Wydajność algorytmów

Poproszony o zbudowanie mostu nad rzeką, łatwo jest zbudować „niepoprawny" most. Most może nie być wystarczająco szeroki dla wymaganych pasów, może nie być wystarczająco mocny, aby przenosić ruch w godzinach szczytu lub w ogóle nie docierać na drugą stronę! Jednak nawet jeśli jest „poprawny", w tym sensie, że w pełni spełnia wymagania operacyjne, nie każdy kandydat na projekt mostu będzie akceptowalny. Możliwe, że projekt wymaga zbyt dużej siły roboczej lub zbyt wielu materiałów lub komponentów. Doprowadzenie do końca może również zająć zbyt dużo czasu. Innymi słowy, chociaż da to dobry most, projekt może być zbyt drogi. Dziedzina algorytmiki jest podatna na podobne problemy. Algorytm może być zbyt kosztowny, a zatem niedopuszczalny. I tak, chociaż Część 5 była poświęcona udowodnieniu, że niepoprawne algorytmy są złe i że metody są potrzebne do ustalenia poprawności algorytmicznej, ta Część argumentuje, że nawet poprawny algorytm może pozostawiać wiele do życzenia. Ponieważ siła robocza i inne powiązane koszty nie są tutaj istotne, pozostają nam dwa kryteria pomiaru oszczędności - materiały i czas. W informatyce nazywa się je miarami złożoności przestrzeni pamięci (lub po prostu przestrzeni) i czasu. Pierwsza z nich jest mierzona kilkoma czynnikami, w tym liczbą zmiennych oraz liczbą i rozmiarami struktur danych wykorzystywanych podczas wykonywania algorytmu. Druga jest mierzona liczbą elementarnych czynności wykonywanych przez procesor w takim wykonaniu. Zarówno przestrzeń, jak i czas wymagany przez algorytm zazwyczaj różnią się w zależności od danych wejściowych, a zatem wydajność algorytmu odzwierciedla sposób, w jaki te zasoby są zużywane w miarę zmian danych wejściowych. Oczywiste jest, że algorytm sumowania wynagrodzeń trwa dłużej na większych listach. Nie oznacza to, że nie możemy precyzyjnie sformułować jego wykonania w czasie; oznacza to jedynie, że sformułowanie będzie musiało uwzględniać fakt, że czas działania algorytmu zależy lub jest funkcją długości listy wejściowej. To samo dotyczy miejsca w pamięci. W rozwiązaniu dynamicznego planowania problemu zmęczonego podróżnika z Części 4 obliczyliśmy wiele optymalnych podróży częściowych i musieliśmy je przechowywać w wektorze, ponieważ były one wykorzystywane w obliczeniach innych. Ich liczba, a co za tym idzie ilość pamięci wykorzystywanej przez algorytm, zależy bezpośrednio od liczby węzłów w wejściowym grafie miasta. Chociaż w tym rozdziale koncentrujemy się na mierze czasu, należy zauważyć, że kwestie całkiem analogiczne do poruszonych tutaj dotyczą również miary przestrzeni.

Potrzebne są ulepszenia

Naszą dyskusję na temat efektywności czasowej w algorytmice należy rozpocząć od całkowitego odrzucenia mitu dotyczącego szybkości komputera. Niektórzy uważają, że komputery są tak niewiarygodnie szybkie, że nie ma problemu z czasem. Cóż, ta opinia jest bezpodstawna. Oto dwa krótkie przykłady, o których powiemy więcej w dalszej części. Załóżmy, że interesuje nas znalezienie najkrótszej trasy dla podróżnika, który chce odwiedzić każde z, powiedzmy, 200 miast. W tej chwili nie ma komputera, który mógłby znaleźć trasę w czasie krótszym niż miliony lat obliczeniowych! Podobnie żaden komputer nie jest w stanie rozłożyć na czynniki (czyli znaleźć liczb pierwszych, które dzielą) dużych liczb całkowitych, powiedzmy o długości 300 cyfr, w czasie krótszym niż miliony lat. Status obu tych problemów może się zmienić, ale na razie nikt nie zna ich sensownych rozwiązań. Bardziej szczegółowe omówienie rzeczywistego czasu, jaki zająłby współczesnemu komputerowi rozwiązanie tych problemów, znajduje się w Części 7. Czas jest kluczowym czynnikiem w prawie każdym użyciu komputerów. Nawet w codziennych zastosowaniach, takich jak programy do prognozowania pogody, systemy zarządzania zapasami i programy do wyszukiwania bibliotek, istnieje szerokie pole do ulepszeń. Czas to pieniądz, a czas komputera nie jest wyjątkiem. Co więcej, jeśli chodzi o komputery, czas może być czynnikiem krytycznym. Niektóre rodzaje systemów skomputeryzowanych, takie jak sterowanie lotem, naprowadzanie pocisków i programy nawigacyjne, określane są jako systemy czasu rzeczywistego. Muszą reagować na bodźce zewnętrzne, takie jak naciskanie przycisków, w czasie

„rzeczywistym”; to znaczy prawie natychmiast. Niezastosowanie się do tego może okazać się śmiertelne.

Ulepszenia po fakcie

Istnieje wiele standardowych sposobów na poprawienie czasu działania danego algorytmu. Niektóre z nich są włączane do kompilatorów, zamieniając je w kompilatory optymalizujące, które w rzeczywistości rekompensują pewne złe decyzje ze strony programisty. Wiele zadań kompilatora optymalizującego można traktować jako przeprowadzanie pewnych rodzajów przekształceń programu. Jednym z najpowszechniej stosowanych jest modyfikowanie programu lub algorytmu poprzez przenoszenie instrukcji z wnętrza pętli na zewnątrz. Czasami jest to proste, jak w poniższym przykładzie. Załóżmy, że nauczyciel, w interesie, aby klasa zarejestrowała w miarę dobre wyniki na egzaminie, chce znormalizować listę ocen, dając uczniowi, który uzyskał najlepszy wynik na egzaminie, 100 punktów i odpowiednio podnosząc resztę. Wysokopoziomowy opis algorytmu może wyglądać tak:

(1) obliczyć maksymalny wynik w MAX;

(2) pomnóż każdy wynik przez 100 i podziel go przez MAX.

(Musimy założyć, że istnieje co najmniej jedna niezerowa klasa, aby podział był dobrze zdefiniowany.) Jeśli lista jest podana w wektorze L(1), ... , L(N), obie części można wykonać za pomocą prostych pętli biegnących przez wektor. Pierwsza szuka maksimum w jakiś standardowy sposób, a druga może być napisana:

(2) dla I od 1 do N wykonuję:

(2.1) L(I) L(I) × 100/MAX

Zauważ, że dla każdego stopnia L(I) algorytm wykonuje jedno mnożenie i jedno dzielenie w pętli. Jednak ani 100, ani wartość MAX nie zmienia się w pętli. Stąd czas wykonania tej drugiej pętli można skrócić prawie o dwie (!) obliczając stosunek 100/MAX przed rozpoczęciem pętli. Odbywa się to po prostu wstawiając oświadczenie:

WSPÓŁCZYNNIK ← 100/MAKS.

między krokami (1) i (2) oraz pomnożenie L(I) przez FACTOR w pętli. Wynikowy algorytm to:

(1) obliczyć maksymalny wynik w MAX;

(2) WSPÓŁCZYNNIK←100/MAX;

(3) dla I od 1 do N wykonuję:

(3.1) L(I) L(I) × WSPÓŁCZYNNIK.

Powodem prawie 50% poprawy jest to, że treść drugiej pętli, która pierwotnie składała się z dwóch operacji arytmetycznych, teraz składa się tylko z jednej. Oczywiście nie we wszystkich implementacjach takiego algorytmu czas będzie zdominowany przez instrukcje arytmetyczne; możliwe, że aktualizacja wartości L(I) jest bardziej czasochłonne niż dzielenie liczbowe. Mimo to następuje znaczna poprawa czasu działania i wyraźnie im dłuższa lista, tym więcej czasu zyskuje ta zmiana. Jak wspomniano, modyfikacje takie jak ta są dość proste i wiele kompilatorów jest w stanie przeprowadzić je automatycznie. Jednak nawet proste usunięcie instrukcji czasami wymaga sprytnych sztuczek. Spójrzmy na inny przykład.

Wyszukiwanie liniowe: przykład

Załóżmy, że szukamy elementu X na nieuporządkowanej liście (np. numer telefonu w pomieszanej książce telefonicznej). Standardowy algorytm wywołuje prostą pętlę, w ramach której przeprowadzane są dwa testy: (1) „czy znaleźliśmy X?” oraz (2) „czy doszliśmy do końca listy?” Pozytywna odpowiedź na którekolwiek z tych pytań powoduje zakończenie działania algorytmu - pomyślnie w pierwszym przypadku i bezskutecznie w drugim. Ponownie możemy założyć, że testy te dominują w wydajności czasowej algorytmu wyszukiwania. Drugi test można wykonać poza pętlą, korzystając z następującej sztuczki. Przed rozpoczęciem wyszukiwania wymagany element X jest fikcyjnie dodawany na końcu listy. Następnie wykonywana jest pętla wyszukiwania, ale bez testowania końca listy; w pętli sprawdzamy tylko, czy X został znaleziony. Skraca to również czas całego algorytmu o około 50%. Teraz, odkąd dodaliśmy X do listy, X zawsze zostanie znaleziony, nawet jeśli nie pojawił się na oryginalnej liście. Jednak w tym przypadku znajdziemy się na końcu nowej listy, gdy konfrontujemy się z X po raz pierwszy, podczas gdy znajdziemy się gdzieś w jej obrębie, jeśli X pojawił się na oryginalnej liście. W konsekwencji, po osiągnięciu X, test na znalezienie się na końcu listy jest przeprowadzany raz, a algorytm zgłasza sukces lub porażkę w zależności od wyniku. (Nawiasem mówiąc, częsty błąd może wystąpić tutaj, jeśli zapomnimy usunąć fikcyjnego X z końca listy przed zakończeniem.) Tym razem musieliśmy być nieco bardziej kreatywni, aby zaoszczędzić 50%.

