Perbandingan Kecepatan Algoritma Rekursif dan Memoization pada Deret Fibonacci Menggunakan Delphi

Ade Hastuty[1*], Nurul Hikma[2], Suci Ramadhani[3], Syahrul Mustakim[4]
Program Studi Teknik Informatika, Universitas Muhammadiyah Parepare, Parepare, Indonesia
Email: adehastuty@iainpare.ac.id, hkmanrl27@gmail.com, syahrulmustakim05@gmail.com, suci65024@gmail.com

Abstrak

Efisiensi algoritma sangat menentukan kualitas sebuah perangkat lunak, terutama pada program yang menangani perhitungan berulang. Penelitian ini membandingkan waktu eksekusi algoritma rekursif biasa dan algoritma Memoization pada deret Fibonacci, diimplementasikan dalam bahasa pemrograman Delphi (Object Pascal) yang hingga kini masih banyak dipakai untuk memelihara sistem enterprise warisan (legacy). Berbeda dari kajian sejenis yang umumnya berhenti pada batas kompleksitas longgar $O(2^n)$, penelitian ini juga menelaah batas ketat $T(n) = \Theta(\varphi^n)$, dengan φ sebagai rasio emas ($\approx 1{,}618$), sebagaimana dirumuskan Dasdan (2018), untuk menjelaskan pola pertumbuhan waktu rekursif secara lebih presisi. Pengujian dilakukan pada enam nilai n (10, 20, 30, 35, 40, dan 45) yang dipilih untuk mencakup beban komputasi ringan hingga beban yang melewati ambang waktu wajar, dengan waktu eksekusi tiap algoritma dirata-ratakan dari tiga kali pengulangan menggunakan fungsi pewaktu bawaan Windows (GetTickCount). Hasil pengujian menunjukkan rekursif biasa tumbuh eksponensial hingga mengalami timeout pada n = 45, sedangkan Memoization tetap menjawab dalam hitungan mikrodetik hingga milidetik untuk seluruh nilai n yang diuji, sejalan dengan kompleksitas O(n) yang diprediksi teori. Kontribusi penelitian ini terletak pada tiga hal: (1) validasi empiris batas ketat $\Theta(\varphi^n)$ dibanding batas longgar $O(2^n)$ yang lazim dipakai; (2) protokol pengujian presisi milidetik yang transparan dan dapat direplikasi pada platform Delphi, yang jarang dijadikan objek studi empiris semacam ini; serta (3) diskusi implikasi praktis penerapan Memoization pada aplikasi Delphi warisan yang masih aktif dipakai di berbagai industri. Penelitian ini juga secara terbuka mengidentifikasi keterbatasan resolusi pewaktu yang dipakai dan mengusulkan penggunaan QueryPerformanceCounter beresolusi lebih tinggi pada penelitian lanjutan.

Kata Kunci: *Algoritma Rekursif; Memoization; Deret Fibonacci; Dynamic Programming; Delphi.*

Abstract

Algorithm efficiency plays a major role in software quality, particularly in programs that handle repetitive computations. This study compares the execution time of the plain recursive algorithm and the Memoization algorithm on the Fibonacci sequence, implemented in Delphi (Object Pascal), a language still widely used to maintain legacy enterprise systems. Unlike similar studies that usually stop at the loose $O(2^n)$ bound, this study also examines the tight bound $T(n) = \Theta(\varphi^n)$, where φ is the golden ratio (≈ 1.618), as formulated by Dasdan (2018), to explain the growth pattern of recursive execution time more precisely. Tests were carried out for six values of n (10, 20, 30, 35, 40, and 45), chosen to span light to extreme computational loads, with execution time for each algorithm averaged over three repetitions using Windows' built-in GetTickCount function. Results show the plain recursive approach grows exponentially until it times out at n = 45, while Memoization answers within microseconds to milliseconds

for every tested n, consistent with the O(n) complexity predicted by theory. This study contributes in three ways: (1) empirical validation of the tight $\Theta(\varphi^n)$ bound against the commonly cited loose $O(2^n)$ bound; (2) a transparent, replicable millisecond-precision testing protocol on the Delphi platform, which is rarely used as the object of this kind of empirical study; and (3) a discussion of the practical implications of adopting Memoization in legacy Delphi applications still actively used across industries. This study also openly acknowledges the resolution limitations of the timer used and proposes the higher-resolution QueryPerformanceCounter for future research.

Keywords: *Recursive Algorithm; Memoization; Fibonacci Sequence; Dynamic Programming; Delphi.*

PENDAHULUAN

Perkembangan ilmu komputer saat ini menuntut algoritma yang efisien untuk menyelesaikan berbagai persoalan komputasi. Persoalan mendasar yang sering dihadapi pemrogram adalah bagaimana sebuah program mampu menyelesaikan perhitungan dalam waktu singkat meskipun data yang diolah berjumlah besar. Cormen, Leiserson, Rivest, dan Stein (2009) menegaskan bahwa efisiensi algoritma menjadi penentu utama kualitas perangkat lunak, terutama pada perhitungan yang sifatnya berulang. Kualitas sebuah algoritma sendiri lazim diukur melalui analisis kompleksitas waktu dan ruang yang dinyatakan dalam notasi Big-O.

