(x-4)+1)

Η αρχική περίπτωση (base case) της αναδρομής είναι solve(0)=0 γιατί δεν χρειαζόμαστε κανένα νόμισμα για να συγκεντρώσουμε ποσό ίσο με μηδέν. Για παράδειγμα:

solve(10) = solve(7)+1 = solve(4)+2 = solve(0)+3 = 3

Καταλήγουμε στο ότι η γενική αναδρομική συνάρτηση είναι η εξής:

$$\texttt{solve}(x) = \begin{cases} \infty & x < 0 \\ 0 & x = 0 \\ \min_{c \in \texttt{coins}} \texttt{solve}(x-c) + 1 & x > 0 \end{cases}$$

Αν το x<0 τότε η τιμή θα είναι ∞ γιατί είναι αδύνατον να συγκεντρώσουμε αρνητικό ποσό. Αν το x=0 τότε η τιμή θα είναι 0 γιατί δεν χρειαζόμαστε κέρματα για να συμπληρώσουμε ποσό ίσο με μηδέν. Αν το x>0 τότε η μεταβλητή c θα ελέγξει όλες τις πιθανότητες για να επιλεγεί το πρώτο κέρμα.

Μόλις υλοποιηθεί η αναδρομική συνάρτηση μπορούμε να υλοποιήσουμε πρόγραμμα στη C++ (Η σταθερά INF δηλώνει το άπειρο).

```
int solve(int x) {
    if (x < 0) return INF;
    if (x == 0) return 0;
    int best = INF;
    for (auto c : coins) {
        best = min(best, solve(x-c)+1);
    }
    return best;
}
```

Ακόμα όμως, η συνάρτηση αυτή δεν είναι αποδοτική γιατί μπορεί να υπάρχουν εκθετικά αυξανόμενοι τρόποι για να συμπληρώσουμε το ποσό. Θα δούμε παρακάτω πως να χρησιμοποιήσουμε μια τεχνική (memoization) για να κάνουμε τη συνάρτηση πιο αποδοτική.

Memoization

Η αρχή του Δυναμικού Προγραμματισμού είναι να χρησιμοποιεί memoization (απομνημόνευση) για να υπολογίσει αποτελεσματικά τις τιμές μιας αναδρομικής συνάρτησης. Για να γίνει αυτό, πρέπει οι τιμές της συνάρτησης να αποθηκεύονται σε έναν πίνακα όταν υπολογίζονται. Για κάθε παράμετρο της συνάρτησης, η τιμή που επιστρέφει υπολογίζεται αναδρομικά μόνο μία φορά και ακολούθως μπορεί να ανακτηθεί από τον πίνακα. Για το πιο πάνω πρόβλημα θα γίνει χρήση πινάκων:

```
bool ready[N];
int value[N];
```

Όπου ready[x] σημαίνει ότι η τιμή του solve(x) έχει υπολογιστεί και το αποτέλεσμα έχει αποθηκευτεί στο value[x]. Η σταθερά N έχει δηλωθεί αρκετά μεγάλη ώστε όλα τα απαιτούμενα ποσά να μπορούν να συμπληρωθούν.

Η συνάρτηση τώρα μπορεί να υλοποιηθεί αποδοτικά ως εξής:

```
int solve(int x) {
      if (x < 0) return INF;
      if (x == 0) return 0;
      if (ready[x]) return value[x];
      int best = INF;
      for (int c = 0; c < k; c++) {
        best = min(best, solve(x- coins[c]))+1);
      }
      value[x] = best;
      ready[x] = true;
return best;
}
```

Η συνάρτηση χειρίζεται τις αρχικές περιπτώσεις x<0 και x=0 όπως προηγουμένως. Ακολούθως, ελέγχει από την τιμή ready[x] αν η τιμή της solve(x) έχει ήδη υπολογιστεί και αν αυτό ισχύει τότε η συνάρτηση την επιστρέφει. Αλλιώς, η συνάρτηση υπολογίζει την τιμή της solve(x) αναδρομικά και την αποθηκεύει στην θέση value[x].

Αυτή η συνάρτηση λειτουργεί αποδοτικά επειδή η για κάθε τιμή του x η απάντηση υπολογίζεται αναδρομικά μόνο μια φορά. Μετά που η τιμή της solve(x) θα αποθηκευτεί στην θέση value[x], μπορεί να ανακτηθεί όποτε χρειαστεί χωρίς να υπολογιστεί ξανά. Η χρονική πολυπλοκότητα του αλγόριθμου είναι O($n \cdot k$), όπου n είναι το ποσό που πρέπει να συμπληρωθεί και k είναι το πλήθος των κερμάτων.

