Θεμελιώδη Θέματα της Επιστήμης των Υπολογιστών

Γεώργιος Κυριακόπουλος ΑΜ: el18153

Δεύτερη Σειρά Ασκήσεων

Άσκηση 1:

α) Ουσιαστικά ο αναδρομικός αλγόριθμος για n δίσκους, μεταφέρει πρώτα τους n - 1 δίσκους στο βοηθητικό πάσσαλο, έπειτα μεταφέρει τον n δίσκο στον τελικό πάσσαλο και τέλος μεταφέρει τους n - 1 δίσκους στον τελικό πάσσαλο. Κάνει επομένως T(n) = T(n - 1) + 1 + T(n - 1) = 2 * T(n - 1) + 1. Με την ίδια λογική για n - 1 δίσκους έχουμε T(n - 1) = 2 * T(n - 2) + 1 και για n - 2 δίσκους έχουμε T(n - 2) = 2 * T(n - 3) + 1. Αντικαθιστώντας το T(n - 2) στην παραπάνω εξίσωση παίρνουμε T(n - 1) = 2 * (2 * T(n - 3) + 1) + 1 και αντικαθιστώντας το T(n - 1) στην πιο πάνω εξίσωση παίρνουμε $T(n) = 2^3 * T(n - 3) + 2^2 + 2^1 + 1$. Με γενίκευση έχουμε $T(n) = 2^k * T(n - k) + 2^{k-1} + 2^{k-2} + \dots + 2^2 + 2^1 + 1$. Με βασική συνθήκη T(0) = 0 και βάζοντας όπου k = n λαμβάνουμε $T(n) = 2^n * T(0) + 2^{n-1} + 2^{n-2} + \dots + 2^2 + 2^1 + 1 = 2^{n-1} + 2^{n-2} + \dots + 2^2 + 2^1 + 1$. Το δεξί μέλος αποτελεί μια γεωμετρική πρόοδο που ισούται με $2^n - 1$ και εν τέλει έχουμε $T(n) = 2^n - 1$.

β) Με παρόμοια λογική θα αποδείξω ότι και για τον επαναληπτικό αλγόριθμο ο αριθμός των κινήσεων είναι ίσος με $2^n - 1$. Για n = 1 βλέπω εύκολα πως ισχύει. Υποθέτω ότι ισχύει για n = k, δηλαδή $f(k) = 2^k - 1$. Θα αποδείξω ότι ισχύει για n = k + 1. Για να μετακινηθεί ο k+1 δίσκος, απαιτείται να έχει τρέξει ο αλγόριθμος και να έχει μετακινήσει τους υπόλοιπους k δίσκους στο βοηθητικό πάσσαλο ώστε να ελευθερωθεί ο δίσκος k + 1. Επομένως θα έχουν τρέξει $f(k) = 2^k - 1$ κινήσεις, ενώ στη συνέχεια θα μετακινηθεί ο k + 1 δίσκος στον τελικό πάσσαλο και στη συνέχεια θα τρέξει ο αλγόριθμος άλλες $2^k - 1$ κινήσεις για να μετακινήσει τους k δίσκους πάνω στον k + 1 δίσκο. Σύνολο έχουμε και πάλι $f (k + 1) = 2^k - 1 + 1 + 2^k - 1 = 2 * (2^k - 1) + 1 = 2^{k+1} - 2 + 1 = 2^{k+1} - 1$. Επομένως, από μαθηματική επαγωγή, αφού ισχύει και για n = k + 1. Άρα έχουμε κινήσεις ίσες με $2^n - 1$ και σε αυτόν τον αλγόριθμο. Η απόδειξη μπορούσε με την ίδια λογική να γίνει όπως και στο α ερώτημα, αφού και εδώ με βάση τη σκέψη που κάναμε για τον επαναληπτικό αλγόριθμο έχουμε T(n) = 2 * T(n - 1) + 1 και T(0) = 0.

γ) Έστω T(n) οι κινήσεις που απαιτούνται για να μετακινήσουμε n δίσκους από ένα πάσσαλο σε έναν άλλο. Έχουμε T(0) = 0, T(1) = 1. Για να μετακινήσουμε το n δίσκο, με παρόμοια λογική με τα προηγούμενα ερωτήματα, πρέπει πρώτα να μετακινήσουμε τους n - 1 δίσκους σε έναν άλλο πάσσαλο, στη συνέχεια να μετακινήσουμε τον n δίσκο στον τελικό πάσσαλο και έπειτα να μετακινήσουμε τους n - 1 δίσκους στον τελικό πάσσαλο επίσης. Ξέρουμε επομένως ότι με αυτά τα βήματα μπορούμε σίγουρα να επιτύχουμε τον σκοπό μας, άλλα δεν γνωρίζουμε αν υπάρχει κάποιος πιο γρήγορος αλγόριθμος.

Επομένως, $T(n) \leq 2T(n-1) + 1$ (1), όπου $T(n - 1)$ ο χρόνος που απαιτείται για να μετακινήσουμε τους n - 1 δίσκους.

