Zadanie 2.

2. (1pkt) Udowodnij, że algorytm Kruskala znajduje minimalne drzewa spinające poprzez przyrównanie tych drzew do drzew optymalnych.

Weźmy dowolny spójny graf G. Niech T będzie wynikiem algorytmu Kruskala na G. Jak pierwszy raz napotkamy na krawędź łączącą dwie spójne składowe to na pewno ją weźmiemy, więc T jest spójne.
Minimalność:
Założenie indukcyjne: Niech F będzie zbiorem wybranych krawędzi w danym kroku. Istnieje drzewo optymalne, którego podzbiorem jest F i które nie zawiera żadnej krawędzi, którą algorytm odrzucił.
1. F jest puste :)
2. jeśli następna krawędź nie należy do MST to możemy ją do niego domontować. Powstał cykl, w którym jest jakaś krawędź f, której waga jest większa od e. Wywalamy f do kosza i mamy prawdziwe MST.

Zadanie 3.

Danych jest n odcinków $I_j =< p_j, k_j >$, leżących na osi OX, $j = 1, \dots, n$.
Ułóż algorytm znajdujący zbiór $S \subseteq \{I_1, \dots, I_n\}$, nieprzecinających się odcinków, o największej mocy.

```
GreedySelect(I[1,…,n])
sort(I) // sortuj rosnąco względem k_i
S = {I[1]} // optymalny zbiór
index = 1 // indeks ostatniego odcinka dodanego do S
for i in 2 to n
        if (p[i] ≥ k[index]) // jeśli akt. odcinek nie przecina się z ostatnim z S
                S.push(I[i])
                index = i
return I
```

Wyjaśnienie działania:
Na początku algorytmu sortujemy tablicę odcinków względem ich końca. Potem w każdym kroku dodajemy do S najwcześniej zaczynający się odcinek, który nie przecina się z żadnym z S.

Dowód nie wprost:
Niech rozwiązanie zwrócone przez GreedySelect nazywa się S, natomiast bardziej optymalne rozwiązanie to A. Niech i będzie indeksem pierwszej pozycji, na której te rozwiązania się różnią. Wtedy rozpatrzmy przypadki:
1. $(S)k_i > (A)k_i$ czyli i-ty odcinek A kończy się wcześniej niż S.
Ale nasz algorytm zawsze wybiera odcinki najwcześniej kończące się,

a to by oznaczało, że $(A)k_i$ musiałby zostać wybrany przed $(S)k_i$, co jest sprzeczne z założeniem.
2. $(S)k_i = (A)k_i$ wtedy oba algorytmy mają ten sam wybór, więc sprzeczność z założeniem, że A jest lepsze od S.
3. $(S)k_i < (A)k_i$ czyli i-ty odcinek A kończy się później niż S.
Wtedy rozwiązanie A jest co najwyżej tak samo dobre co S, czyli sprzeczność.

Zadanie 4.

Rozważ następującą wersję problemu wydawania reszty: dla danych liczb naturalnych a, b (a ≤ b) chcemy przedstawić ułamek $\frac{a}{b}$ jako sumę różnych ułamków o licznikach równych 1.
Udowodnij, że algorytm zachłanny zawsze daje rozwiązanie.
Czy zawsze jest to rozwiązanie optymalne (tj. o najmniejszej liczbie składników)?

Algorytm zachłanny jest postaci:

$$\frac{a}{b} = \frac{1}{\left\lceil \frac{b}{a} \right\rceil} + alg\left(\frac{a}{b} - \frac{1}{\left\lceil \frac{b}{a} \right\rceil}\right)$$

Dowód poprawności:
Zacznijmy od tego, że algorytm zawsze się kończy, bo:

$$\frac{a}{b} - \frac{1}{\left\lceil \frac{b}{a} \right\rceil} < \frac{1}{\left\lceil \frac{b}{a} \right\rceil}$$

$$\frac{a}{b} < \frac{2}{\left\lceil \frac{b}{a} \right\rceil}$$

$$\frac{a}{b}\left\lceil \frac{b}{a} \right\rceil < 2$$

Zatem skoro w każdym kolejnym kroku wykonujemy działania na coraz mniejszych liczbach, to w pewnym momencie otrzymamy wynik.