Μπορούμε επίσης, επαναληπτικά (iteratively), να συμπληρώσουμε τον πίνακα value, υπολογίζοντας όλες τις τιμές 0…n.

```
value[0] = 0;
for (int x = 1; x <= n; x++) {
value[x] = INF;
for (int c = 0; c < k; c++) {
   if (x-c >= 0) {
    value[x] = min(value[x], value[x-coins[c]+1);
    }
  }
}
```

Οι περισσότεροι ανταγωνιστικοί προγραμματιστές προτιμούν αυτή την υλοποίηση γιατί είναι συντομότερη και έχει λιγότερες δηλώσεις μεταβλητών. Από τώρα και στο εξής, θα χρησιμοποιούμε επαναληπτικές υλοποιήσεις στα παραδείγματά μας. Όμως, είναι ευκολότερο να κατανοούμε τις λύσεις που βασίζονται στις αναδρομικές συναρτήσεις.

Υλοποιώντας λύσεις

Κάποιες φορές μπορεί να μας ζητηθεί όχι μόνο να βρούμε την τιμή μιας βέλτιστης λύσης αλλά και να δείξουμε με ποιο τρόπο αυτή υλοποιείται. Στον πρόβλημα των κερμάτων, για παράδειγμα, μπορούμε να δηλώσουμε ακόμη έναν πίνακα ο οποίος να υποδεικνύει για κάθε ποσό, το πρώτο κέρμα που χρησιμοποιήθηκε.

```
int first[N];
```

Ακολούθως, μπορούμε να μετατρέψουμε τον αλγόριθμο ως εξής:

```
value[0] = 0;
for (int x = 1; x <= n; x++) {
    value[x] = INF;
    for (auto c : coins) {
        if (x-c >= 0 && value[x-c]+1 < value[x]) {
            value[x] = value[x-c]+1;
            first[x] = c;
        }
    }
}
```

Στη συνέχεια, με τον πιο κάτω κώδικα μπορούμε να εμφανίσουμε τα κέρματα που χρησιμοποιήθηκαν για να συμπληρώσουμε το απαιτούμενο ποσό:

```
while (n > 0) {
    cout << first[n] << "\n";
    n -= first[n];
}
```

Καταμετρώντας το πλήθος των λύσεων

Ας δούμε τώρα άλλη μία εκδοχή του προβλήματος με τα κέρματα όπου το ζητούμενο είναι να βρούμε με πόσους διαφορετικούς τρόπους μπορούμε να συμπληρώσουμε ένα ποσό x χρησιμοποιώντας τα κέρματα. Για παράδειγμα, αν έχουμε coins = {1,3,4} και x = 5, τότε υπάρχουν 6 διαφορετικοί τρόποι:

- 1+1+1+1+1
- 1+1+3
- 1+3+1
- 3+1+1
- 1+4
- 4+1

Όπως προηγουμένως, μπορούμε να επιλύσουμε το πρόβλημα αναδρομικά. Ας υποθέσουμε ότι η solve(x) υπολογίζει το πλήθος των τρόπων για να συμπληρώσουμε το ποσό x. Για παράδειγμα, αν coins = {1,3,4} τότε solve(5)=6, και η αναδρομική συνάρτηση έχει ως εξής:

solve(x) = solve(x-1) + solve(x-3) + solve(x-4)

Τότε, η γενική αναδρομική συνάρτηση θα είναι:

$$\text{solve}(x) = \begin{cases} 0 & x < 0 \\ 1 & x = 0 \\ \sum_{c \in \text{coins}} \text{solve}(x-c) & x > 0 \end{cases}$$

Αν x<0, η τιμή θα είναι 0 γιατί δεν υπάρχουν λύσεις. Αν x=0, η τιμή θα είναι 1 γιατί μόνο ένας τρόπος υπάρχει για να συμπληρωθεί ποσό ίσο με 0. Αλλιώς, υπολογίζουμε το άθροισμα όλων των τιμών solve(x-c) όπου c είναι τα κέρματα.

