Program kształcenia z informatyki dla kół informatycznych - I rok nauczania

Szczegółowy plan zajęć

Zajęcia A-1: “Co to jest algorytmika?”

Cel zajęć i efekty uczenia

Główne cele zajęć / materiał do opanowania:

- Pojęcie algorytmu i programu, interakcja między człowiekiem a maszyną, konieczność przestrzegania formalnej poprawności programu
- Intuicyjne podejście do złożoności obliczeniowej, algorytmy wolne i szybkie

Dodatkowe cele:

- łagodne wprowadzenie do wyszukiwania binarnego
- pojęcie logarytmu dwójkowego

Plan zajęć

Szacunkowy czas trwania: 2 godziny lekcyjne (orientacyjnie, patrz “Uwaga” poniżej).

Uwaga: Zajęcia wprowadzające w algorytmikę i programowanie (A-1, A-2) muszą bardzo mocno zależeć od tego, jak liczna jest grupa, jaki ma początkowy poziom zaawansowania, czy zajęcia odbywają w pracowniach komputerowych, ile jest komputerów i jakie mają oprogramowanie. Ogromne znaczenie - większe niż przy następnych zajęciach - mają indywidualne preferencje nauczyciela i własny sposób “dotarcia” do uczniów. Dlatego też w wypadku tych zajęć szacunkowy czas ich trwania oraz podany plan lekcji należy traktować, jeszcze bardziej niż przy dalszych zajęciach - jako punkt wyjścia do rozwinięcia własnych metod.

1. Dyskusja: co to jest “algorytm”?
    - *Dyskusja ma na celu pobudzenie uczniów do samodzielnego myślenia, nie wypracowanie formalnej “definicji algorytmu”. Tak naprawdę, nie istnieje jedna, obowiązująca “definicja algorytmu”, ani nawet żadna definicja, która byłaby jednocześnie zrozumiała i znacząca. Najlepsze przybliżenie, jakie mogę zalecić to “ciąg poleceń sformułowany w języku zrozumiałym dla maszyny”.*

2. Jak zbudowany jest komputer i jak wykonuje nasze polecenia? Jak formułować polecenia tak, żeby były zrozumiałe dla maszyn? Co to jest “język programowania”?
    - *Ważne jest podkreślenie, że komputer nie “domyśla się” i nie potrafi zrozumieć “o co nam chodziło”. Musi dostawać proste polecenia, odwołujące się tylko do pojęć z wąskiej listy “zrozumiałych dla niego”.*
    - *Można omówić, ale bez wchodzenia w szczegóły i definicje, pojęcia:*
        - *pamięci operacyjnej - podkreślając, że jest podzielona na komórki, i de facto przechowuje wyłącznie liczby*
        - *wejścia i wyjścia programu - najlepiej na przykładach prostych algorytmów (np. dodających dwie liczby)*
        - *programu - ciągu instrukcji umieszczonych w pamięci, które wykonują się zawsze w podanej kolejności (kolejnością można manipulować za pomocą instrukcji, ale nigdy nie jest przypadkowa, zawsze wykonuje się też jedna instrukcja naraz)*
3. Gra w “za dużo, za mało” i wyszukiwanie binarne jako przykład algorytmu
    - *Dawna radiowa gra w “za dużo, za mało” polegała na zgadnięciu pewnej (tajnej) kwoty pieniędzy przez słuchaczy dzwoniących do radia. Słuchacz podawał kwotę, a system odpowiadał, czy kwota jest za duża, czy za mała. Trafienie w dokładną kwotę oznaczało wygranie jej przez dzwoniącego.*
    - *Zakładamy, że jesteśmy jedynym dzwoniącym, a kwota jest między 1 a 1000 złotych. Jak wygrać ją za pomocą co najwyżej 1000 telefonów? Jak wygrać ją w mniejszej liczbie (10) telefonów?*
4. Złożoność algorytmu - łagodne wprowadzenie
    - *Dlaczego algorytm “dzielący na pół” jest lepszy? Omówienie (nieformalne), czym jest złożoność pesymistyczna - zakładamy, ze system zachowuje się zgodnie z regułami (kwota naprawdę jest między 1 a 1000), ale nasz algorytm zawsze ma pecha - kwota jest taka, żeby telefonów było jak najwięcej.*
5. Logarytm dwójkowy
    - *Ile będzie potrzeba telefonów, jeśli kwota jest między 1 a n? Pojęcie logarytmu dwójkowego.*
    - *Nie ma potrzeby ogólnego definiowania logarytmu o dowolnej podstawie, ani nawet żadnej formalnej definicji logarytmu - przez większość kursu wystarczy nam logarytm dwójkowy zaokrąglony do liczby całkowitej. Oczywiście należy nadmienić, że to uproszczone pojęcie i “kiedyś” wprowadzimy je w całości.*

Zajęcia A-2: “Wstęp do programowania”

Cel zajęć i efekty uczenia

Główne cele zajęć / materiał do opanowania:

- Wprowadzenie do języka C++
- Kompilacja i uruchamianie programów
- Instrukcja warunkowa
- Obsługa systemu sprawdzającego

Zadania do rozwiązania na sprawdzarce

Blackjack
Wczytać dwie liczby z klawiatury, obliczyć ich sumę i wypisać na wyjście - jeśli suma jest równa 21, wypisać zamiast niej napis “Blackjack!”.

Plan zajęć

Szacunkowy czas trwania: 4 godziny lekcyjne (bardzo orientacyjnie, patrz “Uwaga I” poniżej).

Uwaga I: Zajęcia wprowadzające w algorytmikę i programowanie (A-1, A-2) muszą bardzo mocno zależeć od tego, jak liczna jest grupa, jaki ma początkowy poziom zaawansowania, czy zajęcia odbywają w pracowniach komputerowych, ile jest komputerów i jakie mają oprogramowanie. Ogromne znaczenie - większe niż przy następnych zajęciach - mają indywidualne preferencje nauczyciela i własny sposób “dotarcia” do uczniów. Dlatego też w wypadku tych zajęć szacunkowy czas ich trwania oraz podany plan lekcji należy traktować, jeszcze bardziej niż przy dalszych zajęciach, jako punkt wyjścia do rozwinięcia własnych metod.

Uwaga II: W czasie kilku początkowych zajęć konieczny jest czas dla uczniów na samodzielną implementację programów. Uczniom wolniej radzącym sobie z programami trzeba pomagać na bieżąco. Bardzo dobrze sprawdza się prowadzenie kółka z drugą osobą, która pomaga przy programowaniu.

1. [jeśli potrzebne] Zapoznanie uczniów z obsługą komputerów i środowiska programistycznego
    - *Na zajęciach prowadzonych w Krakowie komputery będące do dyspozycji prowadzących i uczniów są wyposażone w system Linux - wymaga to dodatkowego wysiłku, aby wprowadzić uczniów w (na ogół) nieznany im system. Może to wymagać poświęcenia całych osobnych zajęć.*
2. Pierwszy program w C++
    - *Klasyczny przykład to “Hello, world!”, wyglądający w C++ mniej-więcej następująco:*

```
#include <iostream>

using namespace std;

int main()
{
	cout << “Witaj, świecie!”;
}
```

- *Kluczowa jest odpowiednia proporcja pomiędzy tym, co uczniowie powinni w tym momencie rozumieć z programu, a tym, co na razie przepisują z tablicy.*
- *Sugerowane wprowadzenie: wytłumaczenie instrukcji cout jako wypisującej na ekran zadany tekst, reszta instrukcji to “tło” umożliwiające działanie - koniecznie trzeba zaznaczyć, że wszystkie instrukcje zostaną wytłumaczone na kolejnych zajęciach.*
- *Należy surowo przestrzegać złotej zasady, a także wyłożyć ją uczniom: **nie ma przepisywania kodów z tablicy bez zrozumienia.***
- *W tym momencie należy zapoznać uczniów z ideą kompilacji programu, można wspomnieć o różnych językach programowania i o różnicy między językami kompilowanymi i interpretowanymi.*

3. Wpisywanie z klawiatury, zmienne, instrukcja warunkowa i zadanie “Blackjack”
    - *Tutaj wprowadzamy pojęcie zmiennej - sugerowana interpretacja dla uczniów to “komórka pamięci, która ma własną nazwę i w której przechowujemy jedną liczbę”.*
    - *Materiał dość prosty i intuicyjny, warto zwrócić uwagę na to, że w C++ porównywanie robi się dwoma znakami równości (==) - z jednym program będzie działał niewłaściwie.*
4. System sprawdzający
    - *Działanie systemu można wyjaśnić uczniom jako “automatyczne wpisywanie danych z klawiatury i sprawdzanie odpowiedzi”.*
    - *Należy uczulić uczniów na to, że system nie myśli - nie zaakceptuje odpowiedzi różniącej się nawet jednym znakiem (typowy przykład: brak wykrzyknika po “Blackjack”), nie pozwoli też na dodatkowe komentarze (“Wpisz teraz liczbę:”)*

Zajęcia A-3: “Pętle, tablice”

Cel zajęć i efekty uczenia

Główne cele zajęć / materiał do opanowania:

- Wprowadzenie pętli typu *for* i *while*
- Wprowadzenie pojęcia tablicy

Dodatkowe cele:

- Zachęcenie uczniów do unikania powtarzania i kopiowania instrukcji przy programowaniu

Zadania do rozwiązania na sprawdzarce

Suma 10 liczb
Na wejściu dane jest 10 liczb, wypisać ich sumę.

Odwróć liczby
Na wejściu dana jest liczba N, a następnie N liczb. Wypisać je w odwrotnej kolejności.

Plan zajęć

Szacunkowy czas trwania: 2 godziny lekcyjne.

Uwaga: W czasie kilku początkowych zajęć konieczny jest czas dla uczniów na samodzielną implementację programów. Uczniom wolniej radzącym sobie z programami trzeba pomagać na bieżąco. Bardzo dobrze sprawdza się prowadzenie kółka z drugą osobą, która pomaga przy programowaniu.

1. Zadanie “Suma 10 liczb” - jak uniknąć 10-krotnej instrukcji “cin” i/lub 10 zmiennych?
2. Konstrukcja pętli “for” i “while”
    - *Warto przez pewien czas unikać składni “i++”, pisząc zamiast niej “i = i+1”, aż uczniowie przyzwyczają się do samej konstrukcji pętli i będzie można wprowadzać w niej uproszczenia. Podobnie zalecam na początku unikać “for(int i = 0; …), zamiast tego deklarując zmienną wcześniej.*
3. Tablice i ich składnia w C++ - deklaracja, używanie, iteracja po tablicy
    - *Specjalną uwagę należy poświęcić iteracji po tablicy - o ile zapis “cout << tablica[2];” typowo nie nastręcza trudności, o tyle “cout << tablica[i];” bywa już wyższą szkołą jazdy.*
    - *W zadaniu “Odwróć liczby” trzeba dodatkowo iterować się w dół tablicy - jest to konstrukcja, którą będzie trzeba jeszcze powtarzać na kolejnych zajęciach, aby uczniowie złapali odpowiednią intuicję.*

Zajęcia A-4: “Pętle II, łańcuchy”

Cel zajęć i efekty uczenia

Główne cele zajęć / materiał do opanowania:

- Ugruntowanie pojęcia pętli, podwójne pętle, pętle iterujące się w dół
- Używanie typu łańcuchowego, podstawowe funkcje klasy *string*
- Opanowanie algorytmu sprawdzania, czy słowo jest palindromem

Zadania do rozwiązania na sprawdzarce

Prostokąt
Wypisać na ekranie ASCII-art w kształcie pustego w środku prostokąta z liter X

Palindrom
Wczytać z wejścia łańcuch znaków i wypisać TAK lub NIE, w zależności od tego, czy łańcuch jest palindromem

Plan zajęć

Szacunkowy czas trwania: 2 godziny lekcyjne.
Uwaga: W czasie kilku początkowych zajęć konieczny jest czas dla uczniów na samodzielną implementację programów. Uczniom wolniej radzącym sobie z programami trzeba pomagać na bieżąco. Bardzo dobrze sprawdza się prowadzenie kółka z drugą osobą, która pomaga przy programowaniu.

