Michał Włodarczyk
Treść zadania, Opracowanie

Karol Pokorski
Program

Dostępna pamięć: 256 MB. *OI, etap II, dzień pierwszy, 12.02.2014*

Superkomputer

Bajtazar stworzył superkomputer o bardzo nowatorskiej konstrukcji. Może on zawierać wiele (takich samych) procesorów. Każdy procesor może w jednostce czasu wykonać jedną instrukcję.

Programy nie są w tym komputerze wykonywane sekwencyjnie, lecz mają strukturę drzewa. Każda instrukcja programu może mieć zero, jedną lub wiele **instrukcji następujących** *po niej. Po wykonaniu każdej instrukcji należy wykonać instrukcje następujące po niej, jednak można je wykonać w dowolnej kolejności; w szczególności można je wykonywać równolegle na różnych procesorach. Jeśli w superkomputerze jest k procesorów, to w każdej jednostce czasu będzie można wykonać równolegle co najwyżej k instrukcji.*

Bajtazar ma do uruchomienia pewien program. Ponieważ chce optymalnie wykorzystywać swoje zasoby, zastanawia się, jak liczba procesorów wpłynie na szybkość obliczeń. Prosi Cię o określenie, dla danego programu i liczby procesorów, najkrótszego możliwego czasu wykonania tego programu z użyciem superkomputera o tylu procesorach.

Wejście

W pierwszym wierszu standardowego wejścia znajdują się dwie liczby całkowite n i q ($1 \leqslant n, q \leqslant 1\,000\,000$) oddzielone pojedynczym odstępem, oznaczające liczbę instrukcji programu Bajtazara oraz liczbę zapytań.

W drugim wierszu wejścia znajduje się ciąg q liczb całkowitych $k_1, k_2, \ldots, k_q$ ($1 \leqslant k_i \leqslant 1\,000\,000$) pooddzielanych pojedynczymi odstępami: k_i oznacza liczbę procesorów, którymi dysponuje Bajtazar w i-tym zapytaniu.

W trzecim i ostatnim wierszu wejścia znajduje się ciąg $n-1$ liczb całkowitych $a_2, a_3, \ldots, a_n$ ($1 \leqslant a_i < i$) pooddzielanych pojedynczymi odstępami: a_i określa numer instrukcji, po której następuje instrukcja numer i. Instrukcje numerowane są kolejnymi liczbami naturalnymi od 1 do n, przy czym instrukcja numer 1 to pierwsza instrukcja programu.

W testach wartych łącznie 35% punktów zachodzi dodatkowo warunek: $n \leqslant 30\,000$, $q \leqslant 50$. W podzbiorze tych testów wartym łącznie 20% punktów zachodzi dodatkowo warunek: $n \leqslant 1000$, $q \leqslant 10$.

Wyjście

Twój program powinien wypisać na standardowe wyjście jeden wiersz zawierający ciąg q liczb całkowitych pooddzielanych pojedynczymi odstępami: i-ta z tych liczb powinna określać minimalny czas wykonania programu przy założeniu, że superkomputer Bajtazara posiada k_i procesorów.

Przykład

Dla danych wejściowych:

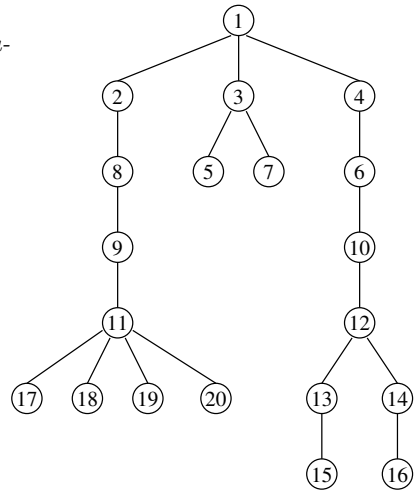
```
20 1
3
1 1 1 3 4 3 2 8 6 9 10 12 12 13 14 11 11 11 11
```

poprawnym wynikiem jest:

```
8
```

Wyjaśnienie do przykładu:
Wykonanie programu może wyglądać następująco:

Czas	*Instrukcje*		
1	*1*		
2	*2*	*3*	*4*
3	*5*	*6*	*7*
4	*8*	*10*	
5	*9*	*12*	
6	*11*	*13*	*14*
7	*15*	*16*	*17*
8	*18*	*19*	*20*



Testy „ocen”:

`1ocen`: *$n = 10$, $q = 2$, drzewo instrukcji jest ścieżką;*

`2ocen`: *$n = 10$, $q = 3$, mały losowy test;*

`3ocen`: *$n = 100$, $q = 3$, wszystkie instrukcje następują po instrukcji nr 1;*

`4ocen`: *$n = 127$, $q = 1$, instrukcje tworzą pełne drzewo binarne;*

`5ocen`: *$n = 1\,000\,000$, $q = 31$, drzewo instrukcji jest długą ścieżką.*

Rozwiązanie

W niniejszym opracowaniu będziemy używać zarówno terminologii z treści zadania, jak i z teorii grafów. Oznaczmy przez $d(x)$ wysokość poddrzewa zaczepionego w wierzchołku-instrukcji x (czyli długość najdłuższej ścieżki biegnącej z x w dół poddrzewa). *Głębokością* x będziemy nazywać długość ścieżki od korzenia (czyli początkowej instrukcji) do x. W szczególności głębokość korzenia wynosi 1. Niech $s(t)$ równa się liczbie wierzchołków na głębokości t. Instrukcję bezpośrednio poprzedzającą x nazwiemy jej *ojcem*, zaś w przypadku pośrednim będziemy mówić o *przodku* x. Aby wyrazić, że zdarzenie miało miejsce w j-tej jednostce czasu, powiemy, że nastąpiło w *turze* j (zamiennie: w momencie bądź w chwili j).

Rozwiązanie dla ustalonej liczby procesorów

Zastanówmy się na początek, jak rozwiązać nasz problem, gdy liczba procesorów jest ustalona – oznaczmy ją przez k. Jeśli $k = 1$, to w każdej jednostce czasu możemy wykonać dokładnie jedną instrukcję, zatem odpowiedzią będzie n. *W dalszej części opracowania będziemy już zakładać, że $k > 1$, co pozwoli uprościć późniejsze rozumowania.*

W pierwszej turze możemy wykonać tylko jedną instrukcję. Natomiast już w kolejnej liczba dostępnych instrukcji może przekroczyć k. Które z nich powinniśmy wykonać od razu, a które mogą poczekać? Okazuje się, że warto w pierwszym rzędzie wykonywać te instrukcje, które mają najdłuższą ścieżkę instrukcji oczekujących po nich.

Twierdzenie 1. *Algorytm zachłanny, który w pierwszej kolejności wybiera instrukcje o największym $d(x)$, generuje plan wykonania programu o minimalnej długości.*

Polecamy Czytelnikowi zastanowić się samemu nad dowodem. Jest to dobre ćwiczenie na szukanie precyzyjnych argumentów do wykazania twierdzenia, które intuicyjnie wydaje się prawdziwe, ale nie jest tak łatwo to uzasadnić. Twierdzenie 1 można udowodnić na kilka sposobów, jednak my skorzystamy z lematu, który przyda się również w dalszej części opracowania. Mianowicie wykażemy, że jeśli w pewnej chwili choć jeden procesor pozostaje bezczynny, to znaczy, że wąskim gardłem jest kształt drzewa instrukcji.

Oznaczmy przez $r_k(t)$ liczbę instrukcji wykonywanych przez algorytm zachłanny w turze t dla k procesorów. Zauważmy, że r_k może zależeć od tego, w jaki sposób algorytm rozstrzyga remisy w sytuacji, gdy gotowych do wykonania jest więcej niż k instrukcji maksymalizujących d. Nie przejmujmy się tym na razie. Ustalmy dowolny algorytm zachłanny i odnośmy się do planu generowanego przez niego – później okaże się, że w istocie jest to bez znaczenia.