Πρέπει επομένως να αποκλείσουμε το ενδεχόμενο ενός πιο αποδοτικού αλγορίθμου. Για να μετακινήσουμε το n δίσκο και να καταφέρουμε να προχωρήσουμε στη λύση του προβλήματος, πρέπει πρώτα σύμφωνα με τα ανώτερα, να έχουμε μετακινήσει τους n - 1 δίσκους, στη συνέχεια τον n δίσκο και τέλος πάλι τους n - 1 δίσκους. Χρειαζόμαστε δηλαδή τουλάχιστον αυτά τα βήματα, επομένως έχουμε ότι $T(n) \geq 2T(n - 1) + 1$ (2). Από τις σχέσεις (1), (2) παίρνουμε ότι $T(n) = 2T(n - 1) + 1$ και μέσω του $T(0) = 0$, $T(1) = 1$ και της διαδικασίας της μαθηματικής επαγωγής που έχουμε εφαρμόσει και στα προηγούμενα ερωτήματα μπορούμε εύκολα να δείξουμε ότι η $T(n) = 2^n - 1$ είναι η συνάρτηση ελάχιστων κινήσεων.

δ) Με παρόμοια λογική με τα παραπάνω ερωτήματα μπορώ να λύσω και το πρόβλημα με τους 4 πασσάλους. Αρχικά διαλέγω ένα k το οποίο ικανοποιεί τη σχέση $1 \leq k < n - 1$. Μεταφέρω τους k κορυφαίους δίσκους σε ένα βοηθητικό πάσσαλο χρησιμοποιώντας και τους 4 διαθέσιμους πασσάλους, μετά μετακινώ τους n - k δίσκους στον τελικό πάσσαλο χρησιμοποιώντας μόνο τους 3 διαθέσιμους πασσάλους και τέλος μετακινώ στον τελικό πάσσαλο τους k κορυφαίους δίσκους. Η πρώτη διαδικασία χρειάζεται σίγουρα λιγότερα από $2^k - 1$ κινήσεις, αφού έχουμε έναν παραπάνω πάσσαλο από την περίπτωση με τους 3 πασσάλους, όπως και η τρίτη, ενώ η δεύτερη $2^{n-k} - 1$. Συνολικά έχουμε κινήσεις σίγουρα λιγότερες από $2 * (2^k - 1) + 2^{n-k} - 1 = 2^{k+1} + 2^{n-k} - 3$. Αυτό είναι σίγουρα μικρότερα από $2^n - 1$, όπως αποδεικνύεται εύκολα με τη χρήση της γεωμετρικής σειράς του α ερωτήματος: $2^{n-1} + 2^{n-2} + \ldots + 2^2 + 2^1 + 1 = 2^n$, δηλαδή $2^{n-1} + 2^{n-2} + \ldots + 2^2 + 2^1 = 2^n - 1$, και αφαιρώντας από τα αριστερά θετικούς όρους εκτός από τους 2^{k+1}, 2^{n-k} (όπου στην οριακή περίπτωση ορισμού του k έχουμε $k + 1 = n - 1$ και $n - k = n - 1$, άρα σίγουρα περιέχεται στην σειρά). Έτσι έχουμε ότι $2^{k+1} + 2^{n-k} - 3 < 2^n - 1$. Μία καλή και λογική προσέγγιση για το k είναι να διαλέξουμε $k = n/2$. Έτσι εν τέλει έχουμε κινήσεις ίσες με $2^{n/2+1} + 2^{n-n/2} - 3 = 2*2^{n/2} + 2^{n/2} - 3 = 3*2^{n/2} - 3$. Άρα έχουμε πολυπλοκότητα $O(2^{n/2})$, αισθητά μικρότερη της πολυπλοκότητας με 3 πασσάλους $O(2^n)$, ειδικά για μεγάλα n.

Άσκηση 2:

α) Θα συντάξουμε ένα πρόγραμμα που θα εκτελεί το test του Fermat σε γλώσσα Python με αριθμό ελέγχων ίσο με 30, έχοντας έτσι μεγάλη πιθανότητα ορθού αποτελέσματος. Σε μία πρώτη έκδοση μπορούμε να χρησιμοποιήσουμε τη συνάρτηση fastmodpower με πολυπλοκότητα O(logn). Υπάρχει όμως και η δυνατότητα χρήση της έτοιμης συνάρτησης pow της Python, η οποία κάνει την ίδια δουλειά σε λιγότερο χρόνο λόγω αποδοτικότερης υλοποίησης.

```python
def fastmodpower(a,n,p):
    res=1
    while (n>0):
        if (n%2==1):
            res=res*a % p
        n=n//2
        a=a*a % p
    return res

def fermat_test(n):
    if n == 2:
        return True
    if n % 2 == 0:
        return False
    for i in range(1,30):
        a = n-i
        if a>1:
            if fastmodpower(a,n-1,n)!= 1:
                return False
        else:
            break
    return True
```

```python
def fermat_test(n):
    if n == 2:
        return True
    if n % 2 == 0:
        return False
    for i in range(1,30):
        a = n-i
        if a>1:
            if pow(a,n-1,n)!= 1:
                return False
        else:
            break
    return True
```

Εκτελώντας το δεύτερο πρόγραμμα για τους αριθμούς 67280421310721, 170141183460469231731687303715884105721, $2^{2281} - 1$ (pow(2,2281) – 1), $2^{9941} - 1$ (pow(2,9941) – 1), $2^{19939} - 1$ (pow(2,19939) – 1) μας επέστρεψε τις τιμές True, False, True, True, False.

