Αλγόριθμοι
2η Σειρά Αναλυτικών Ασκήσεων

Ioannis Daras (daras.giannhs@gmail.com)
Αριθμός μητρώου: 03115018
Ροές: Υ, Λ, Σ

"First, solve the problem. Then, write the code."
— John Johnson

Άσκηση 1

(α)

(α1)

Όλα τα προτεινόμενα κριτήρια οδηγούν σε μη βέλτιστες λύσεις. Κωδικοποιούμε τις ώρες της μέρας με αριθμούς από το 0 εώς το 23. Αναλυτικότερα, έχουμε:

1. Λιγότερες επικαλύψεις

   Το κριτήριο αυτό μας λέει ότι κρατάμε το μάθημα που έχει τις λιγότερες επικαλύψεις. Θεωρούμε το ακόλουθο αντιπαράδειγμα:

   Έστω ότι έχουμε μαθήματα με τους ακόλουθους χρόνους:

   $$(0, 10), (5, 7), (8, 15), (14, 23), (21, 22)$$

   Όλα τα μαθήματα έχουν επικαλύψεις με άλλα δύο μαθήματα. Έτσι, ο αλγόριθμος μας διαλέγει τυχαία κάποιο να κρατήσει. Έστω ότι επιλέγει το μάθημα με χρόνους $(0, 10)$. Στη συνέχεια, πρέπει να διαγράψει όσα μαθήματα επικαλύπτονται με αυτό, δηλαδή τα $(5, 7), (8, 15)$. Στη συνέχεια, μόνο τα μαθήματα $(14, 23), (21, 22)$ επικαλύπτονται οπότε και διαλέγει κάποιο πάλι τυχαία. Έτσι, συνολικά μένουν 2 μαθήματα.

   Η λύση αυτή δεν είναι βέλτιστη καθώς μπορούμε να διαλέξουμε τα μαθήματα: $(5, 7), (8, 15), (21, 22)$ που είναι 3 στο σύνολο. Αφού υπάρχει έστω και ένα στιγμιότυπο του προβλήματος που ο αλγόριθμος μας δεν δίνει βέλτιστη λύση τότε ο αλγόριθμος μας δεν είναι ορθός.

2. Μεγαλύτερη διάρκεια

   Το κριτήριο αυτό μας λέει ότι όσο υπάρχουν επικαλύψεις διώχνουμε το μάθημα με τη μεγαλύτερη διάρκεια.

   Θεωρούμε τα μαθήματα με τους ακόλουθους χρόνους:

   $$(0, 10)(15, 20), (21, 23), (22, 23)$$

   Ο αλγόριθμος μας διαλέγει να διώξει το μάθημα με τη μεγαλύτερη διάρκεια δηλαδή το $(0, 10)$. Συνεχίζουν να υπάρχουν επικαλύψεις οπότε διώχνει το διάστημα με την αμέσως μεγαλύτερη διάρκεια που είναι το $(15, 20)$. Υπάρχουν ακόμα επικαλύψεις οπότε διώχνει το $(21, 23)$ και τελικά μένει 1 μάθημα.

Η λύση αυτή δεν είναι βέλτιστη καθώς μπορούμε να διαλέξουμε να κρατήσουμε τα μαθήματα: $(0, 10), (15, 20), (21, 23)$ που είναι 3 στο πλήθος. Αφού υπάρχει έστω και ένα στιγμιότυπο του προβλήματος που ο αλγόριθμος μας δεν δίνει βέλτιστη λύση τότε ο αλγόριθμος μας δεν είναι ορθός.

3. Περισσότερες επικαλύψεις

   Το κριτήριο αυτό μας λέει να διώξουμε το μάθημα που έχει τις περισσότερες επικαλύψεις με άλλα μαθήματα.

   Θεωρούμε τα μαθήματα με τους ακόλουθους χρόνους:

   $$(0, 10), (5, 8), (9, 20), (18, 23), (21, 22)$$

   Όλα τα μαθήματα επικαλύπτονται με άλλα δύο. Έτσι, ο αλγόριθμος μας διαλέγει κάποιο τυχαία. Έστω ότι διαλέγει το μάθημα με χρόνους $(9, 20)$. Διώχνει αυτό το μάθημα και στη συνέχεια κάθε μάθημα έχει επικάλυψη με άλλο 1. Από το $(0, 10), (5, 8), (18, 23), (21, 22)$ διώχνει τυχαία κάποιο (ας πούμε το $(0, 10)$) και στη συνέχεια διώχνει από τα μαθήματα $(18, 23), (21, 22)$ τυχαία κάποιο άλλο (ας πούμε το $(21, 22)$). Έτσι, μένουν δύο μαθήματα: τα $((5, 8), (18, 23)$.

   Η λύση αυτή δεν είναι βέλτιστη καθώς μπορούμε να κρατήσουμε τα μαθήματα: $(5, 8), (9, 20), (21, 22)$ που είναι τρία σε αριθμό. Αφού υπάρχει έστω και ένα στιγμιότυπο του προβλήματος που ο αλγόριθμος μας δεν δίνει βέλτιστη λύση τότε ο αλγόριθμος μας δεν είναι ορθός.

(α2)

Το πρόβλημα αυτό είναι γνωστό ως weighted interval scheduling.

Αρχικά, διατάσσουμε τα μαθήματα ως προς το χρόνο λήξης. Στη συνέχεια, υπολογίζουμε για κάθε μάθημα ποίο είναι το αμέσως προηγούμενο μάθημα που είναι συμβατό με αυτό. Για να το κάνουμε αυτό για κάθε μάθημα κάνουμε μια δυαδική αναζήτηση πάνω στο διατεταγμένο ως προς το χρόνο λήξης σύνολο μαθημάτων ως προς το μάθημα που τελειώνει πιο πριν από την αρχή του προς εξέταση μαθήματος και πιο κοντά σε αυτή. Ας θεωρήσουμε p(i) τη συνάρτηση που μας δίνει το αμέσως προηγούμενο συμβατό μάθημα. Για να υπολογίσουμε τις τιμές της συνάρτησης αυτής για όλα τα i θέλουμε πολυπλοκότητα $O(nlogn)$.
Η βέλτιστη λύση θα επαληθεύει την ακόλουθη αναδρομική σχέση:

$$OPT(j) = \begin{cases} 0, & j = 0 \\ max\{w_j + OPT\left(p(j)\right), OPT(j-1)\}, & \text{αλλιώς} \end{cases}$$

Αν αφαιρέσουμε από την παραπάνω παράσταση τον τελεστή max παίρνουμε κάθε λύση αφού μας λέει ότι για ένα τυχαίο μάθημα που εξετάζουμε είτε το παίρνουμε (και μαζί τις διδακτικές του μονάδες) και προχωράμε στο αμέσως προηγούμενο **συμβατό** μάθημα είτε δεν το παίρνουμε αλλά προχωράμε στην εξέταση του αμέσως προηγούμενου μαθήματος. Η βέλτιστη λύση απλώς είναι εκείνη που μεγιστοποιεί την παραπάνω ποσότητα.