Lemat 1. Jeśli $r_k(t) < k$, to wszystkie instrukcje na głębokościach nie większych niż t zostają wykonane do chwili t.

Dowód: Dowód przeprowadzimy przez indukcję względem t. Dla $t = 1$ teza jest oczywista. Przyjmijmy zatem $t > 1$.

Jeśli $r_k(t-1) < k$, to na mocy założenia indukcyjnego, ojcowie wszystkich instrukcji na głębokości t zostają wykonani do chwili $t-1$. Skoro $r_k(t) < k$, to znaczy, że algorytm zachłanny wykonał wszystkie dostępne instrukcje.

Rozważmy teraz przypadek $r_k(t-1) = k$. Niech b oznacza liczbę jednostek czasu bezpośrednio poprzedzających t, w których wykonywanych jest po k instrukcji (patrz rys. 1). Wiemy, że $b \leqslant t-2$, ponieważ w pierwszej turze gotowa jest tylko jedna instrukcja.

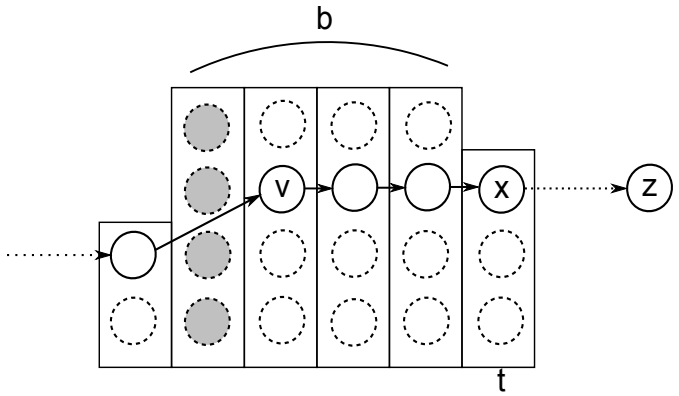
Przypuśćmy, że teza indukcyjna nie zachodzi. To oznacza, że pewna instrukcja o głębokości t zostaje wykonana później. W takim razie algorytm zachłanny przypisze do chwili t pewnego jej przodka (być może pośredniego) – oznaczmy go przez x. Zauważmy, że $d(x) \geqslant 2$.

Czy w chwili $t-1$ zostaje wykonany ojciec x? Jeśli nie, to znaczy, że został wykonany już wcześniej, zatem w chwili $t-1$ instrukcja x była gotowa do wykonania.

Skoro algorytm zachłanny jej nie wybrał, to znaczy, że gotowych było k innych instrukcji $y_1, \ldots, y_k$ spełniających $d(y_i) \geqslant 2$. To by oznaczało, że w chwili t na wykonanie czekało co najmniej k instrukcji będących synami $y_1, \ldots, y_k$, co przeczy założeniu, że $r_k(t) < k$.

Skoro więc ojciec x został wykonany w chwili $t-1$, to cofamy się w czasie z naszym rozumowaniem i pytamy się, czy „dziadek" x został wykonany w chwili $t-2$, czy też wcześniej. Jeśli wcześniej, to w chwili $t-2$ gotowych było k instrukcji $y_1, \ldots, y_k$ spełniających $d(y_i) \geqslant 3$, co prowadzi do sprzeczności analogicznie jak w poprzednim przypadku. Powtarzamy ten argument dla kolejnych przodków x.

