Notatki z AiSD. Nr 5. 16 marca 2005

Programowanie dynamiczne

IIUWr. II rok informatyki. *Opracował: Krzysztof Loryś*

13 Wstęp

Zastosowanie metody Dziel i Zwyciężaj do problemów zdefiniowanych rekurencyjnie jest w zasadzie ograniczone do przypadków, gdy podproblemy, na które dzielimy problem, są niezależne. W przeciwnym razie metoda ta prowadzi do wielokrotnego obliczania rozwiązań tych samych podproblemów. Jednym ze sposobów zaradzenia temu zjawisku jest tzw. *spamiętywanie*, polegające na pamiętaniu rozwiązań podproblemów napotkanych w trakcie obliczeń. W przypadku, gdy przestrzeń wszystkich moźliwych podproblemów jest nieduźa, efektywniejsze od spamiętywania może być zastosowanie metody programowania dynamicznego. Metoda ta polega na obliczaniu rozwiązań dla wszystkich podproblemów, począwszy od podproblemów najprostszych.

Przykład 1.

Problem:

Dane: Liczby naturalne n, k.

Wynik: $\binom{n}{k}$.

Naturalna metoda redukcji problemu obliczenia $\binom{n}{k}$ korzysta z zależności $\binom{n}{k}=\binom{n-1}{k-1}+\binom{n-1}{k}$. Zastosowanie metody Dziel i Zwyciężaj byłoby jednak w tym przypadku nierozważne, ponieważ w trakcie liczenia $\binom{n-1}{k-1}$ jak i $\binom{n-1}{k}$ wywoływalibyśmy rekurencyjnie procedurę dla tych samych danych (tj. dla $n-2$ i $k-1$), co w konsekwencji prowadziłoby do tego, źe niektóre podproblemy byłyby rozwiązywane wykładniczą liczbę razy.

Do spamiętywania możemy wykorzystać tablicę $tab[1..n, 1..k]$.

```
for i=1 to n do
    for j = 0 to k do tab_{i,j} ←"?"
            .....
function nPOk(n, k)
    if tab_{n-1,k-1} ="?" then tab_{n-1,k-1} ←nPOk(n − 1, k − 1)
    if tab_{n-1,k} ="?" then tab_{n-1,k} ←nPOk(n − 1, k)
    tab_{n,k} = tab_{n-1,k-1} + tab_{n-1,k}
    return tab_{n,k}
```

Za zastosowaniem w tym przypadku programowania dynamicznego przemawia fakt, iź liczba różnych podproblemów jakie mogą pojawić się w trakcie obliczania $\binom{n}{k}$ jest niewielka, a mianowicie $O(n^2)$. Podobnie jak w metodzie spamiętywania, algorytm dynamiczny oblicza początkowy fragment trójkąta Pascala i umieszcza go w tablicy *tab*. W przeciwień-

stwie jednak do poprzedniej metody, która jest metodą "top-down" i jest implementowana rekurencyjnie, algorytm dynamiczny jest metodą "bottom-up" i jest implementowany iteracyjnie. To pozwala w szczególności na wyeliminowanie kosztów związanych z obsługą rekursji.

```
for i = 1 to n do tab_{i,0} ← 1
          .....
function nPOk(n, k)
    for j = 1 to k do
        tab_{j,j} ← 1
        for i = j + 1 to n do tab_{i,j} ← tab_{i-1,j-1} + tab_{i-1,j}
    return tab_{n,k}
```

Fakt, źe metoda programowania dynamicznego oblicza w sposób systematyczny rozwiązania wszystkich podproblemów, pozwala często na poczynienie dodatkowych oszczędności w stosunku do metody spamiętywania. W tym przykładzie możemy znacznie zredukować koszty pamięciowe. Jak łatwo zauważyć, obliczenie kolejnej przekątnej trójkąta Pascala wymaga znajomości jedynie wartości z poprzedniej przekątnej. Tak więc zamiast tablicy $n \times k$ wystarcza tablica $n \times 2$, a nawet tablica $n \times 1$.

□

Podobnie jak w przypadku metody dziel i zwyciężaj, kluczem do zastosowania programowania dynamicznego jest znalezienie sposobu dzielenia problemu na podproblemy w taki sposób, by optymalne rozwiązanie problemu można było w prosty sposób otrzymać z optymalnych rozwiązań podproblemów (mówimy wówczas, źe problem wykazuje *optymalną podstrukturę*). Wskazaniem za stosowaniem wówczas programowania dynamicznego a nie metody dziel i zwyciężaj jest sytuacja, gdy sumaryczny rozmiar podproblemów jest duży. Oczywiście, jak juź wspominaliśmy, aby algorytm dynamiczny był efektywny, przestrzeń wszystkich moźliwych podproblemów nie moźe być zbyt liczna.

Przykład 2.

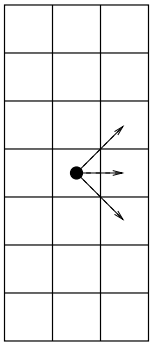
Problem:

Dane: Tablica $\{a_{i,j}\}$ liczb nieujemnych $(i = 1, \ldots, n;\ j = 1, \ldots, m)$

Wynik: Ciąg indeksów $i_1, \ldots, i_m$ taki, źe $\forall_{j=1,\ldots,m-1} |i_j - i_{j+1}| \leq 1$, minimalizujący sumę $\sum_{j=1}^{m} a_{i_j,j}$

Interpretacja: Ciąg $i_1, \ldots, i_m$ wyznacza trasę wiodącą od pierwszej do ostatniej kolumny tablicy a. Startujemy z dowolnego pola pierwszej kolumny i kończymy na dowolnym polu ostatniej kolumny. W każdym ruchu przesuwamy się o jedno pole: albo w prawo na wprost albo w prawo na ukos (jak pokazano na rysunku 5). Chcemy znale"xć trasę o minimalnej długości, rozumianej jako suma liczb z pól znajdujących się na trasie.

Jak łatwo sprawdzić liczba wszystkich prawidłowych tras jest wykładnicza, więc rozwiązanie siłowe nie wchodzi w rachubę.



Rysunek 5: *Możliwe kierunki ruchu w tablicy a.*

Rozważmy najpierw nieco prostsze zadanie, polegające na znalezieniu długości optymalnej trasy. Potem pokażemy w jaki sposób zorganizować obliczenia, by wyznaczenie samej trasy było proste.

Niech $d_{i,k}$ oznacza minimalną długość trasy wiodącej od dowolnego pola pierwszej kolumny do pola $a_{i,k}$, a $P(i,k)$ problem wyznaczenia $d_{i,k}$. Rozwiązanie $P(i,k)$ (dla $k > 1$) można łatwo otrzymać z rozwiązań trzech prostszych podproblemów, a mianowicie $P(i-1,k-1)$, $P(i,k-1)$ i $P(i+1,k-1)$ (w przypadku $P(1,k)$ i $P(n,k)$ - dwóch podproblemów). Problem wykazuje więc optymalną podstrukturę.

