Optimasi Pencarian Kata dalam Scrabble Menggunakan Memoization

Kartini Lovian Simbolon[1] , Siti Nur Aarifah[2] , Elisabeth Claudia Simanjuntak[3],Pardi Octaviando[4],Ahmad Rizqi[5]
Jurusan Sains Data, Fakultas Sains, Institut Teknologi Sumatera
Lampung Selatan, Indonesia
Email: Kartini.122450003@student.itera.ac.id siti.122450006@student.itera.ac.id elisabeth.122450123@student.itera.ac.id pardi.122450132@student.itera.ac.id ahmad.122450138@student.itera.ac.id

1. Latar Belakang

Suatu permainan scrabble memiliki nilai tertentu setiap huruf yang dimiliki yang dapat mempengaruhi skor permainan, Selain itu juga pada papan permainan perlu memperhatikan petak-petak khusus yang dapat menggandakan nilai kata yang sudah dibuat.contoh nya petak *double letter score* yang akan menggandakan nilai huruf tertentu oleh karena itu dibutuhkan strategi untuk bermain permainan scrabble untuk menemukan kata-kata yang *valid* dan mempertimbangkan nilai mencapai skor tertinggi yang menggunakan pemrograman memoization

Dalam meningkatkan efisiensi pencarian kata pada permainan *scrabble* untuk membentuk huruf-huruf yang tersedia ,disini kita akan menggunakan Teknik memorization yang akan sangat berguna untuk menyimpan dan mengakses hasil komputasi yang telah dihitung sebelum nya tanpa menghitung kembali. Sehingga *memoization* akan sangat berguna Ketika terdapat permintaan pencarian yang sama sehingga tidak membuat program banyak bekerja berulangkali karena informasi kata-kata sebelumnya telah ditemukan dan disimpan.

2. Teori Dasar

2.1 *Functools*

Functools adalah modul penting dalam bahasa pemrograman Python yang menyediakan berbagai fitur dan fungsi untuk meningkatkan kegunaan dan kelincahan dalam penulisan kode. Salah satu fitur utamanya adalah kemampuan untuk bekerja dengan fungsi tingkat tinggi (*higher-order functions*), yang memungkinkan fungsi-fungsi untuk menerima fungsi lain sebagai argumen atau mengembalikan fungsi lain sebagai hasil. Dengan menggunakan *decorators*, yang merupakan fitur kunci dalam *functools*, pengembang dapat menambahkan fungsionalitas tambahan ke fungsi-fungsi mereka tanpa harus mengubah implementasi inti dari fungsi tersebut.

Selain itu, *functools* juga menyediakan alat bantu untuk memoization, sebuah teknik yang berguna untuk meningkatkan kinerja fungsi yang sering dipanggil dengan parameter yang sama. Melalui decorator *lru_cache*, modul ini memungkinkan penyimpanan hasil dari pemanggilan fungsi dan pengembalian hasil tersebut dari *cache* ketika fungsi dipanggil kembali dengan parameter yang sama. Dengan demikian, *functools* membantu pengembang Python dalam mengoptimalkan kinerja aplikasi mereka dengan

mudah. Dengan berbagai fitur dan fungsi yang disediakannya, modul *functools* menjadi alat yang sangat berguna dalam pemrograman Python, memungkinkan pengembang untuk menulis kode yang lebih efisien, kelincahan, dan mudah dipelihara.

2.2 *Decorator*

Decorator diterapkan pada fungsi dan memiliki tujuan untuk menambah perilaku mereka. Fungsi yang didekorasi biasanya dikenakan oleh fungsi lain, yang disebut sebagai fungsi asli. Idenya adalah bahwa *decorator* menambahkan beberapa atribut baru ke fungsi asli, menambahkan fungsionalitas baru padanya, dan bahkan menggantinya dengan perilaku yang sama sekali berbeda. Penggunaan *decorator* mengasumsikan sebuah baris kode yang ditempatkan di atas sebuah fungsi atau metode yang dimulai dengan karakter @, diikuti dengan nama dekorator dan argumen. Dekorator digunakan di berbagai *framework***.**

*Decorato*r biasanya merupakan fungsi yang mengambil fungsi lain sebagai argumen, dan kemudian mengembalikan fungsi baru. Fungsi yang dikembalikan digunakan untuk menggantikan fungsionalitas fungsi asli, yang juga disebut fungsi yang dihias. Penggunaan *decorator* secara sintaksis terlihat mirip dengan anotasi di Java dan C#. Namun, perbandingan yang lebih baik adalah membandingkan *decorato*r dengan makro dalam C/C++, Terkadang, dekorator dalam Python tercampur dengan pola desain *decorator*. Meskipun pola *decrator* dapat diimplementasikan menggunakan *decorator*, *decorator* juga dapat digunakan untuk mengimplementasikan pola desain lainnya. *Decorator* dapat digunakan untuk mengimplementasikan pencatat waktu, pengatur waktu, konversi format, kerangka kerja, dan sebagainya. Mereka menawarkan mekanisme yang kuat dan sintaks dekorasi yang mudah untuk menyediakan fungsionalitas yang dimodifikasi dari fungsi *decorator.*