Algorytm nie zawsze zwraca rozwiązanie optymalne, kontrprzykład:

$\frac{a}{b} = \frac{9}{20}$ ma rozwiązanie optymalne $\frac{1}{4} + \frac{1}{5}$, jednak algorytm zwróci rozwiązanie nieoptymalne:

$$\frac{9}{20} = \frac{1}{\left\lceil \frac{20}{9} \right\rceil} + alg\left(\frac{9}{20} - \frac{1}{\left\lceil \frac{20}{9} \right\rceil}\right) = \frac{1}{3} + alg\left(\frac{9}{20} - \frac{1}{3}\right) = \frac{1}{3} + alg\left(\frac{7}{60}\right) =$$

$$= \frac{1}{3} + \frac{1}{\left\lceil \frac{60}{7} \right\rceil} + alg\left(\frac{7}{60} - \frac{1}{\left\lceil \frac{60}{7} \right\rceil}\right) = \frac{1}{3} + \frac{1}{9} + alg\left(\frac{7}{60} - \frac{1}{9}\right) =$$

$$= \frac{1}{3} + \frac{1}{9} + alg\left(\frac{1}{180}\right) = \frac{1}{3} + \frac{1}{9} + \frac{1}{180}$$

Zadanie 5. (1,5pkt)

Ułóż algorytm, który dla danego n-wierzchołkowego drzewa i liczby k, pokoloruje jak najwięcej wierzchołków tak, by na każdej ścieżce prostej było nie więcej niż k pokolorowanych wierzchołków.

Obserwacja 1:
Jeśli k = 1, to można pomalować tylko 1 wierzchołek.
Gdybyśmy pomalowali 2 wierzchołki, to ze spójności drzewa istniałaby ścieżka między nimi z 2 pomalowanymi wierzchołkami, co jest sprzecznością z k = 1.

Obserwacja 2:
Najbardziej opłaca nam się kolorować liście.
Przykładowo, jeśli rozważymy drzewo pełne głębokości 3 (7 wierzchołków) oraz k=2, to możemy pokolorować 4 liście. Jeśli zamiast któregoś z liści pokolorowalibyśmy inny wierzchołek, to istniałaby ścieżka z 3 wierzchołkami pokolorowanymi, sprzeczność z k=2, czyli kolorowanie liści jest optymalne.

Obserwacja 3:
Jeśli k > 2, to możemy pokolorować wszystkie liście, a następnie powtórzyć proces dla $k = \left\lfloor \frac{k}{2} \right\rfloor$ oraz drzewa bez liści.
Łącząc tą obserwację z obserwacją pierwszą okazuje się, że dla k=3 możemy pokolorować maksymalnie L+1 wierzchołków, gdzie L to ilość liści.

Stąd algorytm kolorowania wierzchołków wygląda następująco:
ColorTree(T, k)

```
T' = T
for-each vertice v in T:
        colored = false
while (k > 1)
        for-each leaf in T':
                colored = true //własność T a nie T'
        k = k div 2
        remove all leafs from T'
if (k = 1)
        for 1 random leaf t of T':
        colored = true
return T
```

Zadanie 6. (2 pkt)

6. (2pkt) Ułóż algorytm, który dla danego spójnego grafu G oraz krawędzi e sprawdza w czasie $O(n+m)$, czy krawędź e należy do jakiegoś minimalnego drzewa spinającego grafu G. Możesz założyć, że wszystkie wagi krawędzi są różne.

Krawędź e nie jest maksymalną krawędzią żadnego cyklu ⟺ należy do MST

=> Nie wprost

Jeżeli nie leży na żadnym cyklu to jest mostem i należy do MST.

Jeżeli leży na jakimś cyklu, to możemy ją zamontować do MST i wywalić tą najcięższą krawędź, teraz mamy prawdziwe MST, z e.

