Teoretyczne podstawy informatyki

Wykład 12b:
Metody programistyczne
O złożoności obliczeniowej raz jeszcze

Metody programistyczne

Wyidealizowany scenariusz tworzenia oprogramowania

1. **Definicja problemu i specyfikacja**
   Analiza wymagań użytkownika (często nieprecyzyjne i trudne do zapisania), budowa prostego prototypu lub modelu systemu
2. **Projekt**
   Wyróżniamy najważniejsze komponenty, specyfikujemy wymagania związane z wydajnością systemu, szczegółowe specyfikacje niektórych komponentów
3. **Implementacja**
   Każdy implementowany komponent poddajemy serii testów
4. **Integracja i testowanie systemu**
5. **Instalacja i testowanie przez użytkowników**
6. **Konserwacja**
   Niezwykle istotna jakość stylu programowania (w wielu wypadkach to ponad 50% nakładów poniesionych na napisanie systemu)

Metody programistyczne

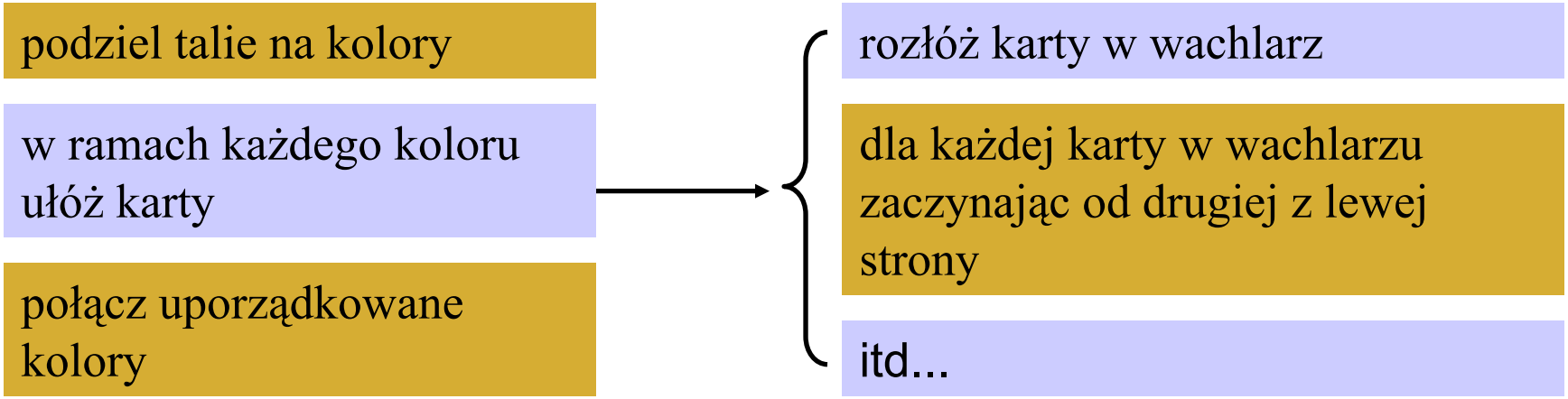
- Przy rozwiązywaniu prostych zadań **podejście „ad hoc”** może dawać szybsze rezultaty. Jednak, gdy mamy do czynienia z bardziej złożonymi problemami efektywniejsze jest **podejście systematyczne**
- **Projektowanie zstępujące (top-down design)**
  - Rozpoczynamy od zdefiniowania problemu który chcemy rozwiązać
  - Problem dzielimy na główne kroki – pod-problemy
  - Pod-problemy są dzielone na drobniejsze kroki tak długo, aż rozwiązania drobnych pod-problemów stają się łatwe .... nazywamy to stopniowym uszczegółowianiem
  - W idealnej sytuacji pod-problemy mogą być rozwiązywane niezależnie, a ich rozwiązania łączone w celu otrzymania rozwiązania całego problemu. W odniesieniu do tworzenia kodu oznacza to, że poszczególne kroki powinny być kodowane niezależnie, przy czym dane wyjściowe jednego kroku są używane jako dane wejściowe do innego
- **Realnie**
  - celem projektowania zstępującego jest **zminimalizowanie tych współzależności**

