Pemecahan Masalah *Longest Increasing Subsequence* Memanfaatkan Program Dinamis dan Binary Search

Chalvin
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
13514032@std.stei.itb.ac.id

***Abstract*— Salah satu persoalan klasik dalam ilmu komputer adalah mencari longest common subsequence. Aplikasi dari hal ini pun tidak terbatas dalam bilang ilmu komputer saja, bidang biologi misalnya, menggunakan solusi permasalahan ini dalam menentukan kemiripan dari beberapa DNA. Beberapa cara mendapatkan solusi permasalahan ini adalah dengan brute force, dynamic programming, serta binary search.**

Keywords—longest increasing subsequent; Dynamic Programming; Binary Search

I. PENGANTAR

Salah satu persoalan klasik dalam ilmu komputer adalah mencari longest common subsequence. Aplikasi dari hal ini pun tidak terbatas dalam bilang ilmu komputer saja, bidang biologi misalnya, menggunakan solusi permasalahan ini dalam menentukan kemiripan dari beberapa DNA.

Dalam makalah ini, penulis ingin menspesifikasi salah satu anak dari persoalan longest common subsequence, yaitu longest increasing subsequence.

II. DASAR TEORI

A. *Exhautive Search*

Dalam ilmu komputer, *Exhautive search* adalah salah satu teknik problem solving yang mengenerasi semua kemungkinan kandidat dari solusi dan mengecek apakah kandidat tersebut memenuhi persyaratan dari permasalahannya. Misalnya dalam mencari faktor dari sebuah bilangan bulat, exhautive search akan mencari semua bilangan dari 1 sampai n, lalu kemudian mengecek apakah n dapat dibagi oleh bilangan tersebut tanpa sisa. Pada persoalan 8 ratu, exhautive search akan mengenumerasi seluruh kemungkinan peletakan 8 ratu, kemudian mengecek apakah ada ratu yang dapat saling serang.

Exhautive search umumnya mudah dalam pengimplementasiannya karna imtuitif dan selalu mendapatkan solusi. Namun, umumnya exhautive search tidak mangkus sehingga hanya dipakai untuk persoalan dengan banyak data yang sedikit.

Beberapa teknik heuristik juga dapat diterapkan pada exhautive search sehingga dapat membatasi jumlah kandidat yang mungkin. Contohnya pada persoalan 8 ratu, daripada menaruh ratu secara acak, kita taruh satu ratu di masing-masing kolom. Pergerakan ratu kita batasi hanya bergerak sepanjang baris, sehingga jumlah kandidat yang mungkin terbentuk berkurang secara drastis.

B. *Binary Search*

Binary search adalah salah satu algoritma dasar dalam ilmu komputer. Binary search digunakan dalam mencari sebuah nilai pada array yang terurut nilainya. Binary search bekerja dengan menggunakan subsequence kontigu yang mana pasti terdapat nilai yang kita cari yang kita sebut ruang pencarian. Awalnya, ruang pencarian binary search adalah seluruh array. Pada tiap tahap, binary search membandingkan median dari subsequence terhadap nilai yang kita cari. Berdasarkan hasil perbandingan, kita dapat memangkas ruang pencarian sampai setengahnya. Sebagai contoh bila kita mencari nilai 55 dari

0, 5, 13, 19, 22, 41, 55, 68, 72, 81, 98

Pertama kita tetapkan ruang pencarian dari indeks ke-0 sampai indeks ke-10. Lalu kita bandingkan nilai median (41) dengan nilai yang kita cari (55). Karena nilai median < nilai yang kita cari, pastilah nilai yang kita cari berada di sebelah kanan median. Ruang pencarian kita dapat dipangkas mencari

55, 68, 72, 81, 98

Kita lakukan kembali hal yang sama dan kita dapatkan

55, 68

Bergantung cara kita menentukan median, kita akan mendapat indeks nilai yang kita cari pada tahap ini atau tahap

selanjutnya. Dan kita dapati bahwa nilai yang kita cari berada pada indeks ke-6.