Na mocy przeprowadzonego rozumowania przodek x stopnia b (oznaczmy go przez y) zostaje wykonany w chwili $t-b$. Przypomnijmy, że x leży na głębokości mniejszej niż t, zatem głębokość y to maksymalnie $t-b-1$. Ale $r_k(t-b-1) < k$, zatem na mocy założenia indukcyjnego y powinien być wykonany do chwili $t-b-1$. Wykazana sprzeczność dowodzi tezy indukcyjnej. ■



Rys. 1: Ilustracja do dowodu lematu 1 dla $k = b = 4$. Instrukcja na głębokości t, która nie zostaje wykonana na czas, jest oznaczona przez z. Przodek x stopnia 4 zostaje wykonany przed momentem $t-4$, zatem przodek stopnia 3 (oznaczony przez v) jest gotowy w turze $t-4$. Z właściwości algorytmu zachłannego wynika, że wszystkie instrukcje pokolorowane na szaro mają następników stopnia co najmniej 4, co jest niemożliwe.

Możemy już teraz udowodnić twierdzenie 1:

Dowód: Niech $t_0 = \max\{t \geqslant 1 : r_k(t) < k \wedge r_k(t+1) > 0\}$. Takie t_0 istnieje zawsze, gdy drzewo ma co najmniej dwa wierzchołki (w przypadku jednego wierzchołka teza jest trywialna). Skoro wszystkie instrukcje na głębokościach nie większych niż t_0 są wykonane do chwili t_0, a algorytm się jeszcze nie zakończył, to musi również zachodzić $s(t_0+1) > 0$. Instrukcje na głębokościach większych niż t_0 są gotowe do realizacji w chwili t_0+1 (oczywiście nie jest możliwe, by były gotowe wcześniej) i są wykonywane po k w turze, poza być może ostatnią turą działania programu. Jest jasne, że żaden plan nie jest w stanie wykonać ich szybciej, zatem plan generowany przez algorytm zachłanny jest optymalny. ■

Przedstawiony algorytm można zaimplementować w złożoności czasowej $O(n \log n)$. Wystarczy dysponować strukturą danych, która pozwala szybko dodawać elementy oraz znajdować i usuwać ten o największym d. Założenia te spełnia *kopiec*[1], na którym oba typy operacji działają w czasie logarytmicznym.

Możemy w ten sposób udzielić odpowiedzi dla każdego zapytania po kolei. Otrzymamy rozwiązanie działające w czasie $O(qn \log n)$, które zdobywało na zawodach 35% punktów. Zostało ono zaimplementowane w plikach `sups3.cpp` i `sups4.pas`.

Rozwiązanie wzorcowe

Jako że potencjalnie liczba zapytań może być tak duża jak n, to rozsądne wydaje się obliczenie najpierw odpowiedzi dla wszystkich $k = 1, \ldots, n$, zapamiętanie ich w tablicy oraz wypisanie na koniec tych pozycji, o które nas proszono. Jest jasne, że dla $k > n$ odpowiedzią jest wysokość drzewa.

Zanim jednak będziemy na to gotowi, musimy lepiej zrozumieć działanie algorytmu zachłannego. Następny lemat daje użyteczne kryterium, mówiące, kiedy wszystkie procesory będą zajęte.

Lemat 2. Dla dowolnych t, i, takich że $\sum_{j=t}^{t+i} s(j) \geqslant (i+1)k$, zachodzi $r_k(t+i) = k$.

Dowód: Załóżmy nie wprost, że $r_k(t+i) < k$. Wiemy z lematu 1, że wszystkie instrukcje znajdujące się na głębokościach $t, t+1, \ldots, t+i$ (oraz być może także niektóre instrukcje z mniejszych głębokości) zostaną wykonane w przedziale $[t, t+i]$. Pociąga to

$$\sum_{j=t}^{t+i} s(j) \leqslant \sum_{j=t}^{t+i} r_k(j) < (i+1)k.$$

Otrzymana sprzeczność dowodzi tezy lematu. ■

Wykorzystując oba lematy, można wykazać, że wartości $r_k(t)$ zależą jedynie od ciągu $s(t)$. Nieistotne są dokładna struktura drzewa oraz sposób, w jaki algorytm zachłanny rozstrzyga remisy. Nie będziemy jednak dowodzić tutaj tego faktu, ponieważ zrobimy to niejako przy okazji podczas konstruowania algorytmu wzorcowego.