Όσον αφορά τους αριθμούς Carmichael, το test του Fermat με 30 ελέγχους αποτυγχάνει μερικές φορές να αποφανθεί αν είναι πράγματι πρώτοι, όπως φαίνεται αν εκτελέσουμε το δεύτερο πρόγραμμα για τους Carmichael αριθμούς 4954039956700380001, 973694665856161, 2199733160881, 1436697831295441, 790689421836863641 όπου μας επιστρέφει τις τιμές False, True, True, False, False.

β) Για την υλοποίηση του test των Miller – Rabin θα συντάξουμε πάλι ένα πρόγραμμα στην Python όπου με τη βοήθεια και κάποιο πρόσθετων εργαλείων, όπως της συνάρτησης randrange του random module. Θα έχουμε πάλι 30 επαναλήψεις, επιτυγχάνοντας έτσι αμελητέα πιθανότητα λάθους.

```
import random
def miller_rabin(n, k):
    if n == 2 or n == 3:
        return True

    if n % 2 == 0:
        return False

    r, s = 0, n - 1
    while s % 2 == 0:
        r += 1
        s //= 2
    for i in range(k):
        a = random.randrange(2,n-1)
        x = pow(a,s,n)
        if x == 1 or x == n - 1:
            continue
        for j in range(r-1):
            x = pow(x,2,n)
            if x == n - 1:
                break
        else:
            return False
    return True
```

Δοκιμάζοντας το παραπάνω πρόγραμμα για τους Carmichael αριθμούς 4954039956700380001, 973694665856161, 2199733160881, 1436697831295441, 790689421836863641, μας επιστρέφει τις τιμές False, False, False, False, False. Παρατηρούμε, επομένως, ότι το test των Miller – Rabin ξεχωρίζει χωρίς σφάλμα με 30 ελέγχους τους πραγματικούς πρώτους από τους Carmichael αριθμούς.

γ) Για να κατασκευάσουμε ένα πρόγραμμα το οποίο θα βρίσκει όλους τους αριθμούς Mersenne, θα χρησιμοποιήσουμε το test των Miller – Rabin ως έτοιμη συνάρτηση και την ιδιότητα πως αν το x δεν είναι πρώτος, ούτε το $2^x - 1$ είναι πρώτος.

```
for i in range(100,3000):
    if(miller_rabin(i,30)):
        if(miller_rabin(pow(2,i)-1, 30)):
            print("2 to the power of",i,"minus 1 is a Mersenne number.")
```

Εκτελώντας το παραπάνω πρόγραμμα λαμβάνουμε την παρακάτω έξοδο, η οποία συμφωνεί πλήρως και με την ιστοσελίδα που αντιπαραβάλλεται.

```
2 to the power of 107 minus 1 is a Mersenne number.
2 to the power of 127 minus 1 is a Mersenne number.
2 to the power of 521 minus 1 is a Mersenne number.
2 to the power of 607 minus 1 is a Mersenne number.
2 to the power of 1279 minus 1 is a Mersenne number.
2 to the power of 2203 minus 1 is a Mersenne number.
2 to the power of 2281 minus 1 is a Mersenne number.
```

Άσκηση 3:

α) Ουσιαστικά έχουμε έναν κατευθυνόμενο γράφο χωρίς κατευθυνόμενους κύκλους. Κάθε ακμή λαμβάνει ως τιμή βάρους της την τιμή που έχει ο κόμβος στον οποίο καταλήγει. Έτσι, αφού οι κόμβοι έχουν μόνο θετικές τιμές, έχουμε ένα γράφο με μόνο θετικά βάρη. Μία λύση είναι η χρήση του αλγορίθμου του Dijkstra, με χρονική πολυπλοκότητα O(|E| + |V|log|V|). Μια πιο αποδοτική λύση, όμως, είναι η χρήση της τοπολογικής ταξινόμησης ενός γράφου και η εύρεση των ελάχιστων διαδρομών για κάθε κόμβο μέσω αυτής με πολυπλοκότητα O(|V|+|E|). Συγκεκριμένα, αρχικοποιούμε τις αποστάσεις σε όλες τις κορυφές ως άπειρο και στην πηγαία κορυφή ως 0. Στη συνέχεια, κάνουμε τοπολογική ταξινόμηση του γράφου και διατρέχουμε μία μία τις κορυφές ανανεώνοντας τις αποστάσεις των γειτόνων της. Παράλληλα σε ένα πίνακα μεγέθους |V| κρατάμε κάθε φορά που ανανεώνουμε μία απόσταση μιας κορυφής την κορυφή από την οποία ερχόμαστε με το βέλτιστο αυτό τρόπο. Έτσι στο τέλος έχουμε ένα γράφο με τις βέλτιστες διαδρομές από την πηγαία κορυφή ως προς κάθε άλλη κορυφή και μέσω του πίνακα μας θα βρίσκουμε το αντίστοιχο μονοπάτι. Η τοπολογική ταξινόμηση έχει πολυπλοκότητα O(|V|+|E|) και η διάσχιση όλων των κορυφών και των γειτονικών τους έχει την ίδια. Επομένως, έχουμε βέλτιστη λύση με O(|V|+|E|) πολυπλοκότητα.

