THE WAY OF LONGEST PALINDROME

Listiarso Wastuargo – 135 08 103

Program Studi Teknik Informatika
Institut Teknologi Bandung
Jl. Ganesha 10, Bandung
e-mail: hallucinogenplus@gmail.com

ABSTRAK

Makalah ini membahas tentang palindrom. Lebih spesifiknya makalah ini melakukan pembahasan bagaimana cara menghitung palindrom terpanjang dalam sebuah string masukan. Makalah ini juga akan melakukan perbandingan antara algoritma-algoritma yang dapat digunakan untuk mencari palindrom terpanjang dalam sebuah string dan mencari yang terbaik diantara algoritma-algoritma yang diberikan.

Kata kunci: palindrom, *string matching*, *dynamic programming*, *string, branch and bound*.

I. PENDAHULUAN

Dahulu manusia sulit berkomunikasi satu sama lain. Namun kini budaya dan bahasa sudah berkembang pesat. Manusia yang berbeda suku pun bisa berkomunikasi satu sama lain.

Kebudayaan yang berkembang pesat ini melahirkan bahasa dan tulisan-tulisan yang menarik. Salah satu tulisan yang menarik adalah tulisan yang dibatasi. Tulisan yang dibatasi adalah tulisan yang memiliki batasan-batasan tertentu yang ditentukan oleh pembuat tulisan. Contoh batasan yang umum adalah membuat tulisan yang diakhiri atau diawali oleh huruf yang sama atau berbunyi bila dilafalkan sama.

Masih mengenai tulisan yang dibatasi, ada satu batasan yang menarik yang dapat dikaji. Batasan tersebut adalah membuat sebuah tulisan yang dibaca sama persis dari kiri ke kanan maupun dari kanan ke kiri. Batasan tersebut disebut palindrom. Palindrom berasal dari bahasa Yunani "palin" yang berarti "lagi" dan "dromos" yang berarti "jalan" atau "arah".

Setelah melakukan pengamatan berulang-ulang kita pun mengetahui bahwa dalam sebuah kata atau kalimat kita dapat membentuk beberapa subkata atau subkalimat atau lebih general dengan nama substring. Substring ini mungkin saja adalah sebuah palindrom. Dan didalam palindrom itu sendiri, mungkin saja terdapat palindrom-palindrom mini yang lain. Sehingga manusia yang selalu ingin tahu pasti mulai bertanya-tanya : "Diberikan sekumpulan simbol (bisa berupa huruf, angka atau simbol-simbol lain) yang disusun menjadi sebuah string, tentukanlah palindrom terpanjang yang ada di dalam string tersebut"

II. Dasar Teori

A. String dan Beberapa Terminologi String

String

Pada contoh pohon di atas, simpul 7 dan 5 adalah anak dari simpul 2, dan simpul 2 adalah orang tua dari simpul 7 dan 5.

Substring

Sebuah string A adalah substring dari string B bila apabila kita memotong string B (dari depan atau dari belakang), kita dapat membuat string A.

Formalnya, sebuah substring dari sebuah string $T = t_1..t_n$ adalah sebuah string $T_s = t_{1+i}..t_{m+i}$ dimana $0 \leq i$ dan $m + i < n$

Prefix

Sebuah prefix dari sebuah string $T = t_1..t_n$ adalah sebuah string $T_s = t_1..t_m$ dimana $m \leq n$. Sebuah proper prefix dari sebuah string adalah sebuah prefix yang bukan merupakan string itu sendiri (memenuhi $0 \leq m < n$).

Suffix

Sebuah suffix dari sebuah string $T = t_1..t_n$ adalah sebuah string $T_s = t_{n-m+1}..t_n$ dimana $m \leq n$. Sebuah proper suffix dari sebuah string adalah sebuah suffix yang bukan merupakan string itu sendiri (memenuhi $0 < m \leq n$).

B. Palindrom

Palindrom adalah sebuah kata, kalimat, frase, atau sekumpulan bilangan yang memiliki arti yang sama bila dibaca dari arah berbeda (dari kiri ke kanan atau dari kanan ke kiri).