Przyda nam się jeszcze jedna prosta obserwacja. Mianowicie jeśli w pewnym momencie wszystkie procesory są zajęte, to na pewno nie zmieni się to po zmniejszeniu k.

Lemat 3. Jeśli $r_k(t) = k$, to $r_{k-1}(t) = k - 1$.

Dowód: Oznaczmy $t_0 = \max\{i < t : r_k(i) < k\}$. Z lematu 1 wiadomo, że instrukcje na głębokościach nie większych niż t_0 są zakończone do chwili t_0. Zatem w przedziale $[t_0+1, t]$ komputer może wykonywać tylko instrukcje o głębokościach z tego przedziału, co implikuje

$$(t - t_0)(k-1) < (t - t_0)k = \sum_{j=t_0+1}^{t} r_k(j) \leqslant \sum_{j=t_0+1}^{t} s(j).$$

Powyższa nierówność w połączeniu z lematem 2 daje tezę lematu. ■

[1] Można o nim poczytać w książce [25].

Algorytm

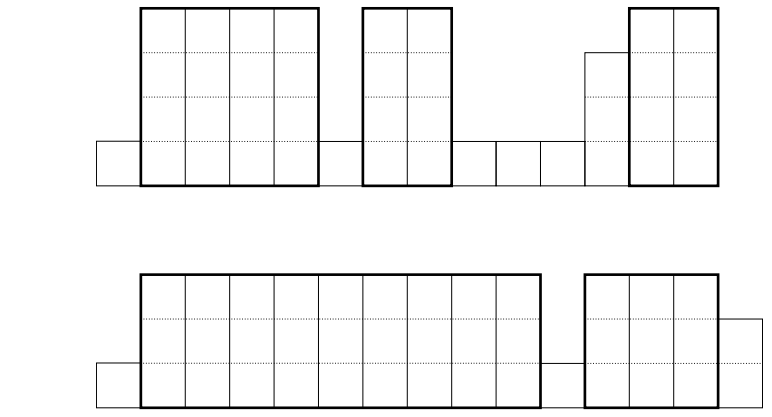
Bogatsi o nową wiedzę, możemy zabrać się do konstruowania efektywnego algorytmu wyliczania $r_k(t)$. Znamy $r_n = s$ i będziemy rozpatrywać kolejne $k = n-1, n-2, \ldots$ Chcielibyśmy wyznaczyć $r_k(1)$, $r_k(2), \ldots$ znając $r_{k+1}(1)$, $r_{k+1}(2), \ldots$ Ponadto musimy zrobić to tak szybko dla wszystkich k, aby całkowity czas działania był $o(n^2)$. Początkowo cel wydaje się nieosiągalny, skoro liczba wszystkich wartości do obliczenia może być rzędu n^2. Trudność tę można ominąć, rozpatrując każdy maksymalny spójny ciąg, na którym $r_k(t) = k$, jako pojedynczy obiekt. Nazwiemy taki ciąg *k-blokiem* lub po prostu blokiem, jeśli wiadomo, o jakie k chodzi. Jeśli $r_k(t) < k$, to przedział jednoelementowy $[t, t]$ będziemy nazywali małym blokiem. Dla symetrii pozostałe bloki będziemy nazywać dużymi.

Załóżmy, że znamy strukturę $(k+1)$-bloków. Aby wyznaczyć k-bloki, zaczynamy od $t = 1$ i przesuwamy się w prawo. Przyjmijmy, że przeprowadziliśmy obliczenia do t_0 i wszystkie instrukcje na głębokościach nie większych niż t_0 zostały już przydzielone.

Jeśli $s(t_0+1) \leqslant k$, to oczywiście $r_k(t_0+1) = s(t_0+1)$. A co w przypadku, gdy $s(t_0+1) \geqslant k+1$? Wówczas w tym miejscu rozpoczyna się duży $(k+1)$-blok. Z lematu 1 wiemy, że nowy k-blok będzie sumą $(k+1)$-bloków. Warunek z lematu 2 gwarantuje, że kolejne bloki możemy scalać zachłannie.