Ο ακόλουθος κώδικας χρησιμοποιεί έναν πίνακα count, έτσι ώστε η τιμή count[x] να είναι ίση με τη τιμή της solve(x) για $0 \leq x \leq n$:

```
count[0] = 1;
for (int x = 1; x <= n; x++) {
    for (auto c : coins) {
        if (x-c >= 0) {
            count[x] += count[x-c];
        }
    }
}
```

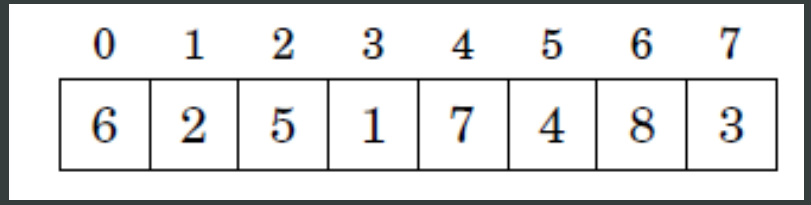
Συνήθως, το πλήθος των λύσεων είναι ένας αρκετά μεγάλος αριθμός και δεν μας ζητείται να υπολογίσουμε τον ακριβή αριθμό παρά μόνο το αποτέλεσμα modulo m, π.χ. $m = 10^9 + 7$. Αυτό γίνεται αλλάζοντας τον κώδικα έτσι ώστε όλοι οι υπολογισμοί να γίνονται modulo m. Ο πιο πάνω κώδικας διαφοροποιείται ως εξής:

```
        count[x] += count[x-c];
        count[x] %= m;
```

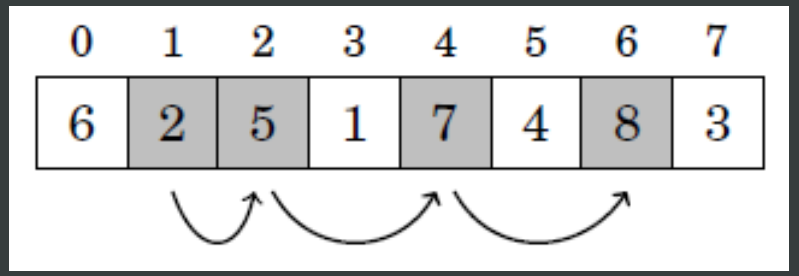
Αφού έχουμε αναλύσει όλες τις βασικές αρχές του Δυναμικού Προγραμματισμού, θα δούμε μία λίστα με προβλήματα που φανερώνουν όλες τις δυνατότητες αυτής της τεχνικής.

Μέγιστη Αύξουσα Υπακολουθία (Longest Increasing Subsequence)

Το πρώτο πρόβλημα είναι να υπολογιστεί η μέγιστη αύξουσα υπακολουθία (LIS) σε έναν πίνακα n στοιχείων. Η LIS ορίζεται ως η μέγιστη υπακολουθία από αριστερά προς δεξιά όπου κάθε στοιχείο είναι μεγαλύτερο από το προηγούμενο και όχι απαραίτητα συνεχόμενο. Για παράδειγμα, στον πίνακα:



Η LIS περιέχει 4 στοιχεία:



Ας υποθέσουμε ότι η length(k) είναι το μέγεθος της LIS μέχρι τη θέση k. Έτσι, αν υπολογίσουμε όλες τις τιμές της length(k) για $0 \leq k \leq n-1$, θα βρούμε το μέγεθος της LIS. Για παράδειγμα, οι τιμές της length(k) για τον πιο πάνω πίνακα θα είναι:

length(0) = 1
length(1) = 1
length(2) = 2
length(3) = 1
length(4) = 3
length(5) = 2
length(6) = 4
length(7) = 2

Για παράδειγμα, length(6)=4 γιατί η LIS μέχρι τη θέση 6 αποτελείται από 4 στοιχεία. Για να υπολογίσουμε την τιμή της length(k), πρέπει να βρούμε μια θέση i<k για την οποία array[i]<=array[k] και η τιμή length(i) μεγιστοποιείται. Τότε γνωρίζουμε ότι length(k)=length(i)+1, γιατί αυτός είναι ένας βέλτιστος τρόπος για να προσθέσουμε το στοιχείο array[k] στην υπακολουθία. Ωστόσο, αν δεν υπάρχει δεν υπάρχει τέτοια θέση i, τότε length(k)=1, που σημαίνει ότι η υπακολουθία περιέχει μόνο το στοιχείο array[k]. Αφού όλες οι τιμές της συνάρτησης μπορούν να υπολογιστούν από τις μικρότερες τιμές, μπορούμε να χρησιμοποιήσουμε Δυναμικό Προγραμματισμό.