III. Pencarian Palindrome Terpanjang

Memastikan apakah sebuah string adalah palindrom adalah sebuah permasalahan yang tergolong kuno dan tidaklah menarik. Sedangkan mencari palindrom terpanjang pada sebuah string menjadi sebuah topik bahasan yang cukup menarik karena kita dapat mengkaji beberapa algoritma yang dapat menyelesaikannya.

A. *The Naive Way*

Cara paling naif adalah dengan memanfaatkan permasalahan yang ada sebelumnya : "Bila diberikan sebuah string, tentukan apakah string tersebut palindrom atau tidak." Hal ini bisa kita lanjutkan menjadi sebuah algoritma :

1. Bentuklah seluruh substring yang mungkin dari string masukan.
2. Dari tiap-tiap substring, cek apakah substring tersebut palindrom atau tidak.
3. Bila substring tersebut palindrom, simpan substring tersebut dan bandingkan dengan substring palindrom terpanjang saat ini.
4. Bila substring yang baru saja didapatkan lebih panjang dari substring palindrom terpanjang saat ini, ubah substring palindrom terpanjang saat ini menjadi substring tersebut.

Kodenya akan menjadi seperti berikut dalam C++ :

```
string longestPalindromeNaive(string inp){
  int strlen = inp.length();
  string result = "";

  for (int i = strlen; i >= 1; --i){
    for (int j = 0 ; j <= strlen - i; ++j){
      string pal = "";
      for (int k = 0; k < i; ++k)
        pal += inp[j + k];

      bool isPal = true;
      for (int k = 0 ; k < i / 2 && isPal; ++k)
        if (pal[k] != pal[i - k - 1])
          isPal = false;

      if (isPal)
        if (result.length() < i)
          result = pal;
    }
  }

  return result;
}
```

Bisa dilihat bahwa kompleksitas pembuatan seluruh substring adalah $O(N^2)$ dan pengecekan kepalindroman sebuah substring adalah O(N) sehingga menghasilkan total kompleksitas $O(N^3)$ untuk algoritma ini.

Algoritma ini masih bisa diperbaiki lagi dengan melakukan penambahan strategi *branch and bound*. Strategi ini diterapkan dengan mengakhiri pencarian palindrom apabila sebuah palindrom telah ditemukan. Hal ini dikarenakan bila kita melakukan pengulangan dengan menguji substring terpanjang hingga substring terpendek, palindrom yang kita temukan pertama pasti merupakan substring palindrom terpanjang saat ini. Kodenya sedikit berubah menjadi :

```
string longestPalindromeBranchAndBound(string
inp){
  int strlen = inp.length();

  for (int i = strlen; i >= 1; --i){
    for (int j = 0 ; j <= strlen - i; ++j){
      string pal = "";
      for (int k = 0; k < i; ++k)
        pal += inp[j + k];

      bool isPal = true;
      for (int k = 0 ; k < i / 2 && isPal; ++k)
        if (pal[k] != pal[i - k - 1])
          isPal = false;

      if (isPal)
        return pal;
    }
  }

  return "";
}
```

B. *The Middle Way*

Dengan mengamati lemma : "Bila diberikan sebuah string A yang tidak palindrom dam 2 buah sembarang karakter b dan c, bAc tidak mungkin palindrom." Dari lemma ini kitadapat mencari palindrom-palindrom yang ada didalam string masukan dengan lebih cepat, langkah langkahnya adalah :

1. Tentukan titik tengah palindrom (bisa berupa satu buah karakter atau buah karakter bila dua karakter yang bersebelahan tersebut sama).
2. Dari titik tersebut, catat karakter paling kiri dan karakter paling kanan sebagai titik tengah palindrom tersebut.
3. Apabila karakter paling kiri sama dengan karakter paling kanan, geser karakter paling kiri ke kiri dan karakter paling kanan ke kanan
4. Ulangi proses hingga karakter paling kiri tidak sama dengan paling karakter kanan atau karakter paling kiri telah mencapai karakter paling kiri string masukan atau karakter paling kanan telah mencapai karakter paling kanan string masukan.
5. Simpan substring ini, bandingkan dengan palindrom terpanjang yang kita miliki saat ini. Update palindrom terpanjang bila diperlukan.
6. Ulangi langkah 1-5 hingga seluruh titik tengah yang mungkin dalam string masukan telah dicoba.

