Zadanie 1 (10 pkt. na pracowni, później 5 pkt.). To zadanie ma dwie niezależne części, należy zatem przesłać/przedstawić dwa programy.

A. Napisz program, który wczytuje ze standardowego wejścia dodatnie liczby *k* oraz *n* (*n* ≤ 30), a następnie *n* liczb naturalnych oznaczających wagi kolejnych elementów w pewnym *n*-elementowym zbiorze. Chcemy wyliczyć, ile podzbiorów zadanego zbioru elementów ma sumaryczną wagę nie większą niż *k*. Do wygenerowania wszystkich podzbiorów można użyć odpowiedniej rekurencji, ale w tym zadaniu użyjemy innego podejścia.

Liczby od 0 do 2^*n*-1 możemy traktować jako reprezentacje wszystkich podzbiorów, gdzie podzbiór zawiera element numer *i* wtedy i tylko wtedy, gdy bit *i* jest zapalony (równy 1). Np. 45 = 101001 oznacza podzbiór (indeksów) {0, 3, 5}, gdzie liczymy bity od najmniej znaczącego (i od zera). Należy więc iterować przez odpowiedni zakres liczb, gdzie każda liczba reprezentuje dokładnie jeden podzbiór, i każdorazowo wyliczać sumę wag elementów zawartych w podzbiorze, a jeśli nie zostanie przekroczone *k*, to zwiększać wynikowy licznik.

Przykładowo, dla wejścia

```
2 3
1 1 1
```

odpowiedź to 7 – możemy wybrać dowolny podzbiór poza całym zbiorem. Natomiast dla wejścia

```
45 5
30 20 1 10 50
```

poprawna odpowiedź to 12: zawsze możemy wziąć lub nie wziąć elementu o wadze 1 oraz (niezależnie) 10, a spośród elementów o wagach 20 i 30 możemy wybrać najwyżej jeden (trzy możliwości), zatem 2x2x3=12.

B. Napisz program wyszukujący wzorzec w binarnym tekście (tj. nad dwuelementowym alfabetem {0, 1}). Powinien on wczytać ze standardowego wejścia dwa ciągi zerojedynkowe (podane w osobnych wierszach). Pierwszy ciąg, o długości co najwyżej 60 znaków, to wzorzec, którego wszystkie wystąpienia chcemy znaleźć w drugim ciągu. Program powinien wypisać indeksy początków wszystkich wystąpień wzorca w tekście (tj. drugim ciągu). Sprawdzenie pojedynczej pozycji należy wykonać w czasie niezależnym od długości wzorca (nie należy zatem porównywać znak po znaku, tj. bit po bicie).

Możemy traktować wzorzec jako liczbę, np. 0110 to 6, i przechodząc przez tekst rozważać każde okienko sąsiednich czterech (w tym przypadku) znaków jako liczbę, którą należy porównać ze wzorcem. Nietrudno przesuwać się pomiędzy kolejnymi okienkami – wystarczy przesunięcie bitowe (które zgubi najstarszy bit, ale to dobrze) i

dopisanie jednego bitu w najmniej znaczącej pozycji. Możemy zatem przy pomocy kilku operacji utrzymywać reprezentację liczbową dla przesuwającego się okienka odpowiedniej długości bitów w tekście.

Przykładowo, dla wejścia

```
010
10101011010111010101011
```

należy wypisać

```
1 3 8 14 16 18
```

(wystąpienia indeksujemy oczywiście od zera).

Uwaga: w tym zadaniu (także w części A) należy użyć operatorów bitowych, np. przesunięć i bitowego AND zamiast (reszty z) dzielenia przez 2. Pliki źródłowe przesłane na SKOS powinny nazywać się *6.1zbior.c* i *6.1wzorzec.c*.

Zadanie 2 (10 pkt.). Rozważamy dwie tablice *S* i *T* tego samego rozmiaru, zawierające liczby naturalne z zakresu [1, 1000], bez powtórzeń w obrębie jednej tablicy. Chcemy (szybko) odpowiadać na zapytania, czy podtablice *S*[*i*...*i*+*k*] i *T*[*j*...*j*+*k*] zawierają dokładnie te same liczby (niekoniecznie w tej samej kolejności). Jednym z rozwiązań jest haszowanie, tzn. przypisywanie podzbiorom (pseudo)losowych liczb je reprezentujących w ten sposób, że dwa różne podzbiory z bardzo niskim prawdopodobieństwem otrzymają taką samą liczbę-reprezentanta. (Haszowanie to bardzo szeroka koncepcja o licznych realizacjach i zastosowaniach, tutaj jest to tylko jeden z wielu przykładów.)

Najpierw wypełniamy pomocniczą tablicę *r*[1001] losowymi liczbami (p. wyjaśnienia w zadaniu z listy 4). Haszem liczby *i* będzie *r*[*i*], natomiast haszem zbioru wielu liczb będzie XOR (bitowa alternatywa wykluczająca) ich haszy. Potrzebujemy również tablic prefiksowych, które zawierają XORy haszy wszystkich prefiksów każdej tablicy, tzn. tablicy prefiksowej *P* dla *S* takiej, że

- *P*[0] = *r*[*S*[0]],
- *P*[1] = *r*[*S*[0]] ^ *r*[*S*[1]],
- *P*[2] = *r*[*S*[0]] ^ *r*[*S*[1]] ^ *r*[*S*[2]] itd.,

jak również *Q* zdefiniowanej analogicznie dla *T*. Przygotowawszy takie struktury, możemy w czasie stałym uzyskać hasz dowolnej podtablicy *S*[*i*...*i*+*k*], którym (z własności operacji XOR – jest przemienna i odwrotna do samej siebie, tj. *x* ^ *y* ^ *y* = *x*) będzie *P*[*i*+*k*] ^ *P*[*i*-1] (zastanów się, co należy zrobić, gdy *i* = 0). Wreszcie (również z przemienności XOR) stwierdzamy, że *S*[*i*...*i*+*k*] zawiera dokładnie ten sam zbiór liczb co *T*[*j*...*j*+*k*] wtedy i tylko wtedy, gdy ich hasze są takie same.