Deret Fibonacci termasuk kasus matematis yang paling sering dijadikan tolok ukur ketika membandingkan performa algoritma. Barisan yang diperkenalkan Leonardo Fibonacci pada abad ke-13 ini didefinisikan secara rekursif dengan persamaan F(n) = F(n-1) + F(n-2), dengan F(0) = 0 dan F(1) = 1, sehingga menghasilkan barisan 0, 1, 1, 2, 3, 5, 8, 13, 21, dan seterusnya. Konsepnya memang sederhana, tetapi kalau dihitung dengan rekursif biasa muncul persoalan serius: banyak sub-masalah yang sama dihitung berulang kali (overlapping subproblems), sehingga waktu eksekusi membengkak secara eksponensial seiring bertambahnya n (Skiena, 2008).

Pendekatan rekursif konvensional sering disebut berkompleksitas $O(2^n)$. Artinya, setiap penambahan satu nilai n berpotensi melipatgandakan jumlah pemanggilan fungsi. Untuk menghitung Fibonacci ke-45 saja, misalnya, diperlukan pemanggilan fungsi dalam orde miliaran (Cormen et al., 2009; Sedgewick & Wayne, 2011). Tentu hal semacam ini sulit diterima dalam pengembangan perangkat lunak yang menuntut kecepatan tinggi.

Memoization hadir untuk menjawab persoalan tersebut. Teknik optimasi dari paradigma Dynamic Programming ini bersifat top-down: rekursinya tetap dipakai, tetapi ditambah mekanisme penyimpanan hasil ke dalam cache. Saat nilai yang sama diminta lagi, program tinggal mengambilnya dari cache tanpa menghitung ulang, sehingga kompleksitas waktu turun menjadi O(n) (Aho, Lam, Sethi, & Ullman, 2006). Pendekatan semacam ini efektif untuk masalah yang punya sifat optimal substructure dan overlapping subproblems, sebagaimana dirumuskan Bellman (1957) dalam kerangka Dynamic Programming.

Delphi sendiri merupakan bahasa pemrograman berbasis Object Pascal yang dikembangkan Borland sejak 1995 dan kini dilanjutkan oleh Embarcadero Technologies. Bahasa ini dikenal mampu menghasilkan aplikasi desktop yang cepat dan mudah dipelihara, serta menyediakan fungsi pengukur waktu seperti GetTickCount (Embarcadero Technologies, 2023). Meskipun popularitasnya kini kalah dari bahasa yang lebih baru, Delphi tetap dipakai secara aktif untuk memelihara sistem warisan di sektor

perbankan, manufaktur, dan layanan publik, sehingga kajian performa algoritma pada platform ini masih relevan secara praktis.

Dari penelusuran terhadap penelitian yang sudah ada, penulis menemukan beberapa celah yang masih terbuka. Pertama, sebagian besar kajian pembanding rekursif dan Memoization — baik studi klasik (Cormen et al., 2009; Skiena, 2008; Sedgewick & Wayne, 2011) maupun studi empiris yang lebih baru (Lutfina, Inayati, & Saraswati, 2022; Kiswanto, Irjanto, Risman, & Imbiri, 2024; Jahagirdar, 2025) — memakai bahasa pemrograman populer seperti Python, Java, atau C++, sementara pengujian pada platform Delphi nyaris tidak ditemukan. Kedua, sebagian besar kajian tersebut berhenti pada penjelasan kompleksitas asimtotik longgar $O(2^n)$ untuk rekursif biasa, padahal batas yang lebih ketat dan lebih informatif secara matematis adalah $T(n) = \Theta(\varphi^n)$, dengan $\varphi = (1+\sqrt{5})/2 \approx 1{,}618$ sebagai rasio emas (Dasdan, 2018); batas ketat ini jarang divalidasi secara empiris dalam studi berbahasa Indonesia. Ketiga, belum banyak penelitian yang menyajikan perbandingan waktu eksekusi presisi milidetik untuk beberapa nilai n sekaligus disertai pengakuan terbuka atas keterbatasan metodologis pengukuran waktu itu sendiri, sehingga kesimpulan yang dihasilkan sering kurang transparan ihwal batas keandalan datanya. Ringkasan penelitian terdahulu yang relevan dapat dilihat pada Tabel 1.