<= Nie wprost

Wywalmy e do kosza. MST nam się rozspójniło na dwie spójne składowe, które w oryginalnym grafie łączyły przynajmniej dwie krawędzie na cyklu, którego e było maksymalną krawędzią. Weźmy tą lżejszą i mamy prawdziwe MST.

Ale zarówno Prim jak i Kruskal działają w czasie gorszym niż O(n + m),
zatem zamiast budować MST i sprawdzać czy e do niego należy wykonamy DFS z jednego z wierzchołków e (nieważne którego).
Jeśli w wyniku DFS dojdziemy do drugiego końca e to znaczy, że e jest maksymalną krawędzią jakiegoś cyklu, czyli nie należy do MST.
W p.p. gdy sprawdzimy każdy wierzchołek (korzystając tylko z krawędzi tańszych od e) i nie dotrzemy do v, to e należy do MST.

```
CheckMST(G, e)
return DFS(G, cost(e), e.start, e.end)
```

```
DFS(G, maxcost, cur, end)
if (cur == end)
        return false
for-each v – neighbour of cur:
        if cost(cur, v) < maxcost:
                result = DFS(G, maxcost, v, end)
                if !result:
                        return false
return true
```

Zadanie 7. (2pkt)

System złożony z dwóch maszyn A i B wykonuje n zadań.
Każde z zadań wykonywane jest na obydwu maszynach, przy czym wykonanie zadania na maszynie B można rozpocząć dopiero po zakończeniu wykonywania go na maszynie A. Dla każdego zadania określone są dwie liczby naturalne a_i, b_i określające czas wykonania i-tego zadania na maszynie A oraz B (odpowiednio).
Ułóż algorytm ustawiający zadania w kolejności minimalizującej czas zakończenia wykonania ostatniego zadania przez maszynę B.

Niech x_i oznacza czas „zmarnowany" na oczekiwanie przez maszynę B na wykonanie i-tego zadania przez maszynę A.
Łatwo zauważyć, że zawsze $x_1 = a_1$.
Następnie mamy $x_2 = \max(0, a_1 + a_2 - b_1 - x_1)$, gdyż aby wykonać drugie zadanie na B, najpierw musimy wykonać pierwsze na obu oraz drugie na A oraz odejmujemy x_1, bo nie chcemy 2 razy liczyć tego samego oczekiwania.
Jeśli A wykonało 2 zadania przed pierwszym B, to ta suma jest ujemna, zatem bierzemy czas oczekiwania równy 0 (bo po wykonaniu 1 w B od razu można zrobić 2 w B). Indukcyjnie można wywnioskować stąd wzór:

$$x_n = \max\left(0, \sum_{k=1}^{n} a_k - \sum_{k=1}^{n-1} b_k - \sum_{k=1}^{n-1} x_k\right)$$

Stąd wynika, że:

$$\sum_{k=1}^{n} x_k = \max_{1 \le i \le n}(K_i)$$

Gdzie:

$$K_n = \sum_{k=1}^{n} a_k - \sum_{k=1}^{n-1} b_k$$

Do optymalnego ułożenia stosujemy zasadę Johnsona która mówi, żeby:

1. Wybrać najkrótsze zadanie z nieprzydzielonych w A lub B,
a) Jeśli dla niego A < B, to zadanie to wrzucamy na początek,
b) Jeśli dla niego A > B, to zadanie to wrzucamy na koniec,
c) Jeśli dla niego A = B, to nie ma znaczenia czy damy je na początek czy koniec.
2. Usunąć to zadanie z listy zadań zarówno A, jak i B.
3. Powtarzać kroki 1,2 tak długo, aż wszystkie zadania zostaną przydzielone.