1. Przypomnienie konstrukcji pętli “for”
    - *Warto zrobić na tablicy kilka(-naście) wariantów: pętle iterujace się od 0/1/2, pętle iterujące się z różnym krokiem, w dół, pętle wypisujące wartość zmiennej sterującej, albo kwadrat tej wartości, etc.*
2. Pętla podwójna, omówienie zadania *Prostokąt*
    - *Można zacząć od narysowania “pełnego” prostokąta m x n jedną podwójną pętlą.*
    - *Najbardziej prostolinijnym sposobem rozwiązania jest wypisanie pierwszego wiersza (XXX…X), potem n-2 razy X (spacje) X, a potem jeszcze raz wiersza XXX...X.*
    - *Alternatywny sposób: pętla jak przy “pełnym” prostokącie, ale z instrukcją warunkową, czy jesteśmy w pierwszym/ostatnim wierszu lub kolumnie. Wtedy wypisujemy “X”, inaczej - spację.*
3. Typ łańcuchowy i znakowy, podstawowe metody klasy *string*
    - *Najłatwiej traktować typ string jak tablicę, z dodatkową metodą length().*
4. Algorytm sprawdzania palindromiczności

Zajęcia A-5: “Algorytm Euklidesa”

Cel zajęć i efekty uczenia

Główne cele zajęć / materiał do opanowania:

- Opanowanie i samodzielna implementacja algorytmu Euklidesa

Dodatkowe cele:

- Utrwalenie zasady działania pętli typu *while*
- Intuicja złożoności logarytmicznej jako efektywnej, a liniowej jako nieefektywnej, jeśli wejściem algorytmu są duże liczby
- Umiejętność krytycznego spojrzenia na algorytm, szukanie przypadków trudnych dla algorytmu

Zadanie do rozwiązania na sprawdzarce

Algorytm Euklidesa
Dane są liczby a, b w zakresie typu int, obliczyć i wypisać NWD(a,b).

Plan zajęć

Szacunkowy czas trwania: 2 godziny lekcyjne.

1. Pojęcie wspólnych dzielników dwóch liczb, pojęcie największego wspólnego dzielnika
    - *Uczniowie powinni mieć świadomość, że wspólnych dzielników może być wiele, a największy dzieli wszystkie pozostałe.*
2. [opcjonalnie] Algorytm z lekcji matematyki (rozłóż na czynniki pierwsze i wybierz wspólne) - czemu go nie stosujemy?
    - *Na tym etapie nie jest jeszcze jasne, jak rozkładać na czynniki pierwsze - ale widać, że nie ma oczywistej metody. Warto wspomnieć, że ostatecznie rozkład na czynniki jest trudnym problemem, znacznie trudniejszym niż NWD, i na nim opiera się algorytmy kryptograficzne.*
3. Własność odejmowania: NWD(a,b) = NWD(b,a-b)
    - *Przejście (a,b) -> (b,a-b) jest zachowuje wszystkie wspólne dzielniki, a więc i największy. Można dla ilustracji rozpisać przykład na małych liczbach.*
4. Algorytm iteracyjny z pętlą *while*.
    - *Uwaga! Wciąż jest mowa o algorytmie z odejmowaniem.*
5. Przetestowanie algorytmu na kilku przykładach
6. Nieefektywność dla dużych liczb (kontrprzykłady)
    - *Tutaj warto zagrać z uczniami w grę: teraz to oni “są sprawdzarką” i muszą dobrać taki test, żeby rozważany program go nie przeszedł.*

    - *Nie operujemy (na razie) na funkcjach, tylko pytamy o konkretne liczby.*
    - *Jeśli znajdą, na przykład (1000000000,1), możemy pytać dalej: jak poradzą sobie z programem, który na wstępie ma rozpatrzone skrajne przypadki b=1 lub b|a.*
    - *Teraz zauważamy, że algorytm może wykonać liczbę operacji bliską a (lub a/2), czyli bardzo dużą. Żeby działał efektywnie, trzeba go przyspieszyć.*
7. Przyspieszenie algorytmu: operacja dzielenia z resztą
    - *Zauważamy, że podzielenie z resztą a%b to "skrót" dla odejmowań, które w algorytmie i tak by nastąpiły - jeśli działała poprzednia wersja, to i ta zadziała.*
8. [opcjonalnie] Szukanie pesymistycznych przypadków (liczby Fibonacciego)
9. [opcjonalnie] Argument na dobrą złożoność algorytmu
    - *W każdym wypadku jedna z liczb zmniejsza się co najmniej dwukrotnie - a zatem nie może być więcej kroków niż log a + log b.*

Zajęcia A-6: “Funkcje / sprawdzanie pierwszości”

Cel zajęć i efekty uczenia

Główne cele zajęć / materiał do opanowania:

- Implementacja “pierwiastkowego” algorytmu sprawdzania pierwszości
- Zdobycie pierwszych intuicji w kwestii notacji O() i asymptotycznej złożoności algorytmu
- Pojęcie funkcji (podprogramu) w języku programowania, implementacja funkcji w C++

Dodatkowe cele:

- Utrwalenie konstrukcji pętli *for* w języku C++
- Intuicja “małych” i “dużych” dzielników, znajdowanie wszystkich dzielników w czasie pierwiastkowym
- Elegancja w projektowaniu algorytmu, unikanie powtarzania kodu, szczególnie “copy-paste”
- Dbałość o szczegóły podczas pisania programu

Zadania do rozwiązania na sprawdzarce

Zielone butelki
Wypisać wiersz o butelkach:

“10 green bottles standing on the wall (…)
and if one green bottle should accidentally fall,
there’d be 9 green bottles standing on the wall”

w wersji dla 100 000 butelek. Zamiast każdej liczby pierwszej należy wypisać BUZZ.

[opcjonalnie] ***Faktoryzacja***
Mając daną liczbę całkowitą N, wypisać jej rozkład na czynniki pierwsze

Plan zajęć

Szacunkowy czas trwania: 4 godziny lekcyjne.

1. Jakie byłoby rozwiązanie zadania, gdyby chodziło tylko o wypisanie wiersza, bez warunku z liczbami pierwszymi?
    - *Warto przypomnieć konstrukcję pętli for i poświęcić chwilę na przypomnienie, jak ją implementować, aby działała “w dół” (od N do 1).*
2. Przypomnienie pojęcia liczby pierwszej
3. Jak sprawdzać pierwszość zadanej liczby N: algorytm “podziel przez wszystkie dzielniki mniejsze niż N”
4. Poprawka algorytmu: dzielenie przez liczby mniejsze od N/2, i/lub przez liczby nieparzyste

- *Warto od razu zastanawiać się nad złożonością na konkretnym, dużym N (~10^9), aby zilustrować niewielki zysk na złożoności.*

5. Intuicja złożoności asymptotycznej i notacji O()
   - *To dobry moment do zilustrowania, czemu poprawki N -> N/2 są z punktu widzenia algorytmiki mniej ważne: ten sam efekt można osiągnąć biorąc szybszy komputer, albo lepszy kompilator języka.*
   - *Zalecam w tym momencie zaprezentować zapis O(N) w znaczeniu "N * jakaś liczba" - przy czym "jakaś liczba" jest stałą, którą możemy w każdej chwili wyznaczyć, ale nie jest nam to na razie potrzebne.*
   - *Warto jednak podkreślić, że w "zawodowym" programowaniu poprawki stałej mogą się okazać bardzo ważne.*
6. Małe i duże dzielniki
   - *Proponuję rozpisać dzielniki liczby 36 (albo np. 144) w parach: $36 = 1 \cdot 36 = 2 \cdot 18 = 3 \cdot 12 = \ldots$ - z tej postaci widać, że połowa dzielników jest "mała" (mniejsza niż sqrt(N), a połowa "duża" (większa niż sqrt(N)).*
7. Modyfikacja do algorytmu "pierwiastkowego"
8. Włączenie algorytmu na pierwszość do zadania - jak uniknąć wielokrotnego przepisywania?
9. Pojęcie podprogramu (funkcji) i implementacja w C++
   - *Zaczynamy od prostej funkcji "podnieś do kwadratu", potem np. "f(x,y) = x+y-1", aby dojść do funkcji "f(x) = 0 jeśli x nie jest pierwsza, f(x) = 1 jeśli x jest pierwsza".*
   - *Uczulamy uczniów na fakt, że napotykając wywołanie funkcji, program zapamiętuje, gdzie był, przeskakuje na chwilę do kodu funkcji, a potem wraca, przywracając wszystko do stanu sprzed wywołania.*
10. [opcjonalnie] Algorytm faktoryzacji - zadanie *Faktoryzacja*
    - *W najprostszej wersji, algorytm po prostu dzieli N przez kolejne liczby naturalne - każda liczba, przez którą uda się podzielić, musi być liczbą pierwszą (nie uda się podzielić np. przez 4, bo wcześniej już dzieliliśmy przez 2). Dzielenie wykonuje się do pierwiastka z N, ostatnia pozostała liczba jest ostatnim dzielnikiem pierwszym N.*

Zajęcia B-1: “Własna arytmetyka”

Cel zajęć i efekty uczenia

Główne cele zajęć / materiał do opanowania:

- Zrozumienie i implementacja algorytmów pisemnego dodawania, [opcjonalnie] odejmowania i mnożenia
- Posługiwanie się typem *vector* jako alternatywą dla tablicy

Dodatkowe cele:

- Opanowanie typu *char*, świadomość różnicy między znakiem a jego kodem ASCII
- Konieczność przemyślenia algorytmu przed jego napisaniem, świadomość wartości dobrze zaprojektowanego algorytmu

Zadania do rozwiązania na sprawdzarce

Dodawanie
Dane są dwie duże (~500 000 cyfr) liczby w systemie dziesiętnym, obliczyć i wypisać ich sumę.

***Odejmowanie** [opcjonalnie]*
Dane są dwie duże (~500 000 cyfr) liczby w systemie dziesiętnym, obliczyć i wypisać ich różnicę (odejmując mniejsza od większej).

[z reguły zadanie “Dodawanie” przeznaczone jest dla uczniów, którzy dopiero zaczynają naukę programowania, “Odejmowanie” dla uczniów z wcześniejszym doświadczeniem]

Mnożenie
Dane są dwie duże (~5000 cyfr) liczby w systemie dziesiętnym, obliczyć i wypisać ich sumę.

Plan zajęć

Szacunkowy czas trwania: 4 godziny lekcyjne.

1. Jak wczytać dużą liczbę z wejścia i jak ją przechowywać w pamięci
    - *Przypominamy, że typ int (a także “long long”) jest ograniczony i nie może przechowywać tak dużych liczb.*
    - *Liczby powinny być czytane jako łańcuchy znaków - wygodniej byłoby jednak trzymać je od najmniej znaczących cyfr do najbardziej znaczących*
2. Typ vector jako “tablica o zmiennej długości”, operacje na nim
    - *Wprowadzamy te operacje, które są nam potrzebne w danym momencie - deklaracja wektora (z rozmiarem i bez), metody size(), resize(), push_back(), pop_back().*
    - *Nie wchodzimy bez potrzeby w szczegóły implementacji wektora - jeśli uczniowie chcą wiedzieć, można opowiedzieć o podwajaniu, sygnalizując jednak, że dopiero w przyszłości dowiedzą się, dlaczego jest to efektywne.*

3. Konwersja łańcucha znaków do tablicy (wektora) liczb - kody ASCII, arytmetyka na znakach w C++
4. Algorytm pisemnego dodawania
    - *To jest w istocie prosty algorytm, uczniowie z reguły są go w stanie wymyślić zupełnie sami, warto skierować ich uwagę na mniej typowe przypadki (liczby różnej długości, suma dłuższa niż argumenty, zera wiodące w niektórych implementacjach).*
5. Algorytm pisemnego odejmowania
    - *Konieczność algorytmu w dwóch fazach - pierwsza pętla porównuje liczby, druga wykonuje odejmowanie*
    - *Odejmowanie jest bardzo podobne do dodawania - warto zwrócić uwagę, że "pożyczanie" możemy wykonać zawsze, nawet jeśli chwilowo pojawiłaby się cyfra -1 na którejś pozycji. Należy zwrócić uwagę na usunięcie zer wiodących i zabezpieczenie się na wypadek różnicy wynoszącej 0.*
6. Pisemne mnożenie - wariant "pomnóż liczbę A przez każdą cyfrę liczby B, dodaj wszystkie wyniki"
    - *Pokrótce opowiadamy ten wariant, zaznaczamy jednak, że będzie bardzo niewygodny w implementacji (oczywiście, chętni odważni uczniowie mogą się z nim zmierzyć)*
7. Pisemne mnożenie - wariant "pomnóż każdą cyfrę przez każdą, na końcu usuń przeniesienia"
    - *Ten wariant jest trudniejszy do zrozumienia, należy koniecznie najpierw zademonstrować go na kilku małych przykładach.*
    - *Warto podkreślać, że mimo trudności koncepcyjnej, ten wariant prowadzi do krótkiego kodu, w którym będziemy się rzadziej mylić - z punktu widzenia programisty jest znacznie lepszy.*

Zajęcia B-2: “Wstęp do rekursji / Szybkie potęgowanie”

Cel zajęć i efekty uczenia

Główne cele zajęć / materiał do opanowania:

- Wprowadzenie pojęcia rekursji i funkcji rekurencyjnej
- Opanowanie i implementacja algorytmu szybkiego potęgowania

Dodatkowe cele:

- Intuicje dotyczące arytmetyki modularnej
- Zrozumienie, jak działają funkcje w programie (stos wywołań)

Zadania do rozwiązania na sprawdzarce

Szybkie potęgowanie
Mając dane liczby a, k i p, obliczyć a^k modulo p.