Pengamatan ini membuat kita tidak perlu melakukan pengecekan setiap substring. Dan kode kita menjadi :

```
  string longestPalindromMiddle(string inp){
    int strlen = inp.length();
    string result = "";

    for (int i = 0; i < strlen; ++i){
      string pal = "";
      int j = 0;
      while (i - j >= 0 && i + j < strlen &&
inp[i - j] == inp[i + j])
        ++j;

      if (2 * j - 1 > result.length()){
        result = "";
        for (int k = i - j + 1; k < i + j; ++k)
          result += inp[k];
      }

      pal = "";
      j = 0;
      while (i - j >= 0 && i + j + 1 < strlen &&
inp[i - j] == inp[i + j + 1])
        ++j;

      if (2 * j > result.length()){
        result = "";
        for (int k = i - j + 1; k <= i + j; ++k)
          result += inp[k];
      }
    }

    return result;
  }
```

Penentuan titik tengah memiliki kompleksitas O(N). Sedangkan pencarian palindrom terpanjang dari sebuah titik tengah memiliki kompleksitas terburuk O(N). Dengan menggabungkan keduanya, kita mendapatkan kompleksitas total dari algoritma ini adalah $O(N^2)$.

C. Algoritma yang Lebih Baik?

Setelah mendapatkan fakta bahwa kita dapat menghitung palindrom terpanjang dengan kompleksitas tercepat $O(N^2)$, pasti terbayang untuk menemukan algoritma yang lebih cepat. Kita mungkin saja mendapatkan komplekstias total O(N log N) atau bahkan O(N).

Bila kita mendapatkan fakta bahwa ada algoritma dengan kompleksitas O(N) untuk permasalahan mencari palindrom terpanjang ini, kita tentunya akan berpikir bahwa algoritma O(N) ini akan dibentuk berdasarkan algoritma $O(N^2)$ (*the middle way*) yang baru saja kita bahas. Sebagai percobaan pertama kita akan melakukan algoritma berikut :

1. Tentukan titik tengah palindrom.
2. Ekspansi ke kiri dan kenanan hingga kita menemukan palindrom terpanjang yang bisa kita dapatkan dari titik tengah ini.
3. Simpan substring ini lalu bandingkan dengan palindrom terpanjang yang kita miliki saat ini. Update palindrom terpanjang bila diperlukan.
4. Ubah titik tengah menjadi karakter terakhir dari palindrom atau karakter terakhir dari palindrom ditambah satu karakter setelah palindrom bila dua karakter ini sama. Posisi titik tengah ini kedepannya akan kita sebut sebagai titik "khusus".
5. Ulangi langkah 1 sampai 4 hingga titik tengah mencapai akhir dari string masukan.

Bisa dilihat bahwa langkah 1 dan langkah 3 adalah langkah yang sama dengan langkah-langkah algoritma *the middle way*. Perbedaannya adalah kita tidak perlu mengecek semua titik tengah yang ada (seluruh karakter di string masukan tersebut), melainkan hanya di titik-titik "khusus". Dan sesuai dengan keinginan kita, algoritma ini memiliki kompleksitas total O(N).

Tapi ada satu hal yang perlu disayangkan. Algoritma ini menghasilkan keluaran yang salah. Sebagai contoh bila kita menjalankan algoritma ini untuk string "abababa", algoritma kita akan mendeteksi palindrom "**a**babababa", "**aba**babaa" hingga akhirnya "ab**ababa**" padahal palindrom terpanjang yang bisa didapat adalah "**abababa**"!

D. Pengamatan Properti Palindrom

Kita bisa membenarkan kesalahan ini dengan mengubah cara kita memilih titik "khusus". Dengan sedikit pengamatan kita bisa mengetahui bahwa titik "khusus" itu adalah titik tengah palindrom yang terkandung dalam palindrom yang sedang diperiksa saat ini. Lebih spesifik lagi, palindrom yang dimulai dari titik "khusus" pasti akan menjadi proper suffix dari palindrom yang sedang kita periksa. Masih belum cukup, titik tengah ini juga harus merupakan proper suffix terpanjang dan palindrom dalam palindrom yang sedang kita periksa. Sehingga algoritma kita akan berubah menjadi seperti berikut :

1. Tentukan titik tengah palindrom.
2. Ekspansi ke kiri dan kenanan hingga kita menemukan palindrom terpanjang yang bisa kita dapatkan dari titik tengah ini.
3. Simpan substring ini lalu bandingkan dengan palindrom terpanjang yang kita miliki saat ini. Update palindrom terpanjang bila diperlukan.
4. Ubah titik tengah menjadi titik tengah proper suffix yang palindrom terpanjang dari palindrom yang sedang kita periksa.
5. Ulangi langkah 1 sampai 4 hingga titik tengah mencapai akhir dari string masukan.