2.3. *Synthetic Sugar*

*Syntactic suga*r memperluas bahasa pemrograman yang sudah ada dengan mengizinkan sintaks ringkas dari pola penggunaan bahasa yang sering muncul. Contoh *synthactic sugar* adalah fitur foreach. Foreach memungkinkan pengembang untuk mengulang setiap elemen dalam koleksi, yang merupakan pola penggunaan umum untuk perulangan, dengan cara yang lebih ringkas daripada menggunakan pernyataan for. Selain itu, meningkatkan keterbacaan kode sumber. Dengan menggunakan foreach sebagai contoh, memahami perulangan foreach jauh lebih mudah dibandingkan dengan perulangan *for-loop* karena kita tidak perlu menginterpretasikan kondisi perulangan secara eksplisit. Namun, semua orang tidak memiliki pendapat yang sama tentang penggunaan *syntactic sugar.* Ada beberapa kritik baik dari kalangan akademis maupun industri. Salah satu pepatah terkenal mengatakan "*Syntactic sugar* menyebabkan kanker titik koma" dalam *Epigrams on Programming* oleh

Allen Perlis . Ilmuwan komputer yang hebat ini menyinggung bahwa pengejaran kenyamanan yang berlebihan akan mengaburkan beberapa penggunaan bahasa pemrograman.

2.4 B. *Top-down approach (Memoization)*

Memoization merupakan Teknik pemrograman yang mempercepat performa dengan caching *return value* sebagai hasil dari suatu fungsi. Fungsi yang *memoized* akan langsung mengeluarkan output berupa *value* solusi dari tahapan tersebut dari hasil yang pernah dicapai sebelumnya. *Memoization* adalah bentuk spesifik dari chacing yang sangat berguna untuk fungsi yang dilakukan pemanggilan berkali-kali dengan argument yang sama. Teknik ini sangat menaikan efisiensi dan juga mengurangi kerja CPU. Kita dapat melihat contohnya dari sebuah pemanggilan fungsi fibonaci.

2.5 *Lru Cache*

Salah satu kebijakan yang paling populer adalah apa yang disebut dengan kebijakan penggantian "*Least Recently Used*" (LRU). Di bawah kebijakan LRU, cache dapat dianggap sebagai antrean. Ketika sebuah berkas baru diminta, berkas tersebut akan ditambahkan ke bagian depan antrean (atau dipindahkan ke bagian depan jika berkas tersebut sudah ada dalam antrean). Jika sebuah berkas mencapai ekor antrian, maka berkas tersebut akan "terdorong keluar" (yaitu dikeluarkan dari cache). Karena *file* yang paling populer adalah *file* yang paling sering diminta, maka file tersebut memiliki kemungkinan yang lebih besar untuk disimpan di cache sehingga pengiriman konten menjadi lebih cepat. Sebagai hasilnya, kinerja dan properti (baik secara teoritis maupun empiris) merupakan topik dari banyak penelitian (dan referensi di dalamnya). Ada banyak perluasan dari kebijakan LRU klasik, termasuk q-LRU dan LRU (m) yang terutama berfokus pada situasi di mana seluruh *file* di-*cache*. Salah satu masalah dengan kebijakan LRU adalah bahwa permintaan berkas yang besar akan menggusur beberapa berkas kecil dalam cache dan dengan demikian dapat merusak kinerja sistem. Untuk menghindari efek ini, makalah ini mengusulkan sebuah generalisasi dari kebijakan penggantian LRU (dilambangkan sebagai gLRU). Secara umum, berkas dapat dibagi menjadi beberapa bagian yang berukuran sama atau potongan-potongan dengan ukuran yang sama. penggantian gLRU berbeda dengan LRU karena ketika sebuah berkas diminta, hanya satu potongan tambahan yang ditambahkan ke dalam *cache* (kecuali jika semua potongan berkas tersebut sudah ada di dalam *cache*). Sebagai contoh, misalkan sebuah dokumen dengan 100 potongan diminta dan 10 potongan saat ini di-*cache*. Di bawah gLRU, 10 potongan tersebut akan dipindahkan ke kepala cache bersama dengan 1 potongan tambahan. Dalam kebijakan LRU, seluruh berkas akan ditambahkan. Kami menyatakan bahwa gLRU menghasilkan kinerja yang lebih baik (misalnya, kecepatan unduh yang lebih cepat, penundaan yang lebih sedikit, dll.) daripada LRU