Αφού το αν θα επιλέξουμε ένα μάθημα εξαρτάται μόνο από παρελθοντικές επιλογές, το πρόβλημα αυτό μπορούμε να το λύσουμε με δυναμικό προγραμματισμό με πολυπλοκότητα O(n) ξεκινώντας το δείκτη j με αρχικοποίηση στο 0 και προχωρώντας τον μέχρι το n κοιτώντας κάθε φορά σε O(1) παρελθοντικές λύσεις.

Στη συνέχεια, δίνεται ο ψευδοκώδικας για τη λύση που περιγράψαμε:

Algorithm 1: Weighted Interval Scheduling

Input: $n, s_1, s_2, ..., s_n, f_1, f_2, ..., f_n, w_1, w_2, ..., w_n$
Output: W: most points that we can get by taking non overlapping courses

```
courses ← sort_by_finish_time([(s_1, f_1), ..., (s_n, f_n)])
for i ← 0 to n − 1 do
    p[i] ← compute_previous_compatible(courses, i);
end
OPT[0] = 0;
for i ← 1 to n − 1 do
    OPT[i] ← max(w_i + OPT[p[i]], OPT[i − 1])
end
return OPT[n]
```

Η συνολική πολυπλοκότητα της λύσης μας είναι: $O(nlogn + n) = O(nlogn)$

(α3)

Ο αλγόριθμος που διαλέγουμε για να λύσουμε το πρόβλημα είναι greedy. Αρχικά, παρατηρούμε ότι τα νέα κτίρια έχει νόημα να τα επισκεφθούμε σε χρονικές στιγμές που αντιστοιχούν σε αρχές μαθημάτων. Η εξήγηση αυτού είναι ότι όταν αρχίζει ένα μάθημα θα έχει τουλάχιστον ίσο αριθμό επικαλύψεων με άλλα μαθήματα ως προς οποιαδήποτε χρονική στιγμή από την αρχή του μέχρι να αρχίσει κάποιο άλλο μάθημα (καθώς το μόνο που μπορεί να συμβεί στο διάστημα αυτό είναι να τελειώσει κάποιο μάθημα). Στη συνέχεια, διατυπώνουμε τον αλγόριθμο με βάση αυτή την παρατήρηση:

1. Διατάσσουμε τα μαθήματα ως προς το χρόνο έναρξης τους. Πολυπλοκότητα πράξης: O(nlogn)
2. Αρχικοποιούμε ένα διάστημα τομής $[s_1, f_n]$
3. Κάθε φορά που έρχεται ένα νέο μάθημα υπολογίζουμε την τομή του με το διάστημα τομής. Αν αυτή η τομή είναι διαφορετική από το κενό συνόλο τότε ανανεώνουμε το διάστημα τομής και προχωράμε για το επόμενο μάθημα. Σε διαφορετική περίπτωση, επισκεπτόμαστε τα νέα κτίρια στην αρχή του διαστήματος τομής και βάζουμε πλέον ως διάστημα τομής το διάστημα του τελευταίου μαθήματος που εξετάσαμε. Προχωράμε επαναληπτικά για τα επόμενα μαθήματα.

Διαισθητική εξήγηση αλγορίθμου

Ο αλγόριθμος μας λέει ότι ξεκινάμε από το πρωί και προσπαθούμε να επισκεφθούμε όσο το δυνατόν πιο πολλά μαθήματα μαζί. Αν κάποιο μάθημα δεν έχει τομή με τα προηγούμενα του τότε επισκεπτόμαστε όλα τα προηγούμενα μαζί και αυτό το μάθημα τοποθετείται σε επόμενο group επίσκεψης. Ο αλγόριθμος μας κρύβει τη βελτιστότητα του στο γεγονός ότι προσπαθούμε να βάλουμε όσο το δυνατόν περισσότερα μαθήματα στο εκάστοτε group επίσκεψης. Αλλάζουμε group μόνο όταν δεν είναι δυνατόν (μηδενική τομή) να επισκεφθούμε αυτό το μάθημα με το προηγούμενο group μαθημάτων.

Απόδειξη ορθότητας

Θα δουλέψουμε με το exchange argument. Έστω μια βέλτιστη λύση S και η δική μας greedy λύση G:

$$S = [t_1, t_2, ..., t_k, \mathbf{t_{k+1}}, ..., t_n]$$
$$G = [t_1, t_2, ..., t_k, \mathbf{t_p}, ..., t_m]$$

Από επαγωγική υπόθεση η βέλτιστη λύση S συμφωνεί με τη δική μας μέχρι τη χρονική στιγμή t_k. Στη συνέχεια, η βέλτιστη λύση λέει να επισκεθούμε τα Νέα Κτίρια τη χρονική στιγμή t_{k+1} ενώ η άπληστη λύση μας τη χρονική στιγμή t_p.

Αφού $t_p > t_k$ ξέρουμε σίγουρα ότι τα μαθήματα που επισκέπτεται ο αλγόριθμος μας τη χρονική στιγμή t_p είναι διαφορετικά από όσα έχει ήδη επισκεφθεί αφού δεν υπάρχει τομή με το προηγούμενο group μαθημάτων. Υπάρχουν τα εξής δύο σενάρια:

1. $t_{k+1} < t_p$.