Pseudokod:

```
struct Job
{
    int a, b, idx;
    bool operator<(Job o) const
    {
        return min(a, b) < min(o.a, o.b);
    }
};

// ustala kolejność zadań maszyn
vector<Job> johnsons_rule(vector<Job> jobs)
{
    sort(jobs.begin(), jobs.end());
    vector<Job> a, b;
    for (Job j : jobs)
    {
        if (j.a < j.b)
            a.push_back(j);
        else
            b.push_back(j);
    }
    a.insert(a.end(), b.rbegin(), b.rend());
    return a;
}

// oblicza czas wykonania zadań na maszynie B (nie jest konieczne do zadania)
pair<int, int> finish_times(vector<Job> const& jobs)
{
    int t1 = 0, t2 = 0;
    for (Job j : jobs)
    {
        t1 += j.a;
```

```
        t2 = max(t2, t1) + j.b;
    }
    return make_pair(t1, t2);
}
```

Zadanie 8. (2pkt)

Niech T = (V, E) będzie drzewem a P(u, v) niech oznacza ścieżką w T (rozumianą jako zbiór krawędzi) łączącą wierzchołki u i v.
Ułóż algorytm, który dla drzewa T znajduje trzy wierzchołki a, b, c, dla których zbiór $\{e \in E: e \in P(a,b) \cup P(a,c) \cup P(b,c)\}$ jest maksymalnie duży.

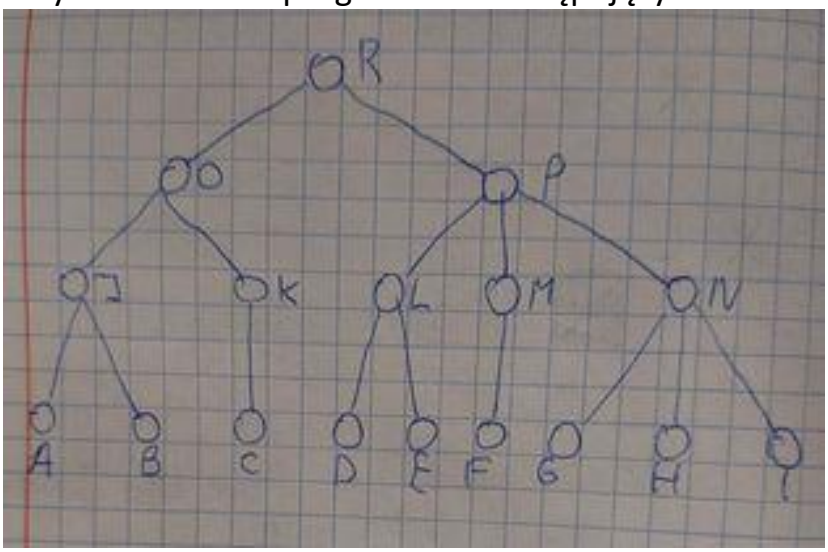
Zadanie jest trudniejszą wersją zadania 1b z listy 1, zatem pomysł jest podobny – DFS i każde wywołanie rekurencyjne zwraca długość 3 najdłuższych, rozłącznych ścieżek wraz z liśćmi, w których się one kończą.

```
// edges to ilość rozłącznych krawędzi, które dany wierzchołek dodaje do rozw.
ThreeVertices(T)
[(edges1, a), (edges2, b), (edges3, c)] = DFS(T)
return (a, b, c)

DFS(T)
if T is leaf
        return [(1, T), (0, null), (0, null)]
p1 = (edges1 = 0, vertice1 = T)
p2 = (edges2 = 0, vertice2 = null)
p3 = (edges3 = 0, vertices3 = null)
ev = (p1, p2, p3)
for-each c – child of T:
        pairs = [(e1, v1), (e2, v2), (e3, v3)] = DFS(c)
        ev.push(pairs)
        sort ev by edges
        remove 3 last pairs from ev
edges1 += 1 // tylko edges1, bo nie chcemy liczyć kraw. wspólnych wielokrotnie
return ev
```

Jako ciekawostka, rozwiązanie zwraca moc maksymalnego zbioru krawędzi o 1 więcej niż w rzeczywistości, bo zwiększanie edges1 zakłada, że istnieje krawędź do ojca, a wiemy, że korzeń nie posiada ojca.
Nie wpływa to jednak na poprawność algorytmu, bo dla wszystkich pozostałych wierzchołków edges1 jest poprawna.