Jeśli za rozmiar $P(i,k)$ przyjmiemy wartość k, to problem rozmiaru k redukujemy do trzech podproblemów rozmiaru $k-1$. To zbyt duże rozmiary, by opłaciło się stosować metodę dziel i zwyciężaj. Z drugiej strony przestrzeń wszystkich podproblemów jest stosunkowo niewielka - składa się z nm elementów (zawiera wszystkie $P(i,j)$ dla $i = 1,\ldots,n, j = 1,\ldots,m$), możemy więc zastosować programowanie dynamiczne.

for $j = 1$ **to** n **do** $d_{0,j} \leftarrow d_{n+1,j} \leftarrow \infty$
for $i = 1$ **to** n **do** $d_{i,1} \leftarrow a_{i,1}$
for $j = 2$ **to** m **do**
    for $i = 1$ **to** n **do** $d_{i,j} \leftarrow a_{i,j} + \min\{d_{i-1,j-1}, d_{i,j-1}, d_{i+1,j-1}\}$
return $\min\{d_{i,m} \mid i = 1,\ldots,n\}$

Pozostaje wyjaśnić, w jaki sposób można odtworzyć optymalną trasę. Niech i_0 będzie wartością i, dla której osiągane jest $\min\{d_{i,m} \mid i = 1,\ldots,n\}$, a więc $a_{i_0,m}$ jest ostatnim polem optymalnej trasy. Przedostatnie pole możemy wyznaczyć sprawdzając, która z trzech różnic $d_{i_0,m} - d_{k,m-1}$ (dla $k \in \{i_0-1, i_0, i_0+1\}$) jest równa $a_{i_0,m}$. Postępując dalej rekurencyjnie wyznaczymy całą trasę. Łatwo zauważyć, że zamiast sprawdzać powyższe różnice, możemy ograniczyć się do sprawdzenia, która z wartości $d_{k,m-1}$ (dla $k \in \{i_0-1, i_0, i_0+1\}$) jest minimalna.

```
procedure trasa(i, j)
{  if j = 1 then return i
   Niech i' będzie takie, że d_{i',j-1} = min{d_{k,j-1} | k ∈ {i-1, i, i+1}}
   return concat( trasa(i', j-1), i))
}
          .....
write( trasa(i_0, m))
```

□

Programowanie dynamiczne jest częstą metodą rozwiązywania problemów optymalizacyjnych. Przykład 2 stanowi ilustrację klasycznego sposobu rozwiązania takiego problemu: najpierw znajdujemy wartość optymalnego rozwiązania a dopiero potem, na podstawie wyliczeń tej wartości, konstruujemy optymalne rozwiązanie.

14 Dalsze przykłady

14.1 Najdłuższy wspólny podciąg.

14.1.1 Definicja problemu

Definicja 4 *Ciąg $Z = \langle z_1, z_2, \ldots, z_k \rangle$ jest* podciągiem *ciągu $X = \langle x_1, x_2, \ldots, x_n \rangle$, jeśli istnieje ściśle rosnący ciąg indeksów $\langle i_1, i_2, \ldots, i_k \rangle$ $(1 \leq i_j \leq n)$ taki, że*

$$\forall_{j=1,2,\ldots k} \;\; x_{i_j} = z_j.$$

Jeśli Z jest podciągiem zarówno ciągu X jak i ciągu Y, to mówimy, że Z jest wspólnym podciągiem *ciągów X i Y.*

Konwencja: Dla wygody, w dalszej części ciągi będziemy traktować jako napisy nad ustalonym alfabetem.

Przykład:
'BABA' jest wspólnym podciągiem ciągów 'ABRACADABRA' i 'RABARBAR', ale nie jest ich najdłuższym wspólnym podciągiem (dłuższym jest np. 'RAAAR').

□

Oznaczenia:

- $LCS(X, Y) = \{Z \;\; | \;\; Z$ jest wspólnym podciągiem X i Y o maksymalnej długości$\}$
- przez X_i oznaczamy i-literowy prefiks ciągu $X = \langle x_1, x_2, \ldots, x_n \rangle$, tj. podciąg $\langle x_1, x_2, \ldots, x_i \rangle$; w szczególności przez X_0 oznaczamy ciąg pusty.

Problem:
Dane: ciągi $X = \langle x_1, x_2, \ldots, x_m \rangle$ i $Y = \langle y_1, y_2, \ldots, y_n \rangle$
Wynik: dowolny ciąg Z z $LCS(X, Y)$

14.1.2 Redukcja problemu

Problem znalezienia ciągu Z z $LCS(X, Y)$ możemy zredukować do prostszych problemów na podstawie następującej obserwacji:

- jeśli ostatnia litera X i ostatnia litera Y są takie same, to litera ta musi być ostatnim elementem każdego ciągu z $LCS(X, Y)$.
- jeśli X i Y różnią się na ostatniej pozycji (tj. $x_m \neq y_n$), to istnieje ciąg w $LCS(X, Y)$, który na ostatniej pozycji ma literę różną od x_m lub istnieje ciąg w $LCS(X, Y)$, który na ostatniej pozycji ma literę różną od x_m.

W pierwszym przypadku problem znalezienia ciągu z $LCS(X_m, Y_n)$ redukujemy do podproblemu znalezienia ciągu z $LCS(X_{m-1}, Y_{n-1})$. Rozwiązaniem będzie konkatenacja znalezionego ciągu i ostatniej litery X-a. W drugim przypadku problem redukujemy do dwóch podproblemów: znalezienie ciągu z $LCS(X_{m-1}, Y_n)$ i znalezienie ciągu z $LCS(X_m, Y_{n-1})$. W tym przypadku rozwiązaniem będzie dłuższy ze znalezionych ciągów.

14.1.3 Algorytm

Najpierw koncentrujemy się na obliczeniu wartości rozwiązania optymalnego, którą w tym przypadku jest długość elementów z $LCS(X, Y)$. Sposobu na obliczenie tej wartości dostarcza nam obserwacja poczyniona w poprzednim paragrafie.

Fakt 9 *Niech $d_{i,j}$ oznacza długość elementów z $LCS(X_i, Y_j)$. Wówczas:*

$$d_{i,j} = \begin{cases} 0 & \text{jeśli } i = 0 \text{ lub } j = 0, \\ 1 + d_{i-1,j-1} & \text{jeśli } i, j > 0 \ i \ \ x_i = y_j, \\ \max(d_{i,j-1}, d_{i-1,j}) & \text{jeśli } i, j > 0 \ i \ \ x_i \neq y_j \end{cases}$$

Tablicę d możemy obliczać kolejno wierszami (lub kolumnami), a wynik odczytamy z $d_{m,n}$.

```
Procedure LCS(X_m, Y_n)
    for i ← 1 to m do d_{i,0} ← 0
    for j ← 0 to n do d_{0,j} ← 0
    for i ← 1 to m do
        for j ← 1 to n do
            if x_i = y_j then d_{i,j} ← 1 + d_{i-1,j-1}
            else d_{i,j} ← max{d_{i-1,j}, d_{i,j-1}}
```

Aby wypisać jakiś element z LCS musimy przejść tablicę d jeszcze raz, począwszy od elementu $d_{n,m}$, w podobny sposób jak to robiliśmy w Przykładzie 2.

Jeśli zależy nam na szybkości algorytmu, możemy nieco przyspieszyć tę jego fazę. W tym celu, w trakcie obliczania tablicy d, możemy w dodatkowej tablicy zapamiętywać "drogę dojścia"do poszczególnych elementów tablicy d. Elementy dodatkowej tablicy przyjmowałyby jedną z trzech różnych wartości, w zależności od tego czy $d_{i,j}$ powstał przez dodanie 1 do $d_{i-1,j-1}$, czy przez przepisanie $d_{i-1,j}$, czy też wreszcie przez przepisanie $d_{i,j-1}$.

14.1.4 Koszt algorytmu

Obliczenie każdego elementu tablicy d odbywa się w czasie stałym. Tak więc całkowity koszt wypełnienia tablicy d jest równy $\Theta(n \cdot m)$. Koszt skonstruowania najdłuższego podciągu na podstawie tablicy d jest liniowy.

14.2 Wyznaczanie optymalnej kolejności mnożenia macierzy.

14.2.1 Definicja problemu

Mamy obliczyć wartość wyrażenia $\mathcal{M}$ postaci $M_1 \times M_2 \times \cdots \times M_n$, gdzie M_i są macierzami. Zakładamy, że wyrażenie jest poprawne, tj. liczba kolumn macierzy M_i jest równa liczbie wierszy macierzy M_{i+1} (dla $i = 1, \ldots, n-1$).