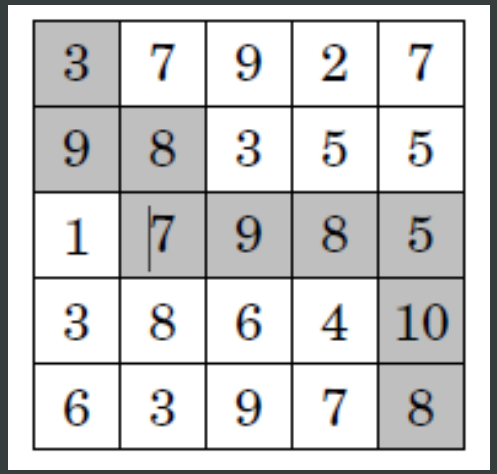
Στον πιο κάτω κώδικα, οι τιμές της συνάρτησης αποθηκεύονται στον πίνακα length.

```
for (int k = 0; k < n; k++) {
    length[k] = 1;
    for (int i = 0; i < k; i++) {
        if (array[i] < array[k]) {
            length[k] = max(length[k],length[i]+1);
        }
    }
}
```

Η πολυπλοκότητα του πιο πάνω κώδικα είναι $O(n^2)$ γιατί έχουμε δύο εμφωλευμένες δομές επανάληψης. Ωστόσο, είναι πιθανόν να υλοποιήσουμε αλγόριθμο πολυπλοκότητας O(nlogn). Μπορείτε να τον βρείτε;

Διαδρομές σε πλέγμα (Paths in a grid)

Στο επόμενο πρόβλημα πρέπει να βρούμε μία διαδρομή από την άνω-αριστερή γωνία μέχρι την κάτω-δεξιά γωνία, σε ένα πλέγμα διαστάσεων N x N, έτσι ώστε να μετακινούμαστε μόνο προς τα κάτω και προς τα δεξιά. Κάθε τετράγωνο περιέχει ένα θετικό ακέραιο και η διαδρομή πρέπει να δημιουργηθεί με τέτοιο τρόπο ώστε το άθροισμα αυτών των τιμών να μεγιστοποιηθεί. Η πιο κάτω εικόνα μας δείχνει μια τέτοια διαδρομή:



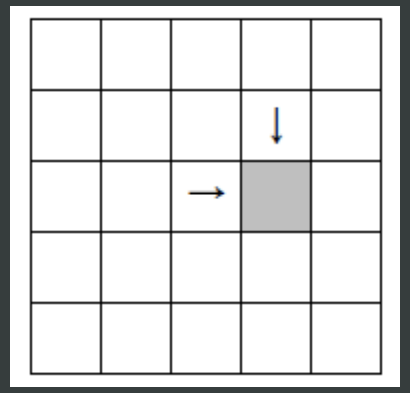
3	7	9	2	7
9	8	3	5	5
1	7	9	8	5
3	8	6	4	10
6	3	9	7	8

Το άθροισμα όλων τετραγώνων της διαδρομής είναι 67 και είναι το μέγιστο πιθανό άθροισμα που μπορεί να σχηματιστεί. Ας υποθέσουμε ότι οι γραμμές και οι στήλες του πλέγματος είναι αριθμημένες από το 1 μέχρι το N και η τιμή value[x, y] είναι η τιμή του ακεραίου που εμφανίζεται στο τετράγωνο square(x,y). Έστω ότι sum(x, y) είναι το μέγιστο άθροισμα της διαδρομής από την άνω-αριστερή γωνία μέχρι το τετράγωνο (x,y). Άρα το sum(N,N) μας δίνει το μέγιστο άθροισμα από την άνω-αριστερή γωνία μέχρι την κάτω-δεξιά γωνία. Για παράδειγμα, στο πιο πάνω πλέγμα sum(5, 5) = 67.