Przykład działania programu na następującym drzewie:



Stąd wartości zwracane przez DFS wyglądają następująco:

Wierzchołek	Co zwraca
A - I	$< (1, self), (0, null), (0, null) >$
J	$< (2, A), (1, B), (0, J) >$
K	$< (2, C), (0, K), (0, null) >$
L	$< (2, D), (1, E), (0, L) >$
M	$< (2, F), (0, M), (0, null) >$
N	$< (2, G), (1, H), (1, I) >$
O	$< (3, A), (2, C), (1, B) >$
P	$< (3, D), (2, F), (2, G) >$
R	$< (4, A), (3, D), (2, C) >$

Stąd wynikiem jest trójka A,D,C.

Zadanie 9. (2pkt)

Operacja swap(i, j) na permutacji powoduje przestawienie elementów znajdujących się na pozycjach i oraz j. Koszt takiej operacji określamy na $|i - j|$. Kosztem ciągu operacji swap jest suma kosztów poszczególnych operacji. Ułóż algorytm, który dla danych π oraz σ - permutacji liczb {1, 2, . . . , n}, znajdzie ciąg operacji swap o najmniejszym koszcie, który przekształca permutację π w permutację σ.

```
FindSequence(pi, sigma)
seq = []
i = length(pi)
while (i > 0) // dopóki istnieje element który należy przesunąć w lewo
        // znajdź indeks najbardziej skrajnie prawego takiego elementu
        while (pi[i] == sigma[i])
                i--
        // znajdź element, który powinien trafić na pozycję i
        j = FindIndexOf(pi, sigma[i])
        k = j + 1, m = i
        while (m > j) // szukamy indeksu elementu do zamiany z j
                m = FindIndexOf(pi, sigma[k])
                k++
        k--
        seq.push_back(<j, k>)
        swap(pi[j], pi[k])
return seq
```

ZŁO, nie ruszać!

~~Pomysł:~~

~~Stworzymy trzecią tablicę indexes, która na i-tym indeksie będzie miała wartość x oznaczającą, że $indexes[i] = \pi[x]$ oraz tablicę moved, która spełnia $\sigma[i] = \pi\left[i + moved[i]\right]$.~~

~~Na początku tworzymy w czasie liniowym jakieś rozwiązanie, a następnie je optymalizujemy.~~

~~Algorytm:~~

```
// tworzy nieoptymalny ciąg operacji swap w czasie O(n) ?
CreateSequence(p1, p2, moved)
seq = list() // nieoptymalny ciąg operacji swap
added = [false for i in 1 to n] // określa, czy dany indeks został dodany do seq
index = 1
while len(seq) < n or index <= n:
        // element jest na dobrym miejscu lub został już odwiedzony
        if (moved[index] == 0 or added[index])
                index++
        else:
                pair = (index, index + moved[index])
                seq.push(pair) // dodaj swapa do sekwencji
                added[index] = true // zaznacz jako odwiedzone
```