Untuk menemukan titik "khusus" ini sekarang tidak semudah dan seefisien algoritma semula. Bila kita ceroboh, mencari titik "khusus" ini bisa memakan waktu O(N), sedangkan proses pencarian palindrom masih O(N) sehingga kompleksitas kembali menjadi $O(N^2)$ yang tidak sesuai dengan harapan. Disinilah saatnya kita memanfaatkan sifat-sifat palindrom dan *dynamic programming*.

Pertama perlu diketahui bahwa proper suffix palindrom kita ini akan muncul paling tidak dua kali di palindrom yang sedang kita cek. Bahkan proper suffix ini adalah sekaligus proper prefix juga. Hal ini terjadi karena palindrom ini adalah palindrom!

E. *The way of O(N)*

Setelah kita memahami sifat palindrom, maka kita bisa memanfaatkan sifat ini dan menggabungkannya dengan teknik pemrograman dinamis untuk menghasilkan algoritma yang efisien.

Kita telah sampai pada tahap bahwa kita diharuskan mencari proper suffix atau proper prefix palindrom terpanjang dari palindrom yang sedang diperiksa. Hal ini sangat sulit dilakukan bila kita tidak memiliki sebuah tabel untuk menyimpan palindrom yang telah kita dapatkan saat ini. Tabel ini digunakan agar kita dapat mengiterasi tabel ini dan mungkin mendapatkan proper suffix atau proper prefix palindrom terpanjang dari palindrom yang sedang diperiksa. Dengan melakukan sedikit pengamatan hal ini berimplikasi bahwa kita hanya perlu menyimpan seluruh palindrom yang ada di string masukan dari awal hingga titik "khusus" sehingga kita pasti paling tidak akan mendapatkan proper prefix dari palindrom yang sedang diperiksa.

Dengan kebutuhan untuk menyimpan seluruh palindrom yang telah kita temukan sampai suatu titik "khusus" kita masih tidak bergerak dari algoritma *the middle way* kita. Tapi sekarang kita berakhir pada pertanyaan yang berbeda. Diberikan titik "khusus", palindrom terpanjang dari titik tersebut, tabel yang berisi palindrom hingga saat ini (hanya perlu menyimpan panjangnya saja); dapatkah kita menemukan titik "khusus" yang baru dan meneruskan isi tabel hingga titik "khusus" tersebut dengan efisien. Sebagai contoh bila kita memiliki sedang pada tahap :

- "ab**a**ba|??", tabelnya [0, 1, 0, 3, 0, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?]
- "aba**b**a|??", tabelnya [0, 1, 0, 3, 0, 5, 0, ?, ?, ?, ?, ?, ?, ?, ?]
- "aba**b**aba|" , tabelnya [0, 1, 0, 3, 0, 5, 0, 7, 0, 5, 0, 3, 0, 1, 0]

Hal terpenting yang perlu kita perhatikan adalah, tabel ini ternyata juga palindrom karena string masukannya sendiri juga palindrom. Bahkan tabel ini memiliki properti yang lebih general : *palindrom terpanjang yang terletak sejauh D kekanan dari titik "khusus" akan paling tidak sepanjang palindrom terpanjang yang terletak sejauh D kekiri dari titik "khusus" bila palindrom D-kekiri secara sepenuhnya berada didalam palindrom yang sedang diperiksa dan sama panjangnya dengan palindrom D-kekanan tersebut bila palindrom D-kekiri tersebut bukanlah prefix dari palindrom yang sedang diperiksa atau palindrom yang sedang diperiksa adalah suffix dari string masukan*. Hal ini berarti kita dapat mengisi palindrom kekanan dari suatu titik "khusus" bila kita mengetahui palindrom kekiri nya. Sebagai contoh bila kita memiliki tabel [0, 1, 0, 3, 0, 5, ?, ?, ?, ?, ?, ?, ?, ?, ?], kita dapat secara intuitif menebak bahwa isi dari tabel selanjutnya adalah [0, 1, 0, 3, 0, 5, 0, ≥3?, 0, ≥1?, 0, ?, ?, ?, ?]. Hal ini juga membuat kita dapat memilih ≥3? sebagai titik tengah yang baru karena palindrom yang sedang diperiksa saat ini bukanlah suffix dari string masukan.