Ponieważ mnożenie macierzy jest działaniem łącznym, wartość $\mathcal{M}$ możemy liczyć na wiele sposobów. Wybór sposobu może w istotny sposób wpłynać na liczbę operacji skalarnych jakie wykonamy podczas obliczeń.

PRZYKŁAD Niech macierze M_1, M_2, M_3 mają wymiary odpowiednio $d \times 1$, $1 \times d$ i $d \times 1$. Rozważmy dwa sposoby obliczenia ich iloczynu:

- $(M_1 \times M_2) \times M_3$
  W wyniku pierwszego mnożenia otrzymujemy macierz $d \times d$, więc jego koszt (niezależnie od przyjętej metody mnożenia macierzy) wynosi co najmniej d^2. W drugim mnożeniu także musimy wykonać $\Theta(d^2)$ operacji.

- $M_1 \times (M_2 \times M_3)$
  Koszt obliczenia $M_2 \times M_3$ wynosi $O(d)$. W jego wyniku otrzymujemy macierz 1×1, więc koszt następnego mnożenia wynosi także $O(d)$.

□

Dalsze rozważania będziemy przeprowadzać przy następującym założeniu[2]:

`Koszt pomnożenia macierzy o wymiarach` $a \times b$ `i` $b \times c$ `wynosi` abc.

PROBLEM:

Dane: $d_0, d_1, \ldots, d_n$ - liczby naturalne
INTERPRETACJA: $d_{i-1} \times d_i$ - wymiar macierzy M_i.

Zadanie: Wyznaczyć kolejność mnożenia macierzy $M_1 \times M_2 \times \cdots \times M_n$, przy której koszt obliczenia tego iloczynu jest minimalny.

14.2.2 Rozwiązanie siłowe

Rozwiązanie siłowe, polegajace na sprawdzeniu wszystkich moźliwych sposobów wykonania obliczeń, jest nieakceptowalne. Liczba tych sposobów dana jest wzorem

$$S(n) = \begin{cases} 1 & \text{jeśli } n = 1 \\ \sum_{i=1}^{n-1} S(i)S(n-i) & \text{jeśli } n > 1 \end{cases}$$

[2]Jest to koszt mnożenia wykonanego metodą tradycyjną; pó"xniej poznamy inne, szybsze metody.

UZASADNIENIE WZORU: Każde z $n-1$ mnożeń jakie występują w ciągu $M_1 \times \ldots \times M_n$ może być ostatnim, jakie wykonamy licząc ten iloczyn. Liczba sposobów mnożenia macierzy, w których i-te mnożenie jest ostatnim, jest równa iloczynowi $S(i)$ (tj. liczby sposobów, na które można pomnożyć i pierwszych macierzy) oraz $S(n-i)$ tj. liczby sposobów, na które można pomnożyć $n-i$ ostatnich macierzy).

Rozwiązaniem powyższego równania jest $S(n) =$"$n-$ta liczba Catalana"$= \frac{1}{n}\binom{2n-2}{n-1} = \Omega(\frac{4^n}{n^2})$. Tak więc koszt sprawdzania wszystkich moźliwych iloczynów jest wykładniczy.

14.2.3 Rozwiązanie dynamiczne

Zauważamy, źe problem wykazuje optymalną podstrukturę. Jeśli bowiem k-te mnoźenie jest ostatnim, jakie wykonamy w optymalnym sposobie obliczeń, to iloczyny $M_1 \times \ldots \times M_k$ oraz $M_{k+1} \times \ldots \times M_n$ teź musiały być obliczone w optymalny sposób.

Na podstawie tej własności moźemy ułoźyć następujący algorytm rekurencyjny obliczający optymalny koszt obliczeń.

```
function matmult(i, j)
    if i = j then return 0
    opt ← ∞
    for k ← i to j − 1 do
        opt ← min(opt, d_{j−1} d_k d_j + matmult(i, k) + matmult(k + 1, j))
    return opt
```

Algorytm ten, jakkolwiek szybszy od metody siłowej, nadal działa w czasie wykładniczym ($\Theta(3^n)$). Przyczyna tkwi w wielokrotnym wykonywaniu obliczeń dla tych samych wartości parametrów (i, j). Unikniemy tego mankamentu stosując programowanie dynamiczne. Niech

$$m_{i,j} = \text{"minimalny koszt obliczenia } M_i \times M_{i+1} \times \cdots \times M_j\text{"}$$

Dla wygody przyjmujemy, źe $m_{i,j} = 0$ (dla $i \geq j$). Wówczas

$$m_{i,j} = \min_{i \leq k < j} (m_{i,k} + m_{k+1,j} + d_{i-1} d_k d_j).$$

Składnik $m_{i,k}$ jest kosztem obliczenia $M_i \times M_{i+1} \times \cdots \times M_k$, składnik $m_{k+1,j}$ - kosztem obliczenia $M_{k+1} \times M_{k+2} \times \cdots \times M_j$, natomiast $d_{i-1} d_k d_j$ to koszt obliczenia iloczynu dwóch powstałych macierzy.

```
procedure dyn − matmult(d[0..n]);
    int m[1..n, 1..n], p[1..n, 1..n]
     for  i ← 1 to n do m_ii ← 0;
     for  s ← 1 to n − 1 do
         for  i ← 1 to n − s do
             j ← i + s
             m_ij ← min_{i≤k<j}(m_{i,k} + m_{k+1,j} + d_{i−1} d_k d_j)
             p_ij ← "to k, przy którym osiągane było minimum dla m_ij"
     return  p[1..n, 1..n]
```

Algorytm oblicza wartości $m_{i,i+s}$ (na podstawie powyższego wzoru) oraz wartości $p_{i,i+s}$, które umoźliwiają pó"xniejsze skonstruowanie rozwiązania.

Koszt algorytmu. Tablicę $m_{i,j}$ liczymy przekątna za przekątną począwszy od głównej przekątnej. Koszt policzenia jednego elementu $m_{i,i+s}$ na s-tej przekątnej wynosi $\Theta(s)$. Ponieważ na s-tej przekątnej znajduje się $n - s$ elementów, koszt algorytmu wynosi

$$T(n) = \sum_{s=0}^{n-1} \Theta(s) \cdot (n - s) = \Theta(n^3).$$

Odtworzenie rozwiązania Odtworzenia rozwiązania dokonujemy w standardowy sposób na podstawie tablicy p. Zwróć uwagę, źe znalezienie rozwiązania na podstawie samych tylko wartości m_{ij} wymagałoby czasu $\Theta(n^2)$.

Programowanie dynamiczne

IIUWr. II rok informatyki. *Opracował: Krzysztof Loryś*

2.3 Problem Plecakowy

Problem plecakowy obejmuje szeroką klasę problemów optymalizacji kombinatorycznej. Dla danego zbioru przedmiotów o określonych wagach i wartościach należy wybrać podzbiór o jak największej sumarycznej wartości i sumarycznej wadze nie przekraczającej zadanego ograniczenia.

Większość problemów plecakowych (w tym obie wersje, które przedstawimy) należy do klasy problemów $\mathcal{NP}$-trudnych, co oznacza, że raczej nie możemy spodziewać się rozwiązań działających w czasie wielomianowym od rozmiaru danych. Algorytmy, które pokażemy są pseudowielomianowe.

2.3.1 Wersja z powtórzeniami

Problem:

Dane: ciąg $w_1, \ldots, w_n \in \mathcal{N}$
ciąg $v_1, \ldots, v_n \in \mathcal{R}$
liczba $W \in \mathcal{N}$

Wynik: wielozbiór zbiór $\{i_1, \ldots, i_k\}$ taki, że $\sum_{j=1}^{k} w_{i_j} \leq W$ oraz $\sum_{j=1}^{k} v_{i_j}$ jest maksymalna

Zakładamy, że waga każdego przedmiotu nie przekracza W.
Podproblemy: mniejszy plecak.