~~index += moved[index] // przejdź do nowego indeksu~~
~~return seq~~

~~// optymalizuje sekwencję zwróconą przez CreateSequence~~
~~OptimizeSequence(seq, n)~~
~~removed = 0~~
~~for index in 1 to n-1:~~
~~i = index - removed~~
~~if (seq[i+1].first > seq[i+1].second) // nieoptymalny fragment: AC,CB~~
~~a = seq[i].first~~
~~c = seq[i].second // równe seq[i+1].first~~
~~b = seq[i+1].second~~
~~if (a <= b)~~
~~seq[i] = (b, c) // zamiast (a,c)~~
~~seq[i+1] = (a, b) // zamiast (c,b)~~
~~if (a == b) // zamiana „sam ze sobą" nie ma sensu~~
~~remove seq[i+1]~~
~~removed += 1~~
~~else // a > b~~
~~seq[i] = (b, a) // zamiast (a,c)~~
~~seq[i+1] = (a, c) // zamiast (c,b)~~
~~return seq~~
~~// główna funkcja wywołująca pozostałe~~
~~FindSequence(p1, p2)~~
~~indexes = [0 for i in 1 to n] // tablica pomocnicza do p1~~
~~moved = [0 for i in 1 to n] // tablica przesunięć w lewo i-tych elementów σ~~
~~for i in 1 to n:~~
~~$indexes[p1[i]] = i$~~
~~for i in 1 to n:~~
~~$moved[i] = indexes[p2[i]] - i$~~
~~seq = CreateSequence(p1, p2, moved)~~
~~seq = OptimizeSequence(seq, n)~~
~~return seq~~

~~Lemat 1:~~
~~Operacje optymalizacji par swapów w OptimizeSequence zmniejszają łączny koszt rozwiązania.~~
~~Dowód:~~
~~Rozważmy przypadki (ify z OPT SEQ), gdzie c jest zawsze największym indeksem:~~

1. a=b, wtedy zamieniamy (a,c), (c,b) na (b,c), których koszty to:

$$C_1 = c - a + c - b = 2c - 2b$$
$$C_2 = c - b$$

Gdzie C_1 to koszt przed optymalizacją, a C_2 to koszt po, czyli chcemy pokazać:
$C_2 < C_1 \rightarrow c - b < 2c - 2b \rightarrow b < c$, co z założenia jest prawdą.

2. a<b, wtedy zamieniamy (a,c), (c,b) na (b,c), (a,b), których koszty to:

$$C_1 = c - a + c - b = 2c - b - a$$
$$C_2 = c - b + b - a = c - a$$

Gdzie C_1 to koszt przed optymalizacją, a C_2 to koszt po, czyli chcemy pokazać:
$C_2 < C_1 \rightarrow c - a < 2c - b - a \rightarrow b < c$, co z założenia jest prawdą.

3. b<a, wtedy zamieniamy (a,c), (c,b) na (b,a), (a,c), których koszty to:

$$C_1 = c - a + c - b = 2c - b - a$$
$$C_2 = a - b + c - a = c - b$$

Gdzie C_1 to koszt przed optymalizacją, a C_2 to koszt po, czyli chcemy pokazać:
$C_2 < C_1 \rightarrow c - b < 2c - b - a \rightarrow a < c$, co z założenia jest prawdą.

Lemat 2:

OptimizeSequence zwraca listę par typu (a, b), gdzie zawsze a<b.

Dowód:

Jak widać w lemacie 1, zawsze kiedy natrafiamy na 2 nieoptymalne pary, to zwracamy 2 pary postaci (a,b) a<b. Wystarczy zatem pokazać, że żadnej pary nie pominęliśmy. Ale to jest proste, bo jeśli dla i-tej pary mamy a=b i ją usuwamy, to następnie patrzymy nadal na i-tą parę (kolejną), bo inkrementacja removed niweluje zwiększenie index. Stąd gdy porównujemy pary i, i+1, to wszystkie poprzednie są już poprawnej postaci, bo każda niepoprawna została poprawiona zgodnie z lematem 1.

Lemat 3.

Operacje optymalizacji par z lematu 1 tworzą pary równoważne im, tzn. jeśli $\pi_1 + swap(a,c) + swap(c,b) = \pi_2$, to $\pi_1 + swap(A) + swap(B) = \pi_2$, gdzie A,B to pary wyznaczone, których wartość jest zależna od relacji a ? b.

Dowód:

Załóżmy, że początkowo $\pi_a = x, \pi_b = y, \pi_c = z$, (x, y, z).
Najpierw rozważmy 2 swapy przed optymalizacją – one są zawsze takie same. Operacja swap(a,c) doprowadza do sytuacji (z, y, x), na których dokonujemy swap(c,b), otrzymując (z, x, y). Teraz trzeba pokazać, że w każdym z 3 przypadków zmiana operacji także zwróci nam (z, x, y).