   Ο αλγόριθμος μας επισκέπεται τη χρονική στιγμή t_p που είναι αρχή ενός μαθήματος. Αφού $t_{k+1} < t_p$ η βέλτιστη λύση δεν επισκέπτεται αυτό το μάθημα. Όμως, δεν επισκέπτεται και περισσότερα μαθήματα αφού $t_{k+1} > t_k$ και μέχρι t_k τόσο ο greedy όσο και η βέλτιστη έχουν επισκεφθεί τα ίδια και ο greedy χωρίζει τα μαθήματα σε group τομών οπότε τη χρονική στιγμή t_p έχει επισκεφθεί ότι μπορεί να επισκεφθεί από τη χρονική στιγμή t_k εώς εκείνη τη χρονική στιγμή, άρα και ότι επισκέπτεται η βέλτιστη λύση. Άρα, η βέλτιστη λύση επισκέπτεται λιγότερα μαθήματα από όσα επισκέπτεται ο greedy αλγόριθμος μας.

2. $t_{k+1} > t_p$.

   Ο αλγόριθμος μας επισκέπτεται πάντα στην αρχή της τομής των μαθημάτων του εκάστοτε group. Αν η χρονική στιγμή t_{k+1} ανήκει στην τομή του προς εξέταση group τότε η βέλτιστη λύση με τη δική μας λύση πετυχαίνουν τα ίδια αποτελέσματα. Σε διαφορετική περίπτωση, η βέλτιστη λύση μας λέει να επισκεφθούμε τα Νέα Κτίρια μετά την τομή του επόμενου group οπότε χάνει κάποια μαθήματα και συνεπώς δεν μπορεί να είναι βέλτιστη.

Αν ανταλλάξουμε τη χρονική στιγμή t_p με τη χρονική στιγμή t_{k+1} η βέλτιστη λύση ή θα μείνει ίδια ή θα βελτιωθεί. Άρα, από το επιχείρημα ανταλλαγής, η greedy λύση που σχεδιάσαμε ταυτίζεται με τη βέλτιστη λύση.

Ψευδοκώδικας

Παρακάτω, παρουσιάζεται ο ψευδοκώδικας για τη συγκεκριμένη άσκηση:

Algorithm 2: Students' gathering

Input: $n, s_1, s_2, ..., s_n, f_1, f_2, ..., f_n$
Output: Time points to visit New Buildings: $t_1, t_2, ..., t_m$

```
courses ← sort_by_start_time([(s_1, f_1), ..., (s_n, f_n)])
time_points ← []
int_start, int_end = courses[0][0], courses[n − 1][1]
for i ← 0 to n − 1 do
    s_i, f_i ← courses[i]
    if int_start < s_i && int_end < s_i then
        times.push(int_start)
        int_start, int_end ← s_i, f_i
    else
        int_start, int_end ← int_start, min(int_end, f_i)
    end
end
```

Άσκηση 2

Ο αλγόριθμος μας είναι ο εξής:

1. Ταξινομούμε τα κουτιά κατά φθίνουσα σειρά χρηματικής αξίας.

2. Ταξινομούμε τα σφυριά κατά φθίνουσα δύναμη.

3. Για κάθε κουτί, κάνουμε δυαδική αναζήτηση ώστε να βρούμε το πιο αδύναμο σφυρί που μπορεί να το σπάσει. Αν το βρούμε χρησιμοποιούμε αυτό το σφυρί. Αλλιώς, χρησιμοποιούμε το πιο αδύναμο σφυρί γενικά. Συνεχίζουμε επαναληπτικά για όλα τα εναπομείναντα κουτιά και τα εναπομείναντα σφυρία.

Διαισθητική εξήγηση του αλγορίθμου

Ο αλγόριθμος μας δουλεύει γιατί βάζει ως προτεραιότητα τα πιο κερδοφόρα κουτιά και για αυτά κάνει το καλύτερο που μπορεί. Συγκεκριμένα, για καθένα από αυτά προσπαθεί να το σπάσει χάνοντας το λιγότερο δυνατό σφυρί και άμα δεν τα καταφέρει απλά πετάει το κουτί και το πιο αδύναμο σφυρί από όλα.

Απόδειξη ορθότητας

Θα δουλέψουμε με το επιχείρημα της ανταλλαγής. Έστω βέλτιστη λύση S και η δική μας Greedy λύση G. Από επαγωγική υπόθεση θεωρούμε ότι η βέλτιστη λύση με τη δικιά μας λύση ταυτίζονται μέχρι ένα σημείο, δηλαδή S,G:

$$S = \left[(f_{s1}, p_{s1}), (f_{s_k}, p_{sk}]), (f_{s_{k+1}}, p_{s_{k+1}}), ..., (f_{sn}, p_{sn})\right]$$

$$S = \left[(f_{s1}, p_{s1}), (f_{s_k}, p_{sk}]), (f_{g_{k+1}}, p_{g_{k+1}}), ..., (f_{gn}, p_{sn})\right]$$

Παρατηρούμε ότι η βέλτιστη λύση αντιστοιχεί το κουτί $p_{s_{k+1}}$ στο σφυρί $f_{s_{k+1}}$ ενώ η δικιά μας λύση στο σφυρί $f_{g_{k+1}}$. Η greedy λύση που σχεδιάσαμε αν μπορεί να σπάσει το κουτί με τα εναπομείναντα σφυριά θα το κάνει και μάλιστα με το ελάχιστο δυνατό σφυρί από τα εναπομείναντα. Δεδομένου ότι συμφωνούν μέχρι το κουτί p_{sk} υπάρχουν τα εξής σενάρια:

1. $f_{gk+1} < p_{s_{k+1}}$

   Τότε δεν υπάρχει εναπομείνον σφυρί που να σπάει το κουτί και άρα ούτε η βέλτιστη λύση καταφέρνει να το σπάσει. Αφού δεν μπορεί να σπάσει το συγκεκριμένο κουτί είναι βέλτιστο να χρησιμοποιήσουμε το σφυρί με τη λιγότερη διαθέσιμη δύναμη από τα εναπομείναντα, άρα ο αλγόριθμος μας κάνει τη σωστή επιλογή.

2. $f_{gk+1} > p_{s_{k+1}}$

   Το κουτί μπορεί να σπάσει από το σφυρί που επιλέγει η άπληστη λύση. Αν η βέλτιστη λύση χρησιμοποιεί σφυρί που δεν σπάει το κουτί τότε δεν παίρνει τους πόντους και άρα πάει χειρότερα από τον άπληστο αλγόριθμο. Από την άλλη, αν διαλέγει σφυρί που σπάει το κουτί το καλύτερο που μπορεί να κάνει είναι να διαλέξει το σφυρί που το σπάει με τη λιγότερη ευκολία ώστε να κρατήσει πιο δυνατά σφυρία για επόμενα κουτιά. Αυτό ακριβώς κάνει και ο άπληστος αλγόριθμος που σχεδιάσαμε και άρα θα συμφωνήσουν ως προς την επιλογή σφυριού.

