Metody algorytmiczne

Wygląda na to, że możemy teraz szczęśliwie kontynuować nasze algorytmiczne obowiązki. Wiemy, jak zbudowane są algorytmy i jak rozmieścić obiekty, którymi manipulują, a także wiemy, jak zapisać je do wykonania przez komputer. Dzięki temu możemy powiedzieć naszemu procesorowi, co powinien robić i kiedy. Jest to jednak zbyt naiwna ocena sytuacji i w miarę postępów zobaczymy różne przyczyny tego stanu rzeczy. Jeden z problemów polega na tym, że nie udostępniliśmy żadnych metod, które można wykorzystać do opracowania algorytmu. Bardzo dobrze mówi się o konstrukcjach, których algorytm może używać – to znaczy o kawałkach, z których może się składać – ale musimy powiedzieć coś więcej o sposobach wykorzystania tych kawałków do stworzenia całości. W tej części dokonamy przeglądu szeregu dość ogólnych metod algorytmicznych, które projektant może zastosować w celu znalezienia rozwiązania problemu algorytmicznego. Trzeba jednak zaznaczyć, że nie ma dobrych przepisów na wymyślanie receptur. Każdy problem algorytmiczny jest wyzwaniem dla projektanta algorytmu. Niektóre problemy są proste, inne skomplikowane, inne kuszące; niniejszy rozdział pokazuje tylko, że pewne algorytmy całkiem dobrze podążają za pewnymi ogólnymi paradygmatami. Morał, że projektant algorytmu może odnieść korzyść z przyjrzenia się im najpierw, próbując sprawdzić, czy można je wykorzystać lub przystosować do użycia w danej sytuacji. Ogólnie rzecz biorąc, projektowanie algorytmiczne to twórcza działalność, która może wymagać prawdziwej pomysłowości, ale która z pewnością może skorzystać na opanowaniu dostępnego zestawu technik i metod.

Wyszukiwania i przechodzenia

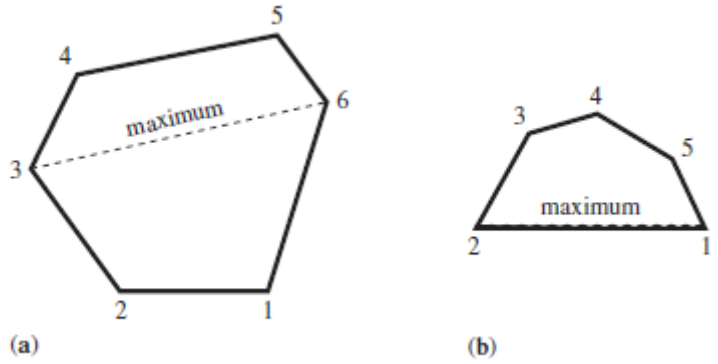
Wiele problemów algorytmicznych rodzi potrzebę przemierzania pewnych struktur. Czasami struktura, którą musimy przejść, jest wyraźnie obecna jako jedna ze struktur danych zdefiniowanych w algorytmie, ale czasami jest to jakaś niejawna abstrakcyjna struktura, której być może nie można faktycznie „zobaczyć", ale która istnieje pod powierzchnią. Czasami szuka się czegoś specjalnego w strukturze („kto jest dyrektorem działu public relations?"), a czasami trzeba wykonać jakąś pracę w każdym punkcie („oblicz średnie oceny wszystkich uczniów") . Na przykład prosty problem sumowania wynagrodzeń, o którym mowa w Części 1, wymaga prostego przejrzenia podanej listy pracowników. Z drugiej strony problem, który dotyczył tylko pracowników zarabiających więcej niż ich menedżerowie, można uznać za wymagający przechodzenia przez wyimaginowaną dwuwymiarową tablicę, w której pracownicy są wykreślani względem pracowników, a wyszukiwanie dotyczy określonego pracownika. /manager par. W takich przypadkach zadaniem jest znalezienie najbardziej naturalnego sposobu przechodzenia przez strukturę danych (czy to jawną, czy niejawną), a tym samym opracowanie algorytmu. Gdy zaangażowane są wektory lub tablice, zwykle pojawiają się pętle i pętle zagnieżdżone, jak wyjaśniono w Części 2, i w tym samym duchu, gdy zaangażowane są drzewa, pojawia się rekurencja, jak to miało miejsce w przykładzie sortowania drzew. Prawdą jest, że idea algorytmu sortowania drzew jest dość subtelna i nie można jej znaleźć po prostu wymyślając najlepszą strukturę kontrolną do przechodzenia przez daną strukturę danych. Jednak gdy już wpadnie na pomysł, drobiazgiem jest uświadomienie sobie, że druga wizyta, w której elementy są wyprowadzane w kolejności, jest niczym innym jak pewnym przechodzeniem drzewa binarnego, z którego nie jest zbyt trudno dojść do wniosku, że należy stosować rekurencję. Przechodzenie wywołane rekurencyjnym przejściem w jest czasami nazywane przeszukiwaniem w głąb ze śledzeniem wstecznym, ponieważ procesor „nurkuje" w drzewo, próbując zejść jak najgłębiej, a kiedy nie może przejść dalej, cofa się niechętnie, zawsze dążąc do wznowienia nurkowania. Jedyną dodatkową cechą jest tutaj wymóg, aby nurkowanie odbywało się maksymalnie w lewo. Istnieje kilka innych sposobów przechodzenia przez drzewa, z których jeden, podwójny do wyszukiwania w głąb, nazywa się wyszukiwaniem wszerz. Trawersowanie w kierunku wszerz oznacza, że poziomy drzewa są kolejno wyczerpane; najpierw korzeń, potem całe jego potomstwo, potem ich potomstwo i tak dalej.

Wyczerpujące poszukiwania, czyli procedura British Museum

Kiedy potrzebujesz znaleźć coś w strukturze danych, możesz po prostu zbadać wszystkie jej elementy jeden po drugim, aż znajdziesz to, czego szukasz. To właśnie zrobiliśmy, gdy szukaliśmy pracowników zarabiających więcej niż ich menedżerowie. Ten prosty pomysł nazywa się wyszukiwaniem wyczerpującym lub, bardziej barwnie, procedurą British Museum. Ten ostatni termin odnosi się do sposobu, w jaki poważnie myśląca osoba przegląda każdy eksponat w muzeum. Wyszukiwanie wyczerpujące jest czasem jedynym sposobem rozwiązania problemu algorytmicznego. Jednak często są znacznie lepsze sposoby. Na przykład, jeśli chciałbyś znaleźć numer w książce telefonicznej za pomocą wyszukiwania wyczerpującego, musiałbyś przejrzeć każde nazwisko w książce, aż znajdziesz to, którego szukasz (lub odkryjesz, że go tam nie ma). To może zająć dużo czasu. Zamiast tego szacujesz z grubsza, gdzie otworzyć książkę, na podstawie pierwszej litery imienia, a następnie szybko korygujesz swoje oszacowanie na podstawie nazwisk na otwieranej stronie i znajdujesz numer, którego szukasz w mniej niż minutę. Ta tak zwana procedura wyszukiwania interpolacyjnego, która jest znacznie bardziej wydajna niż wyczerpujące wyszukiwanie tego problemu, jest zorientowaną na człowieka wersją algorytmu. Dlatego w wielu przypadkach wyczerpujące poszukiwania nie są co najmniej najlepszym sposobem. Mimo to jest to przydatna procedura w przypadkach, w których nie ma innych rozwiązań, a także służy jako punkt odniesienia, z którym można porównać inne rozwiązania.