$K(w) =$ maksymalna wartość plecaka osiągalna dla plecaka o pojemności w.

Fakt 1

$$K(w) = \begin{cases} 0 & \text{jeśli } w = 0, \\ \max_{i:w_i<w}\{K(w-w_i)+v_i\} & \text{jeśli } w > 0 \end{cases}$$

□

Czas działania: $O(nW)$.

2.3.2 Wersja bez powtórzeń

Problem:

Dane: ciąg $w_1, \ldots, w_n \in \mathcal{N}$
ciąg $v_1, \ldots, v_n \in \mathcal{R}$
liczba $W \in \mathcal{N}$

Wynik: zbiór $\{i_1, \ldots, i_k\}$ taki, że $\sum_{j=1}^{k} w_{i_j} \leq W$ oraz $\sum_{j=1}^{k} v_{i_j}$ jest maksymalna.

Podproblemy: mniejszy plecak pakowany podzbiorem przedmiotów.

$K(w, j) =$ maksymalna wartość plecaka osiągalna dla plecaka o pojemności w oraz przedmiotów $\{1, \ldots, j\}$.

Fakt 2

$$K(w,j) = \begin{cases} 0 & \text{jeśli } w = 0 \text{ lub } j = 0 \\ \max\{K(w-w_j, j-1)+v_j, K(w,j-1)\} & \text{wpp} \end{cases}$$

□

Czas działania: $O(nW)$.

2.4 Najkrótsze ścieżki między wszystkimi parami wierzchołków

Do uzupełnienia.

2.5 Przynależność do języka bezkontekstowego

2.5.1 Definicja problemu

Rozpoczynamy od przypomnienia podstawowych pojęć związanych z gramatykami bezkontekstowymi (pojęcia te powinny być znane z wykładu Wstęp do Informatyki).

Definicja 1 Gramatyką bezkontekstową *nazywamy system* $G = \langle V_N, V_T, P, S\rangle$, *gdzie*

- V_N *i* V_T *są skończonymi rozłącznymi zbiorami (nazywamy je odpowiednio* alfabetem symboli nieterminalnych *i* alfabetem symboli terminalnych*);*
- P *jest skończonym podzbiorem zbioru* $V_N \times (V_N \cup V_T)^*$ *(elementy* P *nazywamy* produkcjami*);*
- $S \in V_N$ *i jest nazywany* symbolem początkowym gramatyki.

Zwyczajowo produkcje (A, α) zapisujemy jako $A \to \alpha$.

Definicja 2 *Jeśli każda produkcja gramatyki bezkontekstowej* G *jest postaci:*

- $A \to BC$ *lub*
- $A \to a$,

gdzie $A, B, C \in V_N$ *i* $a \in V_T$, *to mówimy, że* G *jest w* normalnej postaci Chomsky'ego.

Definicja 3 *Niech* $G = \langle V_N, V_T, P, S\rangle$; $\alpha, \beta, \gamma \in (V_N \cup V_T)^*$ *oraz* $A \in V_N$. *Mówimy, że ze słowa* $\alpha A \beta$ można wyprowadzić w G *słowo* $\alpha\gamma\beta$, *co zapisujemy* $\alpha A\beta \Rightarrow \alpha\gamma\beta$, *jeśli* $A \to \gamma$ *jest produkcją z* P.

Definicja 4 *Język* $L(G)$ *generowany przez gramatykę* $G = \langle V_N, V_T, P, S\rangle$ *definiujemy jako*

$$L(G) = \{w \mid w \in V_T^* \text{ oraz } S \stackrel{*}{\Rightarrow} w\},$$

gdzie $\stackrel{*}{\Rightarrow}$ *oznacza tranzytywne domknięcie relacji* $\Rightarrow$.

Przykład 1. Niech

- $V_N = \{S, T, L, R\}$;
- $V_T = \{(,)\}$;
- $P = \{S \to SS\ ;\ S \to LT\ ;\ S \to LR\ ;\ T \to SR\ ;\ L \to (\ ;\ R \to)\ \}$

Jak łatwo sprawdzić $L(G)$ jest językiem zawierający wszystkie słowa zbudowanie z poprawnie rozstawionych nawiasów.
Przykładowe wyprowadzenie słowa w=(()()):

$$S \Rightarrow LT \Rightarrow LSR \Rightarrow LSSR \Rightarrow LLRSR \Rightarrow LLRLRR \Rightarrow (LRLRR \Rightarrow$$

$$(LRL)R \Rightarrow (L)L)R \Rightarrow (L)()R \Rightarrow (()()R \Rightarrow (()())$$

□

Problem:

Dla ustalonej gramatyki bezkontekstowej $G = \langle V_N, V_T, P, S\rangle$ w normalnej postaci Chomsky'ego

Dane: słowo $w = a_1 \dots a_n$ ($a_i \in V_T$ dla $i = 1, \dots, n$)
Wynik: "TAK" - jeśli $w \in L(G)$
"NIE" - w przeciwnym przypadku.

2.5.2 Algorytm naiwny

Niech $M(w)$ oznacza zbiór słów wyprowadzalnych z w w jednym kroku.

```
F_0 ← {S}
for i = 1 to 2|w| − 1 do
    F_{i+1} ← ⋃_{w∈F_i} M(w)
if w ∈ F_{2|w|−1} then return "TAK" else return "NIE"
```

Poprawność: Każda produkcja gramatyki w normalnej postaci Chomsky'ego albo zwiększa o jeden długość wyprowadzanej frazy albo zamienia symbol nieterminalny na terminalny. Tak więc każde słowo z języka o długości n jest wyprowadzane z S po $2n-1$ krokach.

Koszt: Czynnikiem determinującym koszt algorytmu jest koszt pętli wewnętrznej, a ten w głównym stopniu zależy od wielkości zbiorów F_i. Niestety, nawet dla tak prostych gramatyk jak ta z Przykładu 1, zbiory F_i mogą zawierać wykładniczo wiele słów.

2.5.3 Algorytm dynamiczny

Idea:

Jeśli $w = a_1 \dots a_n$ jest słowem z języka $L(G)$, to pierwsza produkcja zastosowana w jego wyprowadzeniu (o ile $n > 1$) musi mieć postać $S \to AB$. Ponieważ dalsze wyprowadzenie z symbolu A jest niezależne od wyprowadzenia z symbolu B, więc musi istnieć i ($1 \le i \le n-1$) takie, że z $A \stackrel{*}{\Rightarrow} a_1 \dots a_i$ oraz $B \stackrel{*}{\Rightarrow} a_{i+1} \dots a_n$.

Na postawie tej obserwacji możemy łatwo zbudować algorym rekurencyjny, jednak czas jego działania może być wykładniczy. W szczególności algorytm taki wielokrotnie może próbować wyprowadzać ten sam fragment słowa w z tego samego symbolu nieterminalnego.

Przykład 2. Niech gramatyka zawiera (między innymi) produkcje $S \to AB$ oraz $A \to AA$. Na drugim poziomie rekursji rekurencyjna procedura może być wywoływana dla A i podsłów $a_1 \dots a_i$ (dla $i = 1, \dots, n-1$); wewnątrz każdego z tych wywołań będzie ona znów wywoływana m.in. dla A i podsłów $a_1 \dots a_j$ $(j = 1, \dots, i-1)$.