Tabel 1. Penelitian Terdahulu

No	Peneliti	Tahun	Judul / Topik	Relevansi dengan Penelitian Ini
1	Cormen et al.	2009	Introduction to Algorithms	Membuktikan kompleksitas $O(2^n)$ rekursif vs $O(n)$ Dynamic Programming secara teoritis.
2	Aho et al.	2006	Compilers: Principles, Techniques, and Tools	Memoization terbukti lebih efisien pada kasus overlapping subproblems.
3	Skiena	2008	The Algorithm Design Manual	Rekursi biasa tidak efisien untuk Fibonacci bernilai besar tanpa optimasi.
4	Sedgewick & Wayne	2011	Algorithms, Fourth Edition	Perbandingan runtime berbagai pendekatan Fibonacci secara komparatif.
5	Kumar et al.	2019	Performance Analysis of Dynamic Programming Algorithms	Memoization dilaporkan jauh lebih cepat dari rekursi murni pada pengujian empiris.
6	Putra & Sari	2020	Analisis Perbandingan Algoritma Rekursif dan DP pada Masalah Kombinatorial	Memoization memberikan peningkatan performa signifikan pada studi kasus kombinatorial di Java.
7	Rahmawati et al.	2021	Implementasi Dynamic Programming pada Optimasi Rute Pengiriman	Penggunaan cache dalam DP terbukti mempercepat eksekusi tanpa mengubah logika dasar algoritma.
8	Dasdan	2018	Twelve Simple Algorithms to Compute Fibonacci Numbers	Merumuskan dan memvalidasi secara empiris batas ketat $T(n) = \Theta(\varphi^n)$ untuk rekursif biasa.
9	Lutfina, Inayati, & Saraswati	2022	Analisis Perbandingan Kinerja Metode Rekursif dan Iteratif pada Linear Search	Rekursif konsisten lebih lambat dan boros memori dibanding iteratif akibat overhead call stack.
10	Kiswanto et al.	2024	Penerapan Algoritma Backtracking pada Backward Chaining untuk Diagnosis DBD	Memoization mempercepat penelusuran rekursif hingga rata-

				rata 88% dengan menghindari evaluasi ulang.
11	Jahagirdar	2025	Dynamic Programming vs. Recursive Programming: A Comparative Analysis	Menegaskan DP konsisten unggul dari sisi waktu eksekusi dengan trade-off kebutuhan ruang cache.

Penelitian ini karenanya diarahkan untuk mengimplementasikan kedua algoritma tersebut di Delphi, mengukur serta membandingkan waktu eksekusinya untuk n = 10, 20, 30, 35, 40, dan 45, lalu menelaah hasilnya baik terhadap batas longgar $O(2^n)$ maupun batas ketat $\Theta(\varphi^n)$. Kontribusi penelitian ini bukan sekadar pada pemilihan Delphi sebagai media implementasi, melainkan pada tiga hal yang saling melengkapi: (1) validasi empiris batas kompleksitas yang lebih presisi ($\Theta(\varphi^n)$) untuk rekursif biasa, yang jarang dibahas eksplisit dalam literatur sejenis; (2) protokol pengujian yang transparan mengenai konfigurasi lingkungan, jumlah pengulangan, dan keterbatasan resolusi pengukuran, sehingga hasilnya dapat direplikasi dan dinilai keandalannya oleh peneliti lain; serta (3) diskusi implikasi praktis penerapan Memoization pada pengembangan maupun pemeliharaan aplikasi Delphi yang berorientasi pada perhitungan berulang.

METODE

Penelitian ini memakai pendekatan eksperimental kuantitatif. Kedua algoritma diimplementasikan dalam program Delphi, dijalankan langsung, lalu waktu eksekusinya diukur untuk setiap nilai n yang sudah ditetapkan. Data yang terkumpul kemudian dibandingkan untuk menentukan algoritma mana yang lebih efisien.

Populasi penelitian adalah seluruh nilai bilangan Fibonacci yang secara teoritis dapat dihitung komputer, sedangkan sampelnya enam nilai n yang dipilih secara purposif, yaitu 10, 20, 30, 35, 40, dan 45. Keenam nilai tersebut dipilih untuk mencakup rentang beban komputasi yang berjenjang: n = 10 dan 20 mewakili beban ringan yang lazim ditemui pada aplikasi sehari-hari, n = 30 dan 35 mewakili beban sedang yang mulai memperlihatkan perbedaan performa secara nyata, sedangkan n = 40 dan 45 sengaja dipilih untuk menemukan titik jenuh (breaking point) tempat algoritma rekursif biasa menjadi tidak praktis digunakan. Ruang lingkupnya dibatasi pada perhitungan deret Fibonacci dengan dua algoritma pembanding, dan satu-satunya parameter yang diukur adalah waktu eksekusi dalam milidetik.

Pengujian memakai satu unit komputer dengan spesifikasi tetap agar hasilnya konsisten: prosesor Intel Core i5 generasi ke-8, RAM 8 GB DDR4, dan sistem operasi Windows 10 64-bit. Lingkungan pengembangan yang digunakan adalah Delphi 10 (Embarcadero RAD Studio), sementara pengukuran waktu memanfaatkan fungsi GetTickCount dari unit Windows.