Metody programistyczne

- Zalety projektowania zstępującego
  - Jest to systematyczna metoda rozwiązywania problemów
    - rozwiązanie jest **modularne**, poszczególne kroki mogą być uruchamiane, modyfikowane i ulepszane niezależnie od pozostałych, rozwiązanie składa się z klarownych fragmentów które można niezależnie zrozumieć
    - dobrze zaprojektowane fragmenty mogą być **ponownie zastosowane** w innych zadaniach
    - można rozpoznać wspólne problemy na samym początku i **uniknąć wielokrotnego ich rozwiązywania**

Metody programistyczne

Przykład: Uporządkuj talie kart:



Metody programistyczne

- **Projektowanie wstępujące** (bottom-up design)
  - Polega na wyjściu od samego języka i wzbogacaniu go nowymi operacjami, dopóki nie będzie można wyrazić rozwiązania problemu w rozszerzonym języku
  - Każde oprogramowanie działające na maszynie cyfrowej rozszerza jej możliwości o nowe funkcje. W efekcie, uruchomienie programu na komputerze tworzy nową maszynę, która nie wykonuje swoich operacji bezpośrednio, ale zmienia je na prostsze wykonywalne przez sprzęt
  - Nową maszynę nazywamy **maszyną wirtualną**, ponieważ istnieje w świecie abstrakcyjnym a nie fizycznym
  - To samo dotyczy sytuacji gdy dodajemy nową operację do języka programowania, pisząc funkcje. Zwiększa ona jego funkcjonalność tworząc nowy, silniejszy język. O zbiorze nowych funkcji można myśleć jako o wirtualnej maszynie zbudowanej na bazie starego języka.

Metody programistyczne

- Przykład: Policz wyrażenie (3/28 + 2/7) * 4/3 –1
  - Aby to wykonać należałoby wzbogacić istniejący język przez dodanie
    - funkcji implementujących działania +, -, * , / na ułamkach,
    - funkcji wczytywania i wypisywania ułamków,
    - ten pakiet funkcji mógłby też zawierać funkcję skracającą ułamki, chociaż nasz program nie musiałby z niej korzystać.
  - Zazwyczaj programy projektuje się **łącząc metodę wstępująca i zstępująca**
    - zaczynamy od podzielenia problemu na pod-zadania,
    - przekonujemy się, że byłby przydatny określony pakiet funkcji,
    - rozstrzygamy, jakie funkcje wejdą w skład pakietu i rozszerzą język,
    - powtarzamy to iteracyjnie, dzieląc problemy na prostsze i wzbogacają tym samym język.... dopóki rozwiązania wszystkich problemów składowych nie dadzą się zapisać bezpośrednio w rozszerzonym języku.

Metody programistyczne

- Abstrakcyjne typy danych (ATD)
  - Jest to istotne ulepszenie metody projektowania programów.
  - Podstawą metody jest **oddzielenie operacji wykonywanych** na danych i sposobów ich przechowywania **od konkretnego typu danych**. Pozwala ona na podzielenie zadania programistycznego na dwie części:
    - wybór struktury danych reprezentującej ATD i napisanie funkcji implementujących operacje
    - napisanie „programu głównego" który będzie wywoływał funkcje ATD
- Program ma dostęp do danych ATD wyłącznie przez wywołanie funkcji; nie może bezpośrednio odczytywać ani modyfikować wartości przechowywanych w wewnętrznych strukturach ATD.