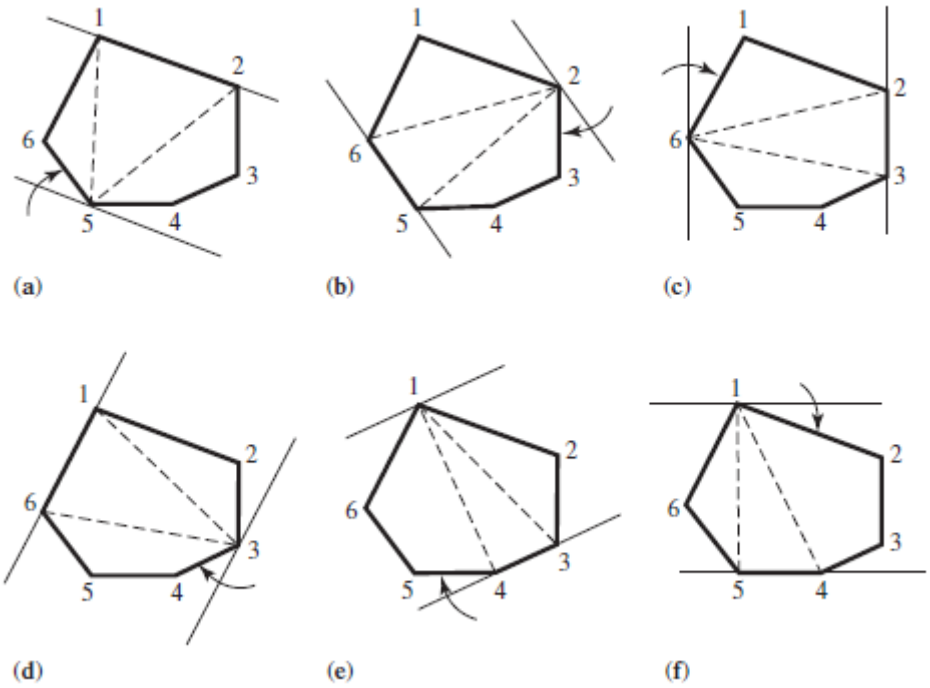
Maksymalna odległość wielokąta: przykład

Wiele interesujących problemów algorytmicznych wiąże się z pojęciami geometrycznymi, takimi jak punkty, linie i odległości, a zatem jest częścią przedmiotu znanego jako geometria obliczeniowa. Wiele problemów z tego obszaru jest zwodniczo łatwych do „rozwiązania” za pomocą ludzkiego układu wzrokowego, ale często prawdziwym wyzwaniem dla projektantów algorytmów. Oto bardzo prosty. Załóżmy, że otrzymaliśmy prosty wielokąt wypukły1, taki jak na rysunku



, i załóżmy, że interesuje nas wskazanie dwóch punktów maksymalnej odległości na jego granicy. Zakłada się, że wielokąt jest reprezentowany przez sekwencję współrzędnych wierzchołków, w kolejności zgodnej z ruchem wskazówek zegara. Ponieważ maksymalna odległość wyraźnie wystąpi dla dwóch wierzchołków (dlaczego?), nie ma potrzeby patrzenia na żadne punkty wzdłuż krawędzi wielokąta inne niż wierzchołki. Trywialne rozwiązanie polegałoby na rozważeniu wszystkich par wierzchołków w pewnej kolejności, utrzymywaniu bieżącego maksimum i pary osiągającej to maksimum w zmiennych. Każda nowa rozważana para jest poddawana prostemu obliczaniu odległości; nowa odległość jest porównywana z bieżącym maksimum, a zmienne są aktualizowane, jeśli nowa odległość okaże się większa, co oznacza, że w rzeczywistości jest to nowe maksimum. Jeśli przez chwilę zastanowimy się nad tym rozwiązaniem, stanie się jasne, że w rzeczywistości przemierzamy wyimaginowaną tablicę, w której wierzchołki są wykreślane względem wierzchołków. Wyimaginowany

element danych w punkcie <I, J > tej tablicy to odległość między wierzchołkami I i J . Przechodzenie można wtedy łatwo przeprowadzić za pomocą dwóch zagnieżdżonych pętli i rzeczywiście zachęcamy do zapisania wynikowego algorytmu. To rozwiązanie uwzględnia jednak zbyt wiele potencjalnych par. Czy nie powinno być możliwe uwzględnienie tylko „przeciwnych" par punktów, takich jak <1, 4>, <2, 5> i <3, > na rysunku 1(a)? Oznaczałoby to przechodzenie tylko przez wektor „specjalnych" par, a nie przez tablicę wszystkich par, co wyraźnie dałoby bardziej wydajny algorytm, który wymaga tylko jednej pętli. To nie jest tak proste, jak się wydaje, ponieważ pożądane przeciwne pary niekoniecznie są tymi, które mają równą liczbę wierzchołków po obu stronach; Rysunek 1(b) pokazuje wielokąt, w którym maksimum występuje w sąsiednich wierzchołkach, co zostałoby pominięte przez algorytm uwzględniający tylko przeciwnie ponumerowane. Lepsze rozwiązanie, które faktycznie wykorzystuje pojedynczą pętlę i uwzględnia tylko „właściwy" rodzaj przeciwstawnych par, pokazano na rysunku 2



Opiszmy, jak to działa. Zrobimy to za pomocą nieformalnych pojęć geometrycznych, chociaż szczegółowy opis algorytmu musiałby przełożyć je na manipulacje numeryczne za pomocą struktur danych i struktur kontrolnych, których tutaj nie będziemy przeprowadzać. Najpierw rysowana jest linia wzdłuż krawędzi między wierzchołkami 1 i 2. Następnie linia równoległa do niej jest stopniowo przesuwana w kierunku wielokąta spoza wielokąta po przeciwnej stronie pierwszej linii, aż do trafienia w jeden z wierzchołków; patrz Rysunek 2(a), z którego jasno wynika, że wierzchołek 5 jest pierwszym, do którego należy w ten sposób dotrzeć. Początkowe przybliżenie do maksimum jest teraz uważane za większą odległość między tym wierzchołkiem (w tym przypadku 5) a wierzchołkami 1 i 2. Następnie rozpoczyna się ruch zgodnie z ruchem wskazówek zegara, którego każdy krok obejmuje:

1. obracanie jednej z tych dwóch linii, aż znajdzie się wzdłuż następnej krawędzi wielokąta w kolejności zgodnej z ruchem wskazówek zegara (na rysunku 2(b) można zobaczyć, że dolna linia obraca się, aby dopasować się do krawędzi od 5 do 6); oraz

2. ustawienie drugiej linii, aby była do niej równoległa (na rysunku 2(b) górna linia jest regulowana).

Dokładnie, która z linii jest obrócona, a która jest dostosowana, określa się przez porównanie wysiłków potrzebnych do obrotu: linia o mniejszym kącie do następnej krawędzi to ta obrócona (na rysunku 2(a)

kąt między dolną linią a krawędź od 5 do jest mniejsza niż między linią górną a krawędzią od 2 do 3). Po zakończeniu rotacji na właśnie obróconej linii pojawia się nowy wierzchołek, którego nie było przed obrotem. Odległość między tym nowym wierzchołkiem a tym na dopasowanej linii jest obliczana i porównywana z bieżącym maksimum, jak poprzednio. Ta procedura jest wykonywana w pełnym okręgu wokół całego wielokąta. Po zakończeniu procedury aktualne maksimum jest żądaną maksymalną odległością. Można wykazać, że wszystkie działania wymagane przez ten algorytm polegają na prostych manipulacjach numerycznych na współrzędnych wierzchołków, które wynikają z elementarnej geometrii analitycznej. Rysunek 2 ilustruje kolejność przekształceń na liniach. Ten przykład został wybrany, aby dodatkowo zilustrować, że rozpoznanie potrzeby przechodzenia i ustalenie, co tak naprawdę ma zostać przebyte, jest ważne i może być bardzo pomocne, ale nie zawsze wystarcza, jeśli chodzi o rozwiązywanie skomplikowanych problemów algorytmicznych; pewien wgląd i duża wiedza na dany temat nie może zaszkodzić.