Plan zajęć

Szacunkowy czas trwania: 2 godziny lekcyjne.

1. Definicja silni, obliczanie metodą iteracyjną
    - *Prosta pętla, uczniowie dochodzą do niej sami*
2. Wyjaśnienie działanie wywołania funkcji (odłóż na stos wywołań, przejdź do funkcji, wróć)
3. Wywołanie rekurencyjne w silni - jak zadziała?
    - *Należy dokładnie opisać działanie - każdy krok wykonany przez program - w wywołaniu np. silnia(4)*
4. Kiedy (i jak) w ogólności działa rekursja
    - *Rekurencja wymaga trzech warunków do działania:*
        - *wywołanie istotnie następuje na mniejszych danych (przejście z silnia(4) do silnia(3), a nie np. do silnia(6)).*
        - *wiemy, jak z wyniku dla mniejszej liczby odtworzyć wynik dla większej (przejście silnia(3) -> silnia(4) jest łatwe).*
        - *znamy początek rekursji (umiemy obliczyć silnia(0) bez wywołania rekurencyjnego).*
5. Implementacja iteracyjna, arytmetyka modularna
    - *Iteracyjna wersja potęgowania jest bardzo prosta dla uczniów, szczególnie znających już silnię*
    - *Reszta z dzielenia - dlaczego nie można obliczyć a^k, a potem wyciągnąć reszty*
    - *Typowe pytanie uczniów: dlaczego nie zadziała funkcja pow(...)?*
    - *Konieczność wyjaśnienia/przypomnienia, jak działa reszta z dzielenia - można brać resztę po każdej operacji, i otrzyma się ten sam wynik*
6. Implementacja rekurencyjna i modyfikacja do wersji “szybkiej” potęgowania
    - *Przepisujemy analogicznie do silni, rysujemy łańcuch wywołań np. dla 2^7*

- *Zauważamy, że dla liczb parzystych (i tylko dla nich) wywołanie a*potega(a,k-1) można zamienić na potega(a,k/2) podniesione do kwadratu*
- *Rozrysowujemy łańcuch wywołań rekurencyjnych np. dla 2^{50}*
- *Argumentujemy, że co drugie wywołanie zmniejsza k o połowę, więc całkowita liczba wywołań nie przekroczy log k.*

Zajęcia B-3: “Systemy liczbowe”

Cel zajęć i efekty uczenia

Główne cele zajęć / materiał do opanowania:

- Opanowanie algorytmu zamiany między systemami o różnych podstawach
- Sprawne posługiwanie się przez uczniów systemem dwójkowym

Dodatkowe cele:

- Ugruntowanie zrozumienia rekursji
- Ćwiczenie operacji na różnych typach danych (*string*, *char*, *int*), czytanie z wejścia zestawu danych z różnymi typami

Zadania do rozwiązania na sprawdzarce

System dwójkowy
Zamienić podaną liczbę z systemu dziesiętnego na dwójkowy

Systemy liczbowe
Zamienić podane liczby z systemów o różnych podstawach na dziesiętny lub odwrotnie

[Pierwsze zadanie jest (dużo) łatwiejszym wariantem drugiego. Można użyć go jako łagodnego wprowadzenia, albo wybrać tylko jeden wariant.]

Plan zajęć

Szacunkowy czas trwania: 2 godziny lekcyjne.

1. Zapis liczby w różnych systemach - przykłady, “ręczne” przeliczanie na system dziesiątkowy
    - *Warto w ramach ciekawostki wspomnieć o systemach używanych w różnych cywilizacjach (dwudziestkowy Majów, sześćdziesiątkowy Sumerów etc.)*
2. Szczególna rola systemu dwójkowego
3. Baza systemu większa niż 10 - system szesnastkowy
    - *Główna rola systemu szesnastkowego: skompresowany zapis dwójkowy. Przykłady (RGB, IPv6).*
4. Przeliczanie na system dziesiętny (algorytm Hornera)
    - *Konieczność czytania liczby w typie string i przeliczenia znaków na liczby, również znaków literowych.*
5. Przeliczanie na system o zadanej podstawie - algorytm iteracyjny i rekurencyjny
    - *Zauważamy, że ostatnią cyfrą liczby X w systemie o podstawie B jest zawsze reszta z dzielenia X przez B, a reszta zapisu X to liczba X/B - stąd łatwo wyprowadzić zarówno algorytm iteracyjny, jak i rekurencyjny. Warto wytłumaczyć oba.*

- *W algorytmie iteracyjnym trzeba pamiętać o odwróceniu na koniec zapisu liczby.*

Zajęcia B-4: “Proste sortowanie”

Cel zajęć i efekty uczenia

Główne cele zajęć / materiał do opanowania:

- Opanowanie algorytmów sortowania kwadratowego: sortowania bąbelkowego, przez selekcję i przez wstawianie

Dodatkowe cele:

- Intuicje dotyczące algorytmów złożoności kwadratowej
- Ćwiczenie operacji na różnych typach danych (*string*, *char*, *int*), czytanie z wejścia zestawu danych z różnymi typami

Zadania do rozwiązania na sprawdzarce

Sortowanie
Posortować podaną tablicę zawierającą ~1000 liczb.

[Można wymagać konkretnego algorytmu - zalecam jednak pozwolić uczniom wybrać dowolny z podanych.]

Plan zajęć

Szacunkowy czas trwania: 2-4 godziny lekcyjne.

1. Sortowanie bąbelkowe
    - *Warto dokładnie przeanalizować, jak działa algorytm - podstawowa wersja ma n^2 operacji (zarówno porównań, jak i zamian).*
    - *Wykonujemy kilka optymalizacji (np. pętla wewnętrzna nie potrzebuje się iterować do końca) - czas działania spada do 1+2+3+...+n = n(n+1)/2.*
    - *Warto poświęcić chwilę czasu tej sumie - uczniowie mają tendencję do mylenia się w niej, potrzebują chwili, żeby się przyzwyczaić.*
    - *Spadek złożoności z n^2 do $n^2/2$ nie jest kluczową zmianą - przypominamy notację O() jako “nieistotnej stałej”.*
2. Sortowanie przez selekcję
    - *Jeśli polecimy uczniom samym wymyślić algorytm sortowania, ten ma prawdopodobnie największe szanse*
    - *Zauważamy, że w algorytmie jest $O(n^2)$ porównań, ale tylko O(n) zamian. Możemy użyć tego faktu, żeby łagodnie wprowadzić intuicję “operacji dominującej”. Zaznaczamy, że dla różnych obiektów operacje porównania i zamiany mogą mieć różny koszt.*
3. Sortowanie przez wstawianie
    - *Wariant pierwotny: ze wstawianiem elementu przez kolejne zamiany.*

- *Sortowanie przez wstawianie ma tendencję do działania najszybciej (z podanych prostych algorytmów), zwłaszcza dla danych, które są już częściowo posortowane.*
- *Wariant z wyszukiwaniem binarnym właściwego miejsca do wstawienia: tylko O(n log n) porównań, ale $O(n^2)$ zamian - nie do uniknięcia w tym algorytmie. W praktyce nie jest to zysk.*

Zajęcia B-5: “Wyszukiwanie binarne”

Cel zajęć i efekty uczenia

Główne cele zajęć / materiał do opanowania:

- Algorytm wyszukiwania binarnego: wariant z rozwiązywaniem równania, szukania elementu w tablicy, modyfikacja dla pierwszego i ostatniego elementu

Dodatkowe cele:

- Intuicje dotyczące pojęcia niezmiennika algorytmu

Zadania do rozwiązania na sprawdzarce

Zagadka Nicolo Tartaglii
Dla zadanych p, q znaleźć liczbę całkowitą x, która spełnia równanie $x^3 + px = q$.

[zadanie pochodzi z kursu algorytmiki z serwisu MAIN2]

Naczelny Statystyk
Dana jest posortowana tablica n liczb, odpowiedzieć na q zapytań postaci “ile razy element x występuje w tablicy?”

Plan zajęć

Szacunkowy czas trwania: 4 godziny lekcyjne.

1. Gra w “za dużo, za mało” - przypomnienie pierwszych zajęć, próba formalizacji wyszukiwania binarnego
    - *Uczniom łatwo jest wymyślić strategię “strzelania w środek”, ale dużo trudniej sformalizować - strategia utrzymywania przedziału, w którym szukamy i stopniowego zmniejszania go może wymagać dłuższego tłumaczenia.*
2. Wyszukiwanie binarne jako sposób na rozwiązywanie równania, na przykładzie równania sześciennego z zadania *Zagadka Nicolo Tartaglii*
    - *Kwestie techniczne w algorytmie:*
        - *czy sprawdzamy w każdej iteracji, że już znaleźliśmy właściwy element (zalecany wariant: nie sprawdzamy)*
        - *który koniec przedziału przesuwamy, i czy nie zawiesimy się na tablicy dwuelementowej: praktycznie zawsze można to naprawić odpowiednio zaokrąglając środek przedziału*
    - *Poprawność algorytmu: właściwy element nie może nigdy opuścić rozważanego przedziału*
    - *Złożoność algorytmu: przedział za każdą iteracją zmniejsza się dwukrotnie*
3. Wyszukiwanie elementu w tablicy - zadanie *Naczelny Statystyk*

- *Przepisanie algorytmu tak, aby znajdował element x w tablicy*
- *Pytanie do uczniów: jeśli jest kilka wystąpień, które wystąpienie znajdzie algorytm (w zalecanym wariancie, zawsze pierwsze wystąpienie)*
- *Jak zmodyfikować algorytm tak, aby znajdował ostatnie wystąpienie? Wystarczy zmienić kilka szczegółów.*
- *Znajdowanie liczby wystąpień jako różnica pierwszego i ostatniego*
- *Alternatywne rozwiązanie - szukanie liczby x, a potem x+1.*

Zajęcia C-1: “Sortowanie za pomocą STL-a”

Cel zajęć i efekty uczenia

Główne cele zajęć / materiał do opanowania:

- Używanie funkcji sort(), pojęcie funkcji porównującej (komparatora)
- Pojęcie klasy/struktury/rekordu

Dodatkowe cele:

- Operacje na liczbach rzeczywistych
- Wprowadzenie do algorytmów geometrycznych: sortowanie kątowe w najprostszej wersji

Zadania do rozwiązania na sprawdzarce

Panorama
Dane są punkty na płaszczyźnie (szczyty górskie). Jaka będzie ich kolejność widziana z punktu (0,0)?

Plan zajęć

Szacunkowy czas trwania: 2 godziny lekcyjne.