Weryfikacja poprawności programu

- Sprawdzenie częściowej poprawności.
  - Weryfikacja opiera się o sprawdzenie specyfikacji, która jest czymś niezależnym od kodu programu.
  - Specyfikacja wyraża, co program ma robić, i określa związek miedzy jego danymi wejściowymi i wyjściowymi.
  - W specyfikacji definiujemy warunek dotyczący danych wejściowych, który musi być spełniony na początku programu, tzw. **warunek wstępny**. Jeżeli dane nie spełniają tego warunku, to nie ma gwarancji że program będzie działał poprawnie ani nawet że się zatrzyma.
  - W specyfikacji definiujemy również **warunek końcowy**, czyli co oblicza program przy założeniu że się zatrzyma. Warunek końcowy jest zawsze prawdziwy jeżeli zachodzi warunek wstępny.

Weryfikacja poprawności programu

- Aby wykazać, że program jest zgodny ze specyfikacją, trzeba podzielić ją na wiele kroków, podobnie jak dzieli się na kroki sam program. Każdy krok programu ma swój warunek wstępny i warunek końcowy.
- Przykład: prosta instrukcja przypisania
  - x=v, x – zmienna, v – wyrażenie
    - warunek wstępny: $x \geq 0$
    - wyrażenie: $x = x+1$
    - warunek końcowy: $x \geq 1$
- Dowodzimy poprawności specyfikacji przez podstawienie wyrażenia **x+1** pod każde wystąpienie **x** w warunku końcowym.
- Otrzymujemy formułę **$x+1 \geq 1$**, która wynika z warunku wstępnego **$x \geq 0$**.
- Zatem specyfikacja tej instrukcji podstawienia jest poprawna.

Weryfikacja poprawności programu

- Istnieją **cztery sposoby** łączenia prostych kroków w celu otrzymania kroków bardziej złożonych.
  1. stosowanie instrukcji złożonych (bloków instrukcji wykonywanych sekwencyjnie)
  2. stosowanie instrukcji wyboru
  3. stosowanie instrukcji powtarzania (pętli)
  4. wywołania funkcji

Weryfikacja poprawności programu

- Każdej z tych metod budowania kodu odpowiada metoda przekształcania warunku wstępnego i końcowego w celu wykazania poprawności specyfikacji kroku złożonego:
  - Aby wykazać prawdziwość specyfikacji instrukcji wyboru, należy dowieść, że warunek końcowy wynika z warunku wstępnego i warunku testu w każdym z przypadków instrukcji wyboru.
  - Funkcje też mają swoje warunki wstępne i warunki końcowe; musimy sprawdzić że warunek wstępny funkcji wynika z warunku wstępnego kroku wywołania, a warunek końcowy pociąga za sobą warunek końcowy kroku wywołania.
  - Wykazanie poprawności programowania dla pętli wymaga użycia **niezmiennika pętli**.

Niezmienniki pętli

- Niezmiennik pętli określa warunki jakie muszą być zawsze spełnione przez zmienne w pętli, a także przez wartości wczytane lub wypisane (jeżeli takie operacje zawiera).
- Warunki te muszą być prawdziwe przed pierwszym wykonaniem pętli oraz po każdym jej obrocie.
  (mogą stać się chwilowo fałszywe w trakcie wykonywania wnętrza pętli, ale musza ponownie stać się prawdziwe przy końcu każdej iteracji)
- Dowodzimy że pewne zdanie jest niezmiennikiem pętli wykorzystując **zasadę indukcji matematycznej**.
- Mówi ona że:
  - jeśli (1) pewne zdanie jest prawdziwe dla $n = 0$
  - oraz (2) z tego, że jest prawdziwe dla pewnej liczby $n \geq 0$ wynika że musi być prawdziwe dla liczby $n+1$,
  - to (3) zdanie to jest prawdziwe dla wszystkich liczb nieujemnych.