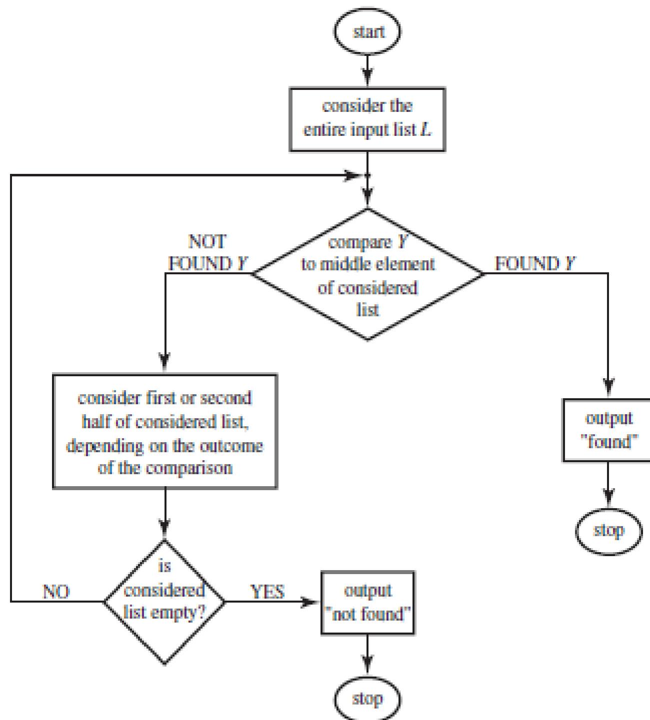
Ulepszenia rzędu wielkości

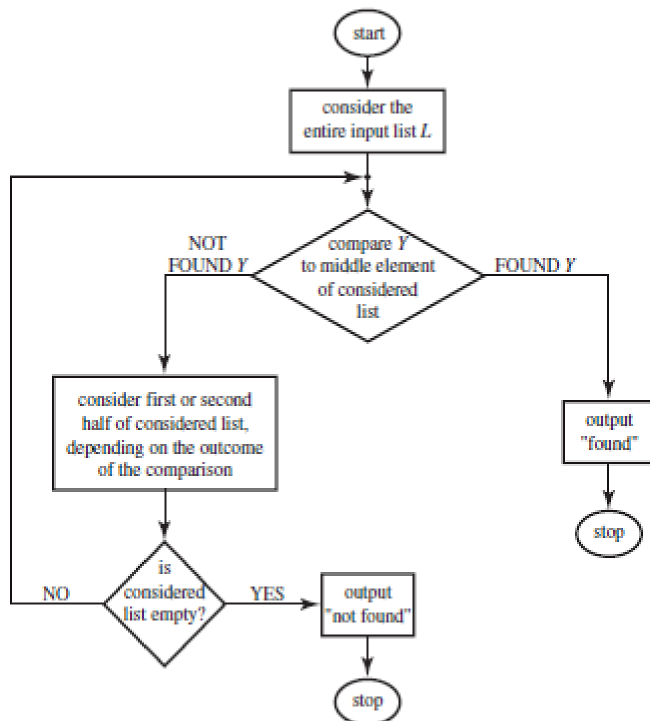
Skrócenie czasu działania algorytmu o 50% jest imponujące. Jak to jednak bywa, w wielu przypadkach potrafimy zrobić dużo lepiej. Kiedy mówimy „lepiej”, nie mamy na myśli tylko stałych spadków o 50%, 60%, a nawet 90%, ale spadki, których stopa staje się coraz lepsza wraz ze wzrostem wielkości danych wejściowych. Jednym z najbardziej znanych przykładów jest wyszukiwanie elementu na uporządkowanej lub posortowanej liście (na przykład wyszukiwanie numeru telefonu w normalnej książce telefonicznej). Mówiąc dokładniej, załóżmy, że dane wejściowe składają się z nazwy Y oraz listy L nazwisk i ich numerów telefonów. Zakłada się, że lista, która ma długość N, jest posortowana alfabetycznie według nazw. Naiwny algorytm, który wyszukuje numer telefonu Y, to ten opisany wcześniej dla nieposortowanej listy: przeglądaj listę L po jednym nazwisku, porównując Y z bieżącą nazwą na każdym kroku i sprawdzając koniec listy listy w tym samym czasie lub stosując sztuczkę opisaną w poprzedniej sekcji i sprawdzając ją tylko raz, gdy zostanie znalezione Y. Nawet jeśli sztuczka zostanie zastosowana, aby skrócić czas działania o 50%, nadal może istnieć przypadek, w którym konieczne jest pełne N porównań; mianowicie, gdy Y w ogóle nie pojawia się w L lub gdy pojawia się na ostatniej pozycji. Mówimy, że algorytm, nazwijmy go A, ma najgorszy możliwy czas działania, który jest liniowy w N lub, używając równoważnego terminu, jest rzędu N. Jest to opisane bardziej zwięźle, mówiąc, że A działa w czas O(N) w najgorszym przypadku, gdzie duże-O oznacza „kolejność”. Notacja O(N) jest dość subtelna. Zauważ, że powiedzieliśmy „A działa w czasie O(N)”. Nie doprecyzowaliśmy stwierdzenia, aby odnosiło się tylko do liczby porównań, chociaż porównania między Y a nazwiskami w L były jedynym rodzajem instrukcji, który liczyliśmy. Powód tego zostanie wyjaśniony później. Na razie wystarczy powiedzieć, że używając notacji duże-O, jak to się czasem nazywa, nie obchodzi nas, czy algorytm przyjmuje czas N (czyli wykonuje N instrukcji elementarnych), czas 3N (czyli wymaga trzykrotnie większej liczby instrukcji elementarnych), czyli 10N, a nawet 100N. Co więcej, nawet jeśli zajmuje tylko ułamek N, powiedzmy N/6, nadal mówimy, że działa w czasie O(N). Jedyną rzeczą, która ma znaczenie, jest to, że czas działania rośnie liniowo wraz z N. Oznacza to, że istnieje pewna stała liczba K, taka, że algorytm działa w czasie nie większym niż K × N w najgorszym przypadku. Rzeczywiście, wersja, która sprawdza koniec listy za każdym razem działa mniej więcej w czasie 2N, a podstępna wersja zmniejsza to do mniej więcej N. Jednak obie mają czas działania wprost proporcjonalny do N. Są to zatem algorytmy czasu liniowego , mający najgorszy czas działania O(N). Termin „najgorszy przypadek” oznacza, że algorytm mógłby prawdopodobnie działać znacznie mniej na niektórych wejściach, być może na większości wejść. Mówimy tylko, że algorytm nigdy nie działa dłużej niż K × N

czasu i że jest to prawdą dla dowolnego N i dla dowolnego wejścia o długości N, nawet najgorszych. Oczywiście, jeśli spróbujemy ulepszyć algorytm wyszukiwania w czasie liniowym, rozpoczynając porównania od końca listy, nadal będą występować równie złe przypadki - Y nie pojawia się w ogóle lub Y pojawia się jako pierwsze imię na liście. W rzeczywistości, jeśli algorytm wymaga dokładnego przeszukania listy, kolejność, w jakiej nazwy są porównywane z Y, nie ma znaczenia. Taki algorytm nadal będzie miał czas działania O(N) w najgorszym przypadku. Mimo to, możemy lepiej przeszukiwać uporządkowaną listę, nie tylko pod względem stałego czynnika „ukrytego wewnątrz" dużego O, ale pod względem samego oszacowania O(N). Jest to poprawa rzędu wielkości i teraz zobaczymy, jak można ją osiągnąć.

Wyszukiwanie binarne: Przykład

Dla konkretności załóżmy, że książka telefoniczna zawiera milion nazwisk, czyli N to 1 000 000 i nazwijmy je X1, X2, ... , X1 000 000. Pierwsze porównanie przeprowadzone przez nowy algorytm nie następuje między Y a imieniem lub nazwiskiem w L, ale między Y i drugim imieniem (lub, jeśli lista jest parzysta, to nazwisko z pierwszej połowy listy), czyli $X_{500\,000}$. Zakładając, że porównywane nazwy okażą się nierówne, co oznacza, że jeszcze nie skończyliśmy, istnieją dwie możliwości: (1) Y poprzedza $X_{500\,000}$ w kolejności alfabetycznej, oraz (2) $X_{500\,000}$ poprzedza Y . Ponieważ lista jest posortowana alfabetycznie, jeśli tak jest w przypadku (1), wiemy, że jeśli w ogóle pojawia się na liście, to musi być w pierwszej połowie, a jeśli tak jest w przypadku (2), to musi pojawić się w drugiej połowie. Możemy więc ograniczyć nasze kolejne poszukiwania do odpowiedniej połowy listy. W związku z tym następne porównanie będzie pomiędzy Y a środkowym elementem tej połowy: X250.000 w przypadku (1) i $X_{750\,000}$ w przypadku (2). I znowu, wynikiem tego porównania będzie zawężenie możliwości do połowy tej nowej, krótszej listy; to znaczy do listy, której długość jest równa jednej czwartej oryginału. Ten proces jest kontynuowany, zmniejszając długość listy lub ogólniej, rozmiar problemu o połowę na każdym kroku, aż do znalezienia Y, w którym to przypadku procedura kończy się, zgłaszając sukces lub trywialną pustą listę zostanie osiągnięty, w takim przypadku kończy się, zgłaszając niepowodzenie . Ta procedura nazywa się przeszukiwaniem binarnym i jest tak naprawdę zastosowaniem paradygmatu dziel i zwyciężaj omówionego w Części 4. Różnica między tym przykładem a wprowadzonymi w nim (przeszukiwanie min&max i sortowanie przez scalanie) polega na tym, że po dzieleniu potrzebujemy tylko podbij jedną z części, a nie obie. Rysunek zawiera schematyczną iteracyjną wersję wyszukiwania binarnego, ale w rzeczywistości możliwe jest również zapisanie prostej wersji rekurencyjnej.





Zachęcamy do tego. Jaka jest złożoność czasowa wyszukiwania binarnego? Aby na to odpowiedzieć, najpierw policzmy porównania. Próba odgadnięcia, ile porównań będzie wymagał algorytm wyszukiwania binarnego w najgorszym przypadku w naszej książce telefonicznej zawierającej milion nazwisk, jest dość pouczające. Przypomnijmy, że naiwne poszukiwania wymagałyby miliona porównań.