□

Podejście dynamiczne polega na obliczeniu dla każdego podsłowa słowa w (począwszy od podsłów jednoliterowych a skończywszy na całym w) zbioru nieterminali, z których da się to podsłowo wyprowadzić. Innymi słowy, celem jest wyznaczenie zbiorów $m_{i,j}$ $(1 \le i \le j \le n)$:

$$m_{i,j} = \{A \mid A \in V_N \ \& \ A \stackrel{*}{\Rightarrow} a_i \dots a_j\}$$

Odpowiedzią algorytmu będzie wartość wyrażenia $S \in m_{1,n}$.

Zbiory $m_{i,j}$ wyznaczyć można na podstawie następujących zależności:

$$m_{i,i} = \{A \mid (A \to a_i) \in P\} \text{ dla } i = 1, \dots, n$$

$$m_{i,j} = \bigcup_{k=i}^{j-1} m_{i,k} \otimes m_{k+1,j} \text{ dla } 1 \le i < j \le n$$

gdzie $m_{i,k} \otimes m_{k+1,j} = \{A \mid (A \to BC) \in P \text{ dla pewnych } B \in m_{i,k} \text{ oraz } C \in m_{k+1,n}\}$

Koszt: Łatwo sprawdzić, że algorytm wykonuje $\Theta(n^3)$ operacji $\otimes$. Ponieważ koszt jednej operacji $\otimes$ jest stały (patrz Uwagi implementacyjne), $\Theta(n^3)$ opisuje koszt całego algorytmu.

Uwagi implementacyjne. Elementy obliczanej tablicy są zbiorami. To stanowi istotną różnicę w stosunku do poprzednich przykładów, gdzie elementy tablicy były prostego typu. Przyjęcie odpowiedniej struktury danych do pamiętania zbiorów $m_{i,j}$ oraz wybór metody obliczania wyniku operacji $\otimes$ może mieć istotny wpływ na koszt algorytmu.

Przykładowo: zbiory $m_{i,j}$ możemy pamiętać jako wektory charakterystyczne lub jako listy. W pierwszym przypadku potrzebujeny $\sim (1/2)n^2|V_N|$ bitów na zapamiętanie tablicy. W drugim przypadku ponosimy spore koszty pamięciowe związane z używaniem wskaźników - jednak mogą one być opłacalne, gdy w średnim przypadku rozmiar zbiorów $m_{i,j}$ jest nieduży. W tym przypadku rozsądną metodą obliczania $m_{i,k} \otimes m_{k+1,j}$ może okazać się zwykłe przeglądanie list:

```
for each B ∈ m_{i,k} do
    for each C ∈ m_{k+1,j} do
        if BC jest prawą stroną produkcji z P
        then m_{i,j} ← m_{i,j} ∪ { symbol z lewej strony tej produkcji }
```

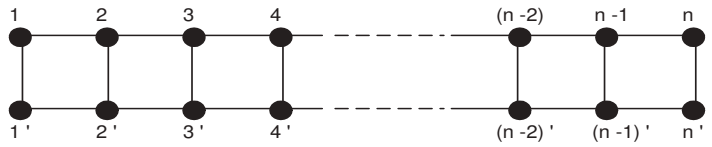
Przy odpowiednim zapamiętaniu informacji o produkcjach, koszt takiego obliczenia nie zależy od liczby produkcji i jest proporcjonalny do iloczynu długości list, co w rozważanym przypadku może być znacznie mniejsze od $|V_N|^2$. Jeśli liczba produkcji jest niewielka opłacalne może być zastosowanie innego sposobu:

```
for each (A → BC) ∈ P do
    if B ∈ m_{i,k} & C ∈ m_{k+1,j} then m_{i,j} ← m_{i,j} ∪ {A}
```

Sposób ten jest szczególnie atrakcyjny przy wektorowej reprezentacji zbiorów, ponieważ wówczas czas odpowiedzi na pytanie o przynależność elementu do zbioru jest stały i koszt powyższej pętli wynosi $\Theta(|P|)$.

2.6 Drzewa rozpinające drabin

Definicja 5 Drabiną *n-elementową nazywamy graf* D_n *przedstawiony na rysunku 1*



Rysunek 1: *Drabina* D_n

PROBLEM:

Dane: liczby naturalne n, k;
ciąg par liczb naturalnych $\{u_i, v_i\}$ $(i = 1, \ldots, m)$
INTERPRETACJA: pary $\{u_i, v_i\}$ określają wyróżnione krawędzie w n-elementowej drabinie;

Wynik: Liczba drzew rozpinających o k krawędziach wyróżnionych.

Ideę algorytmu przedstawimy rozważając prostszy problem, a mianowicie problem wyznaczania liczby drzew rozpinających w D_n (bez uwzględniania krawędzi wyróżnionych). Co prawda, w takim przypadku można w prosty sposób wyprowadzić zwięzły wzór na tę liczbę, lecz nie to jest naszym celem.

W dalszym ciągu, mówiąc o drabinie D_i, będziemy mieć na myśli podgraf drabiny D_i indukowany przez wierzchołki $\{1, \ldots, i, 1', \ldots, i'\}$.

Fakt 3 *Niech T będzie dowolnym drzewem rozpinającym drabiny D_{i+1}, dla dowolnego $i \geq 1$. Wówczas $T \cap D_i$ jest albo*

- *drzewem rozpinającym drabiny D_i albo*
- *lasem rozpinającym grafu D_i złożonym z dwóch drzew; jedno z tych drzew zawiera wierzchołek i a drugie - wierzchołek i'.*

Analogiczna własność zachodzi, gdy T jest lasem rozpinającym drabiny D_{i+1}, złożonym z dwóch drzew, przy czym jedno z tych drzew zawiera wierzchołek $(i+1)$, a drugie - wierzchołek $i'+1$).

Niech S_i oznacza zbiór drzew rozpinających drabiny D_i, a N_i zbiór lasów rozpinających, o których mowa w Fakcie 3, w drabinie D_i. Naszym celem jest policzenie wartości $|S_n|$.

IDEA ALGORYTMU: Kolejno dla $i = 1, \ldots, n$ liczymy wartości $|S_i|$ oraz $|N_i|$, korzystając z zależności przedstawionych w poniższym fakcie:

Fakt 4 *(a)* $|S_1| = |N_1| = 1$

(b) Dla każdego $i > 1$:

$$|S_i| = 3|S_{i-1}| + |N_{i-1}|,$$
$$|N_i| = 2|S_{i-1}| + |N_{i-1}|.$$

DOWÓD:

(a) Oczywiste.

(b) Niech $K_i = \{(i-1, i), ((i-1)', i'), (i, i')\}$ będzie zbiorem krawędzi, którymi D_i różni się od D_{i-1}.

Z dowolnego drzewa rozpinającego $T \in S_{i-1}$ można utworzyć trzy różne drzewa rozpinające z S_i poprzez dodanie dowolnych dwóch krawędzi ze zbioru K_i. Ponadto, dodając wszystkie krawędzie z K_i do dowolnego lasu z N_{i-1} można utworzyć jedno drzewo z S_i. To uzasadnia pierwszy ze wzorów.

Dodając krawędź $(i-1, i)$ do drzewa $T \in S_{i-1}$ otrzymujemy las z N_i. Jedno z jego drzew zawiera wierzchołek i, a drugie z drzew składa się z izolowaneo wierzchołka $\{i'\}$. Analogicznie otrzymujemy jeden las dodając do T krawędź $((i-1)', i')$. Ponadto, z każdego lasu z N_{i-1}, po dodaniu dwóch poziomych krawędzi $(i-1, i)$ oraz $((i-1)', i')$, otrzymujemy jeden las z N_i. To uzasadnia drugi wzór. □

Teraz w prosty sposób możemy tę metodę uogólnić do rozwiązania problemu liczenia drzew rozpinających z wyróżnionymi krawędziami. W tym celu zamiast czterech zbiorów S_i i N_i, rozważamy $2(k+1)$ zbiorów: $S_i(j)$, $N_i(j)$, gdzie parametr j $(j = 0, \ldots, k)$ oznacza liczbę krawędzi wyróżnionych. Przykładowo: $S_i(j)$ będzie się równać liczbie drzew rozpinających w drabinie D_i zawierających dokładnie j krawędzi wyróżnionych. Wyprowadzenie wzorów analogicznych do tych z Faktu 4 pozostawiamy jako proste ćwiczenie.