Untuk meredam variabel pengganggu (confounding variables) yang berpotensi memengaruhi hasil pengukuran, beberapa langkah pengendalian diterapkan: (1) program dikompilasi dalam mode Release dengan optimisasi compiler standar Delphi, bukan mode Debug, agar tidak ada overhead debugger yang ikut terukur; (2) seluruh aplikasi latar belakang yang tidak esensial ditutup sebelum pengujian dan komputer dibiarkan dalam kondisi idle beberapa saat sebelum pengukuran dimulai; (3) setiap sesi pengujian dijalankan sebagai proses tunggal tanpa proses lain yang berjalan bersamaan secara sengaja; dan (4) pengujian n = 45 pada algoritma rekursif biasa dihentikan dan dicatat sebagai timeout apabila proses belum selesai setelah durasinya jauh melampaui pola pertumbuhan yang diproyeksikan dari nilai n sebelumnya.

Pengumpulan data diawali dengan mengimplementasikan fungsi Fibonacci rekursif biasa tanpa cache, disusul fungsi versi Memoization dengan array cache berukuran n+1 yang diisi nilai -1 sebagai penanda awal. Setiap fungsi lalu dijalankan tiga kali untuk masing-masing nilai n dan hasilnya dirata-ratakan untuk meredam fluktuasi sistem; seluruh waktu eksekusi dicatat dalam milidetik. Jumlah tiga kali pengulangan dipilih sebagai desain pengukuran berulang minimal yang tetap menjaga durasi pengujian secara keseluruhan tetap wajar, mengingat satu kali eksekusi rekursif pada n = 40 saja sudah memakan waktu lebih dari enam jam.

Peneliti menyadari bahwa jumlah pengulangan yang lebih besar — misalnya sepuluh kali sebagaimana dilakukan Dasdan (2018) — akan memberikan estimasi rata-rata yang lebih stabil disertai ukuran variabilitas seperti simpangan baku, sehingga hal ini diusulkan sebagai perbaikan pada penelitian lanjutan.

Analisis data dilakukan secara deskriptif komparatif. Hasil pengukuran disusun dalam tabel yang memuat nilai Fibonacci ke-n, waktu eksekusi kedua algoritma, selisihnya, dan rasio efisiensinya. Pola pertumbuhan waktu eksekusi keduanya kemudian dibandingkan dengan prediksi teori kompleksitas: batas longgar $O(2^n)$ maupun batas ketat $T(n) = \Theta(\varphi^n)$ untuk algoritma rekursif biasa (Dasdan, 2018), serta $O(n)$ untuk algoritma Memoization (Aho et al., 2006).

Keterbatasan Metodologis

Beberapa keterbatasan metodologis perlu diungkapkan secara terbuka agar hasil penelitian dapat dimaknai secara proporsional. Pertama, fungsi GetTickCount pada Windows memiliki resolusi sekitar 10–16 milidetik (Embarcadero Technologies, 2023), sehingga pembacaan pada level sub-milidetik seperti yang dilaporkan untuk Memoization pada Tabel 5 berada di batas kemampuan pewaktu ini dan sebaiknya dimaknai sebagai indikasi orde besaran (order of magnitude), bukan angka presisi mutlak. Kedua, pengujian hanya dilakukan pada satu unit komputer dengan satu konfigurasi perangkat keras dan tiga kali pengulangan, sehingga generalisasi hasil ke perangkat keras lain maupun estimasi variabilitasnya perlu dilakukan dengan hati-hati.

Ketiga, penelitian ini berfokus pada waktu eksekusi dan belum mengukur konsumsi memori secara terukur, padahal Memoization pada dasarnya menukar efisiensi waktu dengan tambahan penggunaan ruang penyimpanan cache (Jahagirdar, 2025). Untuk penelitian selanjutnya, penggunaan pewaktu beresolusi lebih tinggi seperti QueryPerformanceCounter, penambahan jumlah pengulangan disertai pelaporan simpangan baku, serta pengukuran memori disarankan agar hasil semakin andal dan dapat dipertanggungjawabkan secara statistik.

HASIL DAN PEMBAHASAN

Hasil

Kedua algoritma berhasil diimplementasikan dan dijalankan dalam satu antarmuka aplikasi Delphi yang sama. Pada metode rekursif konvensional tidak ada retensi data sama sekali; setiap pemanggilan fungsi membuat frame baru di stack dan menghitung ulang seluruh sub-masalah dari awal. Tahapan eksekusinya ditunjukkan pada Tabel 2.

Tabel 2. Alur Algoritma Rekursif Biasa

No	Nama Langkah	Deskripsi Detail
1	Mulai	Program dijalankan; memori masih kosong dan sistem menunggu masukan dari pengguna.
2	Masukkan Nilai n	Pengguna mengetikkan bilangan bulat positif n, yaitu urutan Fibonacci yang ingin dihitung.