Cóż, w najgorszym przypadku (jaki jest przykład jednego? ile jest najgorszych przypadków?) algorytm będzie wymagał tylko 20 porównań! Co więcej, im większe staje się N, tym bardziej imponująca jest poprawa. Międzynarodowa książka telefoniczna zawierająca, powiedzmy, miliard nazwisk, wymagałaby co najwyżej 30 porównań zamiast miliarda! Przypomnij sobie, że każde porównanie zmniejsza długość listy wejściowej L o połowę, a proces kończy się, gdy lista stanie się pusta lub wcześniej. Stąd najgorszą liczbę porównań uzyskuje się, obliczając, ile razy liczba N może być wielokrotnie podzielona przez 2, zanim zostanie zredukowana do 0. (Zasady gry polegają na ignorowaniu ułamków). logarytm o podstawie 2 z N i jest oznaczony przez log2N. W rzeczywistości log2 N zlicza liczbę potrzebną do zredukowania N do 1, a nie do 0, więc tak naprawdę szukamy $1 + \log_2 N$. można śmiało powiedzieć, że algorytm wykonuje porównania O(log N) w najgorszym przypadku. Możemy wyczuć rodzaj ulepszeń, jakie oferuje wyszukiwanie binarne, analizując poniższą tabelę, która przedstawia kilka wartości N w stosunku do liczby porównań wymaganych przez wyszukiwanie binarne w najgorszym przypadku:

N : $1 + \log_2 N$

10 : 4

100 : 7

1000 : 10

milion : 20

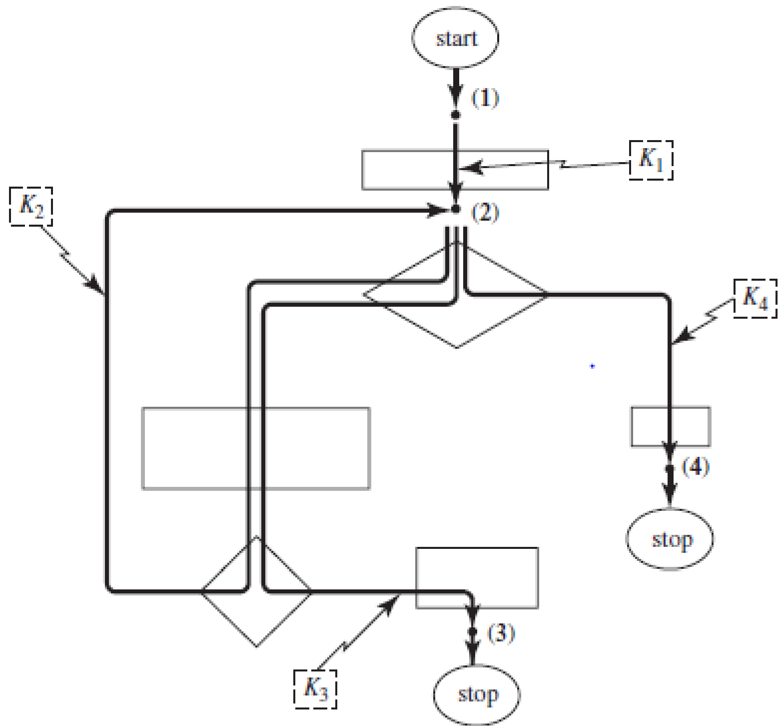
miliard : 30

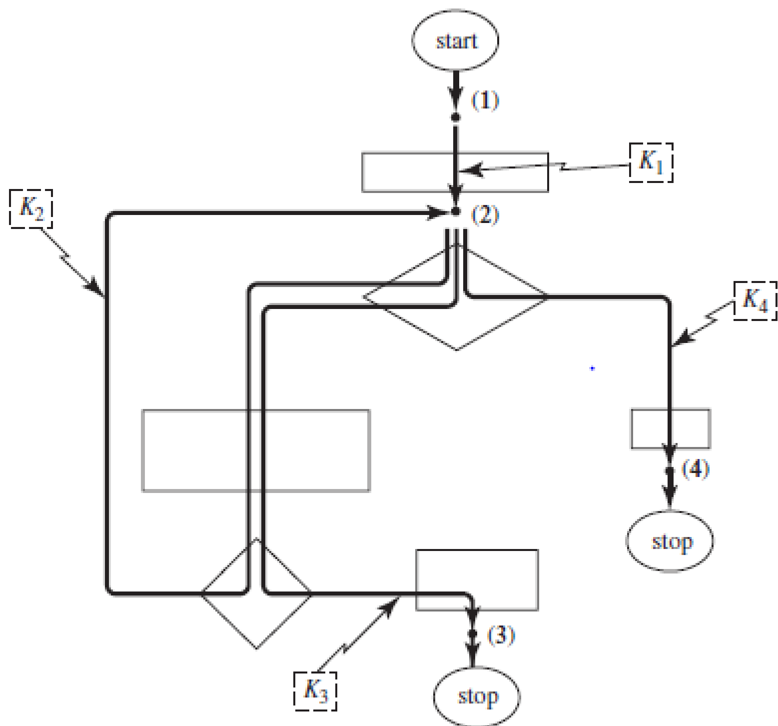
miliard miliardów : 60

Warto zauważyć, że sami posługujemy się wariantem wyszukiwania binarnego, gdy szukamy numeru w książce telefonicznej. Różnica polega na tym, że niekoniecznie porównujemy dane nazwisko z tym, które pojawia się dokładnie w środku książki, a następnie z tym w 1/4 lub 3/4 pozycji i tak dalej. Raczej korzystamy z dodatkowej wiedzy, którą posiadamy, dotyczącej oczekiwanego rozmieszczenia nazwisk w księdze. Ta technika jest często nazywana wyszukiwaniem interpolacyjnym. Jeśli szukamy na przykład „Mary D. Ramsey", otworzymy książkę mniej więcej w punkcie dwóch trzecich. Oczywiście jest to dalekie od precyzji i pracujemy według ogólnych, intuicyjnych reguł, które zostaną omówione pod pojęciem heurystyki w Części 15. Niemniej jednak istnieją precyzyjne sformułowania algorytmów wyszukiwania, które działają w sposób krzywy, podyktowany naturą. i rozmieszczenie elementów. Ogólnie rzecz biorąc, chociaż są one średnio znacznie bardziej ekonomiczne, zmiany te wykazują również podobne zachowanie w najgorszym przypadku jak porównania $O(\log_2 N)$.

Dlaczego wystarczy liczyć porównania?

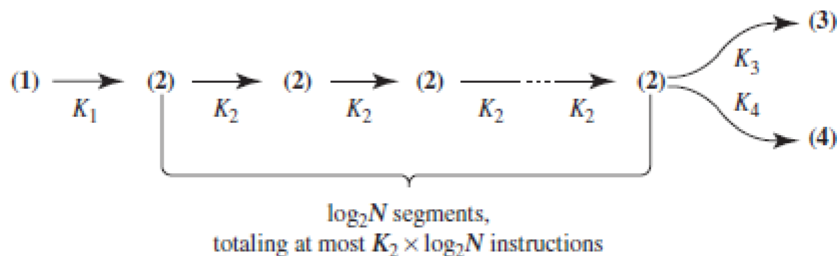
Aby zakończyć naszą analizę złożoności wyszukiwania binarnego, pokazujemy teraz, że jeśli jesteśmy zadowoleni z oszacowania dużego O, wystarczy zliczyć tylko porównania. Czemu? Oczywiście w wyszukiwaniu binarnym wykonywanych jest znacznie więcej instrukcji niż tylko porównania. Ponowne skupienie się na właściwej połowie listy i przygotowanie jej na następny raz wokół pętli prawdopodobnie wiązałoby się z pewnym testowaniem i zmianą indeksów. A potem jest test na pustą listę i tak dalej. Jak się okazuje, pomocne okazują się również punkty kontrolne użyte w poprzednim rozdziale do sprawdzania poprawności. Rysunek przedstawia schemat blokowy z uwzględnieniem punktów kontrolnych i ilustruje możliwe ścieżki lokalne lub „przeskoki" między nimi.





Zauważ, że również tutaj punkty kontrolne przecinają wszystkie pętle - w tym przypadku jest tylko jedna - tak, że cztery ścieżki lokalne są wolne od pętli, jak w przykładzie podanym w Części 5. Fakt ten był ważny dla dowodów poprawności, ponieważ dał początek łatwym do opanowania warunkom weryfikacji, tutaj jest to ważne, ponieważ segmenty bez pętli i podprogramów zajmują maksymalnie stałą ilość czasu. Prawdą jest, że obecność instrukcji warunkowych w takim segmencie może dać kilka różnych sposobów jego przechodzenia. Jednak brak pętli (i podprogramów rekurencyjnych)

gwarantuje, że istnieje ograniczenie tej liczby, a zatem całkowita liczba instrukcji wykonywanych w pojedynczym wykonaniu takiego segmentu nie jest większa niż pewna stała. Jak zobaczymy, często ułatwia to łatwe obliczenie czasu, jaki zajmuje algorytm. Na rysunku skojarzyliśmy stałe od K_1 do K_4 z czterema możliwymi ścieżkami lokalnymi. Na przykład oznacza to, że zakłada się, że każde pojedyncze wykonanie ścieżki lokalnej z punktu kontrolnego (2) z powrotem do samego siebie obejmuje nie więcej niż instrukcje K_2. (Umownie przyjmuje się, że obejmuje to porównanie wykonane na początku segmentu, ale nie na jego końcu). Rozważmy teraz typowe wykonanie wyszukiwania binarnego



Ponieważ punkt kontrolny (2) jest jedynym miejscem w tekście algorytmu dotyczącego porównania, jest on osiągany (w najgorszym przypadku) dokładnie $1 + \log_2 N$ razy, ponieważ, jak omówiono powyżej, jest to całkowita liczba dokonanych porównań. Teraz wszystkie z wyjątkiem ostatniego z tych porównań powodują, że procesor ponownie omija pętlę, a tym samym wykonuje co najwyżej kolejne K instrukcji. Oznacza to, że całkowity koszt czasu wszystkich tych przejść log2 N segmentu (2)→(2) wynosi $K_2 \times \log_2 N$. Aby zakończyć analizę, zauważ, że łączna liczba wszystkich wykonanych instrukcji, które nie są częścią (2) →(2) segmenty, to albo $K_1 + K_3$ albo $K_1 + K_4$, w zależności od tego, czy algorytm zatrzymuje się w punkcie kontrolnym (3) czy (4) ; w obu przypadkach mamy stałą niezależną od N. Jeśli przez K oznaczymy maksimum z tych dwóch sum, możemy wywnioskować, że całkowita liczba instrukcji wykonywanych przez algorytm przeszukiwania binarnego na liście o długości N jest ograniczona za pomocą:

$K + (K_2 \times \log_2 N)$

W związku z tym, ponieważ używamy notacji rzędu wielkości, stałe K i K_2 można „zakopać" pod big-O i możemy po prostu wywnioskować, że wyszukiwanie binarne przebiega w czasie $O(\log_2 N)$ w najgorszym przypadku . Nazywa się to algorytmem logarytmicznym.