Dziel i rządź

Często problem można rozwiązać, sprowadzając go do mniejszych problemów tego samego rodzaju i rozwiązując je, a następnie, przy dodatkowej pracy, łącząc częściowe rozwiązania w celu uzyskania ostatecznego rozwiązania pierwotnego problemu. Jeśli mniejsze problemy są dokładnie tym samym problemem, który mamy pod ręką, ale odnoszą się do „mniejszych" lub „prostszych" danych wejściowych, wówczas w algorytmie można zastosować rekurencję. Ta metoda z oczywistych powodów nazywa się dziel i zwyciężaj. Kilka algorytmów w tej książce urzeczywistnia tę ideę „podziel i uderz". Widzieliśmy to już pośrednio w przykładzie z Wież Hanoi: algorytm rozwiązał problem dla pierścieni N, rozwiązując dwa zadania dla pierścieni N−1 we właściwej kolejności i o odpowiednich parametrach. Inne przypadki pojawiają się później. Oto dwa dodatkowe zastosowania dzielenia i zdobywania. Wyobraź sobie, że dostajesz pomieszaną książkę telefoniczną lub, żeby zabrzmieć głębiej, nieuporządkowaną listę L. Nie jesteśmy zainteresowani sortowaniem L, ale jedynie znajdowaniem największych i najmniejszych elementów, które się w niej pojawiają. Oczywiście możemy po prostu przejrzeć listę raz, zachowując bieżące minimum i bieżące maksimum w zmiennych, porównując każdy element z obydwoma w miarę postępu i aktualizując je, jeśli rozważany element jest mniejszy niż bieżące minimum lub większy niż bieżące maksimum. Jednak poniższy algorytm w prosty sposób wykorzystuje strategię dziel i zwyciężaj i, jak wyjaśniono w rozdziale 6, jest w rzeczywistości nieco lepszy. Schematycznie brzmi:

(1) jeśli L składa się z jednego elementu, to ten element jest traktowany zarówno jako minimum, jak i maksimum; jeśli składa się z dwóch elementów, to mniejszy jest przyjmowany za minimum, a większy za maksimum;

(2) w przeciwnym razie wykonaj następujące czynności:

(2.1) podzielić L na dwie połowy, Lleft i Lright;

(2.2) znajdź ich ekstremalne elementy MINleft, MAXleft, MINright i MAXright;

(2.3) wybierz mniejszy z MINleft i MINright; jest to minimalny element L;

(2.4) wybierz większy z MAXleft i MAXright; jest to maksymalny element L.

(Oczywiście podział w linii (2.1) należy zdefiniować w taki sposób, aby objąć przypadek listy L o nieparzystej długości, powiedzmy, przyjmując, że pierwsza połowa jest dłuższa od drugiej o jeden element.) Teraz, wiersz (2.2) błaga o wykonanie rekurencyjnie, ponieważ problemy do rozwiązania to właśnie problem min&max na mniejszych listach Lleft i Lright. Ta rekurencja nie jest tak prosta, jak się wydaje, ponieważ tutaj, w przeciwieństwie do procedury Wieże Hanoi, wywołanie rekurencyjne musi

dawać wyniki, które są używane w sequelu. W jakiś sposób procesor musi nie tylko zapamiętać swój adres zwrotny i sposób przywrócenia środowiska do stanu sprzed rozpoczęcia obecnego rekursywnego przedsięwzięcia, ale musi także być w stanie przywrócić pewne elementy ze swoich trudów. W takim przypadku najbardziej pomocne byłoby, gdyby procesor mógł wrócić z wywołania rekurencyjnego wraz z minimum i maksimum, które zostało wysłane do obliczeń. Poniższe jest wynikiem odpowiedniego rozszerzenia pojęcia podprogramu i zastosowania go do problemu min&max: subroute find-min&max-of L:

(1) jeśli L składa się z jednego elementu, to ustaw dla niego MIN i MAX; jeśli składa się z dwóch elementów, to ustaw MIN na mniejszy z nich i MAX na większy;

(2) w przeciwnym razie wykonaj następujące czynności:

(2.1) podzielić L na dwie połowy, Lleft i Lright;

(2.2) wywołaj find-min&max-of Lleft, umieszczając zwrócone wartości w MINleft i MAXleft;

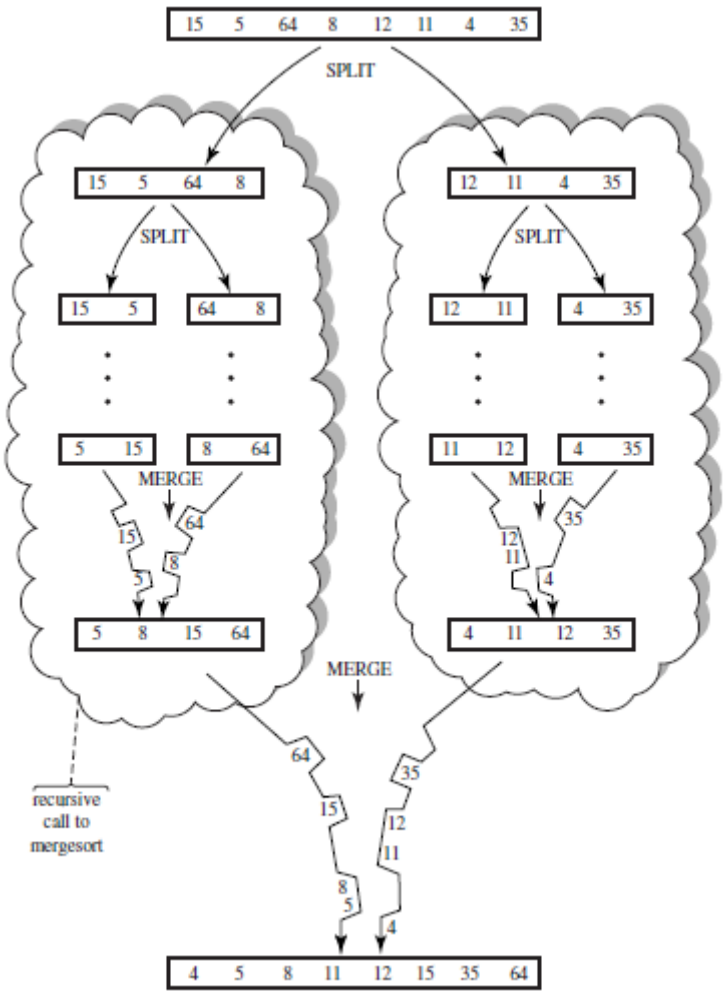
(2.3) wywołaj find- min&max-of Lright, umieszczając zwrócone wartości w MINright i MAXright;

(2.4) ustaw MIN na mniejsze z MINleft i MINright;

(2.5) ustaw MAX na większy z MAXleft i MAXright;

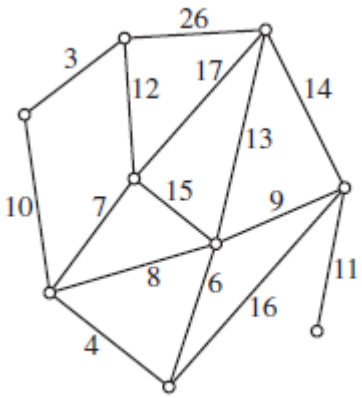
(3) powrót z MIN i MAX.