1. Funkcja sort() w najprostszej postaci - na tablicy liczb całkowitych
    - *Składnia funkcji na tablicy i na STL-owym wektorze*
    - *Łagodne wprowadzenie do iteratorów: sugeruję podejście “zmiennej pokazującej pozycję w tablicy”, bez wchodzenia w szczegóły - trzeba jednak zatrzymać się przy “pozycji za ostatnim elementem”*
2. Struktura w C++
    - *Omówienie na przykładzie punktów w zadaniu (trzy pola: odcięta, rzędna, numer)*
    - *Na razie tylko pola - opcjonalnie można opowiedzieć o klasach i metodach, chociaż w tym momencie potrzbne nie będą*
3. Sortowanie z własnym komparatorem
    - *STL-owa składnia: sort(początek, koniec, funkcja_porównująca)*
    - *Przykład: sortowanie według pierwszej współrzędnej/leksykograficzne*
    - *Własności, jakie musi spełniać komparator (nie może być a < a, nie może być cyklu, etc.)*
4. Sortowanie kątowe - wzory geometryczne
    - *Różnica między sortowaniem leksykograficznym (“od lewej do prawej”) a kątowym - konieczne może być dłuższe zatrzymanie się w tym miejscu, jako że typowo kolejność kątowa nie jest intuicyjna.*
    - *Po zrozumieniu przez uczniów idei kąta nachylenia, należy pokazać (wyprowadzić) wzór geometryczny: punkt A jest na lewo od B, jeśli A.x/A.y < B.x/B.y.*

5. Ostateczna postać funkcji-komparatora
    - *Nie wchodząc w szczegóły, warto zauważyć, że unikamy przy programowaniu liczb rzeczywistych, a szczególnie porównania: zamiast A.x/A.y < B.x/B.y dużo lepiej napisać A.x * B.y < A.y * B.x.*
    - *Kwestia punktów leżących w jednej linii - przy równym kącie najpierw jest leżący bliżej*
    - *Opcjonalnie można zastanowić się, jakie pojawiłyby się problemy, gdyby punkty leżały na całej płaszczyźnie, a nie tylko w górnej połówce.*
    - *Kwestia techniczna: uważać na duże liczby! Niezmieszczenie się w 32-bitowym typie **int** przy porównywaniu kątowym to jeden z najpopularniejszych błędów popełnianych na zawodach informatycznych.*

Zajęcia C-2: “Kolejka”

Cel zajęć i efekty uczenia

Główne cele zajęć / materiał do opanowania:

- Struktura kolejki, szablon *queue* z STL-a
- Pojęcie “abstrakcyjnej” struktury danych
- [opcjonalnie] Wstęp do techniki zwanej “obgryzaniem grafu”

Dodatkowe cele:

- Idea, cele i sensowność optymalizacji pamięciowej

Zadania do rozwiązania na sprawdzarce

Wojna
Dwoje graczy gra w wojnę, każde z własną częścią talii: w jednej turze gracze odkrywają swoje wierzchnie karty, gracz z wyższą kartą zabiera obie i dokłada na koniec swojej talii. Zasymulować podaną liczbę tur rozgrywki.

[opcjonalnie] ***Obrazy i pokoje***
Jest n pokojów, w każdym na początku stoi jedna osoba. W każdym pokoju na ścianie napisany jest numer pewnego innego pokoju. Na dźwięk dzwonka każdy przechodzi ze swojego pokoju do tego, którego numer widzi na ścianie. Wyznaczyć wszystkie pokoje, które nigdy nie staną się puste.

Plan zajęć

Szacunkowy czas trwania: 2 godziny lekcyjne (sama kolejka), lub 4 godziny (z techniką obgryzania).

1. Organizacja danych w postaci kolejki, struktura danych
    - *W kolejce potrzebujemy dopisywać nowe elementy, i wybierać pierwszy dodany element.*
    - *Zwykła tablica nie potrafi tego robić (choćby z uwagi na fakt, że ma stały rozmiar), za to potrafi np. podawać element na zadanej pozycji.*
    - *Różne sposoby uporządkowania danych - **struktury danych** - dobieramy w zależności od tego, czego aktualnie potrzebujemy.*
2. Kolejka w C++
    - *Podobnie jak wektor, kolejkę można napisać na różnych typach danych - typy takie, jak kolejka i wektor nazywają się **szablonami**.*
    - *Warto zwrócić uwagę, że front() tylko podgląda (nie zmieniając) pierwszy element kolejki, a pop() tylko go usuwa (nic nie zwracając).*
3. Własna implementacja kolejki

- *Zauważamy, że samo dokładanie na koniec można zrealizować znanym już typem wektora (jednak wciąż nie wiemy dokładnie, jak działa wektor!).*
- *Zabieranie elementy z początku kolejki - wskaźnik czoła (head) kolejki.*
- *Jak poradzić sobie z pisaniem kolejki bez wektora? Dwa wskaźniki: czoło i ogon kolejki. Ale co, jeśli będzie wiele operacji?*
- *Możliwość I: zadeklarować tablicę o rozmiarze takim, jak liczba operacji na kolejce. Wtedy ogon kolejki nigdy nie dojdzie do końca tablicy.*
- *Możliwość II: kolejka cykliczna, obie operacje modulo długość tablicy. Wtedy zużycie pamięci będzie niższe, ale kolejka zepsuje się, jeśli będzie na niej za dużo elementów.*

4. Której możliwości używać? Czy warto oszczędzać pamięć?
   - *Zawsze patrzymy, do czego będziemy używać kolejki. Jeśli będzie na niej naraz dużo elementów, nie warto robić kolejki cyklicznej. Jeśli jednak będzie dużo operacji, a naraz mało elementów, kolejka cykliczna jest najlepszym pomysłem.*
   - *Dobieranie struktury i implementacji do zadania to bardzo ważna część projektowania algorytmu.*
   - *Warto oszczędzać pamięć, ale nie należy z tym przesadzać - jeśli oszczędność nie jest paląca, a czyni program nieczytelnym, żmudnym do napisania, łatwym do pomylenia się, albo nieoptymalnym czasowo - nie oszczędzamy niepotrzebnie.*
5. Zadanie “Wojna”
   - *Jest to bardzo prosta implementacja kolejki - potrzebne są dwie zmienne tego typu, jedna na każdą talię. Implementacja to zwyczajna symulacja gry, tura po turze.*
   - *Kwestia techniczna: przed każdą turą koniecznie należy sprawdzać, czy kolejka jest pusta, żeby nie wykonać operacji wyciągania z pustej kolejki.*
   - *Operacje na pustej kolejce mogą mieć trudne do przewidzenia skutki, warto uczulić na to uczniów.*
6. Obgryzanie, zadanie “Obrazy i pokoje”
   - *Pokoje, do których nie ma wejścia (nic do nich nie kieruje), opróżnią się w pierwszej turze i już nigdy nie będą używane. Możemy więc je usunąć z gry.*
   - *W wyniku tego usunięcia pewne inne pokoje straciły wszystkie wejścia - one opróżnią się w drugiej turze. Możemy więc teraz z kolei je usunąć, a potem powtarzać tę operację tak długo, aż będzie możliwa.*
   - *Na końcu pozostanie sytuacja, w której wszystkie pokoje są pełne i do każdego jest wejście - pozostałe pokoje będą zatem już zawsze pełne.*
   - *Prosta implemetacja miałaby złożoność $O(n^2)$. Złożoność liniową uzyskujemy w następujący sposób:*
     - *dla każdego pokoju p pamiętamy in[p] - liczbę czynnych wejść do pokoju, na poczatku jest to liczba wszystkich wejść, którą liczymy podczas wczytywania danych;*
     - *pokoje, dla których in[p] = 0, wrzucamy na kolejkę;*
     - *ściągamy pokój p z kolejki, oznaczamy go jako opróżniony, po czym rozważamy pokój x, do którego prowadzi wyjście z p - ponieważ p się*

opróżnił, wejście przestaje być aktywne: zmniejszamy in[x] o 1. Jeśli in[x] jest teraz równe 0, dorzucamy x do kolejki;

- *powtarzamy powyższy krok, aż kolejka się nie opróżni;*
- *pokoje, które pozostaną, mają wartość in[] dodatnią, a więc już zawsze ktoś będzie do nich wchodził.*

Zajęcia C-3: “Stos, Odwrotna Notacja Polska”

Cel zajęć i efekty uczenia

Główne cele zajęć / materiał do opanowania:

- Struktura stosu, szablon *stack* w STLu
- Odwrotna Notacja Polska, algorytm obliczania wyrażenia zadanego w ONP (maszyna stosowa)
- [opcjonalnie] Algorytm zamiany na ONP (“algorytm nastawni kolejowej”)

Dodatkowe cele:

- [opcjonalnie] Definicja nawiasowania przez reguły przepisywania (gramatyka formalna)

Zadania do rozwiązania na sprawdzarce

Nawiasy
Dany jest napis złożony ze znaków nawiasów (,),[,]{,}. Rozstrzygnąć, czy jest poprawnym nawiasowaniem.

Odwrotna Notacja Polska
*Dane jest wyrażenie w Odwrotnej Notacji Polskiej, złożone z liczb oraz znaków + i *. Obliczyć jego wartość.*

[opcjonalnie] ***Odwrotna Notacja Polska kontratakuje***
Dane jest wyrażenie zapisane w standardowej notacji, z liczbami i znakami +,. Zamienić je na Odwrotną Notację Polską.*

Szacunkowy czas trwania: 2-4 godziny lekcyjne.

Plan zajęć

Szacunkowy czas trwania: 2 godziny lekcyjne (sama kolejka), lub 4 godziny (z techniką obgryzania).

1. Struktura stosu, stos w C++/STL i własna implementacja
    - *Większość tłumaczenia odnośnie struktur danych i szablonów, została już wykonana w przypadku kolejki, stos jest nawet prostszy w implementacji.*

    - *Przy stosie nawet częściej o błąd “wyciągania z pustej struktury”, warto zwrócić na to szczególną uwagę.*
2. Zadanie “Nawiasy”
    - *Zaczynamy od jednego rodzaju nawiasu: co to znaczy, że nawiasowanie jest poprawne?*
    - *Możliwa definicja: “wtedy, kiedy nawiasy można dobrać w pary otwierający-zamykający tak, że żadne dwie pary się nie krzyżują”.*
    - *Dla jednego rodzaju nawiasów działa następujący algorytm: idziemy od lewej do prawej, za każdy otwierający nawias dodajemy 1, za zamykający odejmujemy 1. Wyrażenie jest poprawne, jeśli nigdy nie zejdziemy poniżej 0, a na końcu jest dokładnie 0.*
    - *Dla dwóch i więcej rodzajów nawiasów powyższy algorytm nie działa. Warto dać uczniom okazję do poprawiania go samodzielnie - to bardzo dobry moment, żeby wspólnie z nimi wymyślać nowe rozwiązania i wspólnie szukać kontrprzykładów na niedziałające.*
    - *Algorytm ogólny: nawiasy otwierające dorzucamy do stosu. Nawias zamykający musi zawsze pasować do ostatniego nawiasu otwierającego na stosie - jeśli nie, wyrażenie jest niepoprawne, jeśli tak, ściągamy ze stosu ten ostatni otwierający.*
    - *Jeśli na końcu stos jest niepusty, wyrażenie da się uzupełnić do poprawnego, kolejno dopisując zamykające nawiasy pasujące do ostatniego otwartego.*
3. Odwrotna Notacja Polska, algorytm obliczania wyrażenia
    - *Kwestia techniczna: trzeba czytać z wejścia dane, które raz są liczbą, a raz operatorem działania, i rozpoznawać po pierwszym znaku. Liczby następnie zamieniamy na typ liczbowy funkcją atoi() lub stoi().*
4. [opcjonalnie] Odwrotna Notacja Polska, algorytm zamiany na ONP
    - *Warto zacząć od sytuacji, kiedy jest tylko jeden operator (+) i nawiasy. Kiedy już uczniowie rozumieją, jak obsługuje się nawiasy, wprowadza się kwestię priorytetu operatorów.*
    - *Program wymaga dużej ilości instrukcji if/switch, a dodatkowo błędy bywają trudne do wykrycia. Może wymagać dużo pracy od uczniów i poprawiającego ich programy nauczyciela.*

Zajęcia C-4: “Kolejka priorytetowa”

Cel zajęć i efekty uczenia

Główne cele zajęć / materiał do opanowania:

- Struktura kolejki priorytetowej
- [opcjonalnie] Kopiec i operacje na kopcu, sortowanie przez kopcowanie

Dodatkowe cele:

- Przeładowanie operatorów (na przykładzie operatora “<”) w C++

Zadania do rozwiązania na sprawdzarce

Wybory
Jest n partii, każda otrzymała pewną liczbę głosów w wyborach. Rozdzielić m mandatów metodą d’Hondta.