Υπολογίζουμε λοιπόν αναδρομικά τα αθροίσματα ως ακολούθως:

$$sum(x, y) = max(sum(x, y-1), sum(x-1, y)) + value[x][y]$$

Η αναδρομική συνάρτηση βασίζεται στην παρατήρηση ότι μία διαδρομή που καταλήγει στο τετράγωνο square(x, y) μπορεί να προέρχεται από τα τετράγωνα square(x, y-1) ή square(x-1, y):



Έτσι, επιλέγουμε πάντοτε τη διαδρομή που μεγιστοποιεί το άθροισμα. Υποθέτουμε ότι sum(x, y) = 0 και x=0 ή y=0 (γιατί δεν υπάρχει τέτοια διαδρομή), έτσι η αναδρομική συνάρτηση δουλεύει και όταν x=1 και y=1. Αφού η συνάρτηση sum έχει δύο παραμέτρους, ο πίνακας θα είναι δύο διαστάσεων.

Για παράδειγμα, μπορούμε να χρησιμοποιήσουμε τον πίνακα:

```
int sum[N][N];
```

και να υπολογίσουμε τα αθροίσματα ως εξής:

```
for (int y = 1; y <= n; y++) {
    for (int x = 1; x <= n; x++) {
        sum[y][x] = max(sum[y][x-1],sum[y-1][x])+value[y][x];
    }
}
```

Η χρονική πολυπλοκότητα είναι $O(n^2)$.

Knapsack

Ο όρος knapsack αναφέρεται σε προβλήματα όπου μας δίνεται ένα σύνολο αντικειμένων και πρέπει να βρούμε ένα υποσύνολο με συγκεκριμένες ιδιότητες. Τα προβλήματα αυτής της μορφής μπορούν να επιλυθούν με Δυναμικό Προγραμματισμό.

Θα επικεντρωθούμε στο εξής πρόβλημα: Σας δίνεται ένα σύνολο από στοιχεία [w_1, w_2,..., w_n]. Να βρείτε όλα τα πιθανά αθροίσματα που προκύπτουν αθροίζοντας τα στοιχεία του συνόλου. Για παράδειγμα, αν τα στοιχεία του συνόλου είναι [1, 3, 3, 5] τότε τα όλα πιο κάτω αθροίσματα είναι πιθανά:

0	1	2	3	4	5	6	7	8	9	10	11	12
X	X		X	X	X	X	X	X	X		X	X

Σε αυτή την περίπτωση, όλα τα αθροίσματα από το 0 μέχρι το 20 είναι πιθανά εκτός από το 2 και το 10. Για παράδειγμα, το άθροισμα 7 είναι πιθανό αν επιλέξουμε τα στοιχεία [1, 3, 3]. Για την επίλυση του προβλήματος επικεντρωνόμαστε στα υποπροβλήματα όπου χρησιμοποιούμε μόνο τα πρώτα k στοιχεία για να υπολογίσουμε τα αθροίσματα. Έστω ότι possible(x,k)=true αν μπορούμε να υπολογίσουμε το άθροισμα x με τα πρώτα k στοιχεία, αλλιώς possible(x,k)=false. Οι τιμές της συνάρτησης μπορούν να υπολογιστούν αναδρομικά ως εξής:

$$possible(x, k) = possible(x-w_k, k-1) \ V \ possible(x,k-1)$$

Η συνάρτηση βασίζεται στο ότι είτε μπορούμε να χρησιμοποιήσουμε το στοιχείο w_k στο άθροισμα είτε όχι. Αν χρησιμοποιήσουμε το στοιχεί w_k, τότε απομένει να συμπληρώσουμε το άθροισμα x - w_k χρησιμοποιώντας τα πρώτα k-1 στοιχεία. Αν δεν χρησιμοποιήσουμε το στοιχείο w_k, τότε απομένει να συμπληρώσουμε το άθροισμα x χρησιμοποιώντας τα πρώτα k-1 στοιχεία. Οι αρχικές περιπτώσεις (base cases):

$$possible(x,0) = \begin{cases} \text{true} & x = 0 \\ \text{false} & x \neq 0 \end{cases}$$

Αν δεν χρησιμοποιηθούν καθόλου στοιχεία του συνόλου τότε το μόνο άθροισμα που είναι πιθανόν είναι το 0. Ο ακόλουθος πίνακας παρουσιάζει όλες τις τιμές της συνάρτησης για τα στοιχεία [1, 3, 3, 5]. Το 'X' δείχνει τις τιμές που είναι true:

$k \backslash x$	0	1	2	3	4	5	6	7	8	9	10	11	12
0	X												
1	X	X											
2	X	X		X	X								
3	X	X		X	X		X	X					
4	X	X		X	X	X	X	X	X	X		X	X

Μετά που θα υπολογιστούν οι τιμές, η τιμή possible(x,n) μας δείχνει αν μπορούμε να υπολογίσουμε το άθροισμα x χρησιμοποιώντας όλα τα στοιχεία. Έστω ότι το W υποδηλώνει το συνολικό άθροισμα των στοιχείων.