Paradygmat dziel i zwyciężaj może być z pożytkiem wykorzystany do sortowania listy, a nie tylko do znajdowania jej skrajnych elementów. Oto jak. Aby posortować listę L zawierającą co najmniej dwa elementy, podobnie dzielimy ją na połówki, Lleft i Lright, i sortujemy rekurencyjnie oba. Przypadek jednoelementowy jest traktowany oddzielnie, jak w przykładzie min&max. Aby uzyskać ostateczną posortowaną wersję L, łączymy posortowane połówki w jedną posortowaną listę. Aby scalić dwie posortowane listy, wielokrotnie usuwamy i wysyłamy do wyjścia mniejszy z dwóch elementów znajdujących się obecnie na początku dwóch list. Działanie tego algorytmu, zwanego sortowaniem przez scalanie, zilustrowano na rysunku i zachęcamy do szczegółowego zapisania algorytmu.

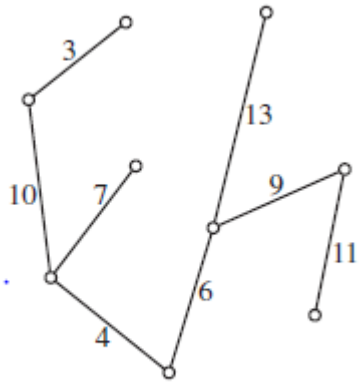


Chciwe algorytmy i wykonawcy kolei

Wiele problemów algorytmicznych wymaga uzyskania jakiegoś najlepszego wyniku z odpowiedniego zestawu możliwości. Weźmy pod uwagę sieć miast i leniwego wykonawcę kolei. Kontrahentowi zapłacono za ułożenie torów tak, aby można było dojechać do każdego miasta z każdego innego. Umowa nie określała jednak żadnych kryteriów, takich jak konieczność zapewnienia niektórych połączeń kolejowych bez międzylądowań, czy maksymalna liczba dozwolonych miast na ścieżce łączącej dowolne dwa inne. Dlatego nasz kontrahent, będąc leniwym, jest zainteresowany ułożeniem najtańszej (czyli najkrótszej) kombinacji odcinków szyny. Załóżmy, że nie wszystkie miasta można połączyć bezpośrednimi odcinkami kolei ze wszystkimi innymi z przyczyn obiektywnych, takich jak przeszkody fizyczne, oraz że odległości są podane tylko między tymi parami miast, które można połączyć. Zakładamy ponadto, że koszt bezpośredniego połączenia miasta A z B jest proporcjonalny do odległości między nimi. Nie dopuszczamy również węzłów kolejowych poza miastami. Taka sieć nazywana jest grafem oznaczonym lub w skrócie grafem. Wykresy są podobne do drzew, z tym wyjątkiem, że drzewa nie mogą się „zamykać”; oznacza to, że nie mogą zawierać cykli ani pętli, podczas gdy wykresy mogą. Rysunek przedstawia przykład grafu miasta i jego minimalnej linii kolejowej.

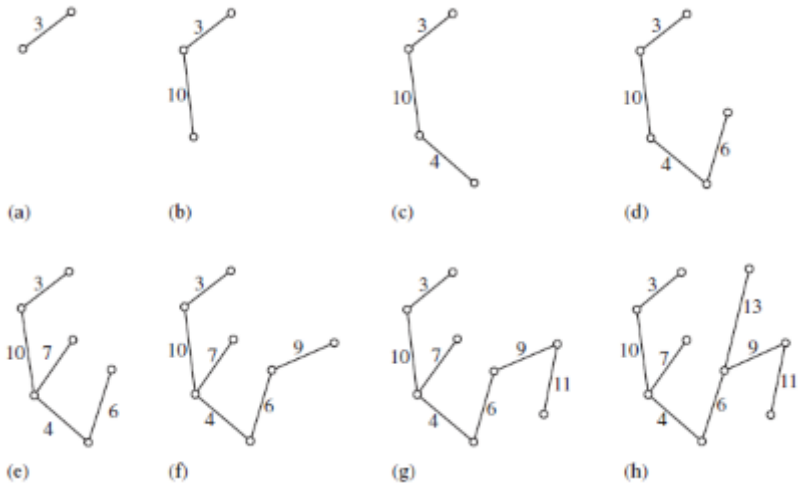


(Not drawn to scale)



Total cost: 63

Zauważ, że wykonawca naprawdę dąży do tego, co czasami nazywamy minimalnym drzewem opinającym. Jest to drzewo, które „rozpina” graf w tym sensie, że dociera do każdego z jego węzłów (czyli w naszym przypadku miast) i jest najtańszym takim drzewem w tym sensie, że suma etykiety wzdłuż krawędzi (czyli w naszym przypadku odległości między miastami) są najmniejsze z możliwych. Łatwo zauważyć, że pożądanym rozwiązaniem musi być drzewo (to znaczy nie może zawierać cykli), ponieważ gdyby miało zawierać jakiś cykl, leniwy wykonawca mógłby uzyskać tańszą linię kolejową, która nadal łączyła wszystkie miasta, eliminując jeden z segmentów tego cyklu. Istnieje algorytmiczne podejście do takich problemów, zwane metodą zachłanną. Zaleca konstruowanie minimalnej krawędzi drzewa rozpinającego po krawędzi, wybierając jako następną krawędź najtańszą z możliwych w obecnej sytuacji. To tak, jakby przyjąć postawę „jedz i pij, bo jutro umrzemy”: rób tyle, ile możesz teraz, bo inaczej możesz żałować, że tego nie zrobiłeś. Rysunek ilustruje budowę takiego drzewa.

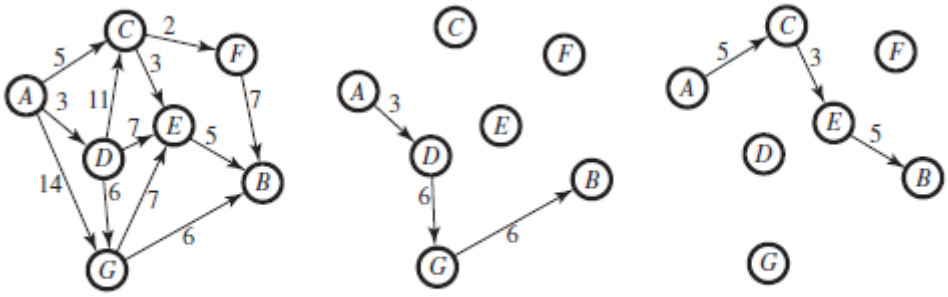


Zacznij od zdegenerowanego drzewa składającego się z najtańszej krawędzi na wykresie, zupełnie samej. Teraz na każdym etapie rozszerz zbudowane do tej pory drzewo, dodając najtańszą krawędź, która nie została jeszcze uwzględniona, o ile spowoduje to powstanie połączonej struktury, która w rzeczywistości jest drzewem. W szczególności nie powinna wprowadzać cyklu; jeśli tak, przejdź do następnej najtańszej krawędzi. Na przykład przejście z rysunku (e) do (f) polega na dodaniu krawędzi oznaczonej 9 zamiast krawędzi oznaczonej 8. Ta ostatnia wprowadziłaby cykl na wykresie. Można wykazać, że ta prosta strategia faktycznie tworzy minimalne drzewo opinające, i zachęcamy do szczegółowego zapisania algorytmu. Algorytmy zachłanne istnieją dla różnych interesujących problemów algorytmicznych. Zwykle są one dość łatwe do zdobycia, niektórych przypadkach są dość

intuicyjne. Najtrudniejszą częścią jest zwykle pokazanie, że strategia chciwości rzeczywiście daje najlepsze rozwiązanie, a jak pokazuje następna sekcja, są przypadki, w których chciwość w ogóle nie popłaca.