Program powinien wczytać długość tablic *S* i *T* i kolejno te dwie tablice, następnie liczbę zapytań *z*, a później *z* zapytań o równość zbiorów na przedziałach, gdzie pojedyncze zapytanie składa się z trzech liczb *i*, *j*, *k*. Przykładowo, dla:

```
8
1 2 3 4 5 6 7 8
7 5 6 8 4 3 2 1
5
0 1 0
7 3 0
0 0 7
0 4 2
0 4 3
```

kolejne odpowiedzi to

- NIE – 1 ≠ 5,
- TAK – 8 = 8,
- TAK – całe tablice mają tę samą zawartość, tylko w innej kolejności,
- NIE – {1, 2, 3} ≠ {4, 3, 2},
- TAK – w obu tablicach na zadanych pozycjach znajduje się zbiór {1, 2, 3, 4}.

Zadanie 3. Powiemy, że słowo *a* da się ułożyć ze słowa *b*, jeśli możemy otrzymać *a* z *b* przez ew. wyrzucenie pewnych liter i ew. przestawienie pozostałych. Np. "kot" jest układalne z "motyka", ale "kotka" nie, bo brakuje jednej litery "k". W tym zadaniu będziemy rozważać zestawy słów i rozstrzygać, czy w danym zestawie jest słowo:

1. z którego da się ułożyć każde inne słowo z zestawu;
2. które da się ułożyć z każdego innego słowa z zestawu;
3. którego nie da się ułożyć z żadnego innego słowa z zestawu;
4. z którego nie da się ułożyć żadnego innego słowa z zestawu.

Wejście

W pierwszym wierszu wejścia znajduje się naturalna liczba $n \leq 16$, a w każdym z kolejnych *n* wierszy opisany jest osobny zestaw słów, dla którego należy wykonać powyższą analizę. Każdy taki wiersz zawiera na początku naturalną liczbę $k \leq 10000$, a po niej *k* niepustych słów składających się z drukowalnych znaków ASCII o kodach nie mniejszych niż 63 (czyli począwszy od *'?'*, w szczególności bez spacji i innych białych znaków). Słowa oddzielone są od siebie (i od *k*) pojedynczymi spacjami i mają długość nie przekraczającą 255.

Wyjście

Jako odpowiedź należy podać jedną liczbę kodującą odpowiedzi na powyższe pytania dla wszystkich zestawów w następujący sposób: odpowiedź twierdząca to 1, przecząca – 0; odpowiedzi dla pojedynczego zestawu mają być zapisane na czterech kolejnych bitach w kolejności takiej, w jakiej zostały wyżej podane pytania, począwszy od najmniej

znaczącego bitu; odpowiedzi dla kolejnych zestawów mają być zapisane na kolejnych czwórkach bitów, począwszy od najmniej znaczącej czwórki. Bity spoza $4 \times n$ najmłodszych mają być wyzerowane, a liczbę należy wypisać w systemie dziesiętnym.

Przykłady

Przykład A

Wejście

```
1
8 ij ji jk kj ijk kij khi ihk
```

Wyjście

```
0
```

W jedynym zestawie nie istnieją słowa z żadną z opisanych własności.

Przykład B

Wejście

```
3
3 ab a b
10 [@ @[ ^ @[@ ]^ [^@@] ]^@ @][@ ^ @]^
2 YYZ ZYY
```

Wyjście

```
861
```

W pierwszym zestawie słowo "ab" ma własności 1 i 3, a "a" oraz "b" własność 4, więc odpowiedź to (dwójkowo) 1101. W drugim zestawie najdłuższe słowo ma własności 1 i 3 i na tym koniec, więc dostajemy 0101. Ostatni zestaw składa się wyłącznie z permutacji tego samego zestawu liter, więc oba jego słowa mają własności 1 i 2, a odpowiedź to 0011.

Przykład C

Wejście

```
4
8 ij ji jk kj ijk kij khi ihk
3 ab a b
10 [@ @[ ^ @[@ ]^ [^@@] ]^@ @][@ ^ @]^
2 YYZ ZYY
```

Wyjście

```
13776
```

Umieszczenie jedynego zestawu z przykładu A (dla którego odpowiedź to 0) przed wszystkimi zestawami z przykładu B odpowiada przesunięciu całej odpowiedzi dla B o cztery bity w lewo, czyli pomnożeniu przez 16.

Uwagi

W tym zadaniu w sprawdzaczce została włączona flaga kompilacji *-O2* skutkująca optymalizacją kodu wynikowego. W połączeniu z innymi naszymi flagami kompilacji,

spowoduje to pojawienie się pewnych nowych ostrzeżeń ze strony kompilatora. W związku z tym:

- testując rozwiązanie na swoich komputerach również warto używać tej flagi kompilacji, żeby móc zawczasu pozbyć się tych błędów;
- prawie na pewno zobaczą Państwo błąd związany z gubieniem wartości wynikowej *scanf* – o ile w bardziej użytkowych programach warto robić z tym coś mądrzejszego (np. przy użyciu instrukcji warunkowej odpowiednio obsłużyć pojawienie się niepoprawnego wejścia) tak na sprawdzaczce poprawność wejścia jest gwarantowana i wystarczy przypisać tę wartość do jakiejś jeszcze nieużywanej zmiennej, którą później możemy czymś nadpisać.