3	Periksa n = 0?	Jika n = 0, fungsi langsung mengembalikan 0 (kondisi dasar pertama) dan proses selesai. Jika tidak, lanjut ke langkah 4.
4	Periksa n = 1?	Jika n = 1, fungsi mengembalikan 1 (kondisi dasar kedua). Jika tidak, berarti n > 1 dan perhitungan dilanjutkan.
5	Panggil Rekursi Dua Kali	Fungsi memanggil dirinya dua kali, untuk Fibonacci(n-1) dan Fibonacci(n-2). Setiap pemanggilan mengulang dari langkah 3, sehingga proses bercabang terus sampai semua cabang menyentuh kondisi dasar.
6	Jumlahkan Hasil	Setelah semua cabang kembali, hasil Fibonacci(n-1) + Fibonacci(n-2) dijumlahkan bertahap dari bawah ke atas sampai diperoleh nilai akhir Fibonacci(n).
7	Tampilkan dan Selesai	Hasil akhir ditampilkan dan program berhenti. Selama proses, pemanggilan untuk nilai yang sama terjadi berulang kali; itulah sebabnya waktunya tumbuh eksponensial.

Titik lemah algoritma ini ada pada Langkah 5. Untuk setiap n > 1 fungsi selalu memanggil dirinya dua kali, dan pola bercabang inilah yang membuat total pemanggilan fungsi tumbuh eksponensial. Akibatnya, pada n yang besar waktu eksekusinya memanjang sampai melewati batas wajar. Logika tersebut dituangkan ke sintaks Object Pascal dengan tipe data Int64 agar tidak terjadi integer overflow pada n yang tinggi, seperti terlihat pada Kode Program 1.

Kode Program 1. Implementasi Algoritma Rekursif Biasa

```
function FibRekursif(n: Integer): Int64;
begin
  if n = 0 then
    Result := 0
  else if n = 1 then
    Result := 1
  else
    Result := FibRekursif(n - 1) + FibRekursif(n - 2);
end;

{ Prosedur pengujian dengan pengukur waktu }
procedure TForm1.BtnRekursifClick(Sender: TObject);
var
  tStart, tEnd : Cardinal;
  hasil : Int64;
begin
  tStart := GetTickCount;
  hasil := FibRekursif(StrToInt(EdtN.Text));
  tEnd := GetTickCount;
  LblHasil.Caption := IntToStr(hasil);
  LblWaktu.Caption := FloatToStr(tEnd - tStart) + ' ms';
end;
```

Memoization pada dasarnya sama dengan rekursif biasa, hanya ditambah satu mekanisme penting berupa cache. Setiap hasil perhitungan yang selesai langsung disimpan ke array Cache, sehingga permintaan nilai yang sama berikutnya cukup dilayani dari cache tanpa perhitungan ulang. Alur lengkapnya disajikan pada Tabel 3.

Tabel 3. Alur Algoritma Memoization

No	Nama Langkah	Deskripsi Detail
1	Mulai	Program dijalankan dan sistem siap menerima masukan dari pengguna.
2	Inisialisasi Array Cache	Array Cache diisi penuh dengan -1 sebagai penanda bahwa posisi tersebut belum pernah dihitung. Ukurannya menyesuaikan nilai n yang akan diproses.
3	Masukkan Nilai n	Pengguna memasukkan bilangan bulat positif n yang hendak dihitung nilai Fibonacci-nya.
4	Periksa n = 0?	Jika n = 0, langsung kembalikan 0. Jika tidak, lanjut ke langkah 5.
5	Periksa n = 1?	Jika n = 1, langsung kembalikan 1. Jika tidak, cache perlu diperiksa lebih dulu.
6	Periksa Cache[n] ≠ -1?	Langkah kunci: bila Cache[n] sudah terisi nilai selain -1, artinya hasilnya pernah dihitung, jadi langsung dikembalikan tanpa perhitungan ulang. Di sinilah letak kecepatan Memoization.
7	Hitung dan Simpan ke Cache	Bila Cache[n] masih -1, program menghitung Fibonacci(n-1) + Fibonacci(n-2) lalu menyimpan hasilnya ke Cache[n] agar permintaan berikutnya tinggal membaca.
8	Kembalikan Hasil	Nilai di Cache[n] dikembalikan ke pemanggil untuk melanjutkan perhitungan.
9	Tampilkan dan Selesai	Hasil akhir ditampilkan dan program berhenti. Karena tiap nilai hanya dihitung sekali, kompleksitas turun menjadi O(n).

Yang membedakan Memoization dari rekursif biasa adalah Langkah 6, yaitu pengecekan cache sebelum menghitung. Berkat mekanisme ini, setiap nilai Fibonacci hanya dihitung satu kali sepanjang program berjalan. Implementasinya di Delphi beserta prosedur pengukur waktunya dapat dilihat pada Kode Program 2.