Solidność notacji Big-O

Jak wyjaśniono, big-Os ukrywają stałe czynniki. W związku z tym w najgorszej analizie wyszukiwania binarnego nie było potrzeby, abyśmy byli bardziej precyzyjni w szczegółach poszczególnych instrukcji składających się na algorytm. Wystarczyło przeanalizować strukturę pętli algorytmu i przekonać się, że pomiędzy dowolnymi dwoma porównaniami jest tylko stała liczba instrukcji w najgorszym przypadku. Jakoś to nie brzmi całkiem dobrze. Z pewnością złożoność czasowa wyszukiwania binarnego zależy od dozwolonych podstawowych instrukcji. Gdybyśmy zezwolili na instrukcję:

search list L for item Y

algorytm składałby się po prostu z pojedynczej instrukcji, a zatem jego złożoność czasowa byłaby O(1), czyli stała liczba w ogóle nie zależna od N. Jak więc możemy powiedzieć, że szczegóły instrukcji są nieistotne? Cóż, złożoność czasowa algorytmu jest rzeczywiście pojęciem względnym i ma sens tylko w połączeniu z uzgodnionym zbiorem instrukcji elementarnych. Niemniej jednak, standardowe rodzaje problemów są zwykle rozważane w kontekście standardowych rodzajów instrukcji elementarnych. Na przykład, wyszukiwanie i sortowanie problemów zazwyczaj obejmuje porównania, aktualizacje

indeksów i testy końca listy, tak że analiza złożoności jest przeprowadzana w odniesieniu do nich. Co więcej, jeśli określony język programowania jest z góry ustalony, to określony zestaw instrukcji elementarnych również został poprawiony, na dobre lub na złe. To, co sprawia, że obserwacje te są istotne, to fakt, że w przypadku większości standardowych rodzajów instrukcji elementarnych możemy sobie pozwolić na dość niejasne, jeśli chodzi o złożoność rzędu wielkości. Napisanie algorytmu w określonym języku programowania lub użycie określonego kompilatora może oczywiście wpłynąć na ostateczny czas działania. Jeśli jednak algorytm używa konwencjonalnych instrukcji podstawowych, różnice te będą składać się tylko ze stałego współczynnika na instrukcję podstawową, a to oznacza, że złożoność big-O jest niezmienna przy takich fluktuacjach implementacyjnych. Innymi słowy, tak długo, jak uzgodniony jest podstawowy zestaw dozwolonych instrukcji elementarnych i tak długo, jak wszelkie skróty stosowane w opisach wysokiego poziomu nie ukrywają nieograniczonych iteracji takich instrukcji, a jedynie reprezentują ich skończone skupiska, duże-O szacunki czasu są solidne. Właściwie dla dobra czytelników, którzy z logarytmami czują się jak w domu, należy dodać, że podstawa logarytmiczna też nie ma znaczenia. Dla dowolnego ustalonego K liczby $\log_2 N$ i logKN różnią się tylko stałym współczynnikiem, a zatem tę różnicę można również ukryć pod notacją big-O. W konsekwencji będziemy odnosić się do wydajności w czasie logarytmicznym po prostu jako O(logN), a nie $O(\log_2 N)$. Solidność zapisu big-O stanowi zarówno jego siłę, jak i słabość. Kiedy ktoś wykazuje algorytm logarytmiczny czasu A, podczas gdy ktoś inny algorytm B działa w czasie liniowym, można bardzo dobrze stwierdzić, że B działa szybciej na przykładowych danych wejściowych niż A! Powód tkwi w ukrytych stałych. Powiedzmy, że skrupulatnie wzięliśmy pod uwagę stałą liczbę elementarnych instrukcji między punktami kontrolnymi, język programowania, w którym zaszyfrowany jest algorytm, kompilator, który tłumaczy go w dół, podstawowe instrukcje maszynowe używane przez komputer z kodem maszynowym oraz samą prędkość tego komputera. Po wykonaniu tej czynności możemy stwierdzić, że algorytm A działa w czasie ograniczonym przez $K \times \log_2 N$ nanosekund, a B działa w ciągu $J \times N$ nanosekund, ale przy K równym 1000, a J równym 10. Oznacza to, że jak możesz zweryfikować, że dla każdego wejścia o długości mniejszej niż tysiąc (w rzeczywistości dokładna liczba to 996), algorytm B jest lepszy od A. Tylko wtedy, gdy osiągnięto dane wejściowe o wielkości 1000 lub większej, różnica między N a $\log_2 N$ stają się widoczne i, jak już wspomniano, kiedy różnica zaczyna się opłacać, robi to bardzo ładnie: do czasu osiągnięcia wielkości wejściowych miliona, algorytm A staje się do 500 razy bardziej wydajny niż algorytm B, a dla nakłady o wielkości miliarda, poprawa jest ponad 330 000 razy! Tak więc użytkownik, który jest zainteresowany tylko danymi wejściowymi o długości poniżej tysiąca, powinien zdecydowanie zastosować algorytm B, pomimo wyższości rzędu wielkości A. Jednak w większości przypadków współczynniki stałe nie są tak daleko od siebie jak 10 i 1000, stąd szacunki dużego O są zwykle znacznie bardziej realistyczne niż w tym wymyślonym przykładzie. Morał tej historii polega na tym, aby najpierw poszukać dobrego i wydajnego algorytmu, podkreślającego wydajność dużego O, a następnie spróbować go ulepszyć za pomocą sztuczek stosowanych wcześniej w celu zmniejszenia zaangażowanych czynników stałych. W każdym razie, ponieważ wydajność big-O może być myląca, kandydujące algorytmy powinny być uruchamiane eksperymentalnie, a ich wyniki czasowe dla różnych typowo występujących rodzajów danych wejściowych powinny być zestawione. Odporność oszacowań dużego O, w połączeniu z faktem, że w większości przypadków algorytmy, które są lepsze w sensie dużego O, są również lepsze w praktyce, sprawia, że badanie złożoności czasowej rzędu wielkości jest najbardziej interesujące dla komputera. naukowcy. W związku z tym, w imię nauki i solidności, w dalszej części skoncentrujemy się głównie na szacunkach big-O i podobnych pojęciach, chociaż mogą one ukrywać kwestie o możliwym znaczeniu praktycznym, takie jak czynniki stałe.

Analiza czasu zagnieżdżonych pętli

Oczywiście skomplikowane algorytmy, które obejmują wiele powiązanych ze sobą struktur kontrolnych i zawierają potencjalnie rekurencyjne podprogramy, mogą być dość trudne do przeanalizowania.

Omówmy pokrótce złożoność czasową niektórych algorytmów pojawiających się w poprzednich częściach. Algorytm sortowania bąbelków z części 2 składał się z dwóch pętli zagnieżdżonych w następujący sposób:

(1) wykonaj następujące N-1 razy:

....

(1.2) wykonaj następujące N - 1 razy:

....

Wewnętrzna pętla jest wykonywana N-1 razy na każdy N-1 razy wykonywana jest pętla zewnętrzna. Jak poprzednio, wszystko inne jest stałe; stąd całkowita wydajność czasowa sortowania pęcherzyków jest rzędu (N - 1) × (N - 1), czyli N2 - 2N + 1. W tym przypadku N2 nazywamy wyrazem dominującym wyrażenia, co oznacza, że inne części, a mianowicie -2N i +1, zostają „połknięte" przez N2, gdy używa się notacji Big-O. W konsekwencji sortowanie bąbelkowe jest algorytmem O(N2), czyli czasu kwadratowego. Przypomnij sobie naszą dyskusję na temat ulepszonej wersji sortowania bąbelkowego, która umożliwiała przechodzenie przez coraz mniejsze fragmenty listy wejściowej. Pierwsze przejście składa się z N − 1 elementów, następne z N − 2 elementów i tak dalej. Dlatego analiza czasu wyniesie w sumie:

(N − 1) + (N − 2) + (N − 3) + ...·+3 + 2 + 1

które można wykazać, aby ocenić do (N2 − N)/2. To mniej niż N2 − 2N + 1 naiwnej wersji, ale nadal jest kwadratowa, ponieważ dominującym czynnikiem jest N2/2, a stały czynnik 12 również zostaje utracony. W ten sposób mamy 50% skrócenie czasu, ale nie dużą poprawę. Łatwo zauważyć, że prosty algorytm sumowania wynagrodzeń (rysunek 2.3) jest liniowy, to znaczy O(N), ale algorytm dla bardziej zaawansowanej wersji, która pociągała za sobą zagnieżdżoną pętlę do wyszukiwania bezpośrednich menedżerów , jest kwadratowy . (Dlaczego?) Algorytm wyszukiwania „pieniędzy", z zastosowaniem procedury lub bez niej może być postrzegany jako liniowy, co należy dokładnie sprawdzić; chociaż mogą istnieć dwa wskaźniki, które rozwijają się oddzielnie, tekst jest przeszukiwany tylko raz, w sposób liniowy. Mając wiedzę zdobytą w niniejszym rozdziale, powinno być całkiem proste ustalenie powodu, dla którego próbowaliśmy ulepszyć algorytm maksymalnej odległości wielokątnej w Części 4. Algorytm naiwny, który przebiega przez wszystkie pary wierzchołków, jest kwadratowy (gdzie N przyjmuje się, że jest liczbą wierzchołków), podczas gdy ulepszony algorytm wykonuje cykle wokół wielokąta tylko raz i można wykazać, że jest liniowy.