Dowodzenie niezmienników pętli

- Stwierdzenie jest prawdziwe przed pierwszym wykonaniem pętli, ale po dokonaniu całej inicjacji; wynika ono z warunku wstępnego pętli.
- Jeśli założymy, że stwierdzenie jest prawdziwe przed jakimś przebiegiem pętli i że pętla zostanie wykonana ponownie, a więc warunek pętli jest spełniony, to stwierdzenie będzie nadal prawdziwe po kolejnym wykonaniu wnętrza pętli.

- Proste pętle maja zazwyczaj proste niezmienniki.
- Pętle dzielimy na trzy kategorie
  - pętla z wartownikiem: czyta i przetwarza dane aż do momentu napotkania niedozwolonego elementu,
  - pętla z licznikiem: zawczasu wiadomo ile razy pętla będzie wykonana,
  - pętle ogólne: wszystkie inne.

Problem „STOP-u”

- Aby udowodnić że program się zatrzyma, musimy wykazać, że zakończą działanie wszystkie pętle programu i wszystkie wywołania funkcji rekurencyjnych.
  - Dla pętli z wartownikiem wymaga to umieszczenia w warunku wstępnym informacji, że dane wejściowe zawierają wartownika.
  - Dla pętli z licznikiem wymaga to dodefiniowania granicy dolnej (górnej) tego licznika.

Uwagi końcowe

- Istnieje wiele innych elementów, które muszą być brane pod uwagę przy dowodzeniu poprawności programu:
  - możliwość przepełnienia wartości całkowitoliczbowych,
  - możliwość przepełnienia lub niedomiar dla liczb zmiennopozycyjnych,
  - możliwość przekroczenia zakresów tablic,
  - prawidłowość otwierania i zamykania plików,
  - itd.

Uwagi końcowe

Na wybór najlepszego algorytmu dla tworzonego programu wpływa wiele czynników, najważniejsze to:

- prostota,
- łatwość implementacji
- efektywność

Złożoność obliczeniowa raz jeszcze

Złożoność zamortyzowana

- W wielu sytuacjach na strukturach danych działają nie pojedyncze operacje ale ich sekwencje.
- Jedna z operacji takiej sekwencji może wpływać na dane w sposób powodujący modyfikacje czasu wykonania innej operacji.

  Jednym ze sposobów określania czasu wykonania w przypadku pesymistycznym dla całej sekwencji jest dodanie składników odpowiadających wykonywaniu poszczególnych operacji.
- Jednak wynik tak uzyskany może być zbyt duży w stosunku do rzeczywistego czasu wykonania. Analiza amortyzacji pozwala znaleźć bliższą rzeczywistej złożoność średnią.

Złożoność zamortyzowana

- Analiza z amortyzacją polega na analizowaniu kosztów operacji, zaś pojedyncze operacje są analizowane właśnie jako elementy tego ciągu. Koszt wykonania operacji w sekwencji może być różny niż w przypadku pojedynczej operacji, ale ważna jest też częstość wykonywania operacji.
- Jeśli dana jest sekwencja operacji op_1, op_2, op_3, … to analiza złożoności pesymistycznej daje daje złożoność obliczeniowa równa:

$$C(op_1, op_2, op_3, \dots) = C_{pes}(op_1) + C_{pes}(op_2) + C_{pes}(op_3) + \dots$$

dla złożoności średniej uzyskujemy

$$C(op_1, op_2, op_3, \dots) = C_{śre}(op_1) + C_{śre}(op_2) + C_{śre}(op_3) + \dots$$

- Nie jest analizowana kolejność operacji, "sekwencja" to po prostu "zbiór" operacji.