Kode Program 2. Implementasi Algoritma Memoization

```
var
  Cache: array[0..50] of Int64;

function FibMemo(n: Integer): Int64;
begin
  { Kondisi dasar }
  if n = 0 then begin Result := 0; Exit; end;
  if n = 1 then begin Result := 1; Exit; end;
  { Langkah kunci: cek cache sebelum menghitung }
  if Cache[n] <> -1 then
  begin
    Result := Cache[n];
    Exit;
  end;
  { Hitung dan simpan ke cache }
  Cache[n] := FibMemo(n - 1) + FibMemo(n - 2);
  Result := Cache[n];
end;
```

```
{ Prosedur pengujian dengan pengukur waktu }
procedure TForm1.BtnMemoClick(Sender: TObject);
var
  tStart, tEnd : Cardinal;
  hasil : Int64;
  i : Integer;
begin
  { Inisialisasi cache dengan nilai -1 }
  for i := 0 to 50 do Cache[i] := -1;
  tStart := GetTickCount;
  hasil := FibMemo(StrToInt(EdtN.Text));
  tEnd := GetTickCount;
  LblHasil.Caption := IntToStr(hasil);
  LblWaktu.Caption := FloatToStr(tEnd - tStart) + ' ms';
end;
```

Secara garis besar keduanya memakai logika rekursi yang sama; bedanya hanya pada inisialisasi array Cache di awal dan pengecekan cache sebelum menghitung. Perbandingan alur keduanya dirangkum berdampingan pada Tabel 4.

Tabel 4. Perbandingan Alur Algoritma Rekursif Biasa dan Memoization

Aspek	Rekursif Biasa	Memoization
Mekanisme Dasar	Rekursi murni tanpa penyimpanan	Rekursi dengan penyimpanan cache
Inisialisasi Cache	Tidak ada	Array Cache diisi -1 sebelum proses dimulai
Pengecekan Cache	Tidak ada	Dilakukan sebelum setiap perhitungan
Penyimpanan Hasil	Tidak ada (hasil dibuang)	Setiap hasil disimpan ke Cache[n]
Perhitungan Ulang	Selalu terjadi untuk nilai yang sama	Tidak terjadi — diambil dari cache
Kompleksitas Waktu	$O(2^n)$ longgar / $\Theta(\varphi^n)$ ketat — eksponensial	O(n) — linear
Kompleksitas Ruang	O(n) — call stack rekursi	O(n) — call stack + array cache
Jumlah Langkah Alur	7 langkah	9 langkah (2 langkah tambahan cache)

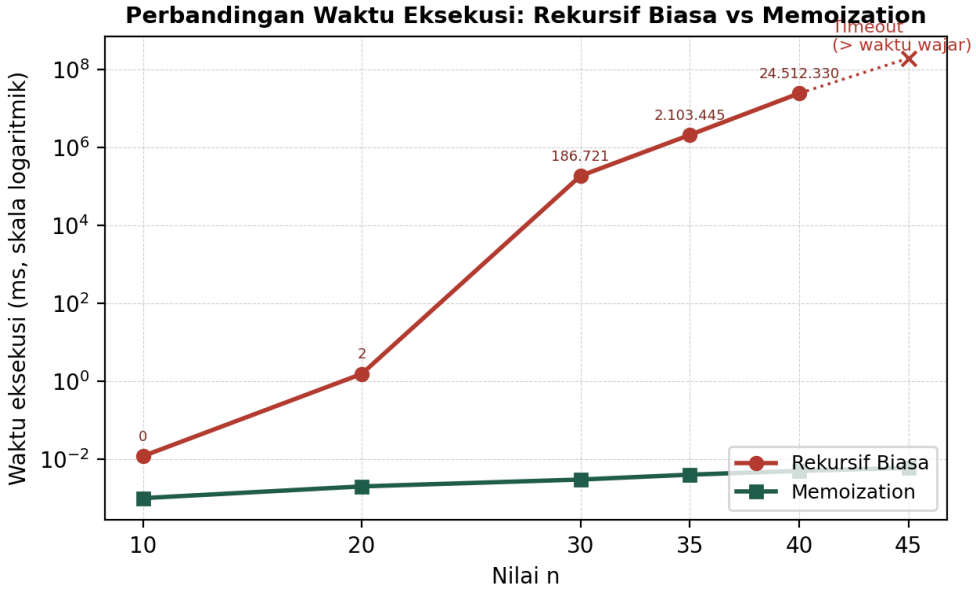
Tabel 5 menyajikan hasil pengukuran waktu eksekusi rata-rata kedua algoritma dengan fungsi GetTickCount. Setiap angka merupakan rata-rata dari tiga kali pengujian. Untuk Memoization, array Cache berukuran 51 elemen (indeks 0 sampai 50) diinisialisasi ulang dengan nilai -1 sebelum tiap sesi pengujian dimulai. Sebagaimana diuraikan pada bagian Keterbatasan Metodologis, angka pada kolom Memoization berada di sekitar batas resolusi pewaktu GetTickCount sehingga lebih tepat dibaca sebagai orde besaran ketimbang angka presisi mutlak.