Jeśli bezpośrednio na prawo znajduje się mały $(k+1)$-blok $[t, t]$ oraz $\sum_{j=t_0+1}^{t} s(j) \geqslant (t-t_0)k$, to $r_k(t) = k$ i dodajemy go do tworzonego k-bloku. W przeciwnym razie musimy wykonać $\sum_{j=t_0+1}^{t} s(j) - k \cdot (t-t_0-1) < k$ instrukcji i pozostawiamy mały blok o większej wartości. Jeśli natomiast napotkamy duży blok, to na pewno będziemy musieli go scalić.

Kiedy zakończymy procedurę scalania, traktujemy prawy koniec bloku jako nowe t_0 i powtarzamy cały proces.



Rys. 2: Przykładowa struktura 4-bloków i 3-bloków dla tych samych danych.

Implementacja i analiza złożoności

Implementacja powyższego algorytmu wymaga zastanowienia, choć gotowy kod okaże się stosunkowo krótki. Należy w szczególności dobrać odpowiednie struktury danych do przechowywania informacji o blokach, aby uniknąć wykonywania liniowej liczby operacji dla pojedynczego k.

Zauważmy, że nie warto interesować się małym blokiem o wysokości s, dopóki nie będziemy analizować $k = s$ (wtedy zmieni się on w duży blok) lub nie będziemy go scalać z innym blokiem. Zamiana w duży blok może oczywiście nastąpić tylko raz dla konkretnego bloku, więc operacji tego typu będzie co najwyżej n. Również sytuacja, w której blok zostaje scalony z blokiem położonym z lewej strony, może zdarzyć się tylko n razy, bo za każdym razem maleje liczba bloków. Pozostaje przeanalizować przypadek, w którym uaktualniamy stan małego bloku, ale nie zostaje on scalony, czyli dalej zachodzi $r_k(t) < k$. Zauważmy, że może mieć to miejsce jedynie na końcu procedury scalania bloków, kiedy pewien duży blok zmniejsza swoją wysokość o 1. Suma wysokości wszystkich dużych bloków, początkowo równa zeru, w trakcie działania algorytmu zwiększy się co najwyżej o n, przy zamianach małych bloków w duże. Suma ta nie może spaść poniżej zera, więc liczba operacji ostatniego typu amortyzuje się do $O(n)$.

Poniższy pseudokod pokazuje inicjalizację struktury danych pozwalającej efektywnie zaimplementować przedstawiony pomysł. Tablica *tasks* (początkowo wypełniona zerami) pamięta wartości r_k dla aktualnego k. Jeśli *first_in_block*[i] równa się **true**, to i jest początkiem bloku o długości *block_size*[i]. Jeżeli *first_in_block*[i] = **false**, to pomijamy i w dalszych obliczeniach (w szczególności, nie dbamy już o aktualizowanie *tasks*[i]).

Z kolei *by_tasks* jest tablicą list. Lista *by_tasks*[i] pamięta te głębokości, dla których r_k równa się i. Zakładamy, że można dodawać do niej elementy w czasie stałym (operacja *push*) oraz przeglądać je w czasie proporcjonalnym do długości listy. Aby ominąć problem usuwania elementów z listy, przed przetworzeniem bloku i będziemy sprawdzać, czy wartość *tasks*[i] jest aktualna.

```
1:  begin
2:    for i := 1 to n do
3:      tasks[d[i]] := tasks[d[i]] + 1;
4:    for i := 1 to n do begin
5:      if tasks[i] > 0 then
6:        by_tasks[tasks[i]].push(i);
7:      block_size[i] := 1;
8:      first_in_block[i] := true;
9:    end
10: end
```