β) Σε αυτό το παράδειγμα έχουμε να κάνουμε έναν κατευθυνόμενο γράφο με θετικές ακμές που έχει πιθανώς κύκλους. Ο πιο αποδοτικός αλγόριθμος είναι ο Dijkstra με τη χρήση Fibonacci heaps. Στο τέλος της εκτέλεσης του αλγορίθμου θα έχουμε στα χέρια μας έναν πίνακα με τις συντομότερες διαδρομές από την πηγαία κορυφή μέχρι οποιαδήποτε άλλη κορυφή, καθώς και την προηγούμενη κορυφή από την οποία ήρθαμε με το βέλτιστο τρόπο. Η πολυπλοκότητα ως γνωστόν είναι O(|E| + |V|log|V|).

γ) Έχουμε πάλι έναν κατευθυνόμενο γράφο με αρνητικά βάρη αυτή τη φορά, τις προσφορές οι οποίες είναι μεγαλύτερες από το κόστος διέλευσης, χωρίς όμως να έχουμε αρνητικούς κύκλους. Για αυτό το σκοπό θα χρησιμοποιήσουμε τον γνωστό αλγόριθμο των Bellman – Ford με πολυπλοκότητα O(|V||E|). Με το πέρας της εκτέλεσης του αλγόριθμου θα έχουμε σε έναν πίνακα τα βέλτιστα μονοπάτια για κάθε κορυφή καθώς και την προηγούμενη κορυφή από την οποία ήρθαμε με το βέλτιστο τρόπο.

Άσκηση 4:

α) Ουσιαστικά έχουμε έναν μη κατευθυνόμενο γράφο με θετικές ακμές (η απόσταση σε χιλιόμετρα δεν γίνεται να είναι αρνητική) και θέλουμε να βρούμε αν υπάρχει μονοπάτι μεταξύ δύο κορυφών, το οποίο όμως να χρησιμοποιεί ακμές με βάρος το πολύ L. Για αυτό τον σκοπό θα χρησιμοποιήσουμε μία τροποποιημένη εκδοχή του γνωστού DFS traversal αλγόριθμου. Συγκεκριμένα, ξεκινάω να τρέχω DFS από μια κορυφή (συγκεκριμένα την αρχική s), όπου για κάθε γείτονα της που δεν έχει ακόμα επισκεφθεί, ελέγχω αν το βάρος της ακμής που οδηγεί από την αρχική στον γείτονα είναι

μεγαλύτερο του L. Αν είναι τότε δεν καλώ το DFS για αυτήν την κορυφή, διαφορετικά καλώ αναδρομικά το DFS για αυτήν. Συνεχίζεται η διαδικασία μέχρι είτε να καλέσω το DFS για την τελική κορυφή t, όπου έχω έξοδο πως είναι εφικτό, είτε να επισκεφτώ όλες τις κορυφές χωρίς να έχω βρει ακόμα μονοπάτι με βάρη μικρότερα ή ίσα του L που να οδηγούν στην τελική κορυφή t, όπου έχω έξοδο πως δεν είναι εφικτό. Η πολυπλοκότητα είναι ουσιαστικά ίση με την πολυπλοκότητα του DFS, O(|V|+|E|) συν |E| ελέγχους για τα βάρη με σταθερή πολυπλοκότητα O(1) ο καθένας, άρα O(|E|) συνολικά. Συνεπώς, έχουμε τελική πολυπλοκότητα O(|V|+|E|).

β) Για να πετύχουμε τον στόχο μας, θα χρησιμοποιήσουμε μία τροποποίηση του αλγόριθμου του Kruskal για να βρούμε το ελάχιστο συνδετικό δένδρο (minimum spanning tree) του συνεκτικού μη κατευθυνόμενου μας γράφου. Ο αλγόριθμος του Kruskal χρησιμοποιεί δομή δεδομένων disjoint-set (union-find) και αυτό μας βοηθάει να τον σταματήσουμε και να επιστρέψουμε το μέγιστο βάρος μέχρι τότε, τη στιγμή που θα ολοκληρωθεί το μονοπάτι του minimal spanning tree από την αρχική κορυφή s στην τελική t. Δηλαδή, κατά την εκτέλεση του αλγορίθμου, σε κάθε επανάληψη θα ελέγχουμε στο τέλος της, εάν οι κορυφές s,t ανήκουν σε ένα ίδιο set της δομής. Αν ανήκουν, διακόπτουμε τον αλγόριθμο και επιστρέφουμε το μέγιστο βάρος μέχρι τότε, το οποίο με βάση τη λογική του αλγορίθμου του Kruskal θα είναι η ελάχιστη αυτονομία καυσίμου που απαιτείται για το ταξίδι από την πόλη s στην πόλη t. Αυτό βασίζεται στο γεγονός ότι τη στιγμή που θα σταματήσουμε τον αλγόριθμο, θα έχουμε προσθέσει στο set μία ακμή με βάρος L (το μέγιστο μέχρι τότε) που θα αποτελεί κομμάτι του μονοπατιού από s σε t, διότι, μόνο τότε και όχι νωρίτερα, θα βρίσκονται οι s,t στο ίδιο set της δομής. Ο αλγόριθμος του Kruskal έχει πολυπλοκότητα O(|E|log|V|), ενώ η αναζήτηση σε disjoin-set data structure θα γίνει το πολύ |E| φορές (όσες οι ακμές που υπάρχουν και για τις οποίες θα τρέξει ο Kruskal) με πολυπλοκότητα log|V|, λόγω χρήσης union by rank. Επομένως, έχουμε πολυπλοκότητα O(|E|log|V|) + O(|E|log|V|) = O(|E|log|V|) = O(|E|log|E|) λόγω της σχέσης $E \leq V^2$, που είναι και το ζητούμενο. Σε μία πιο βέλτιστη περίπτωση, με χρήση path compression και union by rank οι αναζητήσεις μπορούν να γίνουν με πολυπλοκότητα O(α(n)), όπου α(n) < 5 για κάθε n που μπορεί να γραφεί στο φυσικό κόσμο, έχοντας ουσιαστικά σταθερή πολυπλοκότητα, άρα η λύση μας μετατρέπεται σε αλγόριθμο με γραμμική πολυπλοκότητα (όπως θα δούμε και με άλλο τρόπο στο γ ερώτημα).