Η ακόλουθη λύση είναι πολυπλοκότητας $O(n \cdot W)$ και κάνει χρήση της ακόλουθης αναδρομικής συνάρτησης:

```
possible[0][0] = true;
for (int k = 1; k <= n; k++) {
    for (int x = 0; x <= W; x++) {
        if (x-w[k] >= 0) possible[x][k] |= possible[x-w[k]][k-1];
        possible[x][k] |= possible[x][k-1];
    }
}
```

Ωστόσο, υπάρχει μία καλύτερη υλοποίηση που κάνει χρήση ενός μονοδιάστατου πίνακα όπου το στοιχείο possible[x] δείχνει αν μπορούμε να δημιουργήσουμε ένα υποσύνολο με άθροισμα x. Για να το πετύχουμε αυτό πρέπει να ενημερώνουμε τον πίνακα από τα δεξιά προς τα αριστερά για κάθε νέο στοιχείο:

```
possible[0] = true;
for (int k = 1; k <= n; k++) {
    for (int x = W; x >= 0; x--) {
        if (possible[x]) possible[x+w[k]] = true;
    }
}
```

Παρατηρήστε ότι η γενική ιδέα που παρουσιάζεται εδώ, μπορεί να χρησιμοποιηθεί σε διαφορετικές εκδοχές προβλημάτων τύπου knapsack. Για παράδειγμα, αν μας δοθούν στοιχεία με βάρος (weight) και αξία (value) να υπολογίσουμε για κάθε πιθανό άθροισμα βάρους το μέγιστο άθροισμα αξίας ενός υποσυνόλου.

Απόσταση Διόρθωσης (Edit distance - Levenshtein distance)

Η απόσταση διόρθωσης έχει να κάνει με τον υπολογισμό του ελάχιστου αριθμού διαδικασιών που απαιτούνται ώστε να μετατρέψουμε μία συμβολοσειρά σε κάποια άλλη. Οι μόνες διαδικασίες που επιτρέπονται είναι:

- Εισαγωγή χαρακτήρα - insert (π.χ. ABC -> ABCA)
- Διαγραφή χαρακτήρα - remove (π.χ. ABC -> AC)
- Τροποποίηση χαρακτήρα – modify (π.χ. ABC -> ADC)

Για παράδειγμα, η απόσταση διόρθωσης μεταξύ των συμβολοσειρών LOVE και MOVIE είναι 2 γιατί μπορούμε να εφαρμόσουμε, αρχικά, τη διαδικασία τροποποίησης 3 (LOVE -> MOVE) και ακολούθως τη διαδικασία εισαγωγής 1 (MOVE -> MOVIE). Αυτό είναι ο ελάχιστος πιθανός αριθμός διαδικασιών που απαιτείται γιατί είναι προφανές ότι μία μόνο διαδικασία δεν είναι αρκετή.

Ας υποθέσουμε ότι μας δίνεται μία συμβολοσειρά x μεγέθους n και μία συμβολοσειρά y μεγέθους m. Θέλουμε να υπολογίσουμε την απόσταση διόρθωσης μεταξύ των x και y. Για να επιλύσουμε το πρόβλημα, ορίζουμε μία συνάρτηση distance(a,b) η οποία επιστρέφει την απόσταση διόρθωσης μεταξύ των προθεμάτων (prefixes) x[0...a] και y[0...b]. Έτσι, με τη χρήση της συνάρτησης, η απόσταση διόρθωσης μεταξύ της x και της y θα είναι ίση με την τιμή της distance(n-1,m-1). Υπολογίζουμε της τιμές της distance ως εξής:

distance(a,b) = min(distance(a,b-1)+1, distance(a-1,b)+1, distance(a-1,b-1)+cost(a,b))

Εδώ, το cost(a, b) = 0 αν x[a] = y[b], αλλιώς cost(a, b) = 1. Η συνάρτηση υπολογίζει τα ακόλουθα για να επεξεργαστεί τη συμβολοσειρά x:

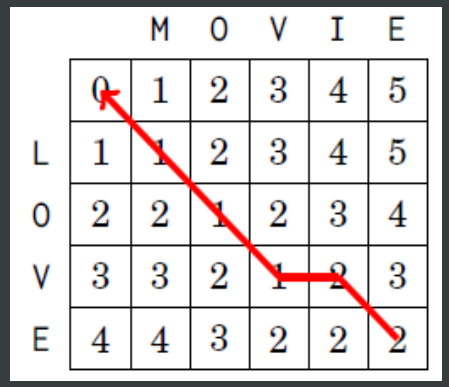
- distance(a, b-1): εισαγωγή ενός χαρακτήρα στο τέλος της x
- distance(a-1,b): διαγραφή του τελευταίου χαρακτήρα της x
- distance(a-1,b-1): αντιστοίχιση ή τροποποίηση του τελευταίου χαρακτήρα της x

Στις πρώτες δύο περιπτώσεις, μόνο μία διαδικασία απαιτείται (εισαγωγή ή διαγραφή). Στην τελευταία περίπτωση, αν x[a] = y[b] μπορούμε να ταυτοποιήσουμε τον τελευταίο χαρακτήρα χωρίς επεξεργασία, αλλιώς απαιτείται μία διαδικασία τροποποίησης (modify).

Ο ακόλουθος πίνακας παρουσιάζει τις τιμές της distance για το πιο πάνω παράδειγμα:

		M	O	V	I	E
	0	1	2	3	4	5
L	1	1	2	3	4	5
O	2	2	1	2	3	4
V	3	3	2	1	2	3
E	4	4	3	2	2	2

Η κάτω-δεξιά γωνία μας υποδεικνύει ότι η απόσταση διόρθωσης μεταξύ των συμβολοσειρών LOVE και MOVIE είναι 2. Ο πίνακας, επίσης, δείχνει και πως να δημιουργήσουμε την ελάχιστη ακολουθία διαδικασιών. Σε αυτή την περίπτωση, η διαδρομή έχει ως εξής:

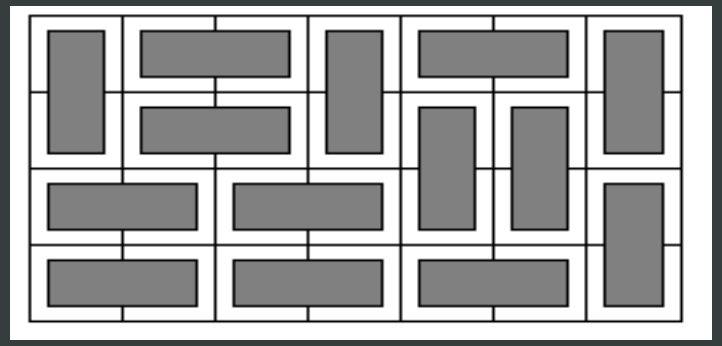


Οι τελευταίοι χαρακτήρες των συμβολοσειρών LOVE και MOVIE είναι οι ίδιοι, οπόταν η απόσταση διόρθωσης μεταξύ τους είναι ίση με την απόσταση διόρθωσης των συμβολοσειρών LOV και MOVI. Μπορούμε να χρησιμοποιήσουμε μία διαδικασία διαγραφής (remove) για να διαγράψουμε τον χαρακτήρα I από τη MOVI. Έτσι, η απόσταση διόρθωσης είναι κατά ένα μεγαλύτερη από την απόσταση διόρθωσης μεταξύ των συμβολοσειρών LOV και MOV, κοκ.

Καταμετρώντας πλακάκια (Counting tilings)

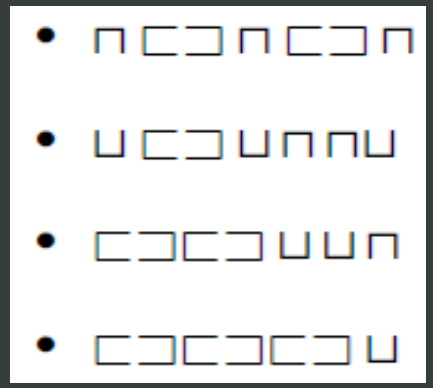
Ορισμένες φορές οι καταστάσεις (states) μίας λύσης βασισμένης σε δυναμικό προγραμματισμό είναι πιο πολύπλοκες από ένα καθορισμένο συνδυασμό από αριθμούς. Για παράδειγμα, θεωρήστε ότι θέλουμε να καταμετρήσουμε τους πιθανούς τρόπους με τους οποίους μπορούμε να καλύψουμε μία επιφάνεια διαστάσεων n x m, χρησιμοποιώντας πλακάκια διαστάσεων 1 x 2 και 2 x 1.