Przypadki dla c będącego największym indeksem:

1. a=b, wtedy zmieniamy (a,c), (c,b) na (b,c).

~~Po optymalizacji mamy tylko swap(b,c), czyli mamy $(x, y, z) \rightarrow (x, z, y)$~~
~~Zatem rzeczywiście operacje te są równoważne.~~

~~2. a<b, wtedy zmieniamy (a,c), (c,b) na (b,c), (a,b).~~
~~Pierwsza operacja to swap(b,c), czyli mamy $(x, y, z) \rightarrow (x, z, y)$.~~
~~Druga operacja to swap(a,b), czyli mamy $(x, z, y) \rightarrow (z, x, y)$~~
~~Zatem rzeczywiście operacje te są równoważne.~~

~~3. a>b, wtedy zmieniamy (a,c), (c,b) na (b,a), (a,c).~~
~~Pierwsza operacja to swap(b,a), czyli mamy $(x, y, z) \rightarrow (y, x, z)$.~~
~~Druga operacja to swap(a,c), czyli mamy $(y, x, z) \rightarrow (z, x, y)$~~
~~Zatem rzeczywiście operacje te są równoważne.~~
~~Dowód Poprawności:~~
~~Permutacje π, σ można zapisać w postaci rozłącznych cykli:~~

$$\frac{\pi}{\sigma} = \left(\frac{\pi_1 \ \dots \ \pi_i}{\sigma_1 \ \dots \ \sigma_i}\right)\left(\frac{\pi_{i+1} \ \dots \ \pi_j}{\sigma_{i+1} \ \dots \ \sigma_j}\right) \dots \left(\frac{\pi_k \ \dots \ \pi_n}{\sigma_k \ \dots \ \sigma_n}\right)$$

~~Zauważmy, że gdy $\pi = \sigma$, to mamy n rozłącznych cykli, każdy identycznościowy długości 1. Stąd faza 1 programu (CreateSequence) znajduje taką sekwencję, której zastosowanie tworzy n rozłącznych cykli, czyli przekształca π w σ. Jednakże, funkcja ta zwraca sekwencję długości n – p, gdzie p to ilość pozycji, które już na starcie się zgadzają, co nie jest optymalnym rozwiązaniem – np. dla $\pi = [3,2,1,4], \sigma = [1,3,2,4]$ CreateSequence zwraca $< (1,3), (3,2), (2,1) >$ o łącznym koszcie 4, większym od optymalnego równego 2.~~
~~Stąd konieczna jest faza 2, czyli zoptymalizowanie tego rozwiązania, co jest dokonywane w OptimizeSequence. Optymalizacja polega na tym, że zamieniamy 2 sąsiednie swapy, w tym jeden z nich „cofa się" na 2 równoważne swapy „bez cofania się" (lemat 3), co zmniejsza koszt (lemat 1). Rozwiązanie nie zawiera żadnych nieoptymalnych par (lemat 2), co kończy dowód.~~

Zadanie 10.

(1pkt) Na wykładzie przedstawiono zachłanny algorytm dla problemu *Pokrycia zbioru*, znajdujący rozwiązania, które są co najwyżej $\log n$ razy gorsze od rozwiązania optymalnego.

Pokaż, że istnieją dane, dla których rozwiązania znajdowane przez ten algorytm są blisko $\log n$ gorsze od rozwiązań optymalnych.

Niech $S_n = \cup\{i\}, S_i = i$ dla $i < n, U = \{1, 2, \dots, n\}$
Niech koszty wynoszą $c_i = \frac{c_n}{n-i} - \epsilon$
Wtedy algorytm pokryje kosztem

$$c = \sum c_i = c_n \times \left(1 + \frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{n}\right) - (n-1)\epsilon$$

Koszt optymalny to c_n
Zatem rozwiązanie będzie $\frac{c}{c_n} \approx \sum \frac{1}{i} \approx ln(n)$ razy gorsze.