7 Algebraic Decision Trees (February 27 and March 4)

7.1 How Hard Is Almost Sorting?

Almost everyone is familiar with the $\Omega(n \log n)$ decision tree lower bound for sorting. Any binary decision tree that sorts must have at least $n!$ leaves, one for each possible permutation, and therefor must have depth at least $\log_2(n!) = \Omega(n \log n)$. It makes absolutely no difference what questions are asked in the internal nodes.

Now consider the following related problems.

- ELEMENT UNIQUENESS: Are any two elements of the input sequence $\langle x_1, x_2, \ldots, x_n \rangle$ equal?
- PARITY: Is the permutation of the input sequence $\langle x_1, x_2, \ldots, x_n \rangle$ even or odd?
- SET INTERSECTION: Given two sets $\{x_1, x_2, \ldots, x_n\}$ and $\{y_1, y_2, \ldots, y_n\}$, do they intersect?
- SET EQUALITY: Given two sets $\{x_1, x_2, \ldots, x_n\}$ and $\{y_1, y_2, \ldots, y_n\}$, are they equal?

We cannot apply the same purely information-theoretic argument to these problems, because there are only two possible outputs: YES and NO. And moreover, there *are* trivial decision trees with only one node that decide these problems.

Probably the most natural model to consider is the comparison tree model that we used to study sorting partial orders. It's possible to prove $\Omega(n \log n)$ lower bounds for the number of comparisons required to solve any of those four problems, using a fairly complicated adversary argument. (Essentially, we *must* sort the data to solve any of them.)

7.2 Linear Decision Trees

However, proving these lower bounds will actually be easier if we use a more powerful model of computation, introduced by David Dobkin and Richard Lipton in the late 1970s. In a *linear decision tree* (with input size n), every internal node is labeled with a vector $(a_0, a_1, \ldots, a_n)$ and has three outgoing edges labeled $-$, 0, and $+$. Given an input vector $(x_1, x_2, \ldots, x_n)$, we decide which way to branch based on the sign of the following expression:

$$a_0 + a_1 x_1 + a_2 x_2 + \cdots + a_n x_n.$$

For example, in each node in a comparison tree, we have $a_i = 1$ and $a_j = -1$ for some i and j, and $a_k = 0$ for all $k \neq i, j$.

Linear decision trees have a very simple geometric interpretation. Our generic problem can be thought of as a function $F : \mathbb{R}^n \to \{0, 1\}$, where the input is a point in n-space and the output is a single bit. Every internal node defines a hyperplane with equation

$$a_1 x_1 + a_2 x_2 + \cdots + a_n x_n = -a_0;$$

this equation describes a line when $n = 2$, a plane when $n = 3$, and so on. We branch at a node depending on whether the input is above, below, or on this hyperplane.[1]

Now consider the set of input points $R(v) \subseteq \mathbb{R}^n$ that reach a particular node v in a linear decision tree. The set $R(v)$ contains all the points that satisfy a set of linear equalities and inequalities; such a set is called a *convex polyhedron*. Recall that a set X is *convex* if for any two points $p, q \in X$, the entire line segment pq is contained in X. The intersection of any two convex sets is clearly convex;

[1] Linear decision trees are exactly the same as the *binary space partition trees* used in computer graphics systems.

any hyperplane divides space into three convex sets: the hyperplane itself and two *halfspaces*. Chugging through the definitions, we discover that convex polyhedra are, in fact, convex. (Whew!)

It is trivial to prove that every convex set is connected.[2] This simple observation gives us our first significant tool to prove lower bound in the linear decision tree model.

Lemma 1. *For any node v in any linear decision tree, $R(v)$ is connected.*

Now let $\#F_1$ denote the number of connected components of the set $F^{-1}(1)$ of points x such that $F(x) = 1$, and define $\#F_0$ similarly.

Lemma 2. *Any linear decision tree that computes the function $F : \mathbb{R}^n \to \{0, 1\}$ has depth at least $\lceil \log_3(\#F_0 + \#F_1) \rceil$.*

Proof: For any point $x \in \mathbb{R}^n$ such that $F(x) = 1$, there must be a 1-leaf ℓ that is reached by x. By the previous lemma, only points in the connected component of $F^{-1}(1)$ containing x can reach ℓ. it follows immediately that there are at least $\#F_1$ 1-leaves. Similarly, there must be at least $\#F_0$ 0-leaves. The lower bound now follows from the usual information-theoretic argument. □

Now we're ready to prove some lower bounds!

Theorem 3. *Any linear decision tree that computes the* Element Uniqueness *function has depth $\Omega(n \log n)$.*

Proof: Let $x = (x_1, x_2, \ldots, x_n)$ and $y = (y_1, y_2, \ldots, y_n)$ be two vectors with distinct coordinates that are sorted in two different orders. In particular, for some pair of indices i and j, we have $x_i < x_j$ and $y_i > y_j$. Any continuous path from x to y must contain a point z such that $z_i = z_j$, by the intermediate value theorem, but then we have $F(z) = 0$. Thus, points with different permutations are in different connected components of $F^{-1}(1)$, so $\#F_1 \geq n!$. The lower bound now follows immediately from Lemma 2. □

Theorem 4. *Any linear decision tree that computes the* Set Intersection *function has depth $\Omega(n \log n)$.*

Proof: For this problem, the input is a point $(x, y) = (x_1, x_2, \ldots, x_n; y_1, y_2, \ldots, y_n) \in \mathbb{R}^{2n}$, where the x- and y-coordinates represent the two sets X and Y. If X and Y are disjoint, then we can partition X and Y into disjoint subsets $X_1, X_2, \ldots, X_k$ and $Y_1, Y_2, \ldots, Y_k$ that satisfy the partial order

$$X_1 < Y_1 < X_2 < Y_2 < \cdots < X_k < Y_k.$$

Here, $A < B$ means that every element of A is less than every element of B, but the elements within A are incomparable. Either X_1 or Y_k or both could be empty.

There are exactly $n!^2$ such partial orders where the x- and y- elements alternate, that is, where each set X_i and Y_i is a singleton, and $k = n$. As in the previous theorem, any two points that obey different partial orders lie in different connected components of $F^{-1}(0)$. Thus, $\#F_0 \geq n!^2$, and the theorem follows immediately from Lemma 2. □

Finally, let's consider a function whose complexity is still open.

- 3Sum: Do any three elements in the input set $\{x_1, x_2, \ldots, x_n\}$ sum to zero?

[2]Careful with those vacuous cases: the empty set is both convex and connected!

The fastest algorithm known for this problem runs in $O(n^2)$ time, and this is conjectured to be optimal.[3] However, the following argument gives the strongest bound known in any general model of computation:

Theorem 5. *Any linear decision tree that computes the* 3SUM *function has depth* $\Omega(n \log n)$.

Proof: As usual, we prove the lower bound by counting the connected components of $F^{-1}(0)$. There are $\binom{n}{3}$ possible triples of input elements that could sum to zero. Each one defines a hyperplane of the form $x_i + x_j + x_k = 0$.

Suppose the Little Birdie tells us that

$$-2 < x_i < -1 \text{ for all } i < n/2, \qquad x_{n/2} = 0, \qquad 1 < x_i < 2 \text{ for all } i > n/2.$$

This set of inequalities defines a convex polyhedron Π. The Little Birdie Principle implies that the complexity of 3SUM is at least the complexity of 3SUM restricted to Π. Since Π is convex, we can apply the same counting arguments as when the input space is unrestricted.