Złożoność zamortyzowana

- Przy analizie z amortyzacją zmienia się sposób patrzenia, gdyż sprawdza się co się stało w danym momencie sekwencji i dopiero potem wyznacza się złożoność następnej operacji:

  $\mathbf{C(op_1, op_2, op_3, \ldots) = C(op_1) + C(op_2) + C(op_3) + \ldots}$

  gdzie **C** może być złożonością optymistyczna, średnią, pesymistyczna lub jeszcze inną - w zależności od tego co działo się wcześniej.
- Znajdowanie złożoności zamortyzowanej tą metoda może być zanadto skomplikowane.
  Znajomość natury poszczególnych procesów oraz możliwych zmian struktur danych używane są do określenia funkcji **C**, którą można zastosować do każdej operacji w sekwencji. Funkcja jest tak wybierana aby szybkie operacje były traktowane jak wolniejsze niż w rzeczywistości, zaś wolne jako szybsze.
- Sztuka robienia analizy amortyzacji polega na znalezieniu funkcji **C**; takiej która dociąży tanie operacje dostatecznie aby pokryć niedoszacowanie operacji kosztownych.

Przykład

- **Przyklad:**
  - dodawanie elementu do wektora zaimplementowanego jako elastyczna tablica
- **Przypadek optymistyczny:** wielkość wektora jest mniejsza od jego pojemności, dodanie elementu ogranicza się do wstawienia go do pierwszej wolnej komórki.

  Koszt dodania nowego elementu to ***O*(1)**.
- **Przypadek pesymistyczny:** rozmiar jest równy pojemności, nie ma miejsca na nowe elementy. Konieczne jest zaalokowanie nowego obszaru pamięci, skopiowanie do niego dotychczasowych elementów i dopiero dodanie nowego.

  Koszt wynosi wówczas ***O*(rozmiar (wektor))**.

Pojawia się nowy parametr, bo pojemność można zwiększać o więcej niż jedną komórkę wtedy przepełnienie pojawia się tylko "od czasu do czasu".

Przykład

- **Analiza z amortyzacją:** badane jest jaka jest oczekiwana wydajność szeregu kolejnych wstawień. Wiadomo, że we przypadku optymistycznym jest to ***O*(1)**, a w przypadku pesymistycznym ***O*(rozmiar)**, ale przypadek pesymistyczny zdarza się rzadko.
- Należy przyjąć pewna hipotezę:
  a) kosztAmort(push(x)) = 1
  niczego nie zyskujemy, łatwe wstawienia nie wymagają poprawek, nie pojawia się jednak zapas na kosztowne wstawienia
  b) kosztAmort(push(x)) = 2
  zyskujemy zapas na łatwych wstawieniach, ale czy wystarczający…? Zależy to od rozmiaru wektora…

Przykład dla założenia że koszt amortyzowany to 2. Operacje prawie cały czas są "na minusie" co jest niedopuszczalne

Rozmiar	Pojemność	Koszt Amortyzowany	Koszt	Zapas
0	0			
1	1	2	0+1	1
2	2	2	1+1	1
3	4	2	2+1	0
4	4	2	1	1
5	8	2	4+1	-2
6	8	2	1	-1
7	8	2	1	0
8	8	2	1	1
9	16	2	8+1	-6
10	16	2	1	-5
…	…	…	…	…
16	16	2	1	1
17	32	2	16+1	-14
18	32	2	1	-13

Przykład dla założenia że koszt amortyzowany to 3.

Rozmiar	Pojemność	Koszt Amortyzowany	Koszt	Zapas
0	0			
1	1	3	0+1	2
2	2	3	1+1	3
3	4	3	2+1	3
4	4	3	1	5
5	8	3	4+1	3
6	8	3	1	5
7	8	3	1	7
…	…	…	…	…
16	16	3	1	17
17	32	3	16+1	3
18	32	3	1	5

c) kosztAmort(push(x)) = 3

- Nigdy nie pojawia się "debet", zaoszczędzone jednostki są niemal w całości zużywane gdy pojawi się kosztowna operacja.