C. *Dynamic Programming*

Program dinamis , seperti *divide-and-conquere,* menyelesaikan masalah dengan menggabungkan solusi pada upamasalah. Pada *divide-and-conquere*, masalah dipecah menjadi upa-upa masalah, menyelesaikan upa-upa masalah tersebut secara rekursif, serta menggabungkan solusinya untuk menyelesaiian persoalan yang paling awal. Pada program dinamis menggunakan cara yang hamper sama dengan *divide-and-conquere*, hanya saja pada program dinamis, upamasalah umumnya mempunyai upaupamasalah (subsubproblem) yang sama. *Divide-and-conquere* menyelesaikan upaupamasalah ini berulang-ulang sehingga menjadi tidak terlalu efisien. Program dinamis menyelesaikan ketidakefisienan tersebut dengan menyimpan hasil penyelesaian upaupamasalah sehingga jika diperlukan lagi, maka upaupamasalah tersebut tidak perlu lagi diselesaikan, melainkan hanya mengambil nilai yang disimpan.'

Umumnya, program dinamis dipakai dalam masalah pengoptimasisasi. Masalah-masalah tersebut akan memiliki banyak solusi penyelesaian.Setiap solusi yang mungkin akan memiliki suatu nilai, dan kita ingin solusi sehinga nilai tersebut teroptimasi(baik memaksimalkan maupun meminimasi).

Dalam menyelesaikan masalah dengan program dinamis, ada 4 tahapan yang akan kita lakukan :

1. Mengkarakteristik struktur dari solusi optimal.
2. Secara rekursif mendefinisikan nilai dari solusi optimal
3. Menghitung nilai dari solusi optimal, umumnya secara bottom-up.
4. Menghasilkan solusi optimal dari informasi yang didapatkan.

Langkah 1-3 adalah inti dari program dinamis. Jika kita hanya memerlukan nilai dari solusi optimal, langkah 4 dapat kita buang. Sebaliknya, bila kita ingin solusi optimal yang menghasilkan nilai optinal tersebut, kita dapat menambahkan informasi tambahan pada langkah 3 sehingga setelah mengetahui nilai optimal, kita dapat mencari solusi optimal tersebut.

D. *Longest Common Subsequence*

Ilmu biologi sering kali membandingkan DNA dari dua atau lebih sample dari organisme berbeda. DNA sendiri terdiri dari 4 macam basa : adenine (A), Guenine(G), Cytosine (C), dan thymine(T). Kita dapat mengekspresikan DNA sebagai string yang terdiri atas huruf A,G,T , dan C. Contohnya DNA pada suatu organisme adalah

S1 = ACCGGTCGAGTGCGCGGAAGCCGGCCGAA

dan DNA dari organisme lain adalah

S2 = GTCGTTCGGAATGCCGTTGCTCTGTAAA.

Salah satu alasan mengkomparasi dua buah DNA adalah untuk mencari tau seberapa mirip kedua DNA tersebut. Suatu DNA dikatakan mirip bila salah satu DNAnya merupakan substring dari DNA yang lain. Pada Contoh diatas, S1 dan S2 bukanlah substring dari satu sama lain. Maka dari itu, kita juga punya definisi lain untuk menyeatakan kedekatan suatu DNA, yaitu melihat berapa banyak perubahan yang harus dilakukan untuk mengubah satu DNA mwnjadi DNA yang lain. Cara lain adalah dengan mencari DNA baru S3, dimana basa S3 muncul pada S1 dan S2 dengan urutan kemunculan yang sama, namun tidak perlu muncul berturut-turut. Contoh, pada contoh diatas, S3 = GTCGTCGGAAGCCGGCCGAA.

Perhitungan kedekatan yang ketika dapat kita reduksi menjadi permasalahan longest-common-sequence. Subsequence dari sebuah string adalah himpunan karakter yang muncul pada S dengan urutan kemunculan yang sama, namun tidak perlu berurutan. Contoh

ACTTGCG

- ACT, ATTC, T, ACTTGC merupakan subsequence dari ACTTGCG
- TTA bukan subsequence dari ACTTGCG

Common-subsequence dari dua buah string adalah subsequence yang muncul pada kedua string. Longest-common-subsequence merupakan common-subsequence yang memiliki panjang maksimal.. Contoh