Dynamiczne programowanie i zmęczeni podróżnicy

Oto problem, który jest podobny do problemu z minimalnym drzewem opinającym, ale ten przeciwstawia się naiwnym zachłannym rozwiązaniom. To również obejmuje sieć miejską, ale zamiast leniwego wykonawcy kolei mamy zmęczonego podróżnika, który jest zainteresowany podróżowaniem z jednego miasta do drugiego. Chociaż obaj mają zadanie do wykonania i obaj chcą zminimalizować całkowity koszt jej wykonania, istnieje zasadnicza różnica: podczas gdy wykonawca musi połączyć wszystkie miasta za pomocą podsieci szyn, podróżny zazwyczaj podróżuje tylko przez niektóre z miasta. Jest zatem jasne, że podróżnik nie szuka minimalnego drzewa opinającego, ale minimalną ścieżkę; czyli najtańsza podróż prowadząca z miasta startowego do wybranego celu. Dla ułatwienia prezentacji zakładamy, że wszystkie linie na wykresie miasta są skierowane, co oznacza, że jeśli dwa miasta są połączone linią na wykresie, to ta linia reprezentuje połączenie w jedną stronę. Zakładamy również, że graf jest połączony, co oznacza, że graf nie składa się z oddzielnych, rozłącznych części. Dalej zakładamy, że graf miasta nie ma cykli, więc naprawdę zmęczony podróżnik nie będzie podatny na kręcenie się w kółko, ponieważ nie będzie w nim nikogo. Taki kompleks nazywa się skierowanym grafem acyklicznym lub DAG w skrócie. I tak otrzymujemy DAG, a podróżnik jest zainteresowany przejściem z punktu A do punktu B. Chciwe podejście do problemu może spowodować, że ścieżka zostanie zbudowana, zaczynając od A i stale dodając do bieżącej niepełnej ścieżki najtańsza krawędź prowadząca z miasta dotychczas osiągniętego do jakiegoś jeszcze nieodwiedzonego miasta, aż do osiągnięcia docelowego miasta B. Rysunek pokazuje przykład zastosowania tego naturalnie wyglądającego algorytmu do grafu, którego minimalna ścieżka od A do B ma długość 13.



Algorytm znajduje ścieżkę o długości 15, która nie jest tak dobra. Chciwość nie popłaca, ponieważ sprytny algorytm musi być na tyle sprytny, aby wziąć krawędź długości 5 do C, a następnie długość 3 do E, mimo że nie są to lokalnie najlepiej wyglądające opcje. Inna, nie zachłanna metoda algorytmiczna, zwana planowaniem dynamicznym, umożliwia dokonywanie tak subtelnych wyborów. (Właściwie ta metoda nazywana jest programowaniem dynamicznym, a nie planowaniem, ale zdecydowaliśmy się użyć tego drugiego słowa, aby lepiej oddać ideę metody i uniknąć konfliktu z algorytmicznym użyciem terminu „programowanie”, który odnosi się do kodowania algorytmu do wykonania przez komputer, jak wyjaśniono w Części 3. Dynamiczne planowanie opiera się na doprecyzowaniu dość prymitywnego kryterium natychmiastowej chciwości i można je abstrakcyjnie opisać w następujący sposób. Załóżmy, że rozwiązanie jakiegoś problemu algorytmicznego składa się z sekwencji wyborów, które mają doprowadzić do jakiegoś optymalnego rozwiązania. Jak już pokazano, jest całkiem możliwe, że samo

wybranie najlepiej wyglądającego wyboru spośród każdego lokalnego zestawu możliwości nie doprowadzi do optymalnego rozwiązania. Jednak często zdarza się, że optimum można uzyskać, rozpatrując wszystkie kombinacje (a) dokonania jednego wyboru oraz (b) znalezienia optymalnego rozwiązania mniejszego problemu reprezentowanego przez pozostałe wybory. Na przykład na rysunku długość najkrótszej drogi z A do B jest najmniejszą z trzech liczb uzyskanych przez wybranie najpierw jednego z miast C, G i D (tych bezpośrednio osiągalnych z A), a następnie dodanie jego odległości z A do długość najkrótszej drogi prowadzącej z niej do B. Symbolicznie, oznaczając długość najkrótszej drogi z X do B przez L(X), możemy napisać:

L(A) = minimum: 5 + L(C), 14 + L(G), 3 + L(D)

Innymi słowy, znajdujemy najkrótszą drogę z A do B, znajdując trzy „prostsze" najkrótsze ścieżki (te z każdego z C, G i D do B), łącząc ich rozwiązania z bezpośrednią krawędzią z A, a następnie wybierając najlepszy z trzech wyników. Oznacza to, że możemy znaleźć optymalne rozwiązanie, znajdując najpierw optymalne rozwiązania trzech „mniejszych" problemów, a następnie dokonując kilku uzupełnień i porównań. Proces ten można następnie kontynuować, pisząc na przykład:

L(D) = minimum: 7 + L(E), 6 + L(G), 11 + L(C)

Gdy takie wyprowadzenia dają klauzule określające L(B) (czyli minimalną odległość od B do siebie) nie są potrzebne dalsze prace, ponieważ nawet wyczerpany podróżnik wie, że L(B) wynosi po prostu 0, ponieważ najlepszym sposobem na przejście z punktu B do punktu B jest pozostanie tam, gdzie jesteś. Te obserwacje prowadzą do dynamicznego algorytmu planowania dla ogólnego problemu zmęczonego podróżnika (czasami zwanego problemem najkrótszej ścieżki), który działa od punktu końcowego B wstecz do A. W przykładzie z rysunku ?? najpierw oblicza najkrótsze ścieżki z F i E do B, czyli L(F) i L(E) (są to jedyne miasta, które prowadzą donikąd poza B). Następnie oblicza L(G) i L(C) (tu G i C są jedynymi miastami, które prowadzą tylko do B, F i E, czyli do miast już rozpatrzonych), potem L(D), a na końcu L (A). Każde obliczenie jest przeprowadzane na podstawie wyników już dostępnych, tak aby np. L(G) było minimum 6 (bezpośrednia odległość od G do B) i 7 + L(E). Wykonując tę procedurę, śledzimy również ścieżkę wsteczną, która jest stopniowo konstruowana od B do A; jest to poszukiwana optymalna ścieżka. Planowanie dynamiczne można traktować jako „dziel i zwyciężaj" doprowadzone do granic możliwości: wszystkie podproblemy są rozwiązywane w kolejności rosnącej, a wyniki są przechowywane w pewnej strukturze danych, aby ułatwić łatwe rozwiązania większych. Metodę można zastosować do wielu bardziej skomplikowanych problemów, które do przechowywania częściowych rozwiązań wymagają struktur danych bardziej złożonych niż zwykłe wektory. Możesz spróbować opracować algorytm dynamicznego planowania dla ściśle powiązanego problemu „podróżującego sprzedawcy"