Σε κάθε περίπτωση, το σφυρί που διαλέγει ο greedy αλγόριθμος μας δίνει τους ίδιους ή περισσότερους πόντους από όσους μας δίνει το σφυρί που μας δίνει η βέλτιστη λύση. Άρα, από το επιχείρημα της ανταλλαγής, ο αλγόριθμος που σχεδιάσαμε είναι ο βέλτιστος.

Ψευδοκώδικας

Παρακάτω φαίνεται ο ψευδοκώδικας για το συγκεκριμένο αλγόριθμο:

Algorithm 3: Television

Input: $n, p_1, p_2, ..., p_n, v_1, v_2, ..., v_n, f_1, f_2, ..., f_n$
Output: V: most money you can get

```
1  sfiria ← sort(f_1, f_2, ..., f_n)
2  koutia ← sort_by_value([(p_1, v_1), (p_2, v_2), ..., (p_n, v_n)])
3  V ← 0
4  for i ← 0 to n − 1 do
5  |   power = koutia[i][0]
6  |   f_k = binary_search(power, sfiria)
7  |   if f_k > power then
8  |   |   V+ = koutia[i][1]
9  |   sfiria.pop(f_k)
10 end
11 return V
```

Πολυπλοκότητα

Το sorting έχει πολυπλόκοτητα $O(nlogn)$. Στη συνέχεια, για κάθε κουτί κάνουμε δυαδική αναζήτηση ως προς τα σφυριά που έχει και αυτό πολυπλοκότητα για όλα τα κουτία $O(nlogn)$. Άρα, η συνολική πολυπλοκότητα του αλγορίθμου μας είναι $O(nlogn)$.

Άσκηση 3

Θα λύσουμε το πρόβλημα με δυναμικό προγραμματισμό. Χωρίς βλάβη της γενικότητας υποθέτουμε ότι έχουμε διαλέξει K αναμνηστικά από κάθε χώρα όπου:

$$K = \max_{i \in [1,...,n]}(k_i)$$

Στις χώρες που έχουν λιγότερα αναμνηστικά από K μπορούμε να βάλουμε ως υπολοιπόμενα αναμνηστικά αναμνηστικά με τεράστιο κόστος και μηδενική συναισθηματική αξία ώστε να μην επιλεγούν πότε από τον αλγόριθμο μας.

Αρχικά, για κάθε χώρα διατάσσουμε τα αναμνηστικά της από το πιο φθηνό στο πιο ακριβό. Στη συνέχεια, ο αλγόριθμος του δυναμικού προγραμματισμού που σχεδιάζουμε βασίζεται στην ακόλουθη αναδρομή:

$$DP[i][j][c] = \begin{cases} max(p_{ij}, p_{ik}), & i = 0 \ \wedge \ c \geq c_{ij} \ \wedge \ 0 \leq k < j \\ DP[i][j-1][c], & j \neq 0 \ \wedge \ c < c_{ij} \\ DP[i-1][K-1][c] + p_{ij}, & i \neq 0 \ \wedge \ j = 0 \ \wedge \ c \geq c_{ij} \\ -\infty, & j = 0 \ \wedge \ c < c_{ij} \\ max\left(DP[i-1][K-1][c-c_{ij}] + p_{ij}, DP[i][j-1][c]\right), & otherwise \end{cases}$$

Υπολογίζουμε την παραπάνω αναδρομική σχέση χρησιμοποιώντας τον πίνακα DP και ξεκινώντας τον δείκτη i από το 0, τον δείκτη j από το 0 και τον δείκτη c από το 0.

Απόδειξη ορθότητας

Θα δουλέψουμε με επαγωγή. Αρχικά, πρέπει να αποδείξουμε ότι είναι ορθή η **επαγωγική βάση**, δηλαδή όλα τα base cases της αναδρομής μας.

Έχουμε τα ακόλουθα base cases:

1. $$DP[i][j][c] = max(p_{ij}, p_{ik}), \quad i = 0 \ \wedge \ c \geq c_{ij} \ \wedge \ 0 \leq k < j$$

Το base case είναι σωστό καθώς για την πρώτη χώρα άμα έχω αρκετά λεφτά να πάρω ένα αναμνηστικό τότε η συναισθηματική αξία που έχω πάρει είναι η μέγιστη συναισθηματική αξία ανάμεσα στις συναισθηματικές αξίες αυτού του αναμνηστικού και των φθηνότερων του.

2.
$$DP[i][j][c] = -\infty, \quad j = 0 \ \wedge \ c < c_{ij}$$

Το base case είναι ορθό καθώς άμα για μια χώρα δεν έχουμε λεφτά να αγοράσουμε το πιο φθηνό αναμνηστικό (j=0), δεν έχουμε λεφτά να αγοράσουμε κανένα αναμνηστικό και συνεπώς δεν μπορεί να βρεθεί λύση που να ικανοποιεί τους περιορισμούς του προβλήματος.

Στη συνέχεια, κάνουμε την **επαγωγική υπόθεση**, δηλαδή υποθέτουμε ότι έχουν υπολογιστεί σωστά οι τιμές του πίνακα DP $\forall k < i, \forall m < j, \forall z < c$.

Πρέπει τώρα να αποδείξουμε το **επαγωγικό βήμα**, δηλαδή ότι θα υπολογιστεί σωστά η τιμή $DP[i][j][c]$. Έχουμε τις ακόλουθες περιπτώσεις:

1.
$$DP[i][j][c] = DP[i][j-1][c], \quad j \neq 0 \ \wedge \ c < c_{ij}$$

Αρχικά παρατηρούμε ότι αναφερόμαστε σε κάποιο προυπολογισμένο από το παρελθόν πρόβλημα αφού στην αναδρομική σχέση το j μικραίνει και το j το ξεκινάμε από το 0 στον υπολογισμό της λύσης μας. Ακόμη, από επαγωγική υπόθεση το προυπολογισμένο πρόβλημα είναι βέλτιστο. Η σχέση είναι ορθή καθώς άμα δεν έχουμε λεφτά να πάρουμε αυτό το αναμνηστικό από αυτή τη χώρα προσπαθούμε να πάρουμε ένα φθηνότερο αναμνηστικό από αυτή τη χώρα.

