Ogólne zasady projektowania algorytmów i programowania

Pracuj nad właściwie sformułowanym problemem

– dokładna analiza nawet małego zadania może prowadzić do ogromnych korzyści praktycznych: skrócenia długości kodu, czasu programowania, czasu wykonania programu
– definicja zadania zajmuje w skrajnym przypadku nawet 90% ogólnego czasu produkcji programu!
– problem często jest psychologiczny a nie techniczny – zleceniodawca, zasugerowany przez jakieś czynniki, często próbuje narzucić programiście niewłaściwe rozwiązanie
– etap ten jest źródłem największej liczby błędów, szczególnie gdy dodatkowe życzenia klienta (zleceniodawcy) są zgłaszane zbyt późno

Zbadaj przestrzeń rozwiązań

– programista musi być nieco leniwy: nigdy nie powinien zabierać się do programowania pierwszego pomysłu, który wpadnie mu do głowy (ale z drugiej strony musi być gotowy do ukończenia zadania w terminie!)
– problem, który wydaje się trudny, może mieć proste, nieoczekiwane rozwiązanie
– przykłady:

- wyszukiwanie binarne zamiast liniowego: zyskujemy na koszcie obliczeniowym z O(n) do O(log n). Ograniczenie – metoda działa na zbiorach posortowanych!

- szukanie największej sumy spójnego fragmentu wektora: różne metody, od działającej w czasie sześciennym do liniowego (patrz plik `w6a.pdf`)

- zbiór anagramów: permutacje liter w wyrazie? Bardzo nieefektywne: n! operacji; porównywanie par wyrazów – n wyrazów x n porównań – co najmniej kilka godzin pracy; alternatywa: posortowanie alfabetyczne liter w wyrazach (patrz plik `w6b.pdf`)
– rozpatruj rozwiązania niekoniecznie związane z koniecznością programowania – np. wykorzystanie arkusza kalkulacyjnego lub bazy danych + kilka makr
– rozpatruj rozwiązania „pozainformatyczne” – np. przesyłanie danych

Przyjrzyj się danym

– stosuj odpowiednie do zadania struktury danych, uzyskując: przyspieszenie działania programu, oszczędność pamięci, łatwiejszą konserwację w przyszłości
– przekształć powtarzający się kod w tablice, przykład: zmienne `->` tablica

– ukrywaj skomplikowane struktury: definiuj je jako abstrakcyjny typ i określ wykonywane na nich operacje w ramach klasy
– pozwalaj danym określać strukturę programu, przykład: korespondencja seryjna

Korzystaj z oszacowań

– szacuj wstępnie czas wykonania zadania oraz zajętość pamięci, aby odrzucić nierealne rozwiązania
– stosuj współczynniki bezpieczeństwa

Pisz poprawne programy

– programiści są optymistami: częsta praktyka (niestety): „napisz swój kawałek kodu, podaj dalej i módl się żeby zadziałał, a błędami niech zajmie się »Sekcja Jakości Kodu i Testowania«”

– stosuj dowodzenie poprawności kodu oraz jego testowanie
– stosuj asercje: precyzyjne wyrażenie zależności między danymi wejściowymi, zmiennymi w programie i danymi wyjściowymi
– wykorzystuj asercje do dowodzenia poprawności instrukcji sekwencyjnych, instrukcji warunkowych, pętli, funkcji
– błędy najczęściej pojawiają się w łatwiejszych częściach kodu, do trudniejszych części programiści bardziej się przykładają

Używaj gotowych komponentów / bibliotek / funkcji systemowych

– stosuj biblioteki / programy dostarczane przez system
– nie odkrywajmy od nowa Ameryki

Buduj prototypy

– pozwalają np. na wybranie najlepszego rozwiązania z kilku dobrych

– można go szybko przekazać do testowania, podczas gdy można dalej pracować np. nad interfejsem

Jeżeli trzeba, idź na ustępstwa

– stosuj kompromis między czasem wykonania programu a rozmiarem wymaganej przez ten program pamięci
– stosuj kompromis pomiędzy stopniem optymalizacji kodu a jego przejrzystością
– przykład z wyliczaniem silni – algorytm siłowy kontra stablicowanie danych
– analogia do wszelkich innych problemów inżynierskich: samolot, zużycie paliwa a przyspieszenie
– ale bez przesady! Kompromis to obustronne ustępstwa!

Nie komplikuj

– Antoine de Saint-Exupery: Konstruktor wie, że osiągnął doskonałość nie wtedy, gdy do jego konstrukcji nic nie można dodać, ale wtedy, gdy nic z niej nie można ująć
– Einstein: rozwiązuj problemy w tak prosty sposób jak to tylko możliwe, ale nie prościej!
– prosty kod jest bezpieczniejszy, odporniejszy i efektywniejszy niż bardziej skomplikowany odpowiednik
– najtańszymi, najszybszymi, najdokładniejszymi i najsolidniejszymi elementami systemu komputerowego są te, których w nim nie ma
– nie pisz dużego programu tam, gdzie wystarczy mały
– ogólniejszy problem może być prostszy
– optymalizuj kod (zarówno pod względem czasu wykonania jak i użycia pamięci) tylko wtedy, jeżeli konieczne trzeba to zrobić, nie optymalizuj za wcześnie

– efektywność staraj się zapewnić na najwyższym poziomie: definicji problemu, struktury systemu, stosowanych algorytmów i doborze struktur danych

Pisz przejrzysty kod

– stosuj formatowanie kodu źródłowego

– pracuj na różnych poziomach abstrakcji (odróżniaj *co* robi projektowany element od tego *jak* to robi)