γ) Θα χρησιμοποιήσουμε τον αλγόριθμο του Camarini. Ουσιαστικά ο αλγόριθμος μας παίρνει έναν γράφο και μία λίστα με τα βάρη του, βρίσκει τον median των βαρών του, χωρίζει τον γράφο σε δύο υπογράφους με βάση τα 2 set που δημιουργεί, ένα με βάρη μεγαλύτερα ή ίσα του median (set A) και ένα με βάρη μικρότερα του median (set B). Στη συνέχεια, παίρνει το δάσος F του υπογράφου G_B, όπου ο υπογράφος G_B έχει για ακμές τις ακμές των οποίων τα βάρη ανήκουν στο set B. Αν αυτό το δάσος είναι ένα spanning tree, δηλαδή εάν μέσω διάσχισης με DFS είναι συνδεδεμένο, τότε ξανατρέχει τον αλγόριθμο για τον υπογράφο G_B με την ίδια λίστα βαρών. Αν δεν είναι spanning tree,

δηλαδή εάν μέσω διάσχισης με DFS δεν είναι συνδεδεμένο, τότε επιστρέφει την ένωση του δάσους F με ότι θα επιστρέψει ο αλγόριθμος που θα τον τρέξει για έναν υπογράφο G_A', ο οποίος θα προκύψει παίρνοντας τις super – vertices, οι οποίες δημιουργούνται ενώνοντας τις κορυφές κάθε αποσυνδεδεμένου μέρους του δάσους σε μία, μαζί με τις ακμές του set A και την ίδια λίστα βαρών. Γενικά, ο αλγόριθμος θα τερματίζει όταν ο αριθμός των ακμών είναι ίσος με 1, επιστρέφοντας τον ίδιο τον γράφο. Όσο το δάσος είναι συνδεδεμένο, δηλαδή δεν περνάει από όλες τις κορυφές, ο αλγόριθμος συνεχίζει τη διαδικασία με το median κτλπ. Όταν μετά από κάποιες αφαιρέσεις ακμών δεν είναι πλέον συνδεδεμένο, ψάχνει να βρει, χτίζοντας πάλι ουσιαστικά βήμα βήμα με εκτέλεση του αλγορίθμου για τον υπογράφου του set A με τις μεγαλύτερες σε βάρος κορυφές, το ένα συνδετικό δέντρο για το οποίο θα ξέρουμε σίγουρα ότι περιέχει τις ελάχιστες δυνατές, σε βάρος, ακμές, επομένως και η αυτονομία που απαιτείται για ένα ταξίδι από μία αρχική σε μία τελική κορυφή, που θα είναι το μέγιστο αυτών των ελάχιστων βαρών, θα είναι η ελάχιστη. Για να βρούμε, τώρα για συγκεκριμένο s,t την ελάχιστη αυτονομία L, αρκεί να τρέξουμε ένα τροποποιημένο DFS στο τελευταίο συνδετικό δέντρο, ξεκινώντας από την κορυφή s και μέχρι να συναντήσουμε την κορυφή t. Κάθε φορά που εκτελούμε το DFS για κάποια κορυφή, προσθέτουμε την τιμή του βάρους της ακμής που ενώνει αυτή με την προηγούμενη από την οποία καλέστηκε, σε μία στοίβα. Εάν φτάσουμε σε φύλλο που δεν είναι η t ή έχουμε ελέγξει όλα τα παιδιά μίας κορυφής, τότε αφαιρούμε την τιμή του βάρους της συνδετικής ακμής από τη στοίβα. Στο τέλος, στη στοίβα θα έχουν μείνει μόνο οι τιμές των βαρών των ακμών που αποτελούν το μονοπάτι από την s στην t, και διασχίζοντας τη μία φορά κρατώντας τη μέγιστη, έχουμε την απάντηση μας.

