Program nauczania w ramach Olimpiady Informatycznej

Wojciech Baranowski

18 lutego 2024

Spis treści

1 Wstęp **2**
1.1 Git . . . 2
1.1.1 Git - przykład liniowy: . . . 2
1.1.2 Git - w praktyce: . . . 2
1.1.3 Repozytorium . . . 2
1.1.4 Repozytorium - przykład . . . 3
1.1.5 Repozytorium - klucze . . . 3
1.1.6 Repozytorium - w praktyce . . . 3
1.2 Złożoność obliczeniowa . . . 4
1.2.1 Symbol O . . . 4
1.2.2 Symbol O - w praktyce . . . 4
1.2.3 Symbol o . . . 4
1.2.4 Symbol o - w praktyce . . . 4
1.2.5 Symbol Ω . . . 5
1.2.6 Symbol Ω - w praktyce . . . 5
1.2.7 Symbol ω . . . 5
1.2.8 Symbol ω - w praktyce . . . 5
1.2.9 Symbol Θ . . . 5
1.2.10 Symbol Θ - w praktyce . . . 6
1.3 Szacowanie wzorcowej złożoności . . . 6
1.3.1 Koncept . . . 6
1.3.2 Szacowanie . . . 6
1.3.3 Przykład . . . 6
1.3.4 W praktyce . . . 7

1 Wstęp

W tej sekcji umieszczone będą pojęcia, których zrozumienie oraz znajomość praktyczna są niezbędne do dalszej kontynuacji nauki.

1.1 Git

Git - system kontroli wersji opracowany przez Linusa Torvalds'a w 2005 roku. Służy on do utrzymywania historii zmian danego zbioru plików i katalogów w potencjalnie rozdystrybuowanym środowisku. Pozwala na zrównoleglenie pracy wielu deweloperów nad jednym projektem, poprzez utrzymywanie nieliniowej struktury zmian.

1.1.1 Git - przykład liniowy:

Alice rozwija aplikację, jednak przydałaby się jej możliwość potencjalnego powrotu do poprzedniej (działającej) wersji programu, jeśli nowa zmiana pójdzie nie po jej myśli. Decyduje się więc skorzystać z git'a. Każdorazowo, gdy jej program znajdzie się w dostatecznie stabilnym stanie, Alice tworzy commit'a utworzonych przez nią zmian. Dzięki temu w dowolnym momencie może wrócić do dowolnego zacommitowanego punktu czasowego.

1.1.2 Git - w praktyce:

Mamy gotowy program, który liczy średnią arytmetyczną dwóch liczb (póki co całkowitych, zwracany wynik także jest całkowity).

1. inicjalizujemy repozytorium git'a: **git init**,
2. tworzymy gałąź (linię czasową) dla naszego programu: **git checkout -b nazwa**,
3. dodajemy nasz program do zbioru plików, które chcemy zacommitować: **git add program.cpp**,
4. commitujemy nasze zmiany: **git commit -m "initial commit"**.

Następnie przerabiamy program tak, by obsługiwał liczby zmiennoprzecinkowe.

1. dodajemy nową wersję pliku do zbioru plików, które chcemy zacommitować: **git add program.cpp**,
2. commitujemy nasze nowe zmiany: **git commit -m "floating points"**,
3. możemy zobaczyć historię naszych commitów przy pomocy następującej komendy: **git reflog**.

Załóżmy, że chcemy się cofnąć do poprzedniego commita, w którym program działał jedynie dla liczb całkowitych, a następnie zmienić jego działanie tak, że tylko wynik byłby liczbą zmiennoprzecinkową.

1. wylistowywujemy historię commitów: **git reflog**,
2. znajdujemy hash interesującego nas commita, do którego chcemy wrócić,
3. resetujemy program do wersji z wybranego commita: **git reset --hard hash-commita**,

W ten sposób możemy rozpocząć pracę na wcześniejszej wersji programu, a następnie zacommitować alternatywne rozwiązanie.

1.1.3 Repozytorium

Repozytorium gita pozwala na przechowywanie historii wersji w chmurze. Najpopularniejszym serwisem świadczącym takie usługi jest **github.com**. W celu skorzystania z githuba niezbędne jest utworzenie konta w serwisie. Następnie w ramach danego projektu należy utworzyć odpowiadające mu repozytorium, z którym następnie można zsynchronizować lokalnego gita.

1.1.4 Repozytorium - przykład

Dotychczasowo Alice rozwijała swoją aplikację na jednym urządzeniu. Ostatnio stwierdziła, że dostęp do projektu na drugim urządzeniu znacznie ułatwił by jej pracę. Decyduje się więc na zastosowanie repozytorium w serwisie github.com. Alice tworzy w serwisie repozytorium dla jej aplikacji, a następnie synchronizuje z nim swojego gita. Od tej pory Alice może w dowolnym momencie umieścić (wypchnąć) jej lokalne zmiany na githuba, a następnie pobrać (sklonować) na drugie urządzenie.