Plan zajęć

Szacunkowy czas trwania: 2 godziny lekcyjne.

1. Kolejka priorytetowa
    - *Na dobrą sprawę kolejka priorytetowa nie różni się za wiele od zwykłej kolejki, przez co nie trzeba zbyt wiele tlumaczyć. Również STL-owa kolejka priority_queue działa bardzo podobnie do queue.*
    - *Kolejka na własnych strukturach: przeładowanie operatora “<”*
    - *Można (teoretycznie) robić kolejkę z własnym komparatorem (funkcją porównującą), ale w C++ ma to wyjątkowo dziwną składnię (inną niż przy sortowaniu) i jest należy skomplikowana, że raczej należy ją na tym etapie odradzać.*
    - *Złożoność kolejki: log n na operację, gdzie n to liczba elementów, które aktualnie są na kolejce.*
2. Zadanie “Wybory”
    - *Nie możemy stworzyć całej tabelki, bo zużyłoby to zbyt wiele czasu i pamięci komputera. Algorytm powinien działać w czasie O(n log n + m log n), gdzie n to liczba partii, m - mandatów.*
    - *Intuicja do metody d’Hondta: wszystkie partie liczą swoje głosy. Ta, która ma najwięcej, otrzymuje mandat, po czym liczba jej głosów zmniejsza się do połowy oryginalnej. Kolejna partia z największą liczbą głosów otrzymuje mandat, i tak dalej. Jeśli partia z oryginalną liczbą głosów x dostała mandat numer k, to liczba jej głosów będzie teraz wynosiła x/(k+1).*
    - *Ta intuicja wprost przekłada się na algorytm: wszystkie partie są na kolejce priorytetowej, wyciągamy z niej partię o największej liczbie glosów, po czym z powrotem dorzucamy do kolejki z “nową” liczbą głosów.*

    - *Treść zadania narzuca dość skomplikowany operator porównania, warto go dokładnie omówić z uczniami.*
3. [opcjonalnie] Sortowanie i kopiec
    - *Warto zauważyć, że używając kolejki priorytetowej da się napisać sortowanie, które będzie działało w O(n log n).*
    - *Ambitnym uczniom można w tym momencie objaśnić strukturę kopca i operacje kopcowania - ja typowo nie wyjaśniam tego uczniom, jako że bardzo rzadko (nawet na Olimpiadzie Informatycznej) zachodzi potrzeba rozumienia struktury kopca, albo pisania go samodzielnie. Podobne struktury będą wprowadzane przy okazji drzew przedziałowych.*

Zajęcia C-5: “Sortowanie przez scalanie”

Cel zajęć i efekty uczenia

Główne cele zajęć / materiał do opanowania:

- Algorytm sortowania przez scalanie
- Algorytm obliczania liczby inwersji w ciągu

Dodatkowe cele:

- Przypomnienie i utrwalenie idei rekursji

Zadania do rozwiązania na sprawdzarce

Sierżant
Dana jest tablica A zawierająca n liczb. Posortować ją, oraz obliczyć, ile było w niej inwersji (takich par x,y, dla których x<y, ale A[x]>A[y]).

Plan zajęć

Szacunkowy czas trwania: 2 godziny lekcyjne.

1. Scalanie
    - *Algorytm scalania jest opisany w wielu podręcznikach, ale na ogół w wersji z tablicą. **Bardzo** wygodne - i dużo łatwiejsze dydaktycznie - jest użycie w nim wektorów: piszemy funkcję scal(vector<int> &A, vector<int> &B, vector<int> &C), której zadaniem jest scalić dwa posortowane wektory w jeden.*
    - *Przy implementacji scalania warto zatrzymać się nad sytuacją, w której w jednym wektorze już skończyły się elementy. Być może najprostsza do wytłumaczenia implementacja funkcji scal(A,B,C) jest następująca:*

```
i = 0; j = 0;
dla k = 1, 2, …, A.size()+B.size():
        jeśli i < A.size() to x = A[i], inaczej x = nieskończoność
        jeśli j < B.size() to y = B[j], inaczej y = nieskończoność
        jeśli x <= y to:
                C.push_back(x)
                i = i+1
        inaczej:
                C.push_back(y)
                j = j+1
```

2. Rekursja i sortowanie
    - *Przy opisie algorytmu warto przypomnieć zasady rekursji:*

        - *wywoływanie się na mniejszych danych (tutaj: dwa razy mniejszych tablicach wektorach)*
        - *możliwość policzenia ostatecznego wyniku z wyników dla mniejszych danych (tutaj: scalanie)*
        - *znany początek rekursji (tablic długości 1 nie trzeba sortować)*
    - *Znowu, algorytm jest dużo łatwiejszy na wektorach niż tablicach: wektor wejściowy rozbijamy na dwa mniejsze, na każdym wywołujemy się rekurencyjnie, a potem scalamy z powrotem do wyjściowego. Jest to mniej efektywne niż oryginalna implementacja (wektory są dość powolne), ale znacznie prostsza do wyjaśnienia uczniom. W szczególności, funkcja jest jednoargumentowa: piszemy sortuj(A), zamiast mniej intuicyjnego sortuj(A,poczatek,koniec).*
3. Analiza algorytmu, złożoność
    - *Bardzo zalecane jest przeanalizowanie całego algorytm na wybranej tablicy (8-10 elementów), ze wszystkimi wywołaniami rekurencyjnymi i wszystkimi operacjami scalania, żeby uczniowie dokładnie przekonali się, co się dzieje w algorytmie.*
    - *Mając takie drzewo wywołań, już względnie łatwo oszacować złożoność: na każdym poziomie rekursji jest O(n) operacji - na pierwszym n operacji przy scalaniu, na drugim 2*n/2 (dwie tablice po n/2), na trzecim 4*n/4, itd. Zatem złożonosć to n*liczba poziomów rekursji, czyli n*log n.*
4. Liczenie inwersji
    - *Liczba inwersji to liczba par, w których większy element stoi przed mniejszym. Zauważamy, że ponieważ algorytm sortuje tablicę, wystarczy liczyć, ile razy mniejszy element “przeskoczy” nad większym podczas przestawiania go w tablicy. Dzieje się tak tylko podczas wywoływania instrukcji:*

        ```
        C.push_back(y)
        j = j+1
        ```

      Wtedy element y przeskakuje nad tyloma większymi elementami, ile pozostało jeszcze na wektorze A (czyli A.size()-i) - taką liczbę trzeba w tym momencie doliczyć do liczby inwersji.
    - *Trzeba przypomnieć uczniom, że liczba inwersji może być kwadratowa, a więc nie mieści się w 32-bitowym typie int.*

Zajęcia C-6: “Sortowanie szybkie”

Cel zajęć i efekty uczenia

Główne cele zajęć / materiał do opanowania:

- Algorytm sortowania szybkiego (QuickSort)
- Technika dwóch wskaźników (“gąsienicy”) na przykładzie problemu 2-SUM

Dodatkowe cele:

- Łagodne zapoznanie uczniów z algorytmami randomizowanymi (tzw. Las Vegas)

Zadania do rozwiązania na sprawdzarce

Randka w ciemno
Dany jest zbiór osób zawierający, dla każdej, imię oraz pewną liczbę. Znaleźć dwie osoby, których liczby sumują się do ustalonej sumy S.

Plan zajęć

Szacunkowy czas trwania: 2 godziny lekcyjne.

1. Idea algorytmu QuickSort
    - *Stosunkowo łatwym sposobem wprowadzenia QuickSorta jest zaproponowanie idei analogicznej do scalania: zamiast najpierw sortować rekurencyjnie, a potem scalić, trzeba podzielić tak, żeby scalanie nie było już potrzebne - po rekursji tablica była już posortowana. Na ogół to wystarczy uczniom do wymyślenia idei “małe elementy na początek, duże na koniec”.*
2. Podział tablicy (*pivoting*)
    - *Po wybraniu elementu dzielącego (pivota) podział tablicy można zrobić na dwa sposoby: metodą Hoare’a (oryginalny pomysł autora QuickSorta), albo metodą Lomuto (nowszą).*
    - *Metoda Lomuto jest prostsza do wymyślenia i łatwiejsza do wytłumaczenia uczniom, ale ma istotną wadę: źle działa na tablicach, w których jest dużo identycznych elementów (na przykład na tablicy, w której wszystkie elementy są równe), dzieląc je nierówno i powodując kwadratowy czas działania całego algorytmu. Należy to koniecznie uświadomić uczniom, jako że test “wszystkie liczby jednakowe” w zasadzie musi być jednym z testów na sprawdzarce. Można zmodyfikować metodę Lomuto tak, aby uwolnić ją od tej wady - na przykład dzieląc tablicę na trzy części (elementy mniejsze od pivota, równe i większe), ale wymaga to dodatkowego wysiłku.*
    - *Metoda Hoare’a działa zawsze i ma bardzo krótki kod, ale jej zachowanie na małych tablicach i w przypadkach brzegowych jest bardzo nieintuicyjne - dwa wskaźniki, których używa (idące z lewego i prawego końca tablicy), czasem*

wskazują na to samo miejsce, czasem wręcz się mijają, a czasem nawet wychodzą poza tablicę. Z mojego doświadczenia próba jakiegokolwiek formalnego udowodnienia uczniom na lekcji, że ta metoda działa, jest skazana na porażkę. Co gorsza, algorytm ma tendencję do błędnego działania przy najmniejszej, nawet zupełnie niewinnej, zmianie w kodzie.

- *Nie ma w tej sytuacji idealnego rozwiązania. Proponowany sposób na przeprowadzenie lekcji: napisać na tablicy metodę Hoare'a, rozważyć jej ogólne działanie (na dużych tablicach jest dość oczywiste) i zaznaczyć to, co jest napisane w powyższym punkcie - że działanie na przypadkach brzegowych wymaga bardzo dokładnego przepisania algorytmu.*

3. Złożoność algorytmu
    - *Rozważamy dwa skrajne przypadki: algorytm zawsze losuje medianę zbioru (czyli element środkowy pod względem wielkości - wychodzi złożoność jak w sortowaniu przez scalanie, O(n log n)), oraz algorytm zawsze losuje najmniejszy element (złożoność kwadratowa). Dla ilustracji można jeszcze (choć wymaga to więcej czasu i dobrej intuicji odnośnie logarytmów) wytłumaczyć przypadek "zawsze losujemy element, który jest w ⅓ pod względem wielkości"), i przez analogię argumentować, że nawet wybór elementu w 1/10 będzie dobry.*
    - *Wynika z tego, że aby algorytm działał wolno (kwadratowo), wybór musi być wyjątkowo pechowy, czyli dane wyjątkowo złośliwie dobrane. Warto w tym momencie poświęcić chwilę na układanie z uczniami złośliwych przykładów na różne wybory elementu dzielącego (np. na wybieranie go zawsze z początku tablicy/zawsze z końca tablicy/zawsze ze środka tablicy).*
    - *A zatem losowanie elementu dzielącego praktycznie gwarantuje dobrą złożoność - żadne spreparowane dane nie są w stanie "wyłożyć" algorytmu z losowaniem. Tutaj warto wytłumaczyć uczniom, że algorytm, który używa losowania (randomizowany) uznajemy za dobry, jeśli najczęściej działa dobrze. "Najczęściej"* ***nie oznacza*** *"na prawie każdych danych", tylko "przy prawie każdym losowaniu": poprawność zależy "od nas i naszego losowania", a nie od "tego, który układał dane".*
4. Zadanie "Randka w ciemno" i metoda dwóch wskaźników
    - *Pierwszy sposób rozwiązania zadania: sortujemy tablicę. Dla każdego elementu bierzemy jego wartość x, i sprawdzamy, czy w tablicy istnieje liczba S-x, za pomocą wyszukiwania binarnego.*
    - *Alternatywna, polecana metoda: niech A będzie rozważaną tablicą elementów, ustawiamy zmienną i na 0 (początek tablicy), j na ostatni element tablicy. Jeśli suma elementów stojących na miejscach i oraz j jest mniejsza niż S, to znaczy że element A[i] jest zbyt mały (nie może dać sumy S z żadnym innym elementem). Zatem możemy zapomnieć o elemencie A[i] do końca algorytmu, zwiększając wskaźnik i o 1. Analogicznie, jeśli A[i]+A[j]>S, to element A[j] nie będzie już potrzebny, wykonujmy więc j = j-1. Powtarzamy operację do momentu znalezienia sumy S, albo spotkania się wskaźników i oraz j.*

- *Technika ta zwana jest czasem “gąsienicą”, ma kilka wariantów i wiele zastosowań “olimpijskich”. Można rozważyć poświęcenie jej osobnych zajęć.*

Zajęcia D-1: “Grafy, algorytm BFS”

Cel zajęć i efekty uczenia

Główne cele zajęć / materiał do opanowania:

- Pojęcie grafu, spójnej składowej, ścieżki i odległości
- Algorytm przeszukiwania grafu wszerz (BFS)

Dodatkowe cele:

- Graf jako model sieci w świecie rzeczywistym

Zadania do rozwiązania na sprawdzarce

facepalm.bt
W danym grafie wypisać liczbę spójnych składowych, oraz odległości od wierzchołka 1 do pozostałych.