Złożoność zamortyzowana

- W przedstawionym przykładzie wybór funkcji stałej był słuszny, ale zwykle tak nie jest.
- Niech funkcja przypisująca liczbę do konkretnego stanu struktury danych **ds** będzie nazywana **funkcją potencjału**. Koszt amortyzowany definiuje się następująco:

$$\mathbf{koszAmort}(op_i) = \mathbf{koszt}(op_i) + \mathbf{f.potencjału}(ds_i) - \mathbf{f.potencjału}(ds_{i-1})$$

- Jest to faktyczny koszt wykonania operacji op_i powiększony o zmianę potencjału struktury danych **ds** po wykonaniu tej operacji.
  Definicja ta obowiązuje dla pojedynczej operacji.

$$\mathbf{koszAmort}(op_1, op_2, op_3, \ldots, op_m) =$$
$$\sum_{i=1}^{m} (\mathbf{koszt}(op_i) + \mathbf{f.potencjału}(ds_i) - \mathbf{f.potencjału}(ds_{i-1}))$$

- W większości przypadków funkcja potencjału początkowo jest zerem, nigdzie nie jest ujemna, tak że czas amortyzowany stanowi kres górny czasu rzeczywistego.

Kontynuacja przykładu

- f.potencjału (vector$_i$) =

  = 0 (jeśli rozmiar$_i$ = pojemność$_i$ czyli vector jest pełny)

  = 2 rozmiar$_i$ - pojemność$_i$ (w każdym innym przypadku)

- Można sprawdzić że przy tak zdefiniowanej **f.potencjału, koszAmort(op$_i$)** jest faktycznie równy **3** w każdej konfiguracji (tanie wstawianie, kosztowne, tanie po kosztownym)

Struktury danych i algorytmy obróbki danych zewnętrznych

- Podstawowy czynnik rzutujący na różnice między obróbką danych wewnętrznych (czyli przechowywanych w pamięci operacyjnej) a obróbką danych zewnętrznych (czyli przechowywanych w pamięciach masowych) jest specyfika dostępu do informacji.
- Mechaniczna struktura dysków sprawia, że korzystnie jest odczytywać dane nie pojedynczymi bajtami, lecz w większych **blokach**. Zawartość pliku dyskowego można traktować jako listę łączoną poszczególnych bloków, bądź też jako drzewo którego liście reprezentują właściwe dane, a węzły zawierają informację pomocniczą ułatwiającą zarządzanie tymi danymi.

Struktury danych i algorytmy obróbki danych zewnętrznych

- Załóżmy że:

  adres bloku = 4 bajty

  długość bloku = 4096 bajtów
- Czyli w jednym bloku można zapamiętać adresy do 1024 innych bloków.
- Czyli informacja pomocnicza do 4 194 304 bajtów będzie zajmować 1 blok.
- Możemy też budować strukturę wielopoziomową, w strukturze dwupoziomowej blok najwyższy zawiera adresy do 1024 bloków pośrednich, z których każdy zawiera adresy do 1024 bloków danych. Maksymalna wielkość pliku w tej strukturze 1024 * 1024 * 1024 = 4 294 967 296 bajtów = 4GB, informacja pomocnicza zajmuje 1025 bloków.

Struktury danych i algorytmy obróbki danych zewnętrznych

- Nieodłącznym elementem współpracy pamięci zewnętrznej z pamięcią operacyjną są **bufory**, czyli zarezerwowany fragment pamięci operacyjnej, w której system operacyjny umieszcza odczytany z dysku blok danych lub z którego pobiera blok danych do zapisania na dysku.
- **Miara kosztu dla operacji na danych zewnętrznych.**
  - Głównym składnikiem czasu jest czekanie na pojawienie się właściwego sektora pod głowicami. To może być nawet kilkanaście milisekund… Co jest ogromnie długo dla procesora taktowanego kilku-gigahercowym zegarem.
  - Zatem "merytoryczna jakość" algorytmu operującego na danych zewnętrznych będzie zależna od liczby wykonywanych **dostępów blokowych** (odczyt lub zapis pojedynczego bloku nazywamy dostępem blokowym).