1.1.5 Repozytorium - klucze

Aby móc uwierzytelnić naszego klienta konsolowego w serwisie github, a co za tym idzie uzyskać dostęp do wypychania lokalnych zmian do repozytorium, musimy najpierw skonfigurować klucze dostępu dla danego urządzenia.

1. generujemy klucze w naszym systemie, które posłużą do uwierzytelniania w githubie **ssh-keygen -t rsa -b 4096 -C "adres-email-użyty-na-githubie"**,
2. domyślnie klucze dostępne są w katalogu **/home/nazwa-użytkownika/.ssh**,
3. kopiujemy zawartość klucza publicznego (klucz z rozszerzeniem **.pub**),
4. dodajemy klucz na githubie w zakładce **settings/SSH-and-GPG-keys** jako nowy klucz SSH.

Ponadto po stronie lokalnego gita wymagana będzie autoryzcja użytkownika. W konfiguracji gita należy ustawić nazwę oraz email użytkownika:

- git config --global user.name "IMIE NAZWISKO"
- git config --global user.email "EMAIL"

1.1.6 Repozytorium - w praktyce

Założyliśmy wcześniej konto w serwisie github.com. Teraz chcemy udostępnić w nim nasz projekt.

1. tworzymy nowe repozytorium na githubie, które będzie przeznaczone dla naszej aplikacji,
2. łączymy lokalne repozytorium gita z githubem: **git remote add origin adres-repozytorium**,
3. wypychamy lokalne zmiany do repozytorium: **git push origin nazwa-gałęzi**.

Następnie chcemy sklonować projekt z githuba na innym urządzeniu.

1. wchodzimy do katalogu, w którym chcemy przechowywać nasz projekt,
2. klonujemy projekt za pomocą linku: **git clone link-do-sklonowania-repozytorium**.

W przypadku, gdy wprowadziliśmy zmiany na jedym urządzeniu i chcemy zaktualizować stan projektu na innym urządzeniu, wykonujemy następujące czynności.

1. wchodzimy do katalogu projektu,
2. aktualizujemy gita o informacje odnośnie stanu repozytorium na githubie: **git fetch**,
3. aktualizujemy lokalnego gita: **git pull**.

W przypadku wystąpienia niezgodnych wersji wersjonowania pomiędzy gitem lokalnym oraz githubem, wystąpić mogą tzw. konflikty, czyli miejsca w projekcie, gdzie programista musi zadecydować jaką wersję części aplikacji git ma uznać za poprawną. W zależności od narzędzie konflikty można rozwiązywać na różne sposoby, dlatego przytoczę jedynie scenariusze, w których konieczne może okazać się zastosowanie działań forsownych.

1. Jeśli próba aktualizacji gita lokalnego się nie powiedzie, natomiast interesująca nas wersja programu znajduje się na githubie, najprościej jest po prostu usunąć lokalny katalog z projektem, a następnie ponownie go sklonować.
2. Jeśli próba wypchnięcia zmian na githuba się nie powiedzie, natomiast interesująca nas wersja programu znajduje się na lokalnym gicie, to możemy dokonać wypchnięcia siłowego: **git push --force origin nazwa-gałęzi**.

1.2 Złożoność obliczeniowa

Złożoność obliczeniowa jest miarą wzrostu liczby operacji wykonywanych przez algorytm w miarę wzrostu rozmiaru danych. W praktyce miara ta zakłada jedynie asymptotyczną ocenę wzrostu, pomijając czynniki mniej znaczące lub stałe. Stosowana jest głównie z uwagi na niezależność od środowiska wykonującego algorytm oraz trudności w szacowaniach opartych na fizycznych aspektach obliczeń takich jak częstotliwość cyklów zegarowych lub opóźnienia na poszczególnych komponentach jednostki wykonującej operacje.

1.2.1 Symbol O

Niech $g : R_{\geq 0} \to R_{\geq 0}$ będzie funkcją zmiennej x. Oznaczmy przez $O(g)$ zbiór funkcji $f : R_{\geq 0} \to R$ takich, że dla pewnego $c \in R_{\geq 0}$ oraz $x_0 \in R_{\geq 0}$ mamy $f(x) \leq cg(x)$ dla wszystkich $x \geq x_0$. Równoważnie:

$$f(x) = O(g(x)) \iff \lim_{x\to\infty} \frac{f(x)}{g(x)} = c \geq 0.$$

1.2.2 Symbol O - w praktyce

- $f(x) = 2x + 100 = O(x)$,
- $f(x) = 3x^2 + 16x = O(x^2)$,
- $f(x) = 5xlog_4(x) = O(xlog(x))$,
- $f(x) = \frac{1}{x^2-4} = O(1)$,
- $f(x) = 72 = O(1)$,
- $f(x) = x + log_2(x) + 120log_4(log_2(x)) = O(x)$,
- $f(x) = x^x + x! = O(x^x)$.