Mnóstwo pracy i terminowa praca

Wiele algorytmów opiera się na sprytnym doborze struktur danych o właściwościach dostosowanych do potrzeb konkretnego algorytmu. Na przykład algorytm sortowania drzewa opisany w rozdziale 2 wykorzystuje drzewa wyszukiwania binarnego, które ograniczają umieszczanie elementów w taki sposób, że mniejsze elementy trafiają na lewo, a większe na prawo. Istnieje wiele rodzajów struktur danych, niektóre o ciekawych nazwach, takich jak „stosy Fibonacciego" lub „czerwono-czarne drzewa", z których każda nadaje się do określonego celu algorytmicznego. Załóżmy, że potrzebujemy algorytmu, który odbiera elementy w określonej kolejności i musi być w stanie dostarczyć najmniejszy z nich przy każdym zapytaniu. Rozważmy na przykład harmonogram drukarni. W każdej chwili klient może przyjechać z jakimś drukiem. Każda praca ma swój własny termin; niektóre są pilne, a inne mogą poczekać. Kiedy kserokopiarka staje się dostępna, kierownik sklepu musi znaleźć najpilniejszą pracę do wykonania. Tutaj elementami, które algorytm musi obsłużyć, są zadania drukowania uporządkowane

według ich terminów. Odpowiednią strukturą danych dla tego problemu jest sterta, która jest drzewem binarnym z tą właściwością, że wartość każdego węzła jest mniejsza niż wartości wszystkich jego potomków. Dzięki tej właściwości najmniejszy element sterty zawsze znajduje się w korzeniu drzewa i jest natychmiast dostępny. Kiedy rzeczywiście uzyskujemy dostęp do tego elementu, musimy go usunąć ze sterty (jak w przykładzie z drukarnią). Ale robiąc to, musimy zachować charakterystyczną właściwość sterty, zastępując ten najmniejszy element mniejszym z jego potomstwa, które właśnie stało się najmniejszym elementem sterty. Jeśli jednak po prostu przesuniemy ten nowy minimalny element do korzenia, stworzymy „dziurę" w drzewie, którą musi wypełnić mniejsze z jego potomstwa. To rzeczywiście się dzieje, a proces jest kontynuowany w dół, aż dotrzemy do liścia. Tyle o usunięciu najmniejszego elementu. Wkładanie nowego elementu do pryzmy jest podobne, z tą różnicą, że zaczyna się on od jednego z liści i przesuwa się w kierunku korzenia, aż do znalezienia właściwego miejsca (czyli do momentu, gdy wejście w górę naruszyłoby charakterystyczną właściwość hałdy). Skuteczna implementacja stert wykorzystuje wektor. Sterta zawierająca N elementów zajmie komórki od 1 do N wektora, przy czym dwa potomstwo węzła znajdującego się w komórce I znajduje się w komórkach 2 · I i 2 · I + 1. W tej reprezentacji , usunięcie elementu minimum nie może być wykonane w sposób opisany powyżej, ponieważ może to spowodować powstanie „dziury" w środku wektora. Zamiast tego element z ostatniej pozycji w wektorze zastępuje pierwszy element (ten właśnie usunięty), a następnie jest „bąbelkowany" w dół, aż znajdzie się we właściwym miejscu w stosie. Ta reprezentacja wektorowa ma pewne właściwości, które sprawiają, że algorytmy manipulacji stertą są dość wydajne

Niedestrukcyjne algorytmy

Inny elegancki przykład użycia struktur danych pochodzi z paradygmatu programowania funkcyjnego, omówionego w Części 3. Paradygmat ten sprawia, że programy są łatwiejsze do zrozumienia i zrozumienia, za cenę wyższego poziomu abstrakcji, który używa tylko niemodyfikowalnych obiektów. Zazwyczaj algorytmy są opisywane w trybie imperatywnym, przy użyciu struktur danych, które algorytm może modyfikować w miarę postępu. Z drugiej strony algorytmy funkcjonalne muszą traktować swoje struktury danych z większym szacunkiem, ponieważ nie można ich modyfikować. W tym widoku zamiast zmieniać struktury danych, zawsze tworzymy nowe. Na przykład operacja dodania elementu do stosu zwraca nowy stos, zawierający wszystkie elementy oryginalnego stosu oraz nowy element na górze. Pierwotny stos nie jest zmieniany i w razie potrzeby jest dostępny do dalszych obliczeń. Ponieważ do stosów można uzyskać dostęp tylko z jednego końca, dość łatwo jest zaimplementować stosy za pomocą połączonych list. Dodanie elementu do stosu to po prostu dodanie elementu na początek listy (a dokładniej stworzenie kolejnego linku, który wskazuje na oryginalną listę, która, jak wspomniano, nie ulega zmianie). Podobnie usunięcie górnego elementu stosu oznacza po prostu przejście do następnego elementu na liście . W przeciwieństwie do stosów kolejki są trudniejsze do zaimplementowania w języku funkcjonalnym, ponieważ umożliwiają dostęp (a zatem wymagają modyfikacji) z obu stron. Usunięcie elementu jest łatwe i odbywa się tak, jak w przypadku stosu. Jednak dodanie elementu oznacza dodanie go na końcu listy, jak w przykładzie JAVA, który widzieliśmy w Części 3. W języku imperatywnym można to łatwo zrobić, modyfikując wskaźnik „następny" ostatniego łącza na liście, ale jest to niemożliwe w języku funkcjonalnym. Oczywiście moglibyśmy skopiować całą listę i dodać nowy element na końcu nowej listy, ale byłaby to strata czasu i pamięci. Sprytny pomysł pozwala nam na realizację kolejek funkcjonalnych z taką samą wydajnością jak przy implementacji imperatywnej. Sztuczka polega na zaimplementowaniu każdej kolejki jako dwóch stosów. Stos „front" zawiera elementy z przodu kolejki, uporządkowane tak, że górny element stosu jest elementem przednim kolejki, a stos „back" zawiera elementy z tyłu kolejki, w odwrotnej kolejności order - górny element stosu będący ostatnim elementem kolejki . Dodanie elementu z tyłu kolejki jest teraz łatwe: jest umieszczany na tylnym stosie. Usuwanie elementu z kolejki jest równie proste: wystarczy usunąć

element znajdujący się na górze przedniego stosu. A co, jeśli przedni stos jest pusty, a tylny nie? W tym przypadku najpierw zamieniamy stos tylny w stos przedni, przesuwając wszystkie jego elementy do stosu przedniego w odwrotnej kolejności. Tylny stos staje się teraz pusty, co jest w porządku, ponieważ nigdy nie musimy niczego z niego usuwać. (W rzeczywistości nigdy nie pozwalamy, aby stos przedni stał się pusty, gdy stos tylny jest niepusty, wykonując to odwrócenie, gdy ostatni element jest usuwany ze stosu przedniego). tylny stos. Zauważ jednak, że każdy element kolejki przejdzie z tylnego stosu do przedniego stosu co najwyżej raz. Kosztem tej operacji można zatem „obciążyć" przenoszony element, tak że po zsumowaniu i uśrednieniu z pełnego wykonania algorytmu, każda operacja wstawiania lub usuwania zajmuje określoną ilość czasu. Ta metoda księgowa, zwana kosztem zamortyzowanym, jest ważną techniką analizy wydajności algorytmów