Analiza czasu rekurencji

Rozważmy teraz problem min&max z Części 4, który wymagał znalezienia elementów ekstremalnych na liście L. Algorytm naiwny przechodzi przez listę iteracyjnie, aktualizując dwie zmienne, które przechowują bieżące elementy ekstremalne. Jest wyraźnie liniowy. Oto procedura rekurencyjna, która w Rozdziale 4 została uznana za lepszą:

podprogram znajdź-min&maks-z 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.

Okaże się, że ta rekurencyjna procedura również działa w czasie liniowym. (W rzeczywistości, jak przekonujemy później, żaden algorytm dla problemu min&max nie może być podliniowy, powiedzmy logarytmiczny.) Jednak procedura rekurencyjna ma mniejszą stałą pod Big-O. Aby zobaczyć dlaczego, potrzebna jest bardziej wyrafinowana analiza złożoności. Algorytm iteracyjny działa, przeprowadzając dwa porównania dla każdego elementu na liście, jedno z bieżącym maksimum, a drugie z bieżącym minimum. Stąd daje całkowitą liczbę porównawczą 2BA. Interesującą częścią jest sposób, w jaki liczy się czas dla rutyny rekurencyjnej. Ponieważ nie wiemy od ręki, ile porównań rutyna wymaga, używamy specjalnie dostosowanej notacji abstrakcyjnej: niech C(N) oznacza (najgorszy przypadek) liczbę porównań wymaganych przez rekurencyjną procedurę min&max na listach o długości N. Teraz, chociaż nie znamy wyraźnej wartości C(N) jako funkcja N wiemy dwie rzeczy:

1. Jeśli N wynosi 2, to przeprowadzane jest dokładnie jedno porównanie - to, które wynika z wiersza (1) procedury; jeśli N wynosi 3, przeprowadzane są trzy porównania, co można zweryfikować.

2. Jeżeli N jest większe od 3, to przeprowadzone porównania składają się dokładnie z dwóch zestawów porównań dla list o długości N/2, ponieważ są dwa wywołania rekurencyjne i dwa dodatkowe porównania — te pojawiające się w wierszach (2.4) i (2.5). (Jeśli N jest nieparzyste, listy mają długość (N + 1)/2 i (N − 1)/2.) Możemy zatem zapisać następujące równania, czasami nazywane relacjami powtarzalności, które wychwytują odpowiednio dwie obserwacje właśnie zrobiłem:

i. $C(2) = 1$

ii. $C(N) = 2 \times C(N/2) + 2$

(Dotyczy to przypadku, w którym N jest potęgą liczby 2. Ogólny przypadek jest nieco bardziej skomplikowany.) Chcielibyśmy znaleźć funkcję C(N) spełniającą te ograniczenia. I rzeczywiście, istnieją metody rozwiązywania takich równań rekurencyjnych iw tym przypadku, jak łatwo sprawdzić, rozwiązanie okazuje się być:

$C(N) = 3N/2 - 2$

Oznacza to, że C(N) jest mniejsze niż 1,5N. Nawet w ogólnym przypadku, dla liczb, które nie są potęgami 2, procedura rekurencyjna wymaga mniej niż 1,7N porównań dla list o długości N, co jest lepsze niż 2N wymagane przez rozwiązanie iteracyjne. Jak wspomniano, jest to nadal O(N), niemniej jednak jest to poprawa, zwłaszcza jeśli porównania są bardzo czasochłonne w pożądanej aplikacji. Przedstawiono tutaj procedurę rekurencyjną min&max, aby dać przykład analizy czasowej algorytmu rekurencyjnego. Warto jednak zauważyć, że faktycznie możemy osiągnąć lepsze zachowanie, używając tylko 1,5N porównań dla ogólnego przypadku znajdowania minimum i maksimum na liście za pomocą (innego) algorytmu iteracyjnego w następujący sposób. Najpierw ułóż N elementów w pary, a następnie porównaj dwa elementy w każdej parze, zaznaczając większy z nich. Kosztuje to N/2 porównań. Następnie przejdź przez N/2 większe elementy, śledząc bieżące maksimum, i podobnie przez N/2 mniejsze elementy, śledząc minimum. Kosztuje to dwa razy więcej porównań N/2, co daje łącznie 1,5N.

W Częściach 2 i 4 zostały pokrótce opisane dwa dodatkowe algorytmy sortowania, sortowanie po drzewach i sortowanie przez scalanie. Nie będziemy tutaj zajmować się szczegółową analizą czasu, z wyjątkiem następujących uwag: Sortowanie drzew, jeśli jest zaimplementowane naiwnie, jest algorytmem kwadratowym. Powód wynika z możliwości, że pewne sekwencje elementów dają w wyniku bardzo długie i wąskie binarne drzewa poszukiwań. Chociaż przechodzenie przez takie drzewo w pierwszym przejściu w lewo jest procedurą w czasie liniowym, można wykazać, że konstruowanie drzewa z sekwencji wejściowej jest kwadratowe. Możliwe jest jednak zastosowanie schematu samodopasowania, omówionego w Części 2 i trzymaj drzewo szerokie i krótkie. Włączenie takiego schematu do fazy konstrukcyjnej sortowania drzew spowoduje poprawę o rząd wielkości, dając algorytm, który używa czasu w kolejności iloczynu N i logarytmu N (a nie N); w symbolach jest to algorytm O(N × log N). Poniższa tabela powinna dać wyobrażenie o oszczędnościach, jakie zapewnia ta poprawa, chociaż jej wpływ na czynniki stałe nie jest widoczny. Przechodząc do sortowania przez scalanie, pozostaje ci interesujące ćwiczenie pokazujące, że ten algorytm sam w sobie jest O(N × log N). Mergesort jest tak naprawdę jednym z kilku wydajnych czasowo algorytmów sortowania i jest zdecydowanie najłatwiejszy do opisania spośród algorytmów O(N × log N). Należy jednak zauważyć, że wykorzystuje nową listę do przechowywania wyników cząstkowych, a zatem wymaga dodatkowej liniowej ilości miejsca, co jest wadą w porównaniu z niektórymi innymi metodami sortowania. W ten sposób mergesort jest jednym z najbardziej efektywnych czasowo procedur sortowania, a także jednym z najłatwiejszych do opisania. Chcielibyśmy przypisać rekurencji obie zalety: ułatwia ona opisanie algorytmu, a także zapewnia przejrzysty mechanizm dzielenia problemu na dwa mniejsze problemy o połowę mniejsze, co stanowi pierwiastek O(N × log) algorytmu N) wydajność czasowa. Inne podejście wykorzystuje stertę zamiast binarnego drzewa wyszukiwania, jak w treesort. Algorytm heapsort najpierw wstawia wszystkie elementy listy wejściowej do sterty, a następnie wielokrotnie wyodrębnia minimalny element, aby utworzyć posortowaną listę. Reprezentacja sterty jako wektora (patrz rozdział 4) zapewnia, że sterta jest zawsze zrównoważona, a zatem każda operacja wstawiania i wydobywania z minimum zajmuje czas logarytmiczny, dając całkowity czas działania O(N × log N).

Złożoność średnich przypadków

Najgorszy przypadek naszych analiz czasowych można interpretować jako wadę. Prawdą jest, że nie można zagwarantować, powiedzmy, wydajności algorytmu w czasie liniowym, chyba że ograniczenie czasowe ma zastosowanie do wszystkich danych wejściowych zgodnych z prawem. Może się jednak zdarzyć, że algorytm jest bardzo szybki dla większości standardowych rodzajów danych wejściowych, a te, które powodują wzrost wydajności w najgorszym przypadku, są mniejszością, którą jesteśmy skłonni zignorować. W związku z tym istnieją inne przydatne szacunki wydajności czasowej algorytmu, takie jak jego zachowanie w średniej wielkości. Tutaj interesuje nas czas wymagany przez algorytm przeciętnie, biorąc pod uwagę cały zestaw danych wejściowych i ich prawdopodobieństwo wystąpienia. Nie będziemy tu wchodzić w szczegóły techniczne, z wyjątkiem uwagi, że analiza przeciętnego przypadku jest znacznie większa trudne do przeprowadzenia niż analiza najgorszego przypadku. Wymagana matematyka jest zwykle znacznie bardziej wyrafinowana i istnieje wiele algorytmów, dla których badacze w ogóle nie byli w stanie uzyskać średnich szacunków. Pomimo fundamentalnej różnicy, wiele algorytmów ma ten sam limit czasu Big-O zarówno dla zachowania najgorszego, jak i średniego przypadku. Na przykład proste sumowanie wynagrodzeń działa w ustalony sposób, zawsze biegnąc do samego końca listy, a zatem jest liniowe w najgorszych, najlepszych i przeciętnych przypadkach. Wersja, która szuka bezpośrednich menedżerów, która w najgorszym przypadku jest kwadratowa, może przeciętnie wymagać przejrzenia tylko połowy listy dla każdego menedżera pracownika, a nie całej. Jednak to tylko zmniejsza N^2 do około $N^2/2$, zachowując ograniczenie czasowe $O(N^2)$ nawet w przeciętnym przypadku, ponieważ stała połowa jest ukryta pod big-O. W przeciwieństwie do tego, w przypadku niektórych algorytmów analiza średnich przypadków

może ujawnić znacznie lepszą wydajność. Klasycznym tego przykładem jest jeszcze inny algorytm sortowania, zwany quicksort, którego nie będziemy tutaj opisywać. Quicksort, również naturalnie rekurencyjny algorytm, ma najgorszy przypadek kwadratowy czasu działania, a zatem wydaje się być gorszy zarówno od sortowania przez scalanie, jak i samodostosowującej się wersji sortowania po drzewach. Niemniej jednak można wykazać, że jego wydajność w przypadku średnich przypadków wynosi O (N × log N), co odpowiada wydajności lepszych algorytmów sortowania. Co sprawia, że quicksort jest szczególnie interesujące, to fakt, że jego wydajność w przypadku średniej wielkości obejmuje bardzo małą stałą. W rzeczywistości, tylko przy porównaniach zliczania, wydajność czasowa szybkiego sortowania wynosi średnio nieco ponad 1,4N × log N. Biorąc pod uwagę fakt, że wymaga tylko niewielkiej stałej ilości dodatkowej przestrzeni dyskowej i pomimo gorszej wydajności w najgorszym przypadku, quicksort jest w rzeczywistości jednym z najlepszych znanych algorytmów sortowania i zdecydowanie najlepszym spośród wymienionych tutaj. Aby uzupełnić naszą krótką dyskusję na temat przedstawionych metod sortowania, powinniśmy zauważyć, że sortowanie bąbelkowe jest najgorszym z wielu, ma nawet przeciętne zachowanie O(N2). (Ten fakt jest jednak dość trudny do udowodnienia.) Wiele osób sprzeciwia się opisaniu bąbelkowego sortowania na kursach informatyki, ponieważ jego elegancja jest wystarczająco myląca, a studenci mogą faktycznie zostać zwabieni do używania go w praktyce.