2.
$$DP[i][j][c] = DP[i-1][K-1][c] + p_{ij}, \quad i \neq 0 \ \wedge \ j = 0 \ \wedge \ c \geq c_{ij}$$

Αρχικά, παρατηρούμε ότι αναφερόμαστε σε παρελθοντικό πρόβλημα αφού ο δείκτης i ξεκινά από το 0 και στην αναδρομική σχέση μικραίνει και ακόμα όλοι οι δείκτες j υπολογίζονται πριν προχωρήσουμε στο επόμενο i. Η σχέση αυτή είναι ορθή καθώς για κάθε χώρα αν πάρουμε το φθινότερο αναμνηστικό η συναισθηματική αξία που συγκεντρώνουμε είναι αυτή που μας δίνει το συγκεκριμένο αναμνηστικό και αυτή που θα μας δώσει το πιο ακριβό αναμνηστικό από την προήγουμενη χώρα. Ο λόγος που επιλέγουμε το πιο ακριβό αναμνηστικό είναι ότι αυτό σε αυτό το cell θα υπάρχει η μεγαλύτερη συναισθηματική αξία από όλα τα αναμνηστικά της χώρας καθώς η συνάρτηση μας είναι αύξουσα όσο ανεβαίνουν οι τιμές των αναμνηστικών: ένα πιο ακριβό αναμνηστικό με λιγότερη συναισθηματική αξία θα έχει τη συναισθηματική αξία του φθινότερου αναμνηστικού όπως ορίζει η σχέση βάσης 1.

3.
$$DP[i][j][c] = max\left(DP[i-1][K-1][c-c_{ij}] + p_{ij}, DP[i][j-1][c]\right), \quad otherwise$$

Για τους λόγους που αναφέραμε στα 1. , 2., πάλι αναφερόμαστε σε προυπολογισμένα προβλήματα. Η αναδρομική σχέση αυτή είναι ορθή καθώς η βέλτιστη επιλογή είναι να διαλέξουμε την maximum συναισθηματική αξία από τις δύο επιλογές που έχουμε (και που τελικά αναφέρονται σε βέλτιστα πρόβλήματα από επαγωγική υπόθεση): είτε να πάρουμε το συγκεκριμένο αναμνηστικό και να πάμε σε άλλη χώρα είτε να μην το πάρουμε και να κοιτάξουμε κάποιο άλλο αναμνηστικό από τη χώρα που είμαστε.

Τελική λύση

Η τελική λύση βρίσκεται στο κελί: DP[N-1][K-1][C]. Προκειμένου να βρούμε ποιά αναμνηστικά αγοράσαμε τρέχουμε ξεκινώντας από αυτό το κελί ανάποδα την αναδρομή και βλέπουμε κάθε φορά από ποιά επιλογή προέκυψαν οι αντίστοιχες τιμές. Για παράδειγμα, για την κάθε χώρα έχοντας στη διάθεση μας c λεφτά βλέπουμε σε ποίο κελί της εμφανίσθηκε πρώτη φορά η συναισθηματική αξία για τη χώρα αυτή και αυτό είναι το αναμνηστικό που επιλέχθηκε.

Κώδικας

Ο κώδικας για την επίλυση του παραπάνω προβλήματος είναι ουσιαστικά η εκτέλεση με ένα τριπλό for της παραπάνω αναδρομής και φαίνεται αναλυτικά παρακάτω(συγγνώμη για το formatting, το χαλάει το LaTeXστο paste):

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <limits.h>
using namespace std;

int main(void){
int N, K, C;
cin >> N >> K >> C;
int values[N][K];
int emotions[N][K];
for (int i=0; i<N; i++){
for (int j=0; j<K; j++){
cin >> values[i][j];
cin >> emotions[i][j];
}
}