S 1 = AAACCGTGAGTTATTCGTTCTAGAA

S 2 = CACCCCTAAGGTACCTTTGGTTC

Salah satu common-subsequence terpanjangnya adalah

S 1 = AA**ACC**G**T**G**AGTTA**TT**C**G**TTC**T**A**G**AA**

S 2 = C**A**CC**CCTA**AG**GTAC**C**TTTG**GTTC

LCS = ACCTAGTACTTG

E. *Longest Increasing Subsequence*

Longest-Increasing-subsequence adalah bentuk khusus dari Longest-common-subsequence, Dimana kita membandingkan sebuah string dengan hasil sorting dari string tersebut. Sebagai contoh

0, 8, 4, 12, 2, 10, 6, 14, 1, 9, 5, 13, 3, 11, 7, 15

Dapat dicari longest-increasing-subsequencenya dengan mensortir data tersebut, lalu dicari longest-common-subsequencenya.

0, 8, 4, 12, **2**, 10, **6**, 14, 1, **9**, 5, 13, 3, **11**, 7, **15**

0, 1, **2**, 3, 4, 5, **6**, 7, 8, **9**, 10, **11**, 12, 13, 14, **15**

LCI = 0, 2, 6, 9, 11, 15

III. ANALISIS DAN IMPLEMENTASI

A. *Solusi Naïf*

Solusi naif dalam mencari longest increasing subsequence adalah secara rekursif, mencari semua increasing subsequence dan mencari increasing subsequence terpanjang. Pseudocodenya adalah sebagai berikut

```
Lis(A[1..N]):
  max <- 0
  for each subsequence B of A do
    if B is increasing subsequence and
length(B) > max
      max <- length(B)

  -> max
```

Kompleksitas dari solusi ini adalah O(N 2^N), karena ada 2^N buah subsequence yang mungkin ada serta kompleksitas untuk mengecek apakah B merupakan increasing subsequence adalah O(N). Solusi ini tentunya tidak dapat dipakai karena tidak mangkus.

B. Solusi Program Dinamis

Kita dapat merepresentasikan masalah ini hanya dengan satu state i. Misalkan LIS(I) adalah LIS(longest-increasing-subsequence) yang berakhir di indeks I. Kita tahu bahwa LIS(0) = 1 karena angka pertamanya merupakan salah satu subsequent yang valid. Untuk I >= 1, LIS(I) menjadi sedikit lebih kompleks. Kita harus mencari suatu indeks J sehingga J < I dan A[J} < A[I} dengan LIS(J) adalah yang terbesar. Setelah menemukan J, kita tahu bahwa LIS(I) = 1 + LIS(J). Fungsi rekursifnya sendiri dapat kita tuliskan sebagai berikut

1. LIS(0) = 1 {BASIS}
2. LIS(I) = max (LIS(J) + 1), Dimana 0 < J < I-1 dan A[J] < A[I] {REKURSIF}

Misalkan kita ingin mencari longest increasing sequence dari

7, 10, 9, 2, 3, 8, 8, 1

- LIS(0) = 1
- LIS(1) = 2, karena kita dapat memperpanjang LIS(0) = {-7} dengan {10} untuk membentuk {-7, 10} dengan panjang 2. J yang diambil adalah 0.
- LIS(2) = 2, karena kita dapat memperpanjang LIS(0) = {-7} dengan {9} untuk membentuk {-7, 9} dengan panjang 2. J yang diambil adalah 0.
- LIS(3) = 2, karena kita dapat memperpanjang LIS(0) = {-7} dengan {2} untuk membentuk {-7, 2} dengan panjang 2. J yang diambil adalah 0.
- LIS(4) = 3, karena kita dapat memperpanjang LIS(3) = {-7, 2} dengan {3} untuk membentuk {-7, 2, 3} dengan panjang 3. J yang diambil adalah 3.
- LIS(5) = 4, karena kita dapat memperpanjang LIS(4) = {-7, 2, 3} dengan {8} untuk membentuk {-7, 2, 3, 8} dengan panjang 4. J yang diambil adalah 4.
- LIS(6) = 4, karena kita dapat memperpanjang LIS(4) = {-7, 2, 3} dengan {8} untuk membentuk {-7, 2, 3, 8} dengan panjang 4. J yang diambil adalah 4.
- LIS(7) = 2, karena kita dapat memperpanjang LIS(0) = {-7} dengan {1} untuk membentuk {-7, 1} dengan panjang 2. J yang diambil adalah 0.
- Jadi, LIS dari

  7, 10, 9, 2, 3, 8, 8, 1

  Adalah LIS(5) dan LIS(6), yaitu 4.