A point $x \in \Pi$ has three coordinates that sum to zero if and only if $x_i = -x_j$ for some pair of indices $i < n/2$ and $j > n/2$. In other words, this restriction of 3SUM is exactly the same as the set intersection problem, for which we already have an $\Omega(n \log n)$ lower bound. In particular, $F^{-1}(0) \cap \Pi$ has at least $(n/2)!^2$ connected components. □

It may come as a surprise that this is the best lower bound we can prove for 3SUM using this method. Any set H of hyperplanes in $\mathbb{R}^n$ defines a cell complex called the *arrangement*. The full-dimensional cells are the connected components of $\mathbb{R}^n \setminus H$. A relatively straightforward inductive argument (in any computational geometry book) implies that the number of full-dimensional cells in an arrangement of N hyperplanes in $\mathbb{R}^n$ is at most

$$\sum_{d=0}^{n} \binom{N}{d} < N^n.$$

(The summation bound is exact if the hyperplanes are in *general position*: the intersection of any $d \le n$ hyperplanes has dimension $n - d$.) The 0-set for 3SUM consists of the full-dimensional cells in the arrangement of $\binom{n}{3}$ hyperplanes in $\mathbb{R}^n$, so its has *less than* $\binom{n}{3}^n = O(n^{3n})$ connected components. Thus, we have no hope of proving a $\omega(n \log n)$ lower bound by counting connected components.[4]

7.3 Algebraic Decision Trees

Now let's consider the following obvious generalization of linear decision trees, first proposed by Guy Steele and Andy Yao. In a *dth-order algebraic decision tree*, every node v is labeled with a polynomial $q_v \in \mathbb{R}[x_1, x_2, \ldots, x_n]$ of degree at most d. As in the linear case, each node has three branches labeled $-$, 0, and $+$, and computation at node v branches according to the sign of the polynomial expression $q_v(x)$. A 1st-order algebraic decision tree is just a linear decision tree.

[3] And in fact, that bound *is* optimal in a weak special case of the linear decision tree model of computation; see my PhD thesis!

[4] In fact, there is no general method to derive $\omega(n \log n)$ lower bounds in any of these decision/computation tree models, as long as the problem is defined by a polynomial number of equalities and inequalities. There are several similar techniques for proving lower bounds that use different notions of the "complexity" of a semi-algebriac set—the number of components in an intersection with a subspace, the volume, the Euler characteristic, the number of boundary features of each dimension, various Betti numbers, etc. The POTM Theorem and its generalizations imply that the complexity of the set 3SUM, for any reasonable notion of "complexity", is only $n^{O(n)}$.

Unfortunately, we can't use the same connectedness argument for algebraic decision trees as we did in the linear case. It's quite easy to come up with polynomials that divide $\mathbb{R}^n$ into more than three connected components, or pairs of polynomials p and q such that the sets $\{x \mid p(x) > 0\}$ and $\{x \mid q(x) > 0\}$ are connected, but their intersection is not.

However, the following theorem gives us a bound on the number of connected components of any semi-algebraic set.[5]

Theorem 6 (Petrovskiĭ, Oleĭnik, Thom, Milnor). *Let X be a semi-algebraic subset of $\mathbb{R}^n$ defined by m polynomial equations and h polynomial inequalities, each of degree at most $d \geq 2$. Then X has at most $d(2d-1)^{n+h-1}$ connected components.*

Corollary 7. *Any dth order algebraic decision tree that computes a function $F : \mathbb{R}^n \to \{0, 1\}$ has depth $\Omega(\log_d(\#F_0 + \#F_1) - n)$.*

Proof: Suppose F can be computed by a dth order algebraic decision tree with depth h. By the POTM Theorem, the set of points that reaches any leaf has at most $d(2d+1)^{n+h-1}$ connected components. Since there are less than 3^h leaves, we have the inequality

$$3^h d(2d+1)^{n+h-1} \geq \#F_0 + \#F_1.$$

Solving for h completes the proof.

$$\begin{aligned}
(6d+3)^h &\geq \frac{\#F_0 + \#F_1}{d(2d+1)^{n-1}}, \\
h &\geq \log_{6d+3} \frac{\#F_0 + \#F_1}{d(2d+1)^{n-1}} \\
&= \frac{\ln(\#F_0 + \#F_1)}{\ln(6d+3)} - (n-1)\frac{\ln(2d+1)}{\ln(6d+3)} - \frac{\ln d}{\ln(6d+3)} \\
&= \Omega(\log_d(\#F_0 + \#F_1) - n)
\end{aligned}$$

□

Corollary 8. *Any algebraic decision tree that computes the* Element Uniqueness *function has depth $\Omega(n \log n)$.*

Corollary 9. *Any algebraic decision tree that computes the* Set Intersection *function has depth $\Omega(n \log n)$.*

Corollary 10. *Any algebraic decision tree that computes the* 3sum *function has depth $\Omega(n \log n)$.*

7.4 Algebraic Computation Trees

Consider the following alternative algorithm for solving the Element Uniqueness problem. Given the input vector $(x_1, x_2, \ldots, x_n)$, we compute its *discriminant*

$$\prod_{1 \leq i < j \leq n} (x_i - x_j)$$

[5]This used to be called "Milnor's theorem", since John Milnor proved it in the late 1960s. Then some time in the 1980's, someone noticed that René Thom had proved the same theorem a few years earlier. Then in the late 1990s, some Russians pointed out that two Russian mathematicians, Petrovskiĭ and Oleĭnik, had proved the theorem several years before Thom. Finally, in the early 2000s, someone noticed that Milnor's paper actually cited Petrovskiĭ and Oleĭnik's earlier paper.

and compare it to zero. Clearly, the discriminant is zero if and only if some pair of elements is equal. It's possible to compute the discriminant in $O(n \log^2 n)$ time using Fast Fourier Transforms. In other words, we can solve the ELEMENT UNIQUENESS problem in near-linear time *without* sorting the input first, using a straight-line program without branches.

A further generalization of algebraic decision trees, proposed by Michael Ben-Or, captures algorithms of this type. An *algebraic computation tree* is a tree with two types of internal nodes.

- **Computation:** A computation node v has an associated value f_v determined by one of the instructions

$$f_v \leftarrow f_u + f_w \qquad f_v \leftarrow f_u - f_w \qquad f_v \leftarrow f_u \cdot f_w \qquad f_v \leftarrow f_u / f_w \qquad f_v \leftarrow \sqrt{f_u}$$

  where f_u and f_w are either values associated with ancestors of v, input values x_i, or arbitrary real constants.[6] Every computation node has one child.

- **Branch:** A branch node u contains one of the test instructions

$$f_u > 0 \qquad f_u \geq 0 \qquad f_u = 0$$

  where f_u is either a value associated with an ancestor of v or an input value x_i. Every branch node has two children.

Given an input $(x_1, x_2, \ldots, x_n)$, we follow a path from the root of the tree down to a leaf. At each computation node, we perform the corresponding arithmetic operation; at each branch node, we branch according to the result of the corresponding test. When we reach a leaf, its value is returned as algorithm's output. As usual, the running time of the algorithm is the length of the path traversed, and the worst-case running time is the depth of the tree.

The algebraic computation tree model can be equivalently described using a *real random access machine* or *real RAM*. A real RAM is pretty close to an actual (abstract) computer: it has random access memory, instructions, control flow, an execution stack, and so forth. The big difference is that the main memory in a real RAM stores *arbitrary real numbers*. Real arithmetic operations $+, -, \cdot, /, \sqrt{}$ and comparisons between real numbers all take constant time. A real RAM may also have integer variables, but **we are not allowed to convert between integer variables and real variables**.[7] In particular, the real RAM model does not allow use of the floor function $\lfloor x \rfloor$.