int DP[N][K][C+1];
int max_emotion = emotions[0][0];
for (int j=0; j<K; j++){
for (int c=0; c<=C; c++){
if (c >= values[0][j]){
if (emotions[0][j] > max_emotion) max_emotion = emotions[0][j];
DP[0][j][c] = max_emotion;
}
else{
DP[0][j][c] = DP[0][j-1][c];
}
}
}
for (int i=1; i<N; i++){
for (int j=0; j<K; j++){
for (int c=0; c<=C; c++){
// an eisai sto 0 kai exeis lefta parto kai des allo provlima
if (j == 0 && c >= values[i][j]) DP[i][j][c] = emotions[i][j] + DP[i-1][K-1][c-values[i][j
    ]];
// an eisai sto 0 kai den exeis lefta, eisai doomed case.
else if (j==0 && c<values[i][j]) DP[i][j][c] = INT_MIN;
// an den mporeis na to pareis, dokimase na pareis kapoio fthinotero.
else if (c<values[i][j]) DP[i][j][c] = DP[i][j-1][c];
// an mporeis na to pareis, des an se simferei na to pareis.
else DP[i][j][c] = max(DP[i-1][K-1][c - values[i][j]] + emotions[i][j], DP[i][j-1][c]);
}
}
}
cout << DP[N-1][K-1][C] << endl;
}
```

Πολυπλοκότητα

Η πολυπλοκότητα της λύσης όπως φαίνεται από τον παραπάνω κώδικα είναι η πολυπλοκότητα των εμφωλευμένων for, δηλαδή: $O(N \cdot C \cdot K)$

Άσκηση 4

Θα λύσουμε το πρόβλημα με δυναμικό προγραμματισμό. Θα χρησιμοποιήσουμε έναν πίνακα DP[i][j] όπου το i θα δείχνει από ποιό σοκολατάκι ξεκινάμε και το j θα δείχνει πόσα σοκολατάκια έχουμε φάει. Το στοιχείο

του πίνακα DP[i][j] θα δείχνει το πόσα λεπτά χρειαζόμαστε αν ξεκινήσουμε από το κουτί i έχοντας φάει j σοκολατάκια. Αρχικά, τοποθετούμε σε κάθε θέση του πίνακα αυτού μια τεράστια τιμή που υποδηλώνει την άγνοια μας σχετικά με το πόσα βήματα χρειαζόμαστε για να φάμε όλα τα σοκολατάκια ξεκινώντας από τη θέση i. Στη συνέχεια, ανανεώνουμε τις τιμές αυτού του πίνακα με δυναμικό προγραμματισμό ακολουθώντας την αναδρομή:

$$DP[i][j] = \begin{cases} 0, & Q - j \leq q_i \\ min\left(DP[k][min(Q, j + q_i)] + abs(i - k)\right), & \forall k \in \mathcal{N} : k \neq i \ \wedge \ 0 \leq k \leq N \ \wedge \ q_i < q_k \ \wedge \ t_i \neq t_k \end{cases}$$

Ο αλγόριθμος μας υπολογίζει την παραπάνω αναδρομή ξεκινώντας τον δείκτη i από το N-1 και τον δείκτη j από το Q ενώ τον δείκτη k τον τρέχει από το 0 στο N-1. Με αυτόν τον τρόπο, τα υποπροβλήματα που συναντά τα έχει ήδη υπολογίσει.

Απόδειξη ορθότητας

Θα δουλέψουμε με επαγωγή. Αρχικά, πρέπει να αποδείξουμε ότι είναι ορθή η **επαγωγική βάση**, δηλαδή τα base cases της αναδρομής μας. Έχουμε:

$$DP[i][j] = 0, \quad Q - j \leq q_i$$

Η σχέση αυτή είναι ορθή καθώς αν τα σοκολατάκια που μας μένουν να φάμε $(Q - j)$ είναι λιγότερα ή ίσα με όσα υπάρχουν στο κουτί δεν χρειάζεται να κάνουμε κανένα βήμα αν ξεκινήσουμε από το κουτί αυτό.

Στη συνέχεια, κάνουμε την **επαγωγική υπόθεση**, ότι δηλαδή έχουν υπολογιστεί σωστά τα:

$$DP[k][l] \quad \forall k, l \in \mathcal{N} : \ i + 1 \leq k < N, \ j + 1 \leq q <= Q$$

Στη συνέχεια, πρέπει να αποδείξουμε το **επαγωγικό βήμα**, δηλαδή την ορθότητα της πρότασης:

$$DP[i][j] = min\left(DP[k][min(Q, j + q_i)] + abs(i - k)\right), \quad \forall k \in \mathcal{N} : k \neq i \wedge 0 \leq k \leq N \wedge q_i < q_k \wedge t_i \neq t_k$$

Αφού το i ξεκινάει από το N-1 και το k ξεκινάει από το 0 βλέπουμε ότι ως προς τον πρώτο δείκτη αναφερόμαστε πάντα σε τιμές του πίνακα που έχουμε αναφερθεί. Όμοια και για τον δεύτερο δείκτη, αφού στην αναδρομή αθροίζουμε πηγαίνουμε σε μεγαλύτερους δείκτες στους οποίους έχουμε ήδη αναφερθεί παλιά αφού το j ξεκινάει από το Q. Έτσι, τα προβλήματα στα οποία αναφερόμαστε έχουν προυπολογιστεί. Από τα προβλήματα αυτά, που λόγω επαγωγικής υπόθεσης είναι βέλτιστα, παίρνουμε το βέλτιστο (operator min) ως προς το κριτήριο που θέλουμε να υπολογίσουμε, δηλαδή παίρνουμε υπόψη το κόστος μετακίνησης σε ένα άλλο κουτί και τα βήματα που θα χρειάζεται το νέο κουτί αν φάμε και τα q_i σοκολατάκια του κουτιού που βρισκόμαστε. Στην όλη διαδικασία, λαμβάνουμε υπόψη μόνο κουτία που υπόκεινται στους περιορισμούς του προβλήματος, που δηλαδή περιέχουν περισσότερα σοκολατάκια και που ο τύπος της σοκολάτας που περιέχουν είναι διαφορετικός από τον τύπο της σοκολάτας του κουτιού που είμαστε. Αφού παίρνουμε το βέλτιστο από τις βέλτιστες λύσεις, ο αλγόριθμος μας είναι βέλτιστος.

Τελική λύση - Υπολογισμός για θέση p

Ο παραπάνω αλγόριθμος υπολογίζει για κάθε θέση i το μικρότερο κόστος για να φάμε Q σοκολατάκια ικανοποιώντας τους περιορισμούς του προβλήματος άμα ξεκινήσουμε από αυτή τη θέση στο κελί DP[i][0]. Το πρόβλημα όμως μας ζητά να ξεκινήσουμε από τη θέση p. Αυτό σημαίνει ότι μπορούμε να κάνουμε δύο πράγματα: είτε να φάμε τα σοκολατάκια της θέσης p (και άρα να απαντήσουμε την τιμή DP[p][0]) είτε να πάμε σε κάποια άλλη θέση και να ξεκινήσουμε από εκεί, λαμβάνοντας υπόψη το κόστος που εισάγει η μετακίνηση από τη θέση p στην άλλη θέση. Για να απαντήσουμε το βέλτιστο, παίρνουμε το min αυτών των δύο επιλογών (εξετάζοντας φυσικά για όλες τις θέσεις εκκίνησης).

Σε περίπτωση που εκτός από τον ελάχιστο χρόνο θέλουμε να βρούμε και ποιά πραγματικά είναι τα κουτιά που καταναλώθηκαν και με ποιά σειρά τρέχουμε από το τέλος προς την αρχή την αναδρομική σχέση και βλέπουμε ποιο k επιλέγεται κάθε φορά από το min και συνεχίζουμε επαναληπτικά.

Κώδικας

Όλα τα παραπάνω, φαίνονται αναλυτικά, υλοποιημένα στο παρακάτω κομμάτι κώδικα (συγγνώμη για το formatting - το χαλάει το LaTeX):