Algorytmy on-line

Załóżmy, że niektórzy rodzice po raz pierwszy w życiu zabierają swoje dzieci do ośrodka narciarskiego. Nie da się powiedzieć, jak bardzo polubią jeździć na nartach, a na ile będą chcieli je uprawiać. Narty można wypożyczyć lub kupić. Jeśli okaże się, że dzieci będą chciały dużo jeździć na nartach, taniej byłoby raz na zawsze kupić narty. Jeśli jednak nie, taniej byłoby po prostu wypożyczyć narty, kiedy tylko masz na to ochotę. Gdyby rodzice z góry wiedzieli, ile dzieci będą chciały jeździć na nartach, wybór byłby oczywisty. Jednak informacje te są nieznane i pozostaje nam przyjęcie pewnej strategii ogólnej formy, która wymaga rozpoczęcia od wypożyczenia nart kilka razy, aby zobaczyć, jak się sprawy mają, a następnie podjęcia decyzji o faktycznym zakupie. Jaka byłaby najlepsza strategia rodziców? Rozwiązanie tego jest przykładem interesującej klasy problemów, których rozwiązanie wymaga tak zwanych algorytmów on-line. Nazwa pochodzi od faktu, że te algorytmy muszą podejmować decyzje na bieżąco, nie znając wszystkich istotnych informacji; konkretnie, nie wiedzą, jakie wnioski mogą zostać złożone w ich sprawie w przyszłości. Pierwszym pytaniem przy analizie algorytmów on-line jest to, jak przeanalizować ich koszt. Zwykłą metodą jest porównanie każdego algorytmu z algorytmem wszechwiedzącym off-line - tym, który potrafi poprawnie przewidzieć przyszłość (w naszym przypadku, do jakiego stopnia dzieci będą cieszyć się jazdą na nartach). Oczywiście żaden algorytm on-line nie może działać lepiej niż ten algorytm offline; najlepszy algorytm on-line to ten, który jest najbliżej tego celu. W naszym przykładzie okazuje się, że najlepszym algorytmem on-line dla problemu z nartami jest wypożyczenie, aż koszt wypożyczenia będzie równy kosztowi zakupu, a następnie kupno. Ten algorytm jest mniej niż dwa razy gorszy od algorytmu wszechwiedzącego. Aby to zobaczyć, załóżmy, że koszt zakupu nart jest równy kosztowi wypożyczeń M i zastanów się, ile razy dzieci będą chciały jeździć na nartach. Jeśli jeżdżą na nartach mniej niż M razy, oba algorytmy zapłacą tę samą kwotę (wszystkowiedzący off-line wie z góry, że będzie jeździł mniej niż M razy, więc wynajmie, a nie kupi). Jeśli dzieci będą chciały jeździć na nartach więcej niż M razy, algorytm on-line wypożyczy M−1 razy, a następnie kupi, za łączny koszt równy $2 \cdot M - 1$ wypożyczeń. Algorytm off-line, znając przyszłość, kupi natychmiast, za cenę wynajmu M. W każdym razie algorytm on-line nigdy nie przekracza dwukrotności kosztu algorytmu off-line. Można wykazać, że żadna inna strategia nie okaże się ogólnie lepsza.4 Strategia, w ramach której rodzice kupią narty po wypożyczeniu K, gdzie K jest ściśle mniejsze niż M − 1, może kosztować K + M wynajem jednostek, podczas gdy algorytm off-line płaci tylko K + 1. (Ile razy dzieci muszą chcieć jeździć na nartach, aby tak się stało?) Jeśli K jest większe niż M − 1, algorytm on-line może ponownie kosztowało rodziców K+M, podczas gdy algorytm off-line wzywa do natychmiastowego zakupu nart, za cenę tylko M wypożyczeń. W obu przypadkach koszt algorytmu on-line jest co najmniej dwukrotnością kosztu algorytmu wszechwiedzącego off-line.

Badania nad metodami algorytmicznymi

Naprawdę bardzo niewiele jest powszechnie akceptowanych paradygmatów, które są wystarczająco ogólne, by zasługiwały na specjalną nazwę i tytuł „metoda algorytmiczna", a większość bardziej

znanych została już opisana. Mimo to, bez szczególnego dążenia do ogólnych paradygmatów, informatycy nieustannie poszukują lepszych metod rozwiązywania coraz bardziej złożonych problemów algorytmicznych. Trochę trudno jest tu dalej omawiać te próby, gdyż kwestie efektywności wkradają się na bardzo wczesnym etapie, a efektywność została omówiona szczegółowo dopiero w rozdziale 6. Ponadto pojęcia współbieżności i probabilizmu, omówione w rozdziałach 10 i 11, stają się coraz bardziej kluczowe dla najnowszych osiągnięć w projektowaniu algorytmicznym. Omawiając te tematy, zobaczymy kilka dodatkowych sposobów na wymyślenie dobrych algorytmów.

Ćwiczenia

W kolejnych ćwiczeniach drzewo jest podawane przez (wskaźnik do) jego korzenia. Węzeł V drzewa z zewnętrznym stopniem N ma N potomstwa, oznaczony jako pierwszy do N-tego i zawiera pewną pozycję danych. Liść jest węzłem bez potomstwa. Drzewo binarne ogranicza liczbę potomstwa każdego węzła do maksymalnie dwóch. Głębokość węzła drzewa jest następująca: głębokość korzenia wynosi 0, a jeśli głębokość V wynosi N to głębokość jego potomstwa wynosi N + 1. Dla węzła V dostępne operacje obejmują pobranie jego zawartość, sprawdzenie, czy ma I-te potomstwo, a jeśli tak, przypisanie wskaźnika do tego potomstwa.

1. Rozważ problem sumowania wynagrodzeń pracowników zarabiających więcej niż ich bezpośredni przełożony, zakładając, że każdy pracownik ma jednego przełożonego. Pracownicy są oznaczeni etykietami 1, 2 itd. Napisz algorytmy rozwiązujące problem dla każdej z poniższych reprezentacji danych wejściowych:

(a) Dane wejściowe są liczbą całkowitą N i dwuwymiarową tablicą A, gdzie N to liczba pracowników, A[I, 1] to wynagrodzenie I pracownika, a A[I, 2] to wynagrodzenie etykietę swojego menedżera.

(b) Dane wejściowe są podawane przez drzewo binarne skonstruowane w następujący sposób: Korzeń drzewa reprezentuje pierwszego pracownika. Dla każdego węzła V drzewa reprezentującego I pracownika,

- V zawiera wynagrodzenie I pracownika;

- pierwsze potomstwo V to listek zawierający etykietę kierownika I pracownika; oraz

- jeśli jest więcej niż I pracowników, drugie potomstwo V jest węzłem reprezentującym I + 1 pracownika.

2.

(a) Napisz algorytm, który mając dane drzewo T , oblicza sumę głębokości wszystkich węzłów T .

(b) Napisz algorytm, który mając drzewo T i dodatnią liczbę całkowitą K, oblicza liczbę węzłów w T na głębokości K.

(c) Napisz algorytm, który przy danym drzewie T sprawdza, czy ma on jakiś liść na równej głębokości.

3. Napisz algorytmy, które rozwiązują następujące problemy, wykonując przechodzenie wszerz danych drzew. Możesz założyć dostępność kolejki Q. Operacje na Q obejmują dodanie przedmiotu z tyłu, odzyskanie i usunięcie przedmiotu z przodu oraz testowanie Q pod kątem pustki.