2.6 Fronzenset

Frozenset adalah tipe data dalam bahasa pemrograman Python yang mirip dengan *set*, tetapi bersifat tidak berubah (*immutable*), artinya setelah dibuat, elemen-elemen di dalamnya tidak dapat diubah. Ini berguna ketika kita ingin menggunakan set sebagai kunci dalam kamus atau sebagai elemen dari set lainnya. `frozenset` biasanya digunakan ketika kita ingin membuat set yang tetap konsisten dan tidak dapat diubah setelah pembuatan, terutama dalam konteks memoization, caching, atau saat bekerja dengan struktur data yang memerlukan kunci yang tidak berubah. Dengan `frozenset`, kita dapat memastikan kestabilan dan keandalan data dalam berbagai situasi, membantu dalam membuat kode yang lebih aman dan dapat diandalkan.

3. .Metode

Metode yang dipakai dalam permain scrabble pada kasus ini menggunakan tiga teknik yaitu memoization, syntatic sugar, dan decorator. Fungsi *lru_cahce* yang ada pada *library functools* yang berguna untuk menyimpan hasil pemanggilan atau disebut sebagai teknik *memoization.* Pada teknik *memoization* ini kita memakai list untuk menambahkan kata baru dan memakai dictionary untuk menambahkan skor untuk setiap huruf baru. Pada proses permainan scrabble ini kita menggunakan syntatic kode untuk membantu pembuatan kode agar lebih ringkas dan mudah dipahami.

Teknik selanjutnya untuk menyelesaikan permainan ini kita pakai *decorator* yang diletakan pada fungsi mencari_kata_terbaik, yang dimana setiap fungsi ini dipanggil maka dia akan menggunakan *decorator* untuk *memoization* atau hasil dari pemanggilan fungsi akan disimpan dalam cahce sehingga jika ada pemanggilan yang sama tidak perlu untuk melakukan perhitungan ulang yang dapat menyebabkan *overhead* jika terlalu banyak

4. Hasil dan Pembahasan

```
# Menambahkan kata-kata baru ke dalam dictionary
dictionary = [
    "apple", "pie", "pear", "bear", "ant", "bee", "gorilla", "fox", "giraffe",
    "zebra", "quartz", "jazz", "quiz"
]

# Memperbarui skor untuk setiap huruf termasuk huruf yang baru
scores = {
    "a": 1, "p": 3, "l": 1, "e": 1, "i": 1, "r": 1, "b": 3, "n": 4, "t": 1,
    "g": 2, "o": 1, "f": 4, "x": 8, "z": 10, "q": 10, "u": 1, "j": 8
}
```

Gamabar 4.1 Kode *dictionary* dan *scores*

Kode di atas merupakan contoh implementasi pencarian kata terbaik berdasarkan kumpulan karakter yang diberikan, dengan memanfaatkan memoization

untuk meningkatkan efisiensi. Pertama-tama, sebuah kamus (`dictionary`) yang berisi sejumlah kata telah didefinisikan, bersama dengan skor untuk setiap huruf dalam kamus skor (`scores`).

```
def menghitung_score(kata):
    return sum(scores.get(char, 0) for char in kata)
```

Gambar 4.2. Kode menghitung *scores*

Fungsi `menghitung_score(kata)` digunakan untuk menghitung skor dari suatu kata berdasarkan skor huruf yang telah ditentukan. Kemudian, fungsi `mencari_kata_terbaik(kata)` didekorasi dengan `@lru_cache(maxsize=None)`, sehingga memoization diterapkan pada fungsi tersebut. Ini berarti bahwa hasil dari setiap pemanggilan fungsi akan disimpan dalam cache untuk penggunaan kembali di masa mendatang, mengurangi jumlah perhitungan yang perlu dilakukan.