```
1  #include <iostream>
2  #include <algorithm>
3  #include <vector>
4  #include <climits>
5  using namespace std;
6  const int max_num = 1e6;
7
8  int main(void){
9  int N, Q;
10 cin >> N >> Q;
11 int q[N];
12 int start;
13 for (int i=0; i<N; i++){
14 cin >> q[i];
15 }
16 cin >> start;
17
18 int DP[N][Q+1];
19
20 for (int i=0; i<N; i++){
21 for (int j=0; j<=Q; j++){
22 DP[i][j] = max_num;
23 }
24 }
25 for (int i=0; i<N; i++){
26 for (int j=0; j<=Q; j++){
27 if (Q-j <= q[i]){
28 DP[i][j] = 0;
29 }
30 }
31 }
32
33 for (int i=N-1; i>=0; i--){
34 for (int j=Q; j>=0; j--){
35 if (DP[i][j] == max_num){
36 for (int k=0; k<N; k++){
37 if (k!=i && q[k] > q[i]){
38 int upper_limit = min(j+q[i], Q);
39 if (DP[k][upper_limit] + abs(i-k) < DP[i][j]){
40 DP[i][j] = DP[k][upper_limit] + abs(i-k);
41 }
42 }
43 }
44 }
45 }
46 }
47
48 int min = max_num;
49 for (int i=0; i<N; i++){
50 if (DP[i][0] + abs(start -i ) < min){
51 min = DP[i][0] + abs(start - i);
52 }
53 }
54 cout << min << endl;
55
56 }
```

Σημείωση: Στον κώδικα που παρουσίαζουμε δεν έχουμε λάβει καθόλου υπόψιν τους τύπους που έχουν τα σοκολατάκια. Η προσθήκη αυτή είναι πολύ απλή: στη γραμμή 37, που ελέγχουμε για το αν είναι κατάλληλη η επιλογή του k, προσθέτουμε αυτή τη συνθήκη.

Πολυπλοκότητα

Η πολυπλοκότητα του αλγορίθμου μας, όπως φαίνεται από τον παραπάνω κώδικα, είναι η πολυπλοκότητα των τριών εμφωλευμένων for, δηλαδή: $O(n^2 \cdot Q)$

Άσκηση 5

Greedy αλγόριθμος

Ο αλγόριθμος μας δουλεύει ως εξής:

1. Σορτάρουμε τις κεραίες με βάση το κόστος $T_i - R_i$
2. Αρχικοποιούμε ένα segment tree από έναν πίνακα με n μηδενικά. Κάθε μηδενικό υποδηλώνει ότι το στοιχείο σε αυτό το index δεν έχει χρησιμοποιηθεί ως πομπός.
3. Διαλέγουμε από τις εναπομείνουσες κεραίες την κεραία με το ελάχιστο κόστος. Κάνουμε ένα query στο segment tree για το άθροισμα των indexes που είναι δεξιά της. Αυτό μας επιστρέφει σε χρόνο $logn$ το άθροισμα που είναι το πλήθος των άσσων, δηλαδή το πλήθος των πομπών. Αν το πλήθος των πομπών είναι ίσο με τον αριθμό των στοιχείων δεξιά της κεραίας που εξετάζουμε, σημαίνει ότι η κεραία μας δεν μπορεί να χρησιμοποιηθεί ως πομπός, οπότε και προχωράμε επαναληπτικά στην επόμενη κεραία προς εξέταση. Σε διαφορετική περίπτωση, αν δηλαδή το άθροισμα δεν είναι ίσο με το πλήθος των στοιχείων δεξιά, μπορούμε να κάνουμε την κεραία πομπό, κάνοντας update το index της στο segment tree. Η πολυπλοκότητα του update είναι logN.
4. Όταν τελειώσουν οι εναπομείνουσες κεραίες παίρνουμε τα αντίστοιχα κόστη από τα φύλλα του segment tree.

Υπολογιστική πολυπλοκότητα

Για το sorting θέλουμε O(nlogn). Για κάθε query/query+update στο segment tree θέλουμε O(logn). Για κάθε κεραία γίνεται ένα query οπότε συνολικά το segment tree θέλει πολυπλοκότητα O(nlogn). Έτσι, η συνολική πολυπλοκότητα του αλγορίθμου μας είναι O(nlogn).

Απόδειξη ορθότητας

Θα δουλέψουμε με το επιχείρημα της ανταλλαγής. Έστω μια optimal λύση S που ως πομπούς έχει τις κεραίες με indexes: $S = [a_1, a_2, ..., a_k, a_{k+1}, ..., a_{n/2}]$ και η δικιά μας Greedy λύση που ως πομπούς έχει τις κεραίες με indexes: $G = [a_1, a_2, ..., a_k, b_{k+1}, ..., b_{n/2}]$.

Από επαγωγική υπόθεση, η άπληστη λύση μας συμφωνεί με την optimal μέχρι και τον πομπό στη θέση a_k. Ο επόμενος πομπός που διαλέγει η optimal λύση είναι ο a_{k+1} ενώ η δικιά μας άπληστη λύση διαλέγει τον πομπό b_{k+1}. Αφού η άπληστη λύση διαλέγει τον πομπό b_{k+1} σημαίνει ότι ο πομπός αυτός έχει μικρότερο κόστος $T_{b_{k+1}} - R_{b_{k+1}}$ από το κόστος του πομπού της βέλτιστης λύσης. Επίσης, αφού οι πομποί παρατίθενται στις τελικές λύσεις με τη σειρά σημαίνει ότι η βέλτιστη λύση διάλεξε τον b_{k+1} για δέκτη. Άρα, η βέλτιστη λύση διάλεξε έναν πομπό που οδηγεί σε μεγαλύτερο κόστος αφήνοντας έξω έναν πομπό που θα οδηγούσε σε μικρότερο κόστος, δηλαδή άτοπο. Αφού οδηγηθήκαμε σε άτοπο, η λύση που περιγράψαμε είναι η βέλτιστη λύση για το πρόβλημα αυτό.

Αλγόριθμος δυναμικού προγραμματισμού

Αφού δώσαμε μια λύση για τον άπληστο αλγόριθμο, θα δώσουμε και μια λύση δυναμικού προγραμματισμού. Χρησιμοποιούμε έναν πίνακα DP[i][j] που εκφράζει το κόστος της κεραίας i αν έχει j receivers δεξιά της. Ο αλγόριθμος μας πρακτικά ακολουθεί την αναδρομική σχέση:

$$DP[i][j] = \begin{cases} T_i, & i = 1 \ \wedge \ j = N/2 \\ \infty, & i = 1 \ \wedge \ j \neq N/2 \\ T_i + DP[i-1][j], & i \neq 1 \ \wedge \ j = N/2 \\ R_i + DP[i-1][j+1] & i \neq 1 \ \wedge \ j = 0 \\ min\left(T_i + DP[i-1][j], R_i + DP[i-1][j+1]\right), & \text{otherwise} \end{cases}$$

Ο αλγόριθμος μας υπολογίζει την παραπάνω αναδρομή ξεκινώντας τον δείκτη i από το 1 και τον δείκτη j από το N ώστε οι αναδρομικές κλήσεις να είναι προυπολογισμένες για κάθε i,j που δεν αντιστοιχεί σε base case περίπτωση.

Απόδειξη ορθότητας

Αρχικά, θα δείξουμε ότι είναι ορθή η **βάση της επαγώγής**, δηλαδή ότι τα base cases μας είναι σωστά. Έχουμε:

1. $$DP[i][j] = T_i, \quad i = 1 \ \wedge \ j = N/2$$

   Προφανώς, η πρώτη κεραία, που έχει N/2 receivers δεξιά της είναι πομπός, άρα πληρώνει το κόστος του πομπού.

2. $$DP[i][j] = \infty, \quad i = 1 \ \wedge \ j \neq N/2$$

   Προφανώς, η πρώτη κεραία πρέπει να έχει N/2 receivers δεξιά της. Άμα δεν έχει, βάζουμε ένα penalty score στο συγκεκριμένο κελί ίσο με το ∞.

Αφού δείξαμε την ορθότητα των base cases, κάνουμε την **επαγωγική υπόθεση** ότι δηλαδή είναι υπολογισμένα βέλτιστα τα:

$$DP[k][l] \forall k, l \in \mathcal{N} : 0 \leq k < i \ \wedge j < l < N$$

Πρέπει να αποδείξουμε το **επαγωγικό βήμα**. Έχουμε τις εξής περιπτώσεις:

1. $$DP[i][j] = T_i + DP[i-1][j], \quad i \neq 1 \ \wedge \ j = N/2$$

   Αρχικά, παρατηρούμε ότι γίνεται αναφορά σε προηγούμενα βέλτιστα (από επαγωγική υπόθεση) υποπροβλήματα. Η σχέση αυτή είναι ορθή καθώς άμα έχουμε $N/2$ receivers δεξιά τότε σίγουρα πρέπει η κεραία που εξετάζουμε να γίνει πομπός γιατί αλλιώς η κεραία αριστερά της θα είχε δεξιά της $N/2+1$ receivers.

2. $$DP[i][j] = R_i + DP[i-1][j+1] \quad i \neq 1 \ \wedge \ j = 0$$

   Και εδώ αναφερόμαστε σε προυπολογισμένα βέλτιστα λόγω επαγωγικής υπόθεσης υποπροβλήματα. Η σχέση αυτή είναι ορθή καθώς μας λέει ότι αν δεν υπάρχει δεξιά μας receiver τότε η συγκεκριμένη κεραία πρέπει να γίνει οπωσδήποτε receiver και πλέον η κεραία που θα είναι στα αριστερά της θα έχει δεξιά της έναν receiver.

3. $$min\left(T_i + DP[i-1][j], R_i + DP[i-1][j+1]\right), \quad \text{otherwise}$$

   Και η σχέση αυτή αναφέρεται σε βέλτιστα από επαγωγική υπόθεση υποπροβλήματα. Η συγκεκριμένη σχέση μας δίνει βέλτιστη λύση καθώς μας δίνει το βέλτιστο από τις δύο διαθέσιμες επιλογές που οδηγούν σε μικρότερα βέλτιστα προβλήματα: αν αυτή η κεραία μπορεί να γίνει και πομπός και δέκτης τότε ή θα γίνει πομπός και η κεραία στα αριστερά της θα έχει τον ίδιο αριθμό receivers στα δεξιά της ή θα γίνει δέκτης και η κεραία στα αριστερά της θα έχει έναν receiver παραπάνω στα δεξιά της. Σε κάθε περίπτωση, πληρώνονται τα αντίστοιχα κόστη.

Λύση προβλήματος

Η λύση του προβλήματος βρίσκεται στο κελί $DP[N][0]$, δηλαδή εξετάζοντας την N κεραία με 0 receivers στα δεξιά της. Αν θέλουμε να βρούμε ποιές κεραίες είναι πομποί και ποιές δέκτες απλώς τρέχουμε ανάποδα την αναδρομή που αντιστοιχεί στο αντίστοιχο κελί και βλέπουμε από ποιά επιλογή προκύπτει η αντίστοιχη τιμή.

Κώδικας

Όλα τα παραπάνω, υλοποιούνται στον ακόλουθο κώδικα σε C++ (συγγνώμη για το formatting, το χαλάει το LaTeXστο paste):

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <climits>
using namespace std;
const int max_num = 1e6;
int main(void){
int N;
cin >> N;
pair<int, int> *keraies = new pair<int, int>[N+1];
for (int i=1; i<=N; i++){
int t, r;
cin >> t >> r;
keraies[i] = make_pair(t, r);
}

int DP[N+1][N/2+1];
// i: poia keraia exetazw
// j: posous receivers exei dexia tis
for (int i=1; i<=N; i++){
for (int j=N/2; j>=0; j--){
/* base case */
if (i==1 && j == N/2) DP[i][j] = keraies[1].first;
/* penalize wrong cases */
else if (i==1) {
DP[i][j] = max_num;
}
else{
/* an exei N/2 receivers dexia tote tha einai sigoura pompos */
if (j==N/2) DP[i][j] = keraies[i].first + DP[i-1][j];
/* an exei 0 receivers dexia tote tha einai sigoura dektis */
else if (j==0) DP[i][j] = keraies[i].second + DP[i-1][j+1];
/* se diaforetiki periptwsi mporei na einai eite pompos, eite dektis */
else DP[i][j] = min(keraies[i].first + DP[i-1][j], keraies[i].second + DP[i-1][j+1]);
}
}
}

cout << DP[N][0] << endl;
}
```

Υπολογιστική πολυπλοκότητα

Όπως φαίνεται στην υλοποίηση που δώσαμε έχουμε πρακτικά δύο εμφωλευμένα for που ξεκινάμε από το 1 και καθένα από αυτά πάει σε μια γραμμική συνάρτηση του n. Έτσι, η συνολική πολυπλοκότητα της λύσης μας είναι: $O(n^2)$.