Górna i dolna granica

Pokazaliśmy wcześniej, jak naiwny algorytm czasu liniowego do przeszukiwania uporządkowanej listy można ulepszyć do czasu logarytmicznego za pomocą wyszukiwania binarnego. Dokładniej, pokazaliśmy, że istnieje algorytm, który przeprowadza takie wyszukiwanie, używając nie więcej niż log2 N porównań w najgorszym przypadku na liście o długości N. Czy możemy zrobić lepiej? Czy w najgorszym przypadku możliwe jest wyszukanie elementu w książce telefonicznej o milionach wpisów z mniej niż 20 porównaniami? Czy można znaleźć algorytm dla problemu przeszukiwania uporządkowanej listy, który wymaga tylko $\sqrt{\log_2 N}$), a może tylko $\log_2(\log_2 N)$, porównania w najgorszym przypadku? Aby spojrzeć na te pytania z odpowiedniej perspektywy, wyobraźmy sobie, że każdy problem algorytmiczny znajduje się w miejscu, wyposażony w nieodłączne optymalne rozwiązanie, do którego dążymy. Następnie pojawia się ktoś z algorytmem, powiedzmy $O(N^3)$ (określany jako czas sześcienny), a tym samym zbliża się do pożądanego rozwiązania „z góry". Mając ten algorytm jako dowód, wiemy, że problem nie może wymagać czasu działania wyższego niż sześcienny; nie może być gorszy niż O(N3). Później ktoś inny odkrywa lepszy algorytm, powiedzmy taki, który działa w czasie kwadratowym, zbliżając się w ten sposób do pożądanego optymalnego rozwiązania, również z góry. Jesteśmy teraz przekonani, że problem nie może być z natury gorszy niż $O(N^2)$, a poprzedni algorytm staje się przestarzały. Pytanie brzmi, jak daleko w dół mogą zajść te ulepszenia? Mając na uwadze metaforę „podejścia z góry" , mówi się, że odkrycie algorytmu nakłada górną granicę na problem algorytmiczny. Lepsze algorytmy sprowadzają najbardziej znane ograniczenie czasowe problemu w dół, bliżej nieznanej, nieodłącznej złożoności samego problemu. Pytania, które zadajemy, dotyczą dolnej granicy problemu. Jeśli potrafimy rygorystycznie udowodnić, że problem algorytmiczny P nie może być rozwiązany przez żaden algorytm, który w najgorszym przypadku wymaga mniej niż, powiedzmy, czasu kwadratowego, to ludzie próbujący znaleźć wydajne algorytmy do rozwiązania P mogą zrezygnować, jeśli i kiedy znajdą algorytm czasowy, bo nie ma mowy, żeby mogli zrobić to lepiej. Taki dowód stanowi dolne ograniczenie problemu algorytmicznego, ponieważ pokazuje, że żaden algorytm nie może poprawić ograniczenia $O(N^2)$. W ten sposób odkrycie sprytnego algorytmu pokazuje, że nieodłączna wydajność czasowa problemu nie jest gorsza niż niektóre ograniczenia, a odkrycie dowodu dolnego ograniczenia pokazuje, że nie jest on lepszy niż niektóre ograniczenia. W obu przypadkach odkryto właściwość problemu algorytmicznego, a nie właściwość konkretnego algorytmu. Może to zabrzmieć mylące, zwłaszcza że dolna granica problemu wymaga rozważenia wszystkich

algorytmów dla niego, podczas gdy górną granicę osiąga się, konstruując jeden konkretny algorytm i analizując jego wydajność czasową . Osiągnięcie dolnej granicy wydaje się niemożliwe. Jak udowodnić coś na temat wszystkich algorytmów? Skąd możemy mieć pewność, że ktoś nie odkryje bardzo subtelnego, ale wydajnego algorytmu, którego nie przewidzieliśmy? Nie są to łatwe pytania, ale być może kontemplacja poniższego przykładu da nam częściowe odpowiedzi.

Dolna granica wyszukiwania w książce telefonicznej

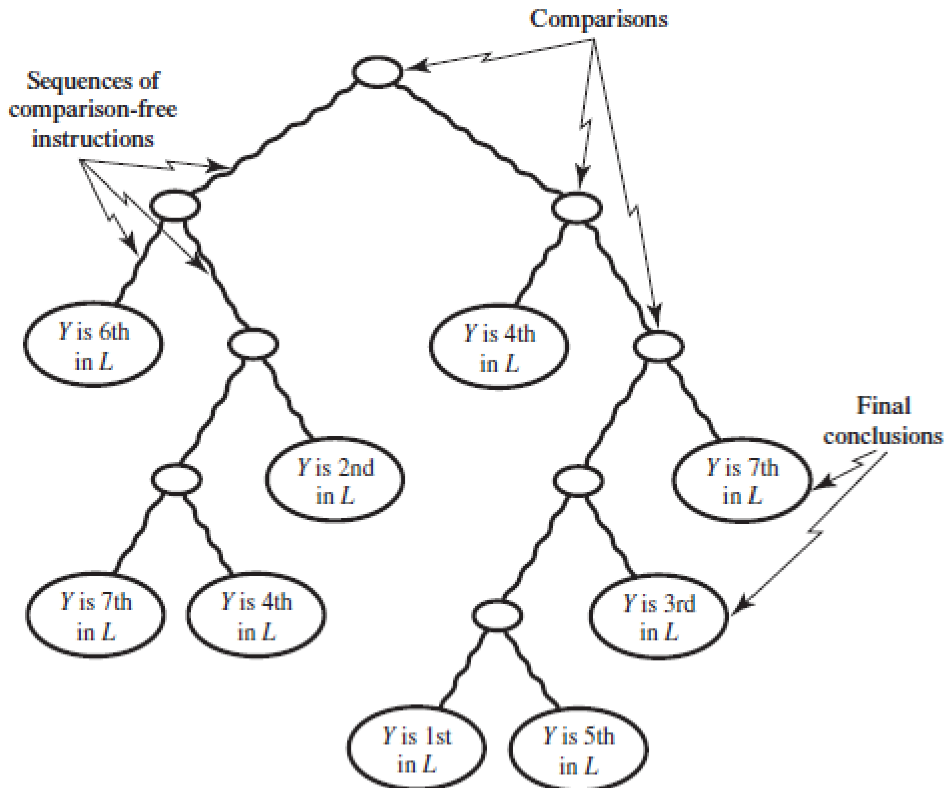
Zanim jednak przytoczymy przykład, powinniśmy ponownie podkreślić fakt, że wszelkie omówienie nieodłącznej złożoności czasowej, w tym dolnych granic, musi być przeprowadzone w odniesieniu do zestawu dozwolonych instrukcji podstawowych. Nikt nie może udowodnić, że problem wyszukiwania wymaga więcej niż jednej jednostki czasu, jeśli podstawowe instrukcje, takie jak:

lista wyszukiwania L dla pozycji Y

są dozwolone. Dlatego zasady gry muszą być dokładnie określone, zanim podejmiemy jakąkolwiek próbę udowodnienia niższych granic. Teraz chcemy pokazać, że wyszukiwanie binarne jest optymalne. Innymi słowy, chcielibyśmy udowodnić, że generalnie nie możemy znaleźć nazwiska w książce telefonicznej o długości N w porównaniach mniejszych niż log2 N w najgorszym przypadku; 20 porównań to minimum dla miliona nazwisk, 30 dla miliarda, 60 dla miliarda miliardów i tak dalej. Mówiąc bardziej ogólnie, nie ma algorytmu dla problemu przeszukiwania uporządkowanej listy, którego wydajność w najgorszym przypadku w czasie jest lepsza niż logarytmiczna.

Zasady gry są tutaj dość proste. Jedynym sposobem, w jaki proponowany algorytm może wydobyć jakiekolwiek informacje z danych wejściowych, jest przeprowadzenie dwukierunkowych porównań. Algorytm może porównać między sobą dwa elementy z listy wejściowej L lub element z L z elementem wejściowym Y, którego pozycja w L jest poszukiwana. Żadne inne zapytania nie mogą być kierowane do wejść L i Y . Jednak nie ma ograniczeń dotyczących instrukcji niezwiązanych z L lub Y ; każdy z nich może zostać uwzględniony, a każdy będzie kosztować tylko jedną jednostkę czasu. Aby pokazać, że dowolny algorytm dla tego problemu wymaga porównań log2N, będziemy argumentować następująco. Mając dany algorytm, dowolny algorytm, pokażemy, że nie może on w żaden sposób poprawić deklarowanej dolnej granicy log2N dla problemu. W związku z tym załóżmy, że wymyśliliśmy algorytm A dla problemu przeszukiwania uporządkowanej listy, który jest oparty na porównaniach dwukierunkowych. Udowodnimy, że istnieją listy wejściowe o długości N, na których algorytm A z konieczności przeprowadzi porównania $\log_2 N$.

W tym celu przyjrzyjmy się zachowaniu A na typowej liście L o długości N. Aby ułatwić sobie życie, pozostańmy przy szczególnym przypadku, w którym L składa się dokładnie z liczb 1, 2, 3, ... , N w kolejności i gdzie Y jest jedną z tych liczb. Możemy również bezpiecznie założyć, że wszystkie porównania mają tylko dwa wyniki, „mniejsze niż” lub „więcej niż”, ponieważ jeśli porównanie daje „równe”, Y zostało znalezione (chyba że przeprowadzono jakieś bezużyteczne porównanie elementu z samym sobą, w którym to przypadku algorytm może się bez niego obejść). Algorytm A, pracujący na naszej specjalnej liście L, może wykonać kilka innych akcji, ale w końcu osiąga swoje pierwsze porównanie. Następnie przypuszczalnie rozgałęzia się w jednym z dwóch kierunków, w zależności od (dwukierunkowego) wyniku. W każdym z nich A może jeszcze trochę popracować przed kolejnym porównaniem, które ponownie dzieli zachowanie A na dwie dalsze możliwości. Zachowanie A można zatem schematycznie opisać jako drzewo binarne w którym krawędzie odpowiadają możliwym sekwencjom działań nieporównywania, a węzły odpowiadają porównaniom i ich wynikom rozgałęziania.