Dalam penyelesaian persoalan diatas, jelas terlihat bahwa banyak sekali submasalah yang dipanggil berulang-ulang karena dalam menghitung LIS(I), kita harus menghitung LIS(J) dengan J antara 0 sampai I-1. Tapi, hanya ada N buah state yang berbeda (yaitu panjang arraynya) sehingga kompleksitas dari program dinamis ini adalah O(N^2).

C. Solusi N log N

Misalkan A = {2, 5, 3}. Dari pengamatan, kita tahu bahwa LISnya adalah {2,5} atau {2,3}. Bila kita menambahkan 2 buah elemen, misalkan 7 dan 11, maka longest increasing subsequence kita menjadi {2, 3, 7, 11} dan {2,5,7,11}.

Bila kita menambahkan satu lagi elemen, yaitu 8, larik kita akan menjadi {2, 5, 3, 7, 11, 8}. Pertanyaannya, apakah kita menambahkan 8 ke dalam LIS kita? Bila iya bagaimana? Jika kita ingin menambahkan 8 ke dalam LIS yang sudah ada, maka 8 harus ditempatkan setelah 7 dan menggantikan 11.

Karena kita tidak tau input selanjutnya, kita tidak akan yakin apakah menambahkan akan membuat LIS kita kedepannya menjadi lebih panjang. Misalkan ada elemen 9 pada larik input, misalnya {2, 5, 3, 7, 11, 8, 7, 9}, kita dapat mengganti elemen 11 pada LIS kita menjaadi 8 karena ada elemen yang dapat memanjangkan LIS kita nantinya

Dari pengamatan diatas, misalkan E adalah elemen paling belakang dari sequence terpanjang. Kita dapat menambahkan elemen saat ini A[I] ke dalam LIS yang sudah ada jika ada suatu elemen A[J] dimana J > I sehingga A[I] < A[J] dan E > A[I]. Pada contoh diatas, E = 11, A[i] = 8 dan A[j] = 9.

Misalkan pada array A = {2, 5, 3}, kita ingin menambahakn elemen baru yaitu 1. Bagaimana dia bisa memanjangkan LIS yang sekarang ada, yaitu {2, 5} dan {2, 3}? Tentu saha tidak bisa. Namun, bisa saja elemen tersebut merupakan awal dari LIS yang lebih panjang daripada LIS yang sudah ada. Misalnya {2, 5, 3, 1, 2, 3, 4, 5, 6}. Menjadikan 1 sebagai awal dari LIS baru tentu menciptakan LIS dengan panjang maksimal.

Dari observasi ini, kita melihat bahwa bila kita menemukan angka terkecil yang baru, angka tersebut berpotensial menjadi awal dari LIS baru.

Secara umum, kita punya himpunan seluruh LIS yang mungkin. Kita menambahkan elemen A[I] ke dalam LIS. Kita

cari seluruh elemen terbesar LIS untuk mencari elemen terbesar yang lebih kecil dari A[i].

Bila:

1. A[i] lebih kecil dibanding seluruh elemen terakhir LIS, buat LIS baru dengan A[I] sebagai elemennya.

2. A[i] lebih besar dibanding seluruh elemen terakhir LIS, kita biat LIS baru dengan mengambil isi dari LIS terpanjang, dan tambahkan A[i] di Lis tersebut.

3. A[I] ada di antara, kita cari Lis yang memiliki elemen terakhir terbesar yang lebih kecil dibanding A[i].

Contoh

0, 8, 4, 12, 2, 10, 6, 14, 1, 9, 5, 13, 3, 11, 7, 15

A[0] = 0. Kasus 1. Tidak ada LIS, buat LIS baru

0.

==========

A[1] = 8. Kasus 2. kopi dan buang

0.

0, 8.

==========

A[2] = 4. Kasus 3. copy, panjangkan, buang

0.