Sortowanie

- **Sortowanie zewnętrzne** to sortowanie danych przechowywanych na plikach zewnętrznych.
- Sortowanie przez łączenie pozwoli na posortowanie pliku zawierającego **n** rekordów, przeglądając go jedynie ***O*(log n)** razy.
- Wykorzystanie pewnych mechanizmów systemu operacyjnego – dokonywanie odczytów i zapisów we właściwych momentach – może znacząco usprawnić sortowanie dzięki zrównoleglowieniu obliczeń z transmisją danych.

Sortowanie przez łączenie

- Polega na organizowaniu sortowanego pliku w pewna liczbę serii, czyli uporządkowanych ciągów rekordów. W kolejnych przebiegach rozmiary serii wzrastają a ich liczba maleje, ostatecznie (posortowany) plik staje się pojedynczą serią.
- Podstawowym krokiem sortowania przez łączenie dwóch plików, f_1 i f_2, jest zorganizowanie tych plików w serie o długości **k**, tak że:
  - liczby serii w plikach f_1, f_2, z uwzględnieniem "ogonów" różnią się co najwyżej o jeden
  - co najwyżej w jednym z plików f_1, f_2, może się znajdować ogon
  - plik zawierający "ogon" ma poza nim co najmniej tyle serii ile jego partner.
- Prosty proces polega na odczytywaniu po jednej serii (o długości **k**) z plików f_1, f_2, łączenia tych serii w dwukrotnie dłuższą i zapisywania tak połączonych serii na przemian do plików g_1, g_2.
- Całkowita liczba dostępów blokowych w całym procesie sortowania jest rzędu $O((n \log n)/b)$ gdzie **b** jest liczba rekordów w jednym bloku.

Przykład

Pliki oryginalne:	28 3 93 10 54 65 30 90 10 69 8 22 31 5 96 40 85 9 39 13 8 77 10
Pliki zorganizowane w serie o długości 2:	28 31 93 96 54 85 30 39 8 10 8 10 3 5 10 40 9 65 13 90 69 77 22
Pliki zorganizowane w serie o długości 4	3 5 28 31 9 54 65 85 8 10 69 77 10 40 93 96 13 30 39 90 8 10 22
Pliki zorganizowane w serie o długości 8:	3 5 10 28 31 40 93 96 8 8 10 10 22 69 77 9 13 30 39 54 65 85 90
Pliki zorganizowane w serie o długości 16:	3 5 9 10 13 28 30 31 39 40 54 65 85 90 93 96 8 8 10 10 22 69 77
Pliki zorganizowane w serie o długości 32:	3 5 8 8 9 10 10 10 13 22 28 30 31 39 40 54 65 69 77 85 90 93 96

Podsumowanie

- *„Nie przejmuj się efektywnością algorytmu… wystarczy poczekać kilka lat.”*
- Taki pogląd funkcjonuje w środowisku programistów, nie określono przecież granicy rozwoju mocy obliczeniowych komputerów. Nie należy się jednak z nim zgadzać w ogólności. Należy zdecydowanie przeciwstawiać się przekonaniu o tym, ze ulepszenia sprzętowe uczynią prace nad efektywnymi algorytmami zbyteczna.
- Istnieją problemy których rozwiązanie za pomocą zasobów komputerowych jest teoretycznie możliwe, ale praktycznie przekracza możliwości istniejących technologii. Przykładem takie problemu jest rozumienie języka naturalnego, przetwarzanie obrazów (do pewnego stopnia oczywiście) czy “inteligentna” komunikacja.
- Pomiędzy komputerami a ludźmi na rozmaitych poziomach.
- Kiedy pewne problemy staja się “proste”… Nowa grupa wyzwań, które na razie można sobie tylko próbować wyobrażać, wytyczy nowe granice możliwości wykorzystania komputerów.