To drzewo przechwytuje zachowanie A dla wszystkich możliwych list składających się z liczb 1, 2, 3, ..., N. Teraz drzewo jest zdecydowanie skończone, ponieważ A musi zakończyć się na wszystkich dozwolonych danych wejściowych, a jego liście odpowiadają osiągnięciu ostatecznej decyzji dotyczącej pozycji Y w L, czyli innymi słowy, dotyczącej wartości Y z wśród 1, 2, ... , N. Oczywiście nie ma dwóch różnych wartości Y, które mogą być reprezentowane przez ten sam liść, ponieważ oznaczałoby to, że A daje takie same dane wyjściowe (czyli pozycję w L) dla dwóch różnych wartości Y - oczywiście absurd. Ponieważ Y ma N możliwych wartości, drzewo musi mieć co najmniej N możliwych liści. (Może mieć więcej, ponieważ może istnieć więcej niż jeden sposób wyciągnięcia tego samego wniosku na temat pozycji Y.) Używamy teraz następującego standardowego faktu o drzewach binarnych, których prawdziwość powinieneś spróbować ustalić:

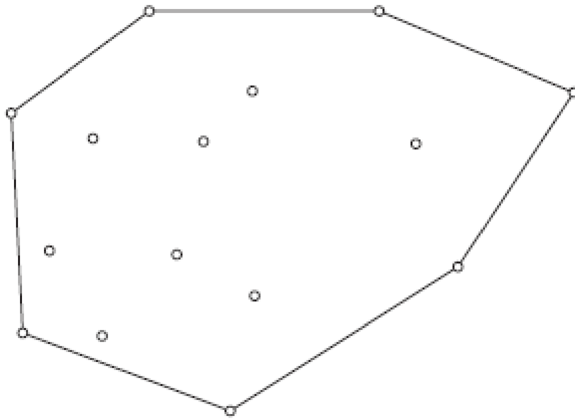
Każde drzewo binarne mające co najmniej N liści ma głębokość co najmniej $\log_2 N$; oznacza to, że drzewo zawiera co najmniej jedną ścieżkę od korzenia, której długość wynosi $\log_2 N$ lub więcej. Wynika z tego, że nasze drzewo porównawcze zawiera ścieżkę o długości co najmniej $\log_2 N$. Ale ścieżka w drzewie odpowiada dokładnie wykonaniu proponowanego algorytmu A na jakiejś liście wejściowej o długości N, przy czym węzły oznaczają porównania dokonane w tym właśnie wykonanie . Stąd, jeśli w drzewie istnieje ścieżka o długości co najmniej $\log_2 N$, wynika z tego, że istnieje pewna lista wejściowa, która wymaga od A wykonania co najmniej log2N porównań. To kończy dowód, że log2N jest dolnym ograniczeniem liczby porównań, a zatem, że algorytmiczny problem wyszukiwania w uporządkowanej liście ma dolne ograniczenie w czasie logarytmicznym. Zwróć uwagę, że argument został sformułowany na tyle ogólnie, aby można go było zastosować do dowolnego proponowanego algorytmu, o ile używa porównań jako jedynego środka do wyodrębnienia informacji z listy wejściowej.

Zamknięte problemy i luki algorytmiczne

Dolne granice można ustalić dla wielu innych problemów algorytmicznych. Na przykład wyszukiwanie na nieuporządkowanej liście jest kwintesencją problemu czasu liniowego i można łatwo wykazać, że w najgorszym przypadku wymaga N porównań. Dlatego ma dolną granicę O(N) dla algorytmów opartych na porównaniach. Możesz chcieć wypracować argument na to. Kilka problemów, o których mówiliśmy, może być interpretowanych jako odmiany nieuporządkowanego wyszukiwania i w konsekwencji są one również ograniczone od dołu przez O(N). Należą do nich problem min&max, proste sumowanie wynagrodzeń, maksymalna odległość wielokąta i tak dalej. Zauważ, że dla każdego z nich rzeczywiście dostarczyliśmy algorytmy czasu liniowego. Oznacza to, że górna i dolna granica faktycznie się spotykają (poza możliwymi różnymi czynnikami stałymi). Innymi słowy, te problemy algorytmiczne są zamknięte, jeśli chodzi o oszacowania czasu Big-O. Mamy algorytm liniowy i wiemy, że nie możemy zrobić nic lepszego. Wyszukiwanie w uporządkowanej liście, jak pokazaliśmy, jest również zamkniętym problemem; ma górną i dolną granicę czasu logarytmicznego. Sortowanie też jest zamknięte: z jednej strony mamy algorytmy, takie jak mergesort, heapsort lub samodostosowująca się wersja treesort, które są O(N × log N), a z drugiej możemy udowodnić O(N × log N). N) dolna granica sortowania listy o długości N. W obu tych przypadkach granice są oparte na modelu porównawczym, w którym informacje o danych wejściowych uzyskuje się tylko przez porównania dwukierunkowe. Wiele problemów algorytmicznych nie ma jednak jeszcze właściwości zamknięcia. Ich górna i dolna granica się nie spotykają. W takich przypadkach mówimy, że powodują one luki algorytmiczne, przy czym najlepiej znana górna granica różni się od (a zatem jest wyższa) od najlepiej znanej dolnej granicy. W Części 4 przedstawiliśmy kwadratowy algorytm znajdowania minimalnych linii kolejowych (problem minimalnego drzewa rozpinającego), ale najbardziej znane dolne ograniczenie jest liniowe. To znaczy, chociaż możemy udowodnić, że problem wymaga czasu O(N) (tu N jest liczbą krawędzi w linii kolejowej wykresu, a nie liczby węzłów), nikt nie zna algorytmu czasu liniowego, a zatem problem nie jest zamknięty. W Części 7 zobaczymy kilka uderzających przykładów luk algorytmicznych, które są niedopuszczalnie duże. Na razie wystarczy zdać sobie sprawę, że jeśli z jakiegoś problemu powstaje luka algorytmiczna, to niedoskonałości nie tkwi w samym problemie, ale w naszej wiedzy na jego temat. Nie udało nam się albo znaleźć najlepszego algorytmu do tego, albo udowodnić, że lepszy algorytm nie istnieje, albo w obu przypadkach.

Zabarykadowanie śpiących tygrysów: przykład

Poniższy algorytm wykorzystuje sortowanie w dość nieoczekiwany sposób i dziedziczy również granice złożoności. Załóżmy, że mamy do czynienia z N śpiącymi tygrysami i aby uniknąć zjedzenia, gdy jeden lub więcej z nich się obudzi, jesteśmy zainteresowani zbudowaniem wokół nich ogrodzenia. Algorytmicznie otrzymujemy dokładną lokalizację tygrysów i chcemy znaleźć najmniejszy wielokąt, który je wszystkie otacza. Oczywiście problem sprowadza się do znalezienia minimalnej sekwencji lokalizacji niektórych tygrysów, które połączone liniowymi fragmentami ogrodzenia (które nazwiemy segmentami linii) obejmą wszystkie pozostałe (patrz rysunek 1). Ta obudowa nazywana jest wypukłą kadłubem zbioru punktów. Zauważ, że ten problem został przedstawiony w zoologicznej postaci tylko po to, by brzmiał trochę bardziej zabawnie. Problem wypukłego kadłuba jest właściwie jednym z podstawowych problemów pojawiających się w grafice komputerowej. Znalezienie szybkich algorytmów dla tego problemu i wielu innych problemów w geometrii obliczeniowej może mieć dużą różnicę w szybkości i możliwości zastosowania wielu aplikacji graficznych. Przedstawiony teraz algorytm opiera się na następującej prostej obserwacji. Aby jakiś odcinek łączący dwa punkty był częścią wypukłego kadłuba, konieczne i wystarczające jest, aby wszystkie pozostałe punkty znajdowały się po tej samej stronie (a raczej jego przedłużenia do pełnej linii).



Ta obserwacja daje początek prostemu algorytmowi: rozważ po kolei każdy potencjalny odcinek linii i sprawdź, czy wszystkie N − 2 punkty, które nie znajdują się na nim, są po jednej jego stronie. Test decydujący, do której strony linii należy dany punkt, można łatwo przeprowadzić w stałym czasie przy użyciu elementarnej geometrii analitycznej. Ponieważ jest N punktów, to jest N2 odcinków potencjału, z których jeden może łączyć każdą parę punktów i każdy z nich musi być porównywany z N − 2 punktami. Daje to sześcienny (czyli algorytm O(N3)), jak łatwo zauważyć. Jest jednak sposób na zrobienie tego znacznie lepiej. Oto główne kroki w proponowanym algorytmie:

(1) znajdź „najniższy" punkt P_1;

(2) posortuj pozostałe punkty według wielkości kąta, jaki tworzą z osią poziomą w połączeniu z P_1 i niech otrzymaną listą będzie P2, ... , PN ;

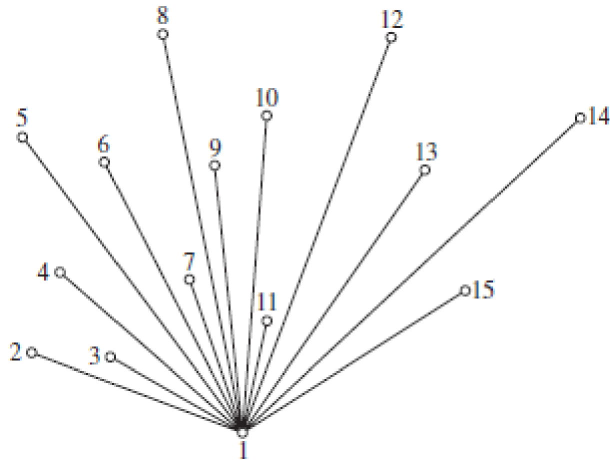
(3) wystartować z P_1 i P_2 w obecnym kadłubie;

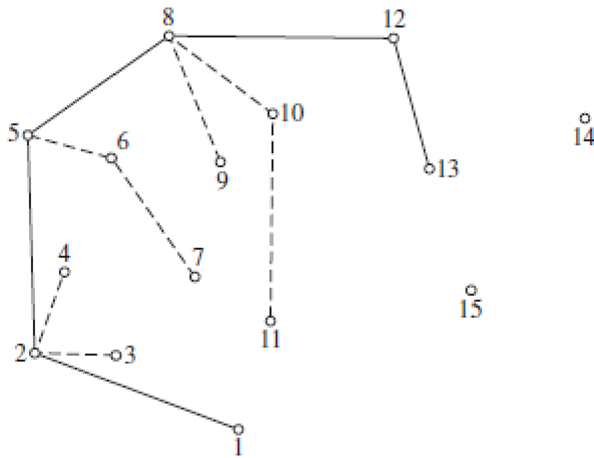
(4) dla I od 3 do N wykonaj następujące czynności:

(4.1) wstępnie dodać P_I do aktualnego kadłuba;

(4.2) działa wstecz przez bieżący kadłub, eliminując punkt PJ, jeśli dwa punkty P1</> i PI leżą po różnych stronach linii między P_{J-1} i P_J , i kończąc skanowanie wstecz, gdy P_J, którego nie trzeba eliminować napotkany.