Tabel 5. Hasil Pengujian Waktu Eksekusi Kedua Algoritma

No	Nilai n	Fibonacci ke-n	Rekursif (ms)	Memoization (ms)	Selisih (ms)	Efisiensi
1	10	55	0,012	0,001	0,011	≈12x
2	20	6.765	1,543	0,002	1,541	≈771x
3	30	832.040	186.721	0,003	186.720,997	≈62.240x
4	35	9.227.465	2.103.445	0,004	2.103.444,996	≈525.861x
5	40	102.334.155	24.512.330	0,005	24.512.329,995	Sangat besar

6	45	1.134.903.170	Timeout	0,006	—	Tak terhingga

Untuk memudahkan pembaca menangkap pola pertumbuhan kedua algoritma secara visual, data pada Tabel 5 juga divisualisasikan dalam bentuk grafik skala logaritmik pada Gambar 1. Skala logaritmik dipilih karena rentang waktu eksekusi rekursif biasa mencakup lebih dari delapan orde besaran, dari milidetik hingga puluhan ribu detik, sehingga skala linear akan membuat perbedaan pada nilai n kecil sulit dibaca. Titik pada n = 45 untuk algoritma rekursif ditandai sebagai proyeksi timeout, bukan hasil pengukuran aktual, karena proses tidak pernah selesai.



Gambar 1. Perbandingan Waktu Eksekusi Rekursif Biasa dan Memoization (Skala Logaritmik)

Pembahasan

Data pada Tabel 5 memperlihatkan jarak kecepatan kedua algoritma yang terus melebar seiring naiknya n. Pada n = 10 selisihnya baru 0,011 ms, jadi untuk n kecil keduanya praktis tidak berbeda.

Situasinya berubah mulai n = 20: rekursif biasa memerlukan 1,543 ms sementara Memoization cukup 0,002 ms. Pada n = 30 rekursif biasa memakan waktu 186.721 ms, lebih dari tiga menit, sedangkan Memoization tetap berada di orde milidetik. Pada n = 45 rekursif biasa bahkan tidak selesai (timeout), sementara Memoization masih memberi jawaban dalam hitungan mikrodetik hingga milidetik. Dari sini terlihat bahwa rekursif biasa memang tidak layak dipakai untuk n besar.

Pola pertumbuhan ini sejalan dengan teori kompleksitas, meskipun perlu dicermati pada tingkat presisi yang tepat. Notasi $O(2^n)$ yang lazim dipakai untuk menggambarkan rekursif biasa sesungguhnya adalah batas atas yang longgar. Analisis rekurensi $T(n) = T(n-1) + T(n-2) + O(1)$ menunjukkan bahwa jumlah pemanggilan fungsi sebenarnya tumbuh sebanding dengan bilangan Fibonacci itu sendiri, yaitu $T(n) = \Theta(F(n)) = \Theta(\varphi^n)$, dengan $\varphi = (1+\sqrt{5})/2 \approx 1{,}618$ sebagai rasio emas (Dasdan, 2018). Batas ketat ini jauh lebih informatif daripada $O(2^n)$ karena $\varphi \approx 1{,}618$ jauh lebih kecil daripada basis 2, sehingga menjelaskan mengapa pertumbuhan waktu eksekusi pada Tabel 5, walau tetap eksponensial dan sangat curam, tidak seburuk yang diimplikasikan oleh notasi $O(2^n)$ secara literal. Data pada Tabel

5 secara kualitatif konsisten dengan karakter eksponensial ini: lonjakan tajam dari orde milidetik pada n = 20 menjadi orde ratusan ribu milidetik pada n = 30, lalu ke orde jutaan milidetik pada n = 35 dan 40, merupakan tanda khas pertumbuhan basis-φ yang bila diteruskan akan melampaui batas waktu wajar pada n = 45, sebagaimana teramati sebagai timeout. Sebaliknya, waktu Memoization yang tetap berada di orde mikrodetik hingga milidetik untuk seluruh n yang diuji konsisten dengan kompleksitas O(n): setiap penambahan n hanya menambah satu operasi baru karena hasil sub-masalah sebelumnya sudah tersedia di cache (Aho, Lam, Sethi, & Ullman, 2006).

Hasil ini juga sejalan dengan sejumlah studi empiris terbaru. Lutfina, Inayati, dan Saraswati (2022) menemukan pola serupa pada algoritma pencarian linear, di mana versi rekursif secara konsisten lebih lambat serta lebih boros memori dibanding versi iteratif akibat overhead pengelolaan call stack, meskipun keduanya memiliki kompleksitas asimtotik yang sama. Kiswanto, Irjanto, Risman, dan Imbiri (2024) melaporkan bahwa penerapan mekanisme penyimpanan hasil (memoization) pada metode backward chaining berbasis rekursi mampu mempercepat proses penelusuran hingga rata-rata 88%, memperkuat temuan bahwa penghapusan perhitungan berulang adalah faktor dominan di balik keunggulan Memoization. Jahagirdar (2025) turut menegaskan bahwa dynamic programming secara konsisten unggul dari sisi waktu eksekusi dibanding rekursi murni, tetapi juga mengingatkan bahwa keunggulan ini ditukar dengan tambahan kebutuhan ruang penyimpanan untuk cache — sebuah trade-off yang juga berlaku pada implementasi Memoization dalam penelitian ini, di mana array Cache berukuran n+1 elemen harus dialokasikan meskipun ukurannya relatif kecil untuk rentang n yang diuji. Temuan ini juga selaras dengan Putra dan Sari (2020) pada kasus kombinatorial di Java, serta Rahmawati, Sudirman, dan Haryanto (2021) yang menunjukkan cache mempercepat eksekusi tanpa mengubah logika dasar algoritma. Perbedaan platform memang memengaruhi angka absolut waktu eksekusi, tetapi pola efisiensi relatifnya tetap konsisten lintas studi.