(a) Mając drzewo T, którego węzły zawierają liczby całkowite, wypisz listę składającą się z sumy zawartości węzłów na głębokości 0, sumy zawartości węzłów na głębokości 1 itd.

(b) Mając dane drzewo T , znajdź głębokość K z maksymalną liczbą węzłów w T . Jeśli jest kilka takich Ks, zwróć ich maksimum.

Wyrażenie arytmetyczne utworzone przez nieujemne liczby całkowite i standardową jednoargumentową operację „−” oraz operacje binarne „+”, „−”, „×” i „/” może być reprezentowane przez drzewo binarne w następujący sposób:

- Liczba całkowita I jest reprezentowana przez liść zawierający I .

- Wyrażenie −E, gdzie E jest wyrażeniem, jest reprezentowane przez drzewo, którego korzeń zawiera „−”, a jego pojedyncze potomstwo jest korzeniem poddrzewa reprezentującego wyrażenie E.

- Wyrażenie E ∗ F, gdzie E i F są wyrażeniami, a „-” jest operacją binarną, jest reprezentowane przez drzewo, którego korzeń zawiera „∗”, jego pierwsze potomstwo jest korzeniem poddrzewa reprezentującego wyrażenie E, a jego drugie potomstwo jest korzeniem poddrzewa reprezentującego F. Zauważ, że symbol „-” oznacza zarówno operacje jednoargumentowe, jak i binarne, a węzły drzewa zawierające ten symbol mogą mieć stopień zewnętrzny 1 lub 2.

4. Zaprojektuj algorytm sprawdzający, czy dane drzewo reprezentuje wyrażenie arytmetyczne.

5. (a) Zaprojektuj algorytm, który oblicza wartość wyrażenia arytmetycznego, biorąc pod uwagę jego reprezentację w postaci drzewa. Zauważ, że dzielenie przez zero jest niezdefiniowane.

(b) Rozszerz swój algorytm, aby najpierw wydrukować wyrażenie reprezentowane przez drzewo wejściowe, a następnie znak równości „=” i jego ocenę. Wyrażenie drukowane powinno być w całości w nawiasach, tj. para pasujących nawiasów powinna obejmować każde zastosowanie operacji binarnej. Mówimy, że dwa wyrażenia arytmetyczne E i F są izomorficzne, jeśli E można uzyskać z F przez zastąpienie niektórych nieujemnych liczb całkowitych innymi. Na przykład wyrażenia (2 + 3) × 6 − (−4) i (7 + 0) × 6 − (−9) są izomorficzne, ale żadne z nich nie jest izomorficzne z żadnym z (−2 + 3) × 6 − (−4) i (7 + 0) + 6 − (−9).

Mówi się, że wyrażenie E jest zrównoważone, jeśli każda operacja binarna w nim jest zastosowana do dwóch wyrażeń izomorficznych. Na przykład wyrażenia -5, (1 + 2) ∗ (3 + 5) i ((−3)/(−4))/((−1)/(−100)) są zrównoważone, natomiast 12 + (3 + 2) i (−3) ∗ (−3) nie są.

6. Zaprojektuj algorytm sprawdzający, czy dwa wyrażenia są izomorficzne, biorąc pod uwagę ich reprezentację w formie drzewa.

7. Zaprojektuj algorytm sprawdzający, czy wyrażenie jest zrównoważone, biorąc pod uwagę jego reprezentację w postaci drzewa. (Wskazówka: wykonaj najpierw przejście wszerz drzewa.)

8. Udowodnij, że maksymalna odległość między dowolnymi dwoma punktami wielokąta występuje między dwoma wierzchołkami.

9. Napisz program implementujący algorytm maksymalnej odległości wielokątnej.

10. Zaprojektuj algorytm, który przy danych (wierzchołkach) niekoniecznie wypukłego wielokąta znajduje parę wierzchołków o minimalnej odległości.

11. Napisz algorytmy, które znajdują dwa maksymalne elementy w danym wektorze składającym się z N odrębnych liczb całkowitych (załóżmy, że N > 1).

(a) Za pomocą metody iteracyjnej.

(b) Stosowanie metody dziel i zwyciężaj.

12. Napisz szczegółowo opisany w tekście zachłanny algorytm znajdowania minimalnego drzewa opinającego. Problem plecaka całkowitoliczbowego polega na znalezieniu sposobu na wypełnienie

plecaka o określonej pojemności niektórymi elementami z danego zestawu dostępnych przedmiotów różnego typu w najbardziej opłacalny sposób. Dane wejściowe do problemu składają się z:

- C, całkowita nośność plecaka;

- dodatnia liczba całkowita N, liczba typów pozycji;

- wektor Q, gdzie Q[I] jest dostępną liczbą elementów typu I;

- wektor W, gdzie W[I] jest wagą każdego elementu typu I spełniającego $0 < W[I] \leq C$;

oraz

- wektor P, gdzie P[I] jest zyskiem z przechowania przedmiotu typu I w plecaku.

Wszystkie wartości wejściowe są nieujemnymi liczbami całkowitymi. Problem polega na wypełnieniu plecaka elementami, których łączna waga nie przekracza C, tak aby łączny zysk z plecaka był maksymalny. Wynikiem jest wektor F, gdzie F[I] zawiera liczbę elementów typu I, które są wkładane do plecaka. Problem plecakowy jest odmianą problemu plecakowego, w którym zamiast dyskretnych przedmiotów są materiały. Różnica polega na tym, że zamiast pracować z liczbą całkowitą liczb, możemy włożyć do plecaka dowolną ilość materiału I, która nie przekracza dostępnej ilości Q[I]. Wektory W i P zawierają teraz odpowiednio wagę i zysk jednej jednostki ilościowej materiału I . Wszystkie wartości wejściowe i wyjściowe są teraz nieujemnymi liczbami rzeczywistymi, niekoniecznie liczbami całkowitymi.

13. (a) Zaprojektuj dynamiczny algorytm planowania dla problemu plecakowo-całkowitego.

(b) Jaki jest wynik twojego algorytmu dla danych wejściowych?

- N = 5

- C = 103

- Q = [3,1,4,5,1]

- W = [10,20,20,8,7]

- P = [17,42,35,16,15]

a jaki jest całkowity zysk z plecaka?

14. (a) Zaprojektuj zachłanny algorytm dla problemu plecakowego.

(b) Jaki jest wynik twojego algorytmu dla danych wejściowych podanych w ćwiczeniu 4.13(b) i jaki jest teraz całkowity zysk z plecaka?

15. (a) W jaki sposób odniesiesz łączne zyski uzyskane dla danego wejściowego typu liczb całkowitych, gdy zostaniesz poddany problemowi plecakowemu i problemowi plecakowemu typu Integer?

(b) Rozważ modyfikację algorytmu, który zaprojektowałeś w ćwiczeniu 4.14(a), która daje w F część całkowitą wielkości obliczonych przez oryginalny algorytm. Udowodnij, że zmodyfikowany algorytm nie rozwiązuje problemu plecakowego. Oznacza to, że podaj dane wejściowe w postaci liczb całkowitych, dla których (zmodyfikowany) algorytm zachłanny wygeneruje akceptowalne wypełnienie liczb całkowitych, które nie jest maksymalnie opłacalne. Znajdź takie dane wejściowe z N, liczbą typów, jak najmniejszą. (Wskazówka: poprawne rozwiązania problemu z plecakiem całkowitym, w

przeciwieństwie do problemu z plecakiem, mogą pozostawić dostępne przedmioty poza plecakiem, nawet jeśli nie jest on pełny.)