Plan zajęć

Szacunkowy czas trwania: 4 godziny lekcyjne.

1. Graf jako rysunek na płaszczyźnie - “węzły połączone kreskami”
2. Przykłady grafów z “prawdziwego życia”: graf sieci drogowej/połączeń komunikacyjnych/sieć społecznościowa
    - *Z mojego doświadczenia wynika, że grafy to jedna z łatwiejszych do pojęcia rzeczy dla uczniów - na przykładzie Facebooka/sieci drogowej błyskawicznie wyrabiają sobie właściwe intuicje*
    - *Warto już w tym momencie zwrócić uwagę, że graf Facebooka jest rzadki - liczba użytkowników jest rzędu 10^9, liczba znajomych przeciętnego użytkownika nie przekracza ~500.*
3. Grafy skierowane i nieskierowane, ścieżki w grafie, spójne składowe, źródło i odległość od źródła.
4. Przechowywanie grafu w pamięci - macierz sąsiedztwa i lista sąsiedztwa.
    - *Tutaj wracamy do przykładu Facebooka: macierz sąsiedztwa okazuje się dla niego kompletnie niepraktyczna. Tak naprawdę, macierze przydają się raczej rzadko.*
    - *Tradycyjnie w implementacji list sąsiedztwa używało się list wskaźnikowych, niektóre starsze podręczniki wciąż tak podają. W praktyce znacznie wygodniejsze jest użycie typu “vector”.*
5. Algorytm przeszukiwania grafu wszerz (BFS)
    - *BFS jest bardzo intuicyjnym algorytmem. Możliwa przykładowa droga wyjaśniania: narysowanie na tablicy grafu, zaznaczenia źródła i kolejne pytania:*

        - *“Które wierzchołki są w odległości 1 od źródła? Które w odległości 2, 3, etc.?”.*
        - *“Znając wierzchołki będące w odległości k, jak wyznaczyć te leżące w odległości k+1?”*
        - *“Jak uniknąć przeglądania za każdym razem wszystkich wierzchołków, żeby znaleźć te w odległości k?”*
    - *Warto szczególnie podkreślić fakt, że BFS nie tylko przegląda dokładnie jeden raz każdy wierzchołek, ale też dokładnie raz przechodzi po każdej liście sąsiedztwa, zatem działa liniowo od liczby krawędzi w grafie.*
    - *Co, jeśli w grafie jest więcej niż jedna spójna składowa? Jak zmodyfikować BFS, żeby odwiedził (i policzył) wszystkie.*

Zajęcia D-2: “Dwukolorowalność i DFS”

Cel zajęć i efekty uczenia

Główne cele zajęć / materiał do opanowania:

- Problem dwukolorowania grafu
- Algorytm przeszukiwania grafu wszerz (DFS)

Dodatkowe cele:

- [opcjonalnie] Obsługa grafu z dodatkową informacją na krawędziach

Zadania do rozwiązania na sprawdzarce

Anty-portal
Podzielić wierzchołki grafu na dwie klasy tak, aby wewnątrz żadnej z nich nie było krawędzi.

[opcjonalnie] ***Autostrady i polityka***
Na każdej krawędzi grafu napisany jest znak “+” albo “-”. W jednym ruchu można wskazać jeden z wierzchołków i zmienić wszystkie znaki wychodzących z niego krawędzi na przeciwne. Rozstrzygnąć, czy i jak można doprowadzić do sytuacji, w której wszystkie znaki na krawędziach są plusami.

Plan zajęć

Szacunkowy czas trwania: 2 godziny lekcyjne.

1. Problem dwukolorowania i jego proste rozwiązanie - ustalenie koloru jednego wierzchołka pociąga za sobą kolory całego grafu (a w zasadzie: całej spójnej składowej)
2. Rozwiązanie problemu za pomocą BFS-a
3. Rekurencyjny algorytm DFS
    - *Na tym etapie podstawową zaletą DFS-a jest prostota - na razie nie umiemy zrobić za jego pomocą nic, czego byśmy nie potrafili za pomocą BFS-a.*
    - *To dobry moment, żeby wprowadzić konstrukcję ze standardu C++11:*
      for *(****int*** *x : lista_sąsiadów) { …. };*
    - *Warto zwrócić uwagę na fakt, że rekursja w DFSie może przeszkadzać: zużycie pamięci przy dużej liczbie wierzchołków jest znacznie większe niż w BFS.*
4. [opcjonalnie] Zadanie “Autostrady i polityka”
    - *To zadanie nie jest wprost na dwukolorowalność, ale algorytm jest bardzo podobny.*
    - *Oczywiste jest, że nie ma sensu dwukrotnie zmieniać znaków przy tym samym wierzchołku. Dodatkowo, jeśli jakiś zbiór wierzchołków jest rozwiązaniem (daje same plusy), to jego dopełnienie - wszystkie pozostałe wierzchołki - też jest*

dobrym rozwiązaniem. Wzadaniu napisane jest, że wierzchołek 1 ma pozostać niezmieniony - to założenie jest tylko po to, żeby rozwiązanie było jednoznaczne.

- *Jeśli wiemy, że nie wolno dotknąć wierzchołka 1, to determinuje działanie przy innych wierzchołkach - jeśli na krawędzi wychodzącej z 1 jest "-", to drugi wierzchołek musi zostać użyty, jeśli jest "+", to nie może być użyty. W ten sposób, analogicznie do dwukolorowania, wyznaczamy kolejno stan wszystkich wierzchołków. Na końcu przechodzimy przez wszystkie krawędzie i sprawdzamy, czy wyznaczone przez nas rozwiązanie zmieni wszystkie na "+"*
- *Zadanie, poza nowym spojrzeniem na ideę algorytmu dwukolorowania, wprowadza model, w którym na każdej krawędzi trzeba trzymać dodatkową informację (znak). Zatem lista sąsiadów wierzchołka musi być listą struktur, nie tylko liczb-numerów sąsiadów. Przyda się to przy grafach ważonych.*
- *Prawdziwy jest następujący fakt: rozwiązanie istnieje wtedy i tylko wtedy, gdy na każdym cyklu w grafie iloczyn znaków jest dodatni. Nie wydaje się jednak możliwe (a w każdym razie nie jest łatwe) rozwiązanie zadania oparte na tym fakcie.*

Zajęcia D-3: “BFS na nietypowych grafach”

Cel zajęć i efekty uczenia

Główne cele zajęć / materiał do opanowania:

- Szczególny rodzaj grafu - plansza z zablokowanymi polami (*grid graph*)
- Tworzenie grafu w czasie działania programu

Dodatkowe cele:

- [opcjonalnie] Nauka podejścia do długich programów, unikanie powtórzeń kodu

Zadania do rozwiązania na sprawdzarce

Loch czarnoksiężnika

[wersja prostsza zadania - “Loch początkującego czarnoksiężnika”]: Dana jest plansza, na której niektóre pola są wolne, niektóre zablokowane. Jedno pole jest polem startowym i stoi na nim pionek, pewne inne pole jest metą. Pionek może poruszać się tylko po wolnych polach, w każdym ruchu przechodząc o 1 pole. Ile minimalnie potrzeba ruchów, aby pionek dotarł do mety?

[wersja trudniejsza zadania] Pionek dodatkowo jest kolorowy - startuje w kolorze czerwonym. Niektóre pola sprawiają, że pionek zmienia kolor - z czerwonego na zielony i odwrotnie. Z kolei na inne pola można wejść tylko, jeśli pionek ma odpowiedni kolor.

Plan zajęć

Szacunkowy czas trwania: 2 godziny lekcyjne.

1. Wersja prosta zadania: graf planszy, z wierzchołkami jako polami, krawędzie między sąsiednimi polami
2. Implementacja BFS-a na podanym grafie
    - *Należy unikać tworzenia i przechowywania całego grafu - prawdopodobnie i tak nie zmieści się w pamięci operacyjnej. Wierzchołki to pola, a więc pary liczb, które możemy bez zmian trzymać na kolejce. Dla każdego pola pamiętamy, co na nim jest (wolne/zablokowane/meta), oraz czy już było odwiedzone (co jest standardem w algorytmie BFS).*
    - *Po ściągnięciu wierzchołka z kolejki należy przeglądnąć jego wszystkie krawędzie. Nie ma potrzeby ich przechowywać - jeśli wierzchołek to pole (x,y), to jego sąsiadami są pola (x+1, y), (x-1, y), (x, y+1), (x, y-1).*
    - *Dla każdego sąsiada sprawdzamy, czy nie jest zablokowanym polem, i czy nie było odwiedzone. Jeśli nie - dokładamy je do kolejki. To powoduje konieczność powtórzenia cztery razy tego samego ciągu instrukcji - warto zachęcać uczniów, aby nie kopiowali kodu (copy-paste): stwarza to dużą możliwość powielenia*

błędów i jest bardzo niewygodne przy jakiejkolwiek konieczności poprawiania. Jedną z możliwości jest na przykład następująca pętla:

```
ściągnij z kolejki pole (x,y);
vector<int> dx = {1, -1, 0, 0};
vector<int> dy = {0, 0, 1, -1};
for (q = 0; q < 4; q++)
        sprawdź pole (x+dx[q], y+dy[q]);
```

3. Wersja trudniejsza zadania
    - *Traktujemy loch, jakby miał dwa "piętra", tzn. pole jest opisane trzema liczbami (k,x,y) gdzie k to kolor pionka (0 lub 1), a (x,y) położenie. Z pola (k,x,y) można zawsze przejść do sąsiadów (pola (k,x+1,y) etc.). Jeśli pole (k, x, y) jest wirem (zmianą koloru), można wtedy również przejść do pól (1-k, x+1, y), (1-k, x-1, y) itd. Poza tym rozwiązanie jest identyczne.*
    - *W tej wersji kod powtarzałby się 8 razy: tym bardziej należy zachęcać uczniów, aby tego unikali.*
    - *Zadanie w pełnej wersji jest jednym z najtrudniejszych kodersko w całym programie; uczniowie od początku mają tendencję do pisania programów "po swojemu", nieraz w sposób nieuporządkowany, czasem ciężko ich przekonać się do zmiany stylu (robienia wcięć, opisowego nazywania zmiennych, używania funkcji zamiast copy-paste); jedną z funkcji tego zadania jest uświadomienie uczniom, że praktyki dobrego i czytelnego pisania mają sens. Należy się jednak przygotować na to, że wielu uczniów będzie potrzebować pomocy przy poprawianiu swoich programów. Czasem można rozważyć środki bardziej stanowcze: polecić uczniowi, aby napisał swój kod całkiem od nowa, jeśli jest źle zaprojektowany i nieczytelny.*

Zajęcia D-4: “Grafy skierowane i sortowanie topologiczne”

Cel zajęć i efekty uczenia

Główne cele zajęć / materiał do opanowania:

- Pojęcia grafu skierowanego, cyklu
- Kolejność topologiczna i sortowanie topologiczne

Dodatkowe cele:

- Podział krawędzi w algorytmie DFS: drzewowe/wsteczne/poprzeczne/wyprzedzające

Zadania do rozwiązania na sprawdzarce

Komisja śledcza
Dany jest graf skierowany. Posortować go topologicznie lub stwierdzić, że ma cykl.