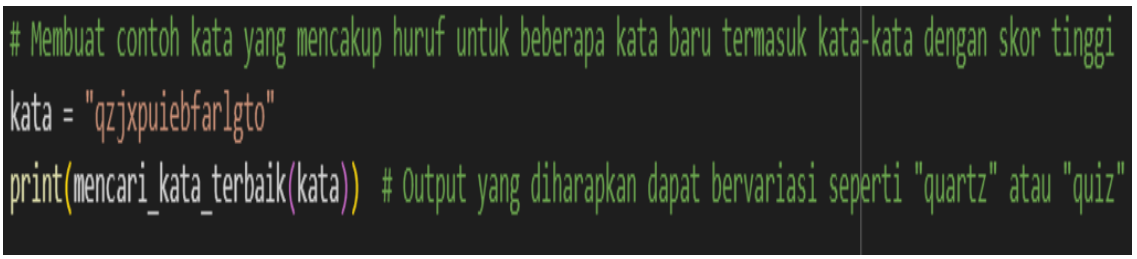
```
# Menggunakan decorator untuk memoization
@lru_cache(maxsize=None)
def mencari_kata_terbaik(kata):
    skor_terbaik = 0
    kata_terbaik = None
    paket_kata = frozenset(kata)

    for kata in dictionary:
        if frozenset(kata).issubset(paket_kata):
            score = menghitung_score(kata)
            if score > skor_terbaik:
                skor_terbaik = score
                kata_terbaik = kata

    return kata_terbaik
```

Gambar 4.3. Kode mencari kata terbaik

Dalam fungsi `mencari_kata_terbaik`, sebuah loop `for` digunakan untuk iterasi melalui setiap kata dalam kamus. Pada setiap iterasi, kami memeriksa apakah set karakter dari kata dalam kamus merupakan subset dari set karakter dari kata yang diberikan. Jika ya, skor kata tersebut dihitung menggunakan fungsi `menghitung_score`, dan jika skor tersebut lebih tinggi dari skor terbaik yang sebelumnya disimpan, kata tersebut dianggap sebagai kata terbaik baru. Akhirnya, kata terbaik yang ditemukan dikembalikan sebagai output dari fungsi.



Gambar 4.4. Kode membuat contoh kata dan *print*

Sebagai contoh, kita membuat sebuah kumpulan karakter *kata* yang mencakup sejumlah huruf. Kemudian, kita memanggil fungsi *mencari_kata_terbaik* dengan kata sebagai argumen, dan mencetak hasilnya. Output yang diharapkan adalah kata terbaik yang dapat dibentuk dari kumpulan karakter tersebut, yang mungkin berupa *quartz* atau *quiz* atau kata lainnya yang memiliki skor tertinggi dari kata-kata yang sesuai. Dengan demikian, implementasi ini menggabungkan memoization untuk meningkatkan efisiensi, dengan pencarian kata terbaik berdasarkan kumpulan karakter yang diberikan.

5. **esimpulan**

Dengan demikian, dapat disimpulkan bahwa kode tersebut mengadopsi berbagai metode untuk meningkatkan pencarian kata dalam permainan *Scrabble*, terutama dengan penggunaan memoization. Metode ini memungkinkan penyimpanan hasil dari pemanggilan fungsi sebelumnya, menghindari perhitungan ulang saat parameter yang sama diberikan. Penggunaan decorator @lru_cache dari modul functools memfasilitasi implementasi memoization dengan baik, menyimpan hasil pemanggilan fungsi dan mengembalikannya dari cache saat fungsi dipanggil kembali dengan parameter yang sama. Ini tidak hanya meningkatkan kinerja, tetapi juga membuat kode lebih sederhana dan mudah dipahami.

Selain itu, kode tersebut memanfaatkan frozenset untuk menunjukkan himpunan karakter dalam kata. Dengan menggunakan frozenset, kode dapat efektif memetakan apakah kata dalam kamus merupakan subkumpulan dari karakter yang ada dalam kata yang diberikan. Penggunaan frozenset membantu dalam mempercepat proses pencarian kata yang sesuai dengan aturan permainan Scrabble, membentuk strategi yang lebih efektif dalam mencapai skor tertinggi.

Secara keseluruhan, kode tersebut berhasil mencapai skor tertinggi dengan mengoptimalkan pencarian kata dalam Scrabble, menerapkan strategi yang efektif, dan mempertimbangkan nilai untuk mencapai skor maksimum. Dengan menggunakan memoization dan metode lain seperti syntactic sugar dan decorator, kode ini memastikan kinerja optimal tanpa mengorbankan kejelasan dan keefektifan implementasi.

6. **Daftar pustaka**

[1]Kim, D., & Yi, G. (2014). Measuring syntactic sugar usage in programming languages: An empirical study of c# and java projects. *Lecture Notes in Electrical Engineering*, *279 LNEE*, 279–284. https://doi.org/10.1007/978-3-642-41674-3_40

[2]*memoization*. (n.d.).

[3]Popovic, T., & Popović, T. (2015). *NAPREDNE TEHNIKE U PYTHON-U: DEKORATORI ADVANCED PYTHON TECHNIQUES: DECORATORS*. https://www.researchgate.net/publication/272833859

[4]Santanapurba, H., Ati Sukmawati, Mk. R., & Ahmad Faisal, Mk. (n.d.). *DASAR-DASAR BAHASA PEMROGRAMAN GOLANG*.