1.2.3 Symbol o

Niech $g : R_{\geq 0} \to R_{\geq 0}$ będzie funkcją zmiennej x. Oznaczmy przez $o(g)$ zbiór funkcji $f : R_{\geq 0} \to R$ takich, że dla dowolnego $c \in R_{>0}$ istnieje $x_0 \in R_{\geq 0}$ takie, że $f(x) < cg(x)$ dla wszystkich $x \geq x_0$. Równoważnie:

$$f(x) = o(g(x)) \iff \lim_{x\to\infty} \frac{f(x)}{g(x)} = 0.$$

1.2.4 Symbol o - w praktyce

- $f(x) = 2x + 100 = o(x^2)$,
- $f(x) = 3x^2 + 16x = o(x^8)$,
- $f(x) = 5xlog_4(x) = o(x^2)$,
- $f(x) = \frac{1}{x^2-4} = o(1)$,
- $f(x) = 72 = o(n)$,
- $f(x) = x + log_2(x) + 120log_4(log_2(x)) = o(xlog(x))$,
- $f(x) = x^x + x! = o(x^{x!})$.

1.2.5 Symbol Ω

Niech $g : R_{\geq 0} \to R_{\geq 0}$ będzie funkcją zmiennej x. Oznaczmy przez $\Omega(g)$ zbiór funkcji $f : R_{\geq 0} \to R$ takich, że dla pewnego $c \in R_{\geq 0}$ oraz $x_0 \in R_{\geq 0}$ mamy $g(x) \leq cf(x)$ dla wszystkich $x \geq x_0$. Równoważnie:

$$f(x) = \Omega(g(x)) \iff \lim_{x\to\infty} \frac{f(x)}{g(x)} = c > 0 \vee \lim_{x\to\infty} \frac{f(x)}{g(x)} = \infty.$$

1.2.6 Symbol Ω - w praktyce

- $f(x) = 2x + 100 = \Omega(x)$,
- $f(x) = 3x^2 + 16x = \Omega(x^2)$,
- $f(x) = 5xlog_4(x) = \Omega(log(x))$,
- $f(x) = \frac{1}{x^2-4} = \Omega(\frac{1}{x^{100}})$,
- $f(x) = 72 = \Omega(1)$,
- $f(x) = x + log_2(x) + 120log_4(log_2(x)) = \Omega(1)$,
- $f(x) = x^x + x! = \Omega(x!)$.

1.2.7 Symbol ω

Niech $g : R_{\geq 0} \to R_{\geq 0}$ będzie funkcją zmiennej x. Oznaczmy przez $\omega(g)$ zbiór funkcji $f : R_{\geq 0} \to R$ takich, że dla dowolnego $c \in R_{\geq 0}$ istnieje $x_0 \in R_{\geq 0}$ takie, że $g(x) < cf(x)$ dla wszystkich $x \geq x_0$. Równoważnie:

$$f(x) = \omega(g(x)) \iff \lim_{x\to\infty} \frac{f(x)}{g(x)} = \infty.$$

1.2.8 Symbol ω - w praktyce

- $f(x) = 2x + 100 = \omega(log(x))$,
- $f(x) = 3x^2 + 16x = \omega(x)$,
- $f(x) = 5xlog_4(x) = \omega(log(x))$,
- $f(x) = \frac{1}{x^2-4} = \omega(\frac{1}{x^3})$,
- $f(x) = 72 = \omega(\frac{1}{x})$,
- $f(x) = x + log_2(x) + 120log_4(log_2(x)) = \omega(log(x))$,
- $f(x) = x^x + x! = \omega(x^{100})$.

1.2.9 Symbol Θ

Niech $g : R_{\geq 0} \to R_{\geq 0}$ będzie funkcją zmiennej x. Oznaczmy przez $\Theta(g)$ zbiór funkcji $f : R_{\geq 0} \to R$ takich, że dla pewnych $c_1, c_2 \in R_{\geq 0}$ oraz $x_0 \in R_{\geq 0}$ mamy $c_1 g(x) \leq f(x) \leq c_2 g(x)$ dla wszystkich $x \geq x_0$. Równoważnie:

$$f(x) = \Theta(g(x)) \iff \lim_{x\to\infty} \frac{f(x)}{g(x)} = c > 0.$$

1.2.10 Symbol Θ - w praktyce

- $f(x) = 2x + 100 = \Theta(x)$,
- $f(x) = 3x^2 + 16x = \Theta(x^2)$,
- $f(x) = 5xlog_4(x) = \Theta(xlog(x))$,
- $f(x) = 72 = \Theta(1)$,
- $f(x) = log(log(x)) = \Theta(log(log(x))$,
- $f(x) = 2^{x+1} = \Theta(2^x)$,
- $f(x) = 2^{1000000} = \Theta(1)$.