Poniżej widzimy główną część programu. W momencie, w którym wysokość bloku przekracza k, obliczana jest zmienna *delta* równa liczbie instrukcji, które nie mieszczą się w bloku. Powiększamy blok w prawo, dopóki *delta* nie równa się 0. Za każdym razem, gdy powiększamy pewien blok, sprawdzamy, czy czas zakończenia nie wzrasta.

```
1:  begin
2:    wynik[n] := d[1];
3:    for k := n − 1 downto 1 do
4:      foreach x ∈ by_tasks[k + 1] do
5:        if first_in_block[x] and tasks[x] = k + 1 then begin
6:          delta := block_size[x];
7:          tasks[x] := k;
8:          by_tasks[k].push(x);
9:          while delta > 0 do begin
10:           y := x + block_size[x];
11:           wynik[k] := max(wynik[k], y + block_size[y] − 1);
12:           if block_size[y] = 1 and tasks[y] + delta < k then begin
13:             tasks[y] := tasks[y] + delta;
14:             delta := 0;
15:             by_tasks[tasks[y]].push(y);
16:           end
17:           else begin
18:             { Gdy tasks[y] = k + 1, delta może wzrosnąć! }
19:             delta := delta − (k − tasks[y]) · block_size[y];
20:             block_size[x] := block_size[x] + block_size[y];
21:             first_in_block[y] := false;
22:           end
23:         end
24:       end
25:   return wynik;
26: end
```

Taka struktura danych nie gwarantuje, że bloki analizowane będą od lewej do prawej. Zachęcamy jednak Czytelnika do wykazania, że nie ma to wpływu na wynik algorytmu. Przedstawione rozwiązanie można znaleźć w pliku `sup2.cpp`.

Inna metoda implementacji to użycie struktury *find-union*. Czas działania algorytmu jest w tym przypadku minimalnie gorszy od $O(n)$, jednak różnica jest niewykrywalna w testach. Rozwiązania korzystające z tego pomysłu znajdują się w plikach `sup.cpp` oraz `sup1.pas`.

Gdyby pominąć pomysł z dużymi blokami i za każdym razem analizować każdą głębokość osobno, otrzymalibyśmy algorytm o złożoności pesymistycznej $O(qn)$, ale radzący sobie z dużymi losowymi testami. Na zawodach mógłby on zdobyć do 60% punktów. Pomysł ten zaimplementowano w plikach `sups5.cpp` oraz `sups6.pas`.

Rozwiązanie alternatywne

Okazuje się, że istnieje zupełnie inne rozwiązanie, sprowadzające problem do zagadnienia geometrycznego. Jest ono godne uwagi, gdyż okazuje się prostsze w implementacji. Główna obserwacja orzeka, że wąskie gardło każdego programu znajduje się na jego

końcu. Zwrócił na to uwagę Paweł Gawrychowski na swoim blogu[2] (uzasadnienie poprawności w opisie na blogu różni się od przedstawionego tutaj).

Twierdzenie 2. *Optymalny czas zakończenia dla k procesorów to*

$$H_k = \max_{h:s(h)>0} \left(h + \left\lceil \frac{\sum_{i>h} s(i)}{k} \right\rceil \right),$$

gdzie przez $\lceil r \rceil$ oznaczamy najmniejszą liczbę całkowitą nie mniejszą niż r.

Dowód: Łatwo zauważyć, że czas zakończenia żadnego planu nie może być mniejszy niż H_k. Dla każdego h wierzchołki znajdujące się na głębokościach większych niż h mogą być przetwarzane nie wcześniej niż w momencie $h+1$ oraz ich wykonanie zajmie co najmniej $\lceil \sum_{i>h} s(i)/k \rceil$ jednostek czasu.

Zgodnie z dowodem twierdzenia 1, czas zakończenia wykonywania instrukcji programu przez algorytm zachłanny to $h + \lceil \sum_{i>h} s(i)/k \rceil$ dla $h = t_0 = \max\{t \geqslant 1 : r_k(t) < k \wedge r_k(t+1) > 0\}$. W połączeniu z poprzednią obserwacją implikuje to, że czas ten jest równy H_k. ■

Jako że potrzebujemy szybko obliczać H_k dla różnych k, chcielibyśmy liczyć maksimum z jak najprostszych funkcji.