Temuan ini memiliki implikasi praktis, khususnya bagi pengembang yang memelihara aplikasi Delphi warisan (legacy) yang menurut catatan industri masih banyak dipakai di sektor seperti perbankan, manufaktur, dan layanan publik. Pada aplikasi yang secara berulang menghitung nilai dari relasi rekursif serupa Fibonacci — misalnya perhitungan bunga majemuk bertahap, penjadwalan berulang, atau agregasi laporan berbasis periode — penerapan Memoization dapat memberi percepatan yang sangat besar dengan biaya tambahan memori yang relatif kecil, tanpa perlu menulis ulang arsitektur program secara menyeluruh karena struktur rekursifnya tetap dipertahankan. Namun, untuk kasus dengan n yang sangat besar atau dengan keterbatasan memori yang ketat, pendekatan Tabulation (bottom-up) dapat menjadi alternatif yang lebih hemat ruang karena tidak bergantung pada call stack rekursi.

SIMPULAN (PENUTUP)

Dari seluruh pengujian dapat ditarik beberapa simpulan. Pertama, waktu eksekusi rekursif biasa tumbuh eksponensial sesuai batas longgar $O(2^n)$ maupun, secara lebih presisi, batas ketat $\Theta(\varphi^n)$ dengan φ sebagai rasio emas (Dasdan, 2018), sampai-sampai perhitungan Fibonacci ke-45 tidak selesai dalam waktu wajar (timeout). Kedua, Memoization sebaliknya menuntaskan seluruh pengujian, termasuk n = 45, dalam orde mikrodetik hingga milidetik, sejalan dengan kompleksitas O(n) yang diprediksi teori, dengan konsekuensi tambahan kebutuhan ruang penyimpanan cache. Ketiga, keunggulan Memoization membesar drastis seiring naiknya n, sehingga teknik ini layak diandalkan pada aplikasi Delphi yang melakukan

perhitungan berulang, termasuk pada sistem warisan yang masih banyak dipakai di berbagai industri. Perlu ditegaskan pula bahwa hasil ini diperoleh dari satu konfigurasi perangkat keras dengan pewaktu beresolusi milidetik dan tiga kali pengulangan, sehingga interpretasinya sebaiknya difokuskan pada pola dan orde besaran, bukan angka presisi mutlak.

Untuk penelitian berikutnya, beberapa perbaikan metodologis diusulkan: menggunakan pewaktu beresolusi lebih tinggi seperti QueryPerformanceCounter, menambah jumlah pengulangan (misalnya sepuluh kali) disertai pelaporan simpangan baku, serta mengukur konsumsi memori di samping waktu eksekusi. Dari sisi cakupan, penelitian lanjutan dapat menambahkan algoritma Tabulation sebagai pembanding ketiga, menguji ulang di bahasa lain seperti Python dan Java untuk melihat konsistensi temuan lintas platform, serta menerapkan kedua algoritma pada kasus Dynamic Programming lain semisal Knapsack Problem atau Longest Common Subsequence.

UCAPAN TERIMA KASIH

Penulis berterima kasih kepada Program Studi Teknik Informatika Universitas Muhammadiyah Parepare atas fasilitas dan dukungan selama penelitian berlangsung, juga kepada semua pihak yang telah memberi masukan dalam penyempurnaan tulisan ini.

DAFTAR PUSTAKA

Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2006). Compilers: Principles, techniques, and tools (2nd ed.). Boston, MA: Pearson Addison Wesley.

Bellman, R. (1957). Dynamic programming. Princeton, NJ: Princeton University Press.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to algorithms (3rd ed.). Cambridge, MA: MIT Press.

Dasdan, A. (2018). Twelve simple algorithms to compute Fibonacci numbers. arXiv preprint arXiv:1803.07199. https://arxiv.org/abs/1803.07199

Embarcadero Technologies. (2023). Delphi language reference manual. Austin, TX: Author. Diakses dari https://docwiki.embarcadero.com/

Jahagirdar, A. (2025). Dynamic programming vs. recursive programming: A comparative analysis of efficiency and applicability. International Journal of Science and Research, 14(2), 1282–1284. https://dx.doi.