1.3 Szacowanie wzorcowej złożoności

1.3.1 Koncept

Popularną strategią przydatną w procesie projektowania wzorcowego rozwiązania danego problemu algorytmicznego zadanego na Olimpiadzie Informatycznej jest szacowanie pożądanej złożoności. Każde zadanie posiada jasno sprecyzowane dane wejściowe, a w szczególności ich rozmiar, co może pozwolić na określenie złożoności rozwiązania wzorcowego, poprzez oszacowanie maksymalnej liczby operacji jaką potencjalnie może wykonać program, skonfrontowanie jej z limitami czasowymi i szybkością procesora, a następnie dopasowanie do jednej ze standardowych postaci funkcji wyrażającej złożoność.

Dla przykładu podane są najpopularniejsze złożoności wzorcowych rozwiązań zadań z Olimpiady Informatycznej (podane w kolejności rosnącej):

- $O(n)$
- $O(nlog(n))$
- $O(n\sqrt(n))$
- $O(n^2)$
- $O(n^3)$
- $O(2^n)$

1.3.2 Szacowanie

Biorąc pod uwagę specyfikację procesorów używanych do wykonywania zadań na środowiskach olimpiady w celu ich testowania (taktowanie 2GHz), Możemy założyć, że w przeciągu jednej sekundy będzie on w stanie wykonać około 2 miliardy poleceń. Z uwagi na architekturę procesorów, w szczególności fakt, że instrukcje znane nam z języka C++ często składają się z wielu podstawowych instrukcji procesora, bezpieczniej jest przyjąć, że procesor będzie w stanie przetworzyć parędziesiąt milionów instrukcji języka C++ na sekundę. Idąc dalej, jeśli chcielibyśmy zastosować powyższe szacowanie w kontekście złożoności obliczeniowej, zależnej od wielkości danych wejściowych, trzeba wziąć pod uwagę wszelakie stałe, które pojęcie złożoności pomija, tak więc ostatecznym szacowaniem, które należałoby zastosować jest kilka milionów operacji na sekundę.

1.3.3 Przykład

Dla przykładu weźmy pod uwagę zadanie, w którym wielkość danych określona jest przez $n \leq 200000$, natomiast algorytm powinien się wykonywać co najwyżej 5 sekund. Opierając się na powyższych założeniach, możemy więc przyjąć, że w przeciągu 5 sekund będzie on w stanie wykonać około kilkanaście milionów operacji. Spróbujmy dopasować liczbę operacji w przypadku największej możliwej wartości n ($n = 200000$) do jednej z typowych funkcji wyrażających złożoność:

- $O(n^2)$: $200000^2 = 4 * 10^{10}$, co znacznie wychodzi poza zakres kilkunastu milionów.
- $O(n\sqrt{n})$: $200000 * \sqrt{200000} \approx 450 * 200000 = 9 * 10^7$, co daje wynik o niecały rząd wielkości za duży.
- $O(nlog(n))$: $200000 * log(200000) \approx 15 * 200000 = 3 * 10^6$, co mieści się w naszych szacowaniach.
- $O(n)$, równe po prostu 200000 jest niedoszacowaniem o około 2 rzędy wielkości.

Po powyższej analizie można wywnioskować, że docelową złożonością rozwiązania, której prawdopodobnie spodziewają się autorzy zadania jest złożoność $O(nlog(n))$.

1.3.4 W praktyce

W praktyce ograniczenia czasowe zadań zazwyczaj mieszczą się w przedziale od jednej do kilkunastu sekund. Oznacza to, że dla poszczególnych rozmiarów danych wejściowych możemy przyporządkować złożoności, które z dużym prawdopodobieństwem mogą okazać się złożonościami docelowymi:

- $[1 - 10]$: $O(n!)$
- $[10 - 15]$: $O(3^n)$ / $O(2^n)$
- $[15 - 20]$: $O(2^n)$
- $[20 - 100]$: $O(n^4)$
- $[100 - 300]$: $O(n^3)$
- $[300 - 1000]$: $O(n^2log(n))$
- $[1000 - 5000]$: $O(n^2)$
- $[5000 - 50000]$: $O(n\sqrt{n})$ / $O(nlog^2(n))$
- $[50000 - 500000]$: $O(nlog(n))$
- $[500000 - 1000000]$: $O(nlog(n))$ / $O(n)$
- $[1000000+]$: $O(n)$

Warto zaznaczyć, że jeśli rozwiązanie posiada nienaturanie niskie albo wysokie stałe, to powyższe szacowania mogą okazać się błędne. Należy więc polegać przede wszystkim na swojej własnej intuicji.