Plan zajęć

Szacunkowy czas trwania: 2 godziny lekcyjne.

1. Grafy skierowane, pojęcie cyklu skierowanego
2. Algorytm DFS dokładniej: kolory wierzchołków, rodzaje krawędzi
    - *Algorytm DFS koloruje wierzchołki na biało (jeszcze nieodwiedzone), szaro (DFS się zaczął, ale nie skończyły się jeszcze wszystkie wywołania rekurencyjne) lub czarno (DFS się skończył).*
    - *Krawędź u→v jest wsteczna, jeśli w momencie, kiedy przeglądamy wierzchołek u, wierzchołek v jest szary. Krawędź wsteczna zawsze domyka cykl w grafie. Możemy więc stosunkowo łatwo zrobić algorytm, który wykrywa cykl - modyfikujemy DFS-a tak, aby patrzył na kolor końca przeglądanej krawędzi. Biały oznacza wywołanie rekurencyjne, czarny jest ignorowany, szary oznacza krawędź wsteczną, a więc cykl.*
3. Acykliczny graf skierowany (DAG), pojęcie kolejności topologicznej
    - *Tutaj konieczne jest dłuższe zatrzymanie się i narysowanie kilku przykładów na tablicy.*
    - *Trzeba uczulić uczniów na to, że kolejność topologiczna może być więcej niż jedna - a wręcz może ich być bardzo dużo. Warto zrobić też kilka przykładów z pytaniem uczniów “czy zadana kolejność jest topologiczna”, albo “znajdź wszystkie możliwe porządki topologiczne”.*
4. Znajdowanie kolejności topologicznej za pomocą DFS-a
    - *Standardowy algorytm: “wypisz wierzchołek na końcu DFS-a” ustawia wierzchołki w odwrotnej kolejności topologicznej. Sam algorytm jest więc niezwykle prosty.*
    - *Warto opowiedzieć szkic dowodu poprawności: w znalezionej kolejności nie może być wystąpić najpierw wierzchołek u, a potem v taki, że u ← v. Wtedy bowiem nie moglibyśmy zakończyć DFS(v) nie kończąc wcześniej DFS(u).*

5. [opcjonalnie] Znajdowanie kolejności topologicznej metodą “obgryzania”
    - *Dla każdego wierzchołka v tworzymy zmienną licznik[v], która zlicza, ile krawędzi do niego wchodzi.*
    - *Jeśli w grafie jest wierzchołek z licznik[v] = 0, to zawsze może być pierwszy w kolejności topologicznej. Wypisujemy v i usuwamy go z grafu, zmniejszając licznik wszystkim jego sąsiadom.*
    - *Powtarzamy procedurę dla wierzchołków, dla których licznik spadł do 0. Aby złożoność pozostała liniowa, używamy kolejki, na którą trafiają wszystkie wierzchołki, których licznik się wyzerował.*
    - *Warto przedstawić uczniom obie metody (DFS i obgryzanie) - obie są użyteczne!*

Zajęcia D-5: “Silnie spójne składowe”

Cel zajęć i efekty uczenia

Główne cele zajęć / materiał do opanowania:

- Pojęcie silnie spójnej składowej, algorytm Kosaraju znajdowania silnie spójnych składowych

Zadania do rozwiązania na sprawdzarce

Euro 2102
Dany jest graf skierowany oraz pewna liczba zapytań par wierzchołków. Dla każdej pary odpowiedzieć, czy istnieje ścieżka w obie strony między nimi.

Plan zajęć

Szacunkowy czas trwania: 2 godziny lekcyjne.

1. Pojęcie silnie spójnej składowej
    - *Samo pojęcie jest dość intuicyjnie, dobry przykład-rysunek na tablicy powinien wystarczyć uczniom do zrozumienia.*
    - *Warto w tym momencie zadać pomocnicze zadanie: “Dany jest graf skierowany - wyznacz silnie spójną składową wierzchołka numer 1.” Aby je rozwiązać, wystarczy po prostu znaleźć wszystkie wierzchołki osiągalne z 1, a potem wszystkie wierzchołki, z których można dojść do 1, odwracając krawędzie grafu. Znalezienie tego uproszczonego algorytmu daje dobrą intuicję przed właściwym.*
2. Algorytm Kosaraju znajdowania silnie spójnych składowych
    - *Jest to najbardziej znany algorytm rozwiązujący ten problem, dostępny w wielu miejscach, w tym na Wikipedii. Warto zwrócić uczniom uwagę na kwestie techniczne, w tym najbardziej typowe błędy, jakie można w nim popełnić:*
        - *Algorytm zawiera dwie rekurencyjne procedury DFS. Bardzo częstą pomyłką jest nazwanie ich podobnie (np. “DFS1” i “DFS2”), skopiowanie jednego, otrzymać drugi, ale pozostawienie oryginalnej nazwy przy wywoływaniu rekurencyjnym. W ten sposób np. “DFS2” wywołuje rekurencyjnie “DFS1” zamiast siebie samego.*
        - *Pomiędzy pierwszym a drugim DFS-em trzeba odwrócić graf, albo po prostu przechowywać w pamięci dwie kopie grafu, w tym jedną z odwróconymi krawędziami. Łatwo zapomnieć o odwróceniu, albo też drugi DFS (zwłaszcza przy skopiowaniu!) wywołać znowu na pierwszym grafie.*
        - *Pierwszy DFS znajduje pewną kolejność wierzchołków, drugi DFS należy wykonywać na wierzchołkach w (odwróconej) właśnie kolejności. Łatwo się pomylić i wywołać DFS-a po prostu w kolejności od największych*

numerów wierzchołków [napisać "for(i=n-1; i>=0; i--) dfs(i); zamiast for(i=n-1; i>=0; i--) dfs(tablica_kolejnosci[i]);]. Jest to o tyle problematyczne, że algorytm z tym błędem (podobnie jak w przypadku dwóch powyżej podanych) działa poprawnie na małych przykładach, więc trudno jest zrozumieć, czemu sprawdzarka zgłasza błędną odpowiedź.

3. [opcjonalnie] Dowód poprawności algorytmu
    - *Łatwo widać, że algorytm jest liniowy, znacznie trudniej pokazać, że zawsze właściwie znajduje silnie spójne składowe. Być może warto w ogóle pominąć ten konkretny dowód poprawności.*
    - *Jeśli zdecydujemy się dowód przedstawić uczniom, polecam zaczerpnąć go z książki Jeffa Ericksona "Algorithms", za to odradzam niezbyt fortunny i zagmatwany argument z podręcznika "Wprowadzenie do algorytmów" Cormena i innych.*

Zajęcia E-1: “Wprowadzenie do programowania dynamicznego”

Cel zajęć i efekty uczenia

Główne cele zajęć / materiał do opanowania:

- Programowanie dynamiczne: metoda bottom-up
- (opcjonalnie) Metoda spamiętywania (memoization)

Zadania do rozwiązania na sprawdzarce

Piramida
Plansza składa się z n kolejnych pól, numerowanych 1...n, z których niektóre są zablokowane (niedostępne). Pionek stoi na polu 1 i w każdym ruchu przemieszcza się do przodu co najmniej o jedno, a co najwyżej o 6 pól. Na ile sposobów może dojść do pola n, nie stając po drodze na żadnym zablokowanym polu?

Mosiężny Most
Dany jest ciąg n liczb. Wybrać z niego pewne liczby tak, żeby otrzymać jak największą sumę, nie wybierając nigdy trzech sąsiednich liczb.

Plan zajęć

Szacunkowy czas trwania: 2 godziny lekcyjne.

1. Metoda rekurencyjna i iteracyjna na obliczanie ciągu Fibonacciego
    - *Podstawowa wada metody rekurencyjnej: wielokrotne obliczanie tych samych wartości; metoda iteracyjna w bardzo oczywisty sposób oblicza każdą wartość tylko raz.*
2. Główna idea programowania dynamicznego - obliczyć podzadania jak w rekursji, ale tak, żeby żadnego nie obliczać wielokrotnie.
3. Metoda “top-down”: formułujemy podzadania - mniejsze wersje ostatecznego zadania - i ustawiamy je w łańcuch (w tym wypadku podzadania to wartości F(i)).
    - *Początkowe elementy łańcucha są łatwe do obliczenia (F(0) i F(1))*
    - *Każdy element da się obliczyć z poprzednich (F(i) z F(i-1) i F(i-2))*
    - *Ostatni element to nasze pierwotne zadanie (F(n)).*
    - *Metodę top-down polecam jako główny sposób przedstawiania algorytmów dynamicznych - jest dość intuicyjna i uniwersalna.*
4. [opcjonalnie] Metoda “bottom-up”, czyli spamiętywanie (nieoficjalnie ang. *memoization*):
    - *Wywołujemy normalną rekursję, ale przy wejściu do procedury sprawdzamy, czy wartość była już kiedyś obliczana.*

- *Jeśli nie, obliczamy ją rekurencyjnie i zapamiętujemy w tablicy (ew. STL-owej mapie).*
- *Jeśli już była i jej wartość jest w tablicy, zwracamy ją bez dalszych wywołań rekurencyjnych.*
- *Przykładowy pseudokod obliczania liczb Fibonacciego tą metodą:*

```
F[0..n] = {-1, -1, …, -1}

fibonacci(x):
        jeśli F[x] = -1:
                jeśli x = 0 to F[x] = 0
                jeśli x = 1 to F[x] = 1
                jeśli x > 1, to F[x] = fibonacci(x-1) + fibonacci(x-2)
                zwróć wynik F[x]
        jeśli F[x] ≠ -1:
                zwróć wynik F[x]
```

5. Zadanie “Piramida”.
   - *Jeden z najprostszych przykładów na programowanie dynamiczne: liczba sposobów S[k] dotarcia na pole k to 0, jeśli k jest zablokowane, i S[k-1]+...+S[k-6] w przeciwnym razie.*
6. Zadanie “Mosiężny most”
   - *Znowu, sformułowanie podzadania jest dość proste - jeśli tablica wejściowa to A[1..n], częściowy wynik W[k] to maksymalny zysk na początkowym fragmencie A[1..k].*
   - *Istotnym - i pouczającym - elementem zadania jest zauważenie, że wynik W[k] da się wyliczyć z poprzednich.*
   - *Jeśli nie wybieramy elementu A[k], to W[k] = W[k-1] - wynik jest taki sam, jak na elementach 1...k-1*
   - *Jeśli wybieramy A[k], ale nie wybieramy A[k-1], to W[k] = A[k] + W[k-2].*
   - *Jeśli wybieramy A[k] i A[k-1], to nie możemy wybrać A[k-2], więc wynik to W[k] = A[k]+A[k-1]+W[k-3].*

Zajęcia E-2: “Przyśpieszanie algorytmów dynamicznych”

Cel zajęć i efekty uczenia

Główne cele zajęć / materiał do opanowania:

- Algorytm najdłuższego podciągu rosnącego
- Przykłady podejścia dynamicznego - szukanie podzadań

Zadania do rozwiązania na sprawdzarce

Remont autostrady
W autostradzie jest n dziur, na pozycjach a_1, a_2, …, a_n od początku autostrady. Naprawienie i-tej dziury kosztuje c_i. Można też przykładać długie łaty - każda łata kosztuje M i przykrywa wszystkie dziury na pewnym przedziale domkniętym długości d. Jaki jest minimalny koszt załatania wszystkich dziur?

Chorąży
Dany jest ciąg n liczb. Ile operacji przeniesienia elementu w inne miejsce trzeba wykonać, aby ciąg był posortowany?