Για να αποδείξουμε ότι έχει γραμμική πολυπλοκότητα ο ανώτερος αλγόριθμος, θα δείξουμε ότι όλες οι διαδικασίες του έχουν γραμμική πολυπλοκότητα. Αρχικά, η εύρεση του median σε μη ταξινομημένο πίνακα έχει O(|E|) πολυπλοκότητα (με παρόμοια λογική σαν αυτή που χρησιμοποιείτε στην quicksort), η δημιουργία του set A, set B είναι επίσης O(|E|), αφού απλά διατρέχουμε τον πίνακα των βαρών έχοντας υπόψη μας τον median, η κατασκευή του δάσους με αλγόριθμο DFS είναι O(|E| + |V|) που είναι γραμμικό και πάλι, αφού για μη κατευθυνόμενους γράφους ξέρουμε ότι $|V| \leq 2|E|$, η δημιουργία των υπογράφων G_A, G_B είναι επίσης O(|E|), ο έλεγχος με DFS για το γράφο είναι επίσης O(|E|+|V|) = O(|E|) και το τρέξιμο της στοίβας είναι και αυτό O(|E|). Άρα, γενικά, βλέπουμε μόνο διαδικασίες O(|E|) και κάθε φορά ξέρουμε ότι ουσιαστικά μειώνονται στη μέση οι κορυφές άρα έχουμε O(|E| + |E|/2 + |E|/4 + … + 1) = O(|E|). Επομένως, ο αλγόριθμος αυτός τρέχει σε γραμμική πολυπλοκότητα.

Άσκηση 5:

Αρχικά παίρνουμε έναν πίνακα S[t] που σύμφωνα με την εκφώνηση έχει τιμή 0 για τις μέρες για τις οποίες δεν πηγαίνουμε στη σχολή και 1 για τις μέρες τις οποίες πηγαίνουμε στη σχολή. Παίρνουμε άλλον έναν πίνακα P[t] που για t = k μας δίνει την τιμή του

εισιτηρίου για k μέρες, ενώ στις υπόλοιπες θέσεις έχει τιμή 0. Επίσης έχουμε έναν πίνακα V[t] που θα συμπληρώσουμε αναδρομικά με το ελάχιστο κόστος για t ημέρες, έναν πίνακα E[t] που κρατάει το συνολικό αριθμό εισιτηρίων για κάθε τύπο.

Ξεκινάμε να υπολογίσουμε τον πίνακα V[t]. Μετράμε τις μέρες με το i ξεκινώντας από i=1. Αν S[i] = 1 και i>1, χρησιμοποιούμε τον τύπο $V[i] = \min_{k \le i}(V[i-k] + P[k])$, και στη συνέχεια υπολογίζουμε το ελάχιστο κόστος των υπόλοιπων ημερών με βάσει τις προηγούμενες μέρες και τους διαθέσιμους τύπους εισιτηρίων. Αν S[1] = 1, τότε V[1] = P[1]. Ο πάνω τύπος θα εκτελείται με τη μορφή μιας for loop η οποία θα κρατάει το ελάχιστο κόστος V[i] (αρχικά άπειρο ή ο μεγαλύτερος διαθέσιμος αριθμός σε έναν τύπο δεδομένων μίας συγκεκριμένης γλώσσας) που θα ψάχνει διαθέσιμα k>0 ξεκινώντας από i=k και κατεβαίνοντας ένα ένα κάθε φορά στον πίνακα P[k], για τα οποία P[k] > 0, δηλαδή για τα οποία υπάρχει κάποιος διαθέσιμος τύπος εισιτηρίου. Αν S[i] = 0 και i>1, τότε V[i] = V[i-1]. Αν S[1] = 0, τότε V[1] = 0.

Κατά τη διαδικασία υπολογισμού του V[i] θα συμπληρώνουμε ταυτόχρονα και τον πίνακα E[t]. Συγκεκριμένα, θα κρατάμε και άλλη μία μεταβλητή, η οποία θα ανανεώνεται κάθε φορά που ανανεώνεται και το ελάχιστο της for loop και θα κρατάει το k. Στο τέλος κάθε υπολογισμού του V[i] θα αυξάνουμε κατά ένα το E[k]. Στο τέλος τα μη μηδενικά στοιχεία του E[t] θα μας δίνουν τον αριθμό (E[k]) του τύπου του εισιτηρίου k-ημερών.

Ο αλγόριθμος τρέχει μία for loop από 1 έως T που υπολογίζει τα V[i] και εσωτερικά της μία ακόμα for loop από τον αριθμό του εξωτερικού loop έως 1, που θα υπολογίζει το ελάχιστο των αθροισμάτων, επομένως θα χρειάζεται κάθε φορά να ελέγχει ένα μεγάλο μέρος του πίνακα με τα κόστη των εισιτηρίων, κάτι που θα έχει στη χειρότερη περίπτωση $O(T^2)$ πολυπλοκότητα. Η εύρεση του συνδυασμού των εισιτηρίων έχει γραμμική πολυπλοκότητα O(T). Άρα, τελικά έχουμε πολυπλοκότητα $O(T^2)$ στη χειρότερη περίπτωση.

Άσκηση 6:

α) Για n = 100 και k = 1 η βέλτιστη σειρά δοκιμών είναι να αρχίσω από το n = 1 και να κάθε φορά να ανεβαίνω κατά ένα εκατοστό, μέχρι να σπάσει το μοναδικό μου ποτήρι. Έστω ότι σπάει στο επίπεδο k, θα ξέρω τότε ότι το μέγιστο επίπεδο από το οποίο δεν σπάνε τα ποτήρια αν πέσουν θα είναι το k - 1. Μέγιστο πλήθος δοκιμών είναι το 99 (O(n)), δεδομένου ότι σίγουρα τα ποτήρια σπάνε σε κάποιο ύψος n, αφού η χειρότερη περίπτωση είναι να σπάει το ποτήρι για n = 100, όπου θα δοκιμάσω μέχρι το n=99. Λόγω της διάθεσης ενός μόνο ποτηριού, ο αλγόριθμος αυτός δεν έχει καλή πολυπλοκότητα, ούτε ομαλά κατανεμημένο μέγιστο πλήθος δοκιμών.