Lemat 4. Oznaczmy $f_h(k) = kh + \sum_{i>h} s(i)$. H_k można wyrazić równoznacznie jako

$$\left\lceil \frac{\max_{h:s(h)>0} f_h(k)}{k} \right\rceil.$$

Dowód: Zauważmy, że

$$h + \left\lceil \frac{\sum_{i>h} s(i)}{k} \right\rceil = \lceil f_h(k)/k \rceil.$$

Chcemy się przekonać, że

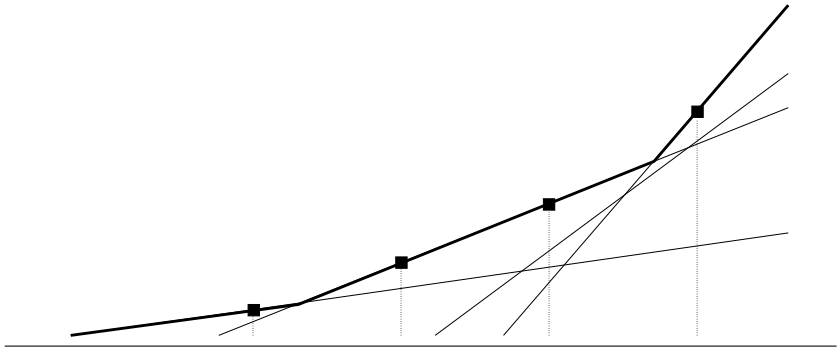
$$\left\lceil \frac{\max_{h:s(h)>0} f_h(k)}{k} \right\rceil = \max_{h:s(h)>0} \lceil f_h(k)/k \rceil.$$

Równość zachodzi, gdyż funkcja $x \longrightarrow \lceil x/k \rceil$ jest monotoniczna, zatem zachowuje maksimum zbioru. ■

Niech m oznacza wysokość wejściowego drzewa. Sprowadziliśmy problem do wyznaczania maksimum m funkcji liniowych w punktach $1, \ldots, n$. Zauważmy, że jeśli $f_h(k_0) > f_i(k_0)$ dla każdego $i < h$, to taka sama zależność będzie zachodzić dla dowolnego $k > k_0$. Innymi słowy, gdy funkcja f_h „przegoni" funkcje o niższych indeksach, to pozostanie już zawsze od nich większa, gdyż rośnie szybciej.

Daje to prosty algorytm wyznaczania górnej otoczki rodziny funkcji liniowych. Po przetworzeniu $f_1, f_2, \ldots, f_h$ trzymamy informacje o kolejnych funkcjach na stosie. Dla każdego i pamiętamy, na jakim maksymalnym przedziale $[left_i, right_i]$ funkcja f_i jest

[2] `http://fajnezadania.wordpress.com/2014/04/13/superkomputer/`



Rys. 3: Górna otoczka czterech funkcji liniowych.

większa od pozostałych. Jeżeli wspomniany przedział jest pusty, zapominamy o tej funkcji.

Jak dodać do struktury funkcję f_{h+1}? Przypuśćmy, że na szczycie stosu znajduje się f_j. Szukamy w czasie stałym najmniejszego k, dla którego $f_{h+1}(k) > f_j(k)$. Jeśli $k \leqslant \mathit{left}_j$, to usuwamy ze stosu f_j i powtarzamy procedurę. W przeciwnym razie uaktualniamy right_j i dodajemy f_{h+1} na czubek stosu.

Oczywiście każdą funkcję usuniemy ze stosu maksymalnie raz, zatem całkowita złożoność obliczeniowa przedstawionego algorytmu jest liniowa. Rozwiązanie alternatywne można znaleźć w pliku `sup3.cpp`.