Plan zajęć

Szacunkowy czas trwania: 2 godziny lekcyjne.

1. Zadanie “Remont autostrady”
    - Podzadanie jest analogiczne jak w poprzednio rozważanych problemach - koszt W[k] załatania dziur a_1, a_2, …, a_k.
    - *Albo dziurę a_k naprawiamy pojedynczo - wtedy koszt to $W[k-1]+c_k$, albo przystawiamy do niej łatę, przykrywając również część poprzednich dziur - wtedy koszt to $M + W[s]$, gdzie s jest największe takie, że $a_s < a_k$ - d.*
    - *Dziurę s możemy wyszukać binarnie, osiągając koszt n log n całego algorytmu.*
2. Najdłuższy podciąg rosnący - algorytm kwadratowy
    - Dla danego ciągu liczb szukamy nadłuższego podciągu (wyboru pewnych elementów ciągu, niekoniecznie sąsiednich), który jest silnie rosnący.
    - *Najbardziej intuicyjny algorytm dynamiczny: dla każdego elementu obliczamy długość podciągu rosnącego, który się na nim kończy - jeśli A[1..n] to tablica wejściowa, a W[1..n] - tablica wyników, to $W[k] = min \{ W[i] : i < k, A[i] < A[k] \}$.*
    - *Algorytm działa w czasie kwadratowym, choć można go przyśpieszyć do O(n log n) wykorzystując np. statyczne drzewa przedziałowe - oczywiście w tym momencie kursu uczniowie jeszcze ich nie znają. Tematem lekcji jest zupełnie inny, szybki algorytm na najdłuższy podciąg rosnący, opisany poniżej.*
3. Zmiana wymiaru - algorytm O(n log n)

- *Odwracamy pytanie z poprzedniego algorytmu: dla każdej możliwej długości podciągu rosnącego trzymamy minimalny element, na którym taki podciąg może się skończyć. Pełny opis i kod można znaleźć np. w tym miejscu.*

Zajęcia E-3: “Dwuwymiarowe programowanie dynamiczne”

Cel zajęć i efekty uczenia

Główne cele zajęć / materiał do opanowania:

- Wprowadzenie do wielowymiarowych algorytmów dynamicznych
- Odtwarzanie wyniku przy programowaniu dynamicznym
- Optymalizacja pamięciowa w algorytmach dynamicznych

Dodatkowe cele:

- Różnica między złożonością czasową i pamięciową - podejście teoretyczne i praktyczne

Zadania do rozwiązania na sprawdzarce

Diamenty
Na niektórych polach kwadratowej planszy n x n umieszczone są diamenty. Pionek przechodzi z pola w lewym górnym rogu do pola w prawym dolnym rogu, poruszając się wyłącznie w prawo i w dół. Znaleźć trasę, w której można zebrać w ten sposób najwięcej diamentów.

Neon
Dane sa dwa napisy. Znaleźć najdłuższy podciąg, który występuje jednocześnie w obu (niekoniecznie jako spójny fragment).
Wersja I: podać tylko długość podciągu, mieszcząc się w stosunkowo niskim (np. 8MB) limicie pamięci.
Wersja II: podać cały podciąg.
[opcjonalnie] Wersja III: podać cały podciąg oraz zmieścić się w niskim limicie pamięci.

Plan zajęć

Szacunkowy czas trwania: 4 godziny lekcyjne.

1. Zadanie “Diamenty”
    - *Podzadanie: maksymalna liczba diamentów, którą można zebrać kończąc na polu (i,j).*
    - *Wartość podzadania: W[i][j] = max(W[i-1][j],W[i][j-1]) [+ 1, jeśli na polu (i,j) jest diament] - wynika to z prostej obserwacji, że ścieżka kończąca na polu (i,j) musi wcześniej przejść przez jedno z pól (i-1,j), (i,j-1).*
    - *Odtwarzanie wyniku: na pole (i,j) zawsze wchodzimy z pola (i-1,j) albo (i,j-1) - z tego z nich, które ma większą wartość W. Zaczynając od pola (n,n) i cofając się po jednym ruchu, odtwarzamy całą drogę.*
2. Zadanie “Neon”

- *Problem najdłuższego wspólnego podciągu jest nie tylko ważnym ćwiczeniem z programowania dynamicznego, ale jest fundamentalny ze względu na zastosowania, od podobieństwa kodów DNA po szukanie plagiatów w tekstach.*
- *Podzadanie: dla słów A[1..n] i B[1..m] wartość NWP[i][j] to najdłuższy wspólny podciąg fragmentów A[1..i] oraz B[1..j]. Pełny algorytm jest opisany w bardzo wielu miejscach, nawet na Wikipedii, a także skrótowo zapisany w poniższej ramce:*

```
najdłuższy wspólny podciąg(A[1..n], B[1..m])
        NWP[i][0] = 0, dla każdego i=1,...,n
        NWP[0][j] = 0, dla każdego j=1,...,m
        dla i = 1,..., n:
                dla j = 1,..., m:
                        jeżeli A[i] = B[j]
                                NWP[i][j] = NWP[i-1][j-1]+1
                        w przeciwnym razie:
                                NWP[i][j] = max(NWP[i-1][j],NWP[i][j-1])
```

- *Warto z uczniami przeanalizować całą procedurę na dwóch przykładowych słowach, wypełniając kolejno całą tablicę NWP[i][j] po jednym polu.*

3. Najdłuższy wspólny podciąg - odtwarzanie wyniku
   - *Główna idea - analogicznie jak w "Diamentach", cofamy się od wyniku (NWP[m][n]) po jednym polu.*
     - *Metoda I: w czasie tworzenia tablicy NWP[i][j] zapisujemy w osobnej tablicy R[][], z którego pola ((i-1,j), (i,j-1) lub (i-1,j-1) weszliśmy na (i,j)) - np. R[i][j] = 1, jeśli NWP[i][j] = NWP[i-1][j], 2 jeśli NWP[i][j] = NWP[i][j-1] lub 3, jeśli NWP[i][j] = NWP[i-1][j-1]+1. Mając zapisane te wartości, możemy odtworzyć wynik.*
     - *Metoda II: zauważamy, ze wartości R[i][j] nie musimy mieć zapisanych, możemy je na nowo obliczyć w czasie cofania się, porównując za każdym razem wartość NWP[i][j] z sąsiednimi wartościami.*
4. Najdłuższy wspólny podciąg - optymalizacja pamięciowa
   - *Zauważamy, że przy obliczaniu NWP[i][j] nie korzystamy z wartości NWP[i'][j] dla i'<i-1, możemy więc usunąć je z pamięci. Najprostszą metodą zmodyfikowania oryginalnego algorytmu jest dopisanie "mod 2" do każdego NWP[i][...], jak w poniższej ramce:*

```
najdłuższy wspólny podciąg(A[1..n], B[1..m])
        NWP[i][0] = 0, dla i = 0, 1
        NWP[0][j] = 0, dla każdego j=1,...,m
        dla i = 1,..., n:
                dla j = 1,..., m:
                        jeżeli A[i] = B[j]
```

NWP[i mod 2][j] = NWP[i mod 2][j][j-1]+1
***w przeciwnym razie**:*
NWP[i mod 2][j] =
max(NWP[i mod 2][j],NWP[i mod 2][j-1])

- *Ta technika, niezwykle prosta w implementacji - najpierw napisanie algorytmu z "kwadratową" tablicą, a potem zredukowanie jednego wymiaru za pomocą operacji modulo - działa bardzo często na dwuwymiarowe algorytmy dynamiczne i warto przekazać uczniom, aby ją zapamiętali.*
- *Optymalizacje pamięciowe nie zawsze są potrzebne, ale w wypadku algorytmów dynamicznych często właśnie ten przypadek zachodzi - na przykład dla danych o wielkości ~20 000, złożoność czasowa $O(n^2)$ jest jak najbardziej akceptowalna, ale złożoność pamięciowa $O(n^2)$ nie jest do udźwignięcia dla standardowych komputerów.*
- *Warto zwrócić uwagę na fakt, że w podanej formie optymalizacja pamięciowa i odtwarzanie wyniku wykluczają się ze sobą - odtwarzanie potrzebuje całej tablicy wyników podzadań, nie tylko dwóch ostatnich wierszy. Aby móc pogodzić te techniki ze sobą, konieczne są triki, takie jak algorytm Hirschberga omówiony w następnym punkcie*

5. [opcjonalnie] Algorytm Hirschberga - odtwarzanie wyniku dla najdłuższego wspólnego podciągu, przy jednoczesnej optymalizacji pamięciowej
   - *Algorytm (opisany np. tutaj) nie jest bardzo trudny matematycznie, ale dość złożony i abstrakcyjny - wymaga dobrego opanowania rekursji, a także głębokiego zrozumienia "oryginalnego", opisanego wyżej, algorytmu. Sugeruję stosować go wyłącznie jako dodatkowy materiał dla najambitniejszych uczniów.*

Zajęcia E-4: “Problem plecakowy”

Cel zajęć i efekty uczenia

Główne cele zajęć / materiał do opanowania:

- Problem plecakowy

Dodatkowe cele:

- Algorytmy wielomianowe vs. pseudowielomianowe

Szacunkowy czas trwania: 2 godziny lekcyjne.

Zadania do rozwiązania na sprawdzarce

Skarb faraona

Klasyczny problem plecakowy: dane jest n przedmiotów, każdy o własnej wadze oraz wartości. W plecaku o pojemności B należy zmieścić przedmioty o możliwie najmniejszej łącznej wadze.

Plan zajęć

Szacunkowy czas trwania: 2 godziny lekcyjne.

1. *Problem plecakowy - opis i implementacja*
    - *Problem i odpowiedni algorytm jest szczegółowo opisany większości podręczników - podaję jedynie sugestie techniczne i prezentacyjne.*
    - *Trochę nieintuicyjne dla uczniów może być “podręcznikowa” definicja wynik[k][x] jako “maksymalna wartość na pierwszych k przedmiotach i plecaka pojemności x”, jako że nie jest jasne, czym jest “pierwszych k przedmiotów”. Dla celów algorytmu przedmioty trzeba uporządkować, chociaż oczywiście dowolna kolejność jest dobra.*
    - *W celu pokonania tych trudności, być warto zdefiniować “tablicę chwilowego wyniku” A[0..B], która podaje aktualny najlepszy wynik (maksymalną wagę przedmiotu) dla każdej pojemności plecaka od 0 do B. Na początku nie ma żadnych przedmiotów, więc A[x] = 0 dla każdego x. Potem dokładamy po jednym przedmiocie i analizujemy, jak zmienia się tablica.*
    - *Przerabiamy z uczniami wybrany przykład 4-5 przedmiotów - klasa powinna stosunkowo szybko “złapać” zmiany w tablicy A i samodzielnie wymyślić ostateczny wzór - przy dokładaniu przedmiotu o wadze S i wartości V wartość A[x] zmienia się na A[x] = max(A[x],V+A[x-S]).*
    - *Można teraz zmienić tablicę A na dwuwymiarową tablicę wynik[k][x], a wzór na wynik[k][x] = max(wynik[k-1][x],V+wynik[k-1][x-S]) - otrzymujemy algorytm o kwadratowej złożoności pamięciowej, który pozwoli też na odtworzenie wyniku.*
    - *Można też pozostać przy jednej tablicy A, ale należy koniecznie zwrócić uwagę, że instrukcja A[x] = max(A[x],V+A[x-S]) musi być wykonana w pętli dla wszystkich*

x koniecznie “w dół” od B do 0, inaczej nadpiszemy wartości A[x], których będziemy za chwilę potrzebować.

2. *Złożoność algorytmu*
    - *Złożoność wynosi oczywiście O(nB), gdzie n jest liczbą przedmiotów, a B - pojemnością plecaka. Warto zaakcentować, że algorytm może działać bardzo długo mimo, że dane są małe (np. $B = 10^{10} = 10\,000\,000\,000$ to 11 znaków w pliku wejściowym, a wymaga ogromnej liczby operacji od algorytmu). Być może warto w tym momencie opowiedzieć uczniom o różnicy między algorytmami wielomianowymi i wykładniczymi.*