Για n = 100 και k = 2 θα χρησιμοποιήσουμε το πρώτο ποτήρι για να προσδιορίσουμε ένα εύρος για το ύψος που σπάνε τα ποτήρια, ώστε με το δεύτερο ποτήρι να διατρέξουμε

αυτό το διάστημα όπως στην περίπτωση n = 100 και k = 1. Ιδανικά θέλουμε ο αλγόριθμος μας να είναι αποδοτικός για κάθε περίπτωση, χωρίς να έχει μεγάλες διαφορές στο μέγιστο πλήθος δοκιμών για τις χειρότερες περιπτώσεις. Επομένως, θα χρησιμοποιήσουμε το πρώτο ποτήρι ανεβαίνοντας σταδιακά ύψη προσπαθώντας να διατηρήσουμε το μέγιστο πλήθος δοκιμών σταθερό. Καθώς κάθε φορά που ανεβαίνουμε ύψος με το πρώτο ποτήρι, αυξάνονται κατά μία οι δοκιμές, μας συμφέρει το εύρος, σε περίπτωση που σπάσει το πρώτο ποτήρι και συνεχίσουμε με το δεύτερο ποτήρι με τη λογική του α ερωτήματος, να μικραίνει κατά ένα. Δηλαδή οι τιμές των αλμάτων – ελέγχων για το πρώτο ποτήρι θα είναι η λύση της εξίσωσης x + (x -1) + (x - 2) + (x - 3) + … + 2 + 1 = [x * (x + 1)] / 2 = 100. Από αυτήν λαμβάνουμε x^2 + x = 200 ή x^2 + x - 200 = 0, θετική λύση της οποίας είναι το 13,65 και με στρογγυλοποίηση προς τα πάνω έχουμε x = 14. Άρα αρχικά θα έχουμε άλμα x = 14, αν δεν σπάσει το πρώτο ποτήρι άλμα 13, δηλαδή n = 27, έπειτα άλμα 12, δηλαδή n = 39 κτλπ. Όταν σπάσει το πρώτο ποτήρι ακολουθεί παρόμοια διαδικασία με το ερώτημα α για το δεύτερο ποτήρι, μόνο που το εύρος n θα είναι σημαντικά περιορισμένο. Στη χειρότερη περίπτωση θα έχουμε μέγιστο πλήθος δοκιμών σταθερό και ίσο με 14 (O(√2n) = O(√n)), είτε για n = 14, όπου έχουμε 1 δοκιμή για το πρώτο ποτήρι και 13 δοκιμές για το δεύτερο, σύνολο 14, είτε για n = 27, όπου έχουμε 2 + 12 για το πρώτο και δεύτερο αντίστοιχα και ομοίως για κάθε n που ανήκει στην ακολουθία του αθροίσματος των αλμάτων.

Γενικεύοντας για κάθε n και k = 2, θα λύνουμε κάθε φορά την εξίσωση x + (x -1) + (x - 2) + (x - 3) + … + 2 + 1 = [x * (x + 1)] / 2 = n, δηλαδή x^2 + x - 2*n = 0 και τη θετική της λύση θα τη στρογγυλοποιούμε προς τα πάνω, όπου αυτό είναι απαραίτητο. Έτσι θα έχουμε τα άλματα των δοκιμών για το πρώτο ποτήρι, το οποίο θα μειώνεται κατά 1 με κάθε επιπλέον άλμα, και θα ξέρουμε και το μέγιστο πλήθος δοκιμών, το οποίο θα ισούται με το πρώτο άλμα x (θετική λύση της παραπάνω εξίσωσης).

β) Για να γενικεύσουμε τον αλγόριθμο για οποιαδήποτε n, k θα χρησιμοποιήσουμε μία ένωση δύο αλγορίθμων. Ο ένας θα είναι η δυαδική αναζήτηση και ο άλλος μία αναδρομική λύση με βάση το ερώτημα α. Αρχικά, ένα έχουμε k ≥ $\log_2$n, τότε μας συμφέρει να χρησιμοποιήσουμε τη δυαδική αναζήτηση, μέσω της οποίας θα έχουμε απάντηση σε $\log_2$n βήματα στη χειρότερη περίπτωση. Σε άλλη περίπτωση, δηλαδή, όπου k < $\log_2$n, πρέπει να γενικεύσουμε τον αλγόριθμο του ερωτήματος α μέσω αναδρομικών σχέσεων, για οποιοδήποτε n, k. Ουσιαστικά, για k = 2, κάθε φορά υπολογίζουμε το αρχικό άλμα μέσω της εξίσωσης x + (x -1) + (x - 2) + (x - 3) + … + 2 + 1 = [x * (x + 1)] / 2 = n, ενώ στη συνέχεια, εάν δεν σπάσει το ποτήρι, λύνουμε την ίδια εξίσωση για να βρούμε το δεύτερο άλμα, αλλά αυτή τη φορά για n – x ύψος, αφού θα έχουμε αποκλείσει τα x πρώτα εκατοστά, δηλαδή λύνουμε μια μορφή της ίδιας εξίσωσης, (x -1) + (x - 2) + (x - 3) + … + 2 + 1 = n – x, όπου θα ξέρουμε ότι το άλμα θα είναι το προηγούμενο μειωμένο κατά ένα. Σε περίπτωση που σπάσει βεβαίως, καλύπτουμε όλο το διάστημα x εκατοστών με το δεύτερο ποτήρι, ξεκινώντας από την αρχή και ανεβαίνοντας ένα εκατοστό τη φορά. Επομένως, σε μία γενικότερη περίπτωση με k > 2 θα εφαρμόσουμε την ίδια λογική με το