Μία έγκυρη λύση για ένα πλέγμα 4 x 7 θα ήταν η πιο κάτω:



και ο συνολικός αριθμός από πιθανούς τρόπους είναι 781.

Το πρόβλημα μπορεί να επιλυθεί χρησιμοποιώντας Δυναμικό Προγραμματισμό διασχίζοντας το πλέγμα γραμμή-γραμμή. Κάθε γραμμή μπορεί να αναπαρασταθεί σαν μία συμβολοσειρά m χαρακτήρες από το σύνολο {⊓,⊔,⊏,⊐} Για παράδειγμα, η πιο πάνω λύση μπορεί να συμβολιστεί με τις παρακάτω συμβολοσειρές:



Ας υποθέσουμε ότι η count(k,x) επιστρέφει το πλήθος από πιθανούς τρόπους για να δημιουργήσουμε μία λύση για τις γραμμές 1…k, έτσι ώστε κάθε συμβολοσειρά x να αντιστοιχεί στην γραμμή k. Μπορούμε να χρησιμοποιήσουμε Δυναμικό Προγραμματισμό γιατί η κατάσταση μίας γραμμής περιορίζεται μόνο από την κατάσταση της προηγούμενης γραμμής.

Μία λύση είναι έγκυρη μόνο αν η γραμμή 1 δεν περιέχει τον χαρακτήρα ␣ , η γραμμή n δεν περιέχει τον χαρακτήρα ⊓ και όλες οι συνεχόμενες γραμμές είναι συμβατές. Για παράδειγμα, οι γραμμές ␣⊏⊐␣⊓⊓␣ και ⊏⊐⊏⊐␣␣⊓, είναι συμβατές, ενώ οι γραμμές ⊓⊏⊐⊓⊏⊐⊓ και ⊏⊐⊏⊐⊏⊐␣ δεν είναι.

Εφόσον μια γραμμή περιέχει m χαρακτήρες και υπάρχουν 4 επιλογές για κάθε χαρακτήρα, το πλήθος των διακριτών (distinct) γραμμών είναι το πολύ 4^m. Έτσι, η χρονική πολυπλοκότητα είναι $O(n4^{2m})$ αφού πρέπει να ελέγξουμε τις $O(4^m)$ πιθανές καταστάσεις για κάθε γραμμή και για κάθε κατάσταση πρέπει να ελέγξουμε και τις $O(4^m)$ πιθανές καταστάσεις της κάθε προηγούμενης γραμμή. Μία καλή πρακτική θα ήταν να περιστρέψουμε το πλέγμα έτσι ώστε η μικρότερη πλευρά να έχει μήκος m, γιατί ο συντελεστής 4^{2m} είναι αποτρεπτικός για την χρονική πολυπλοκότητα.

Είναι πιθανόν να κάνουμε τη λύση πιο αποδοτική υλοποιώντας μία πιο περιορισμένη αναπαράσταση των γραμμών. Όπως αποδεικνύεται, είναι αρκετό να γνωρίζουμε ποιες στήλες στην προηγούμενη γραμμή περιέχουν τον χαρακτήρα ⊓. Έτσι, μπορούμε να αναπαραστήσουμε μία γραμμή χρησιμοποιώντας μόνο τους χαρακτήρες ⊓ και □, όπου □ μόνο 2^m διακριτές γραμμές και η πολυπλοκότητα είναι πλέον $O(n2^{2m})$.

Υπάρχει όμως και μία φόρμουλα απ’ ευθείας υπολογισμού του πλήθος:

$$\prod_{a=1}^{\lceil n/2 \rceil} \prod_{b=1}^{\lceil m/2 \rceil} 4 \cdot (\cos^2 \frac{\pi a}{n+1} + \cos^2 \frac{\pi b}{m+1})$$

Αυτή η φόρμουλα είναι πολύ αποτελεσματική επειδή υπολογίζει τα πλακάκια σε χρόνο O(nm), όμως λόγω του ότι το αποτέλεσμα είναι γινόμενο πραγματικών αριθμών, ένα μειονέκτημα είναι η αποθήκευση των ενδιάμεσων αποτελεσμάτων.

“Until you understand dynamic programming, it seems like magic”

adamos2468@gmail.com