Algoritma menghitung palindrom terpanjang kita berakhir menjadi :

1. Inisialisasi tabel untuk memuat semua titik tengah yang mungkin.
2. Pilih titik tengah pertama sebagai titik "khusus".
3. Ekspansi ke kiri dan kenanan hingga kita menemukan palindrom terpanjang yang bisa kita dapatkan dari titik tengah ini.
4. Masukkan panjang palindrom ini ke posisi tabel yang sesuai.
5. Iterasi kebelakang tabel dan isi palindrom-palindrom yang merupakan suffix dari palindrom yang sedang diperiksa hingga suatu posisi. Ubah titik "khusus" ke posisi ini.
6. Ulangi langkah 1 sampai 5 hingga titik tengah mencapai akhir dari string masukan.
7. Cek kembali seluruh tabel dan dapatkan palindrom yang paling panjang.

Kode kita akhirnya berakhir menjadi seperti ini :

```
string longestPalindromeLinear(string str){
  int len = str.length();
  vector<int> res;
  int i = 0;
  int palLen = 0;

  while (i < len){
    if (i > palLen && str[i - palLen -1] ==
str[i]){
      palLen += 2;
      i += 1;
      continue;
    }

    res.push_back(palLen);

    int right = res.size() - 2;
    int left = right - palLen;
    bool found = false;

    for (int j = right; j > left; --j){
      int now = j - left - 1;

      if (res[j] == now){
        palLen = now;
        found = true;
        break;
      }

      res.push_back(min(now, res[j]));

    }

    if (!found){
      ++i;
      palLen = 1;
    }
  }

  res.push_back(palLen);

  int right = res.size() - 2;
  int left = right - (2 * len + 1 -
res.size());

  for (int i = right; i > left; --i){
    int now = i - left - 1;
    res.push_back(min(now, res[i]));
```

```
    }

    string result = "";
    for (int i = 2; i < res.size(); i += 2){
      int tengah = i / 2;
      if (res[i] > result.length()){
        result = "";
        int gerak = res[i] / 2;
        for (int j = tengah - gerak; j < tengah +
gerak; ++j)
          result += str[j];
      }

      if (i + 1 >= res.size()) break;

      if (res[i + 1] > result.length()){

        result = "";
        int gerak = res[i + 1] / 2;

        for (int j = tengah - gerak; j <= tengah
+ gerak; ++j)
          result += str[j];
      }
    }
    return result;
  }
```

Algoritma ini hanya melakukan pengisian tabel, melakukan ekspansi palindrom dari titik "khusus" atau melakukan perpindahan titik "khusus" pada tiap iterasinya. Karena tabel yang perlu diisi memiliki ukuran dua kali panjang string masukan, algoritma ini berjalan pada kompleksitas linier alias O(N).

IV. Kesimpulan

Kompleksitas terbaik yang bisa didapatkan setelah melakukan beberapa pengamatan algoritma pencarian palindrom terpanjang hingga saat ini adalah O(N). Hal ini merupakan kemajuan yang cukup drastis mengingat kita memulai pengamatan kita dengan algoritma yang memiliki kompleksitas $O(N^3)$.

Dengan melakukan pengamatan berulang-ulang dan menggunakan strategi algoritma yang berbeda untuk menyerang suatu permasalahan, kita bisa mendapatkan solusi terbaik untuk suatu permasalahan.

REFERENSI

[1] http://en.wikipedia.org/wiki/Substring 7 Desember 2010 pukul 13.19

[2] http://en.wikipedia.org/wiki/Palindrome 7 Desember 2010 pukul 14.00

[3] http://www.akalin.cx/2007/11/28/finding-the-longest-palindromic-substring-in-linear-time/ 7 Desember 2010 pukul 16.20

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 7 November 2010

Listiarso Wastuargo 13508103