Może trochę trudno zobaczyć, co robi algorytm, ale rzut oka na rysunki 2 i 3 powinien pomóc.





Rysunek 2 pokazuje punkty posortowane po kroku (2) w kolejności, w jakiej będą brane pod uwagę w kroku (4), a rysunek 3 pokazuje kilka pierwszych dodanych i wyeliminowanych w kroku (4). Na rysunku 3 właśnie dodano punkt P_{13}, a następny krok będzie uwzględniał P_{14}. Jednakże, ponieważ linia między P_{12} i P_{13} przebiega między P_1 i P_{14} punkt P_{13} zostanie wyeliminowany, sprowadzając kadłub wokół punktów powyżej P13. Choć udowodnienie poprawności tego algorytmu jest ćwiczeniem pouczającym, bardziej interesuje nas jego efektywność czasowa. Przyjmiemy zatem, że algorytm jest poprawny i skoncentrujemy się na jego analizie czasowej. Łatwo zauważyć, że krok (1), który polega na prostym wyszukaniu punktu, który leży najniżej na obrazie, czyli tego, który ma najmniejszą współrzędną pionową, zajmuje czas liniowy. Teraz krok (2), sortowanie, można przeprowadzić przez dowolne wydajne sortowanie. W związku z tym, ponieważ obliczenie kąta lub porównanie dwóch kątów można przeprowadzić w stałym czasie, krok (2) zajmuje całkowity czas $O(N \times \log N)$. Interesująca część dotyczy kroku (4). Wygląda na to, że zagnieżdżona struktura pętli kroku (4) zapewnia wydajność w czasie kwadratowym. Jednak właściwym sposobem analizy tej części algorytmu jest zlekceważenie jego struktury i rozliczenie czasu punkt po punkcie. Zauważ, że krok (4.2) eliminuje tylko punkty i zatrzymuje się, gdy napotkany zostanie punkt, którego nie należy eliminować. Teraz, ponieważ żaden punkt nie zostanie wyeliminowany więcej niż raz,łączna liczba punktów branych pod uwagę w zagnieżdżonej pętli kroku (4.2) nie może być większa niż O(N). Ponieważ wszystkie pozostałe części kroku (4) również zabierają nie więcej niż czas liniowy, krok (4) w całości zajmuje czas liniowy. Całkowity czas działania algorytmu wynosi zatem:

Krok (1) O(N)

Krok (2) O(N × log N)

Krok (3) O(1)

Krok (4) O(N)

Razem O(N × log N)

To trochę zaskakujące, że sortowanie ma w ogóle coś wspólnego ze znalezieniem wypukłego kadłuba. Bardziej zaskakujące jest to, że sortująca część algorytmu jest w rzeczywistości dominującą częścią, jeśli chodzi o złożoność obliczeniową.

Badania efektywności algorytmów

Znalezienie wydajnych algorytmów do rozwiązywania problemów algorytmicznych jest jednym z najczęstszych tematów badawczych w informatyce. Rzeczywiście, prawie wszystkie zagadnienia omawiane w tym rozdziale są przedmiotem znacznych wysiłków badawczych, a większość z nich została zebrana pod ogólnym terminem teoria konkretnej złożoności. W tej dziedzinie ludzie są zainteresowani opracowywaniem i wykorzystywaniem struktur danych i metod algorytmicznych w celu ulepszania istniejących algorytmów lub opracowywania nowych. Wiele genialnych pomysłów trafia do wyrafinowanych algorytmów, zapewniając czasami zaskakujące skrócenie czasu działania. Najczęściej odbywa się to z praktycznym celem szybszego rozwiązania problemu za pomocą komputera. Jednak czasami siłą napędową jest po prostu chęć przygwożdżenia nieodłącznej złożoności problemu o rząd wielkości, nawet jeśli wynika to z praktycznego punktu widzenia, nie są lepsze niż te znane wcześniej. Dobrym tego przykładem jest problem układu linii kolejowych (drzewo opinające) z Części 4. Przedstawiliśmy dla tego algorytm w czasie kwadratowym, który z pewnym wysiłkiem można ulepszyć do O(N × log N). Ze wszystkich praktycznych celów ten ulepszony algorytm lub dowolny z wielu innych algorytmów o podobnej złożoności działa dobrze. Jednak teoretycy złożoności nie są z tego zadowoleni, ponieważ najlepiej znana dolna granica jest liniowa. Chcą wiedzieć, czy rzeczywiście można znaleźć algorytm czasu liniowego dla tego problemu. Ostatnio, przy użyciu dość sprytnych i skomplikowanych technik, górna granica została zbliżona do O(N). W szczególności istnieje algorytm, który działa w czasie ograniczonym przez O(f (N) × N), gdzie f (N) jest funkcją, która rośnie niewiarygodnie wolno - dużo, dużo wolniej niż, powiedzmy, log N. Poniższa tabela pokazuje najmniejsze Ns dla kilku pierwszych wartości tej funkcji:

Najmniejsze N : takie, że f (N) to

4 : 2

16 : 3

64 000 : 4

znacznie więcej niż całkowita liczba cząstek w znanym wszechświecie; 5

absolutnie niewyobrażalne : 6

Oczywiście, dla wszystkich praktycznych celów algorytm O(f (N) × N) jest naprawdę liniowy; wartość f (N) wynosi 5 lub mniej dla dowolnej liczby, którą kiedykolwiek będziemy zainteresowani. Musimy jednak pamiętać, że bez względu na to, jak wolno rośnie f (N), w końcu stanie się ono większe niż jakakolwiek stała. Stąd od pewnego momentu, to znaczy dla wszystkich wystarczająco dużych N, każdy dany algorytm czasu liniowego przewyższa algorytm O(f (N) × N) i mówimy, że pierwszy jest

asymptotycznie lepszy od drugiego. Tym samym problem drzewa opinającego jest nadal otwarty, ponieważ wciąż tworzy lukę algorytmiczną, z której badacze nie zrezygnowali. Mamy nadzieję, że w dającej się przewidzieć przyszłości problem zostanie w taki czy inny sposób zamknięty. Albo zostanie odkryty algorytm czasu liniowego, albo zostanie znaleziony dowód nieliniowości dolnej granicy, pasujący do znanej górnej granicy. Jak wspomniano wcześniej, dolne granice są bardzo trudne do znalezienia, a dla wielu problemów nikt nie zna żadnych nieliniowych granic dolnych. Innymi słowy, pomimo tego, że najlepsze algorytmy dla niektórych problemów są kwadratowe, sześcienne lub gorzej, nikt nie jest w stanie udowodnić, że nie istnieje dla nich jakiś algorytm czasu liniowego, który czekałby na odkrycie. Metody udowadniania nieliniowych dolnych granic są niezwykle rzadkie i badacze dokładają wszelkich starań, aby je znaleźć. Wydajność dla przypadków średnich prowadzi do kolejnego trudnego kierunku badań, a wciąż istnieje wiele znanych algorytmów, dla których nie przeprowadzono zadowalających analiz przypadków średnich. W ogóle nie omawialiśmy tutaj złożoności przestrzennej, ale warto stwierdzić, że osiągnięcie dobrych górnych i dolnych granic wymagań dotyczących przestrzeni pamięciowej problemów algorytmicznych jest również przedmiotem wielu badań nad konkretną złożonością. Ludzi interesują nie tylko oddzielne granice w czasie i przestrzeni, ale także wspólna czasowo-przestrzenna złożoność problemu. Może być możliwe osiągnięcie, powiedzmy, górnego ograniczenia w przestrzeni liniowej na problem za pomocą jakiegoś algorytmu i, powiedzmy, górnego ograniczenia w czasie kwadratowym za pomocą innego algorytmu, ale to nie implikuje istnienia algorytmu, który osiąga oba jednocześnie. Możliwe, że problem wiąże się z kompromisem między czasem a przestrzenią, co oznacza, że płacimy za oszczędność czasu większą przestrzenią i za gospodarkę przestrzeni większą ilością czasu. W takich przypadkach badacze starają się udowodnić, że istnieje taki kompromis. Zwykle przybiera to formę dowodu, że działanie dowolnego algorytmu rozwiązującego problem spełnia pewne równanie, które występuje jako ograniczenie dolne lub ograniczenie górne (lub oba). Równanie zazwyczaj ujmuje trójczynnikową zależność między wejściową długością N a (najgorszym przypadkiem) czasem działania i przestrzenią pamięci dowolnego algorytmu rozwiązania. Na przykład załóżmy, że następujące równanie zostało ustalone jako górna i dolna granica złożoności czasowo-przestrzennej problemu P:

$S^2 \times T = O(N^3 \times (\log N)2)$

Oznacza to, że jeśli chcemy spędzić $O(N^3)$ czasu, możemy rozwiązać problem używając tylko przestrzeni O(log N), podczas gdy jeśli nalegamy na spędzanie nie więcej niż O(N2) czasu, potrzebowalibyśmy O(√ N × log N) przestrzeń. Kolejne dwa rozdziały skoncentrują się na teorii złożoności złych wiadomości dla nas. Omówią również pojęcia, które są jeszcze bardziej niezawodne niż notacja duże-O.