Given any algorithm written for the real RAM model, we can extract an algebraic computation tree by recording all possible branches and real arithmetic operations in any execution of the algorithm. Thus, any lower bound derived in the algebraic computation tree model immediately applies in the real RAM model as well.

Lemma 11. *Any algebraic computation tree that computes a function $F : \mathbb{R}^n \to \{0, 1\}$ has depth $\Omega(\log(\#F_0 + \#F_1) - n)$.*

Proof: Suppose F can be computed by an algebraic computation tree T with depth h. I claim that $2^{h+1}3^{n+h-1} \geq \#F_0 + \#F_1$; the lemma follows immediately from this claim by the usual arguments.

Let ℓ be a leaf of T with depth at most h. We can describe the set $R(\ell)$ of points that reach ℓ as a semi-algebraic set by manipulating the arithmetic and test instructions on the path from the

[6] Obviously, dividing by zero or taking the square root of a negative number is never allowed.

[7] If we could freely convert between integers and reals, and still do exact real arithmetic in constant time, we could solve some NP-hard problems in linear time.

root to ℓ. If v is a computation node, we obtain an algebraic equation as follows:

Operation	Equation
$f_v \leftarrow f_u + f_w$	$f_v = f_u + f_w$
$f_v \leftarrow f_u - f_w$	$f_v = f_u - f_w$
$f_v \leftarrow f_u \cdot f_w$	$f_v = f_u \cdot f_w$
$f_v \leftarrow f_u / f_w$	$f_u = f_v \cdot f_w$
$f_v \leftarrow \sqrt{f_u}$	$f_u = f_v^2$

Similarly, for any branch node v, we obtain either a new equation or a new inequality, depending on the outcome of the test.

Test	TRUE	FALSE
$f_u > 0$	$f_u > 0$	$-f_u \geq 0$
$f_u \geq 0$	$f_u \geq 0$	$-f_u > 0$
$f_u = 0$	$f_u = 0$	$f_u \cdot f_v = 1$

Note that an unsuccessful equality test introduces a new variable.

The points $(x_1, x_2, \ldots, x_n, f_1, f_2, \ldots, f_h)$ that satisfy this list of polynomial equalities and inequalities describe a semi-algebraic set $U(\ell) \subseteq \mathbb{R}^{n+h}$. Since every polynomial in this list has degree at most 2, the POTM Theorem implies that $U(\ell)$ has at most $2 \cdot 3^{n+h-1}$ connected components.

A point $x \in \mathbb{R}^d$ reaches leaf ℓ if and only if there exists a point $f = (f_1, f_2, \ldots, f_h) \in \mathbb{R}^h$ such that $(x, f) \in U(\ell)$. In other words, we can obtain $R(\ell)$ by projecting $U(\ell)$ onto its first n coordinates. Since projection can only decrease the number of connected components, we conclude that $R(\ell)$ has at most $2 \cdot 3^{n+h-1}$ components. The claim, and thus the lemma, now follows from the fact that T has at most 2^h leaves. □

The lower bound argument requires only that we pay for multiplications, divisions, roots, and branches; additions and subtractions could be performed for free. In fact, the lower bound holds in a more general model where each computation node computes an arbitrary bilinear function of its ancestor values. At the cost of a factor of d, we can also include an operation that computes the roots of an arbitrary polynomial of degree d, whose coefficients are ancestor values.[8] We can even allow computations with complex numbers, by representing each $z = x + yi$ as a pair (x, y), or equivalently, allowing the projection operators $\Re$ and $\Im$ to be performed at no cost.

Corollary 12. *Any algebraic computation tree that computes the* ELEMENT UNIQUENESS *function has depth* $\Omega(n \log n)$.

Corollary 13. *Any algebraic computation tree that computes the discriminant* $\prod_{1 \leq i < j \leq n} (x_i - x_j)$ *has depth* $\Omega(n \log n)$.

Proof: Once we compute the resultant, we can solve the ELEMENT UNIQUENESS problem with just one more branch. □

More accurately, we have an $\Omega(n \log n)$ lower bound on the number of multiplications required to compute the resultant. This lower bound is actually tight; the resultant can be computed using $O(n \log n)$ multiplications, in $O(n \log^2 n)$ time. (Most of the additional time is additions and subtractions.)

[8] More accurately, the natural cost of computing a root of a polynomial $P(x)$ is the minimum time required to evaluate $P(x)$. In particular, the cost of computing $\sqrt[d]{f_u}$ is $\Theta(\log d)$, by repeated squaring.

Corollary 14. *Any algebraic computation tree that computes the* SET INTERSECTION *function has depth* $\Omega(n \log n)$.

Corollary 15. *Any algebraic computation tree that computes the* 3SUM *function has depth* $\Omega(n \log n)$.

7.5 Generic Width

Now consider the problem of computing the maximum element in a set of n numbers. It's not hard to prove a lower bound of $n-1$ in the comparison tree model using an adversary argument, but this argument doesn't generalize to arbitrary algebraic decision or computation trees. Perhaps there is a faster algorithm to compute the maximum element using higher-degree polynomials! Alas, the following argument of Montaña, Pardo, and Recio implies that there is no such algorithm.[9]

Any closed semi-algebraic set X in $\mathbb{R}^n$ can be written in the canonical form

$$X = \bigcup_{i=1}^{t} \bigcap_{j=1}^{r} \{p_{ij} \geq 0\},$$

where each p_{ij} is a polynomial with n variables. The *width* of X is defined as the minimum r for which such a representation is possible. By convention, the empty set and $\mathbb{R}^n$ both have width zero. The *generic width* of a (not necessarily closed) semi-algebraic set X is defined as

$$\overline{w}(X) = \min\{\text{width}(S) \mid \dim(S \oplus X) \leq n\},$$

where the minimum is taken over all closed semi-algebraic sets S. The generic width of X never exceeds the width of X; take $S = X$.

Lemma 16. *Any algebraic decision or computation tree that decides whether a point $x \in \mathbb{R}^n$ lies inside a fixed semi-algebriac set X has depth at least $\overline{w}(X)$.*

Proof: Let T be an algebraic decision tree, and let ℓ be an arbitrary leaf with depth h. Let $R(\ell)$ be the set of possible inputs that reach ℓ. If the path to ℓ contains any $=$-branches, the dimension of $R(\ell)$ is less than n, so $\overline{w}(R(\ell)) = 0$. Otherwise, $R(\ell)$ is the intersection h *open* algebraic halfspaces, which implies that $\overline{w}(R(\ell)) \leq h$.

Since T correctly decides membership in X, we must have

$$X = \bigcup_{\text{1-leaves } \ell} R(\ell),$$

which implies that

$$\overline{w}(X) \leq \max_{\text{1-leaves } \ell} \overline{w}(R(\ell)), \leq \max_{\text{1-leaves } \ell} \text{depth}(\ell) \leq \text{depth}(T).$$

Unlike most results about algebriac decision trees, this argument does *not* require a constant upper bound on the degree of the query polynomials. Thus, it immediately applies to algebaric computation trees as well. □

Theorem 17. *Any algebriac decision or computation tree that computes the largest element of an n-element set has depth at least $n-1$.*

[9] This result is often credited to Rabin, but his proof has a bug.

Proof (sketch): In fact, we can prove an $n-1$ lower bound for the simpler problem of *verifying* the largest element, or equivalently, determining membership in the polyhedron

$$X = \bigcap_{i=2}^{n} \{x_1 - x_i \geq 0\}.$$

It is a fairly tedious exercise to prove that $\overline{w}(X) \geq n-1$. (It is hopefully obvious that $\overline{w}(X) \leq n-1$; the lower bound is considerbly less trivial.) □