πρώτο ποτήρι του k = 2, απλά για περισσότερα ποτήρια. Επομένως, για να βρούμε το πρώτο άλμα θα λύσουμε την $x_k * (x_k + 1) / 2 = n$, ενώ στη συνέχεια για να βρίσκουμε τα επόμενα άλματα, καθώς και τα πρώτα άλματα του επόμενου ποτηριού θα χρησιμοποιούμε τον τύπο $x_{i-1} * (x_{i-1} + 1) / 2 = x_i$. Φυσικά, όταν i = 1 τότε θα ελέγχουμε το μήκος όλου του διαστήματος που έχει απομείνει, όπως κάναμε και στην περίπτωση του k = 2. Ο αλγόριθμος της δυαδικής αναζήτησης έχει πολυπλοκότητα $O(\log_2 n)$, ενώ ο δεύτερος αναδρομικός αλγόριθμος στη χειρότερη περίπτωση θα κάνει $O(x_{k-1})$ βήματα, άρα έχει πολυπλοκότητα της τάξης $O(\sqrt{n})$. Άρα, στη χειρότερη περίπτωση, ο αλγόριθμος μας έχει $O(\sqrt{n})$ πολυπλοκότητα, αφού $O(\sqrt{n}) > O(\log_2 n)$.

γ) Σε C++ ο αλγόριθμος είναι ο παρακάτω. Ουσιαστικά, τρέχει με δοσμένα n, k, t και επιστρέφει τον αριθμό των ελέγχων μέχρι να βρει το target. Έχουμε υλοποίηση για τους δύο διαφορετικούς αλγορίθμους, την binary_search και την recursive. Στην πρώτη, συγκρίνουμε το mid με το t (target, το σημείο από το οποίο και μετά τα ποτήρια σπάνε) και αναλόγως διαλέγουμε το διάστημα και τρέχουμε αναδρομικά τη συνάρτηση προσθέτοντας 1 έλεγχο στην επιστροφή της αρχικής κλήσης, μέχρι να βρούμε το t με το mid και να επιστρέψουμε τον τελικό μας έλεγχο. Σε αυτήν τη συνάρτηση δεν μας νοιάζει καθόλου πόσα ποτήρια χαλάμε, ή πόσα θα μας μείνουν, καθώς με τον έλεγχο $k \geq \log_2 n$ είμαστε σίγουροι, πως αρκούν και είναι και ο πιο αποδοτικός τρόπος. Στη συνέχεια, με την recursive, εάν έχουμε ένα ποτήρι, τρέχουμε όλο το διάστημα ανεβαίνοντας κατά ένα εκατοστό κάθε φορά, προσθέτοντας έναν έλεγχο στην επιστροφή της αρχικής κλήσης, μέχρι να βρούμε το target και να τερματίσει η συνάρτηση. Σε διαφορετική περίπτωση, υπολογίζουμε το j (jump, άλμα) για το ποτήρι μας και προχωράμε. Αν μέσω του jump μας έχουμε δεν έχουμε περάσει το t, καλούμε αναδρομικά, προσθέτοντας έναν έλεγχο στην επιστροφή της αρχικής κλήσης, τη συνάρτηση για μεγαλύτερο l (low, πάτωμα) και με όρισμα στο τέλος h - j (με τη λογική του $(x - 1) + (x - 2) + (x - 3) + \ldots + 2 + 1 = n - x$). Αν περάσουμε το t, καλούμε αναδρομικά, με την ίδια προσθήκη ενός ελέγχου, τη συνάρτηση για μικρότερο h (high, ταβάνι), k - 1 ποτήρια και όρισμα στο τέλος j (με τη λογική του $x_{i-1} * (x_{i-1} + 1) / 2 = x_i$).

```cpp
#include <iostream>
#include <cmath>
using namespace std;

int binary_search(int l, int h, int t) {
    if(h >= l) {
        int mid = l + (h - l) / 2;

        if(mid == t)
            return 1;
        if(mid > t)
            return binary_search(l, mid - 1, t) + 1;
        return binary_search(mid + 1, h, t) + 1;
    }
    else
        return 0;
}

int recursive(int l, int h, int k, int t, int j) {
    if(k == 1) {
        if(l == t)
            return 1;
        else
            return recursive(l + 1, h, k, t, j) + 1;
    }
    else {
        j = ceil(-0.5 + sqrt(1 + 8*j) / 2);

        if (l + j < t) {
            return recursive(l + j + 1, h, k, t, h - j) + 1;
        }
        else {
            return recursive(l, l + j - 1, k - 1, t, j) + 1;
        }
    }
}

int main() {
    int n, k, t;
    cin >> n, k, t;
    if(k >= log2(n))
        cout << binary_search(0, n, t);
    else
        cout << recursive(0, n, k, t, n);
}
```