0, 4.

0, 8. Dibuang

==========

A[3] = 12. Kasus 2. kopi dan buang

0.

0, 4.

0, 4, 12.

==========

A[4] = 2. Kasus 3. copy, panjangkan, buang

0.

0, 2.

0, 4. Dibuang.

0, 4, 12.

==========

A[5] = 10. Kasus 3. copy, panjangkan, buang

0.

0, 2.

0, 2, 10.

0, 4, 12. Dibuang.

==========

A[6] = 6. Kasus 3. copy, panjangkan, buang

0.

0, 2.

0, 2, 6.

0, 2, 10. Dibuang.

==========

A[7] = 14. Kasus 2. kopi dan buang

0.

0, 2.

0, 2, 6.

0, 2, 6, 14.

==========

A[8] = 1. Kasus 3. copy, panjangkan, buang

0.

0, 1.

0, 2. Dibuang.

0, 2, 6.

0, 2, 6, 14.

==========

A[9] = 9. Kasus 3. copy, panjangkan, buang

0.

0, 1.

0, 2, 6.

0, 2, 6, 9.

0, 2, 6, 14. Dibuang.

==========

A[9] = 9. Kasus 3. copy, panjangkan, buang

0.

0, 1.

0, 2, 6.

0, 2, 6, 9.

0, 2, 6, 14. Dibuang.

==========

A[10] = 5. Kasus 3. copy, panjangkan, buang

0.

0, 1.

0, 1, 5.

0, 2, 6. Dibuang.

0, 2, 6, 9.

==========

A[11] = 13. Kasus 2. kopi dan buang

0.

0, 1.

0, 1, 5.

0, 2, 6, 9.

0, 2, 6, 9, 13.

==========

A[12] = 3. Kasus 3. copy, panjangkan, buang

0.

0, 1.

0, 1, 3.

0, 1, 5. Dibuang.

0, 2, 6, 9.

0, 2, 6, 9, 13.

==========

A[13] = 11. Kasus 3. copy, panjangkan, buang

0.

0, 1.

0, 1, 3.

0, 2, 6, 9.

0, 2, 6, 9, 11.

0, 2, 6, 9, 13. Dibuang.

==========

A[14] = 7. Kasus 3. copy, panjangkan, buang

0.

0, 1.

0, 1, 3.

0, 1, 3, 7.

0, 2, 6, 9. Dibuang.

0, 2, 6, 9, 11.

==========

A[15] = 15. Kasus 2. kopi dan buang

0.

0, 1.

0, 1, 3.

0, 1, 3, 7.

0, 2, 6, 9, 11.

0, 2, 6, 9, 11, 15. <== LIS List

==========

Dalam implementasinya, kita tidak perlu menyimpan seluruh LIS yang ada, cukup menyimpan elemen terakhir dari tiap LIS. Mencari bilangan terbesar yang lebih kecil dari A[i] membutuhkan kompleksitas O(log N) menggunakan binary search, dan kita memproses n buah data, sehingga kompleksitas algoritma ini adalah O(n log n)

IV. Ucapan terima Kasih

Pertama-tama saya ingin berterima kasih kepada Tuhan karena melalui anugrah-Nya saya bisa menyelesaikan makalah ini. Saya juga berterima kasih kepada orang tua saya yang senantiasa mendukung saya dikala duka. Saya juga sangat berterima kasih kepada dosen saya Dr. Ir. Rinaldi Munir dan Dr.Nur Ulfa Maulidevi, karena telah memberikan inspirasi bagi saya. Saya juga berterima kasih kepada Institut Teknologi Bandung, tempat dimana saya menuntut ilmu ketika saya menyelesaikan makalah ini.

V. Daftar Pustaka

[1] Thomas H.Cormen, et al, Introduction to Algorithm, 3rd ed.

[2] http://www.geeksforgeeks.org/longest-monotonically-increasing-subsequence-size-n-log-n/

[3] Steven Halim, Competitive programming 3, 2013

[4] Rinaldi Munir, Diktat Kuliah IF2211 Strategi Algoritma, Program

[5] Studi Teknik Informatika ITB, 2009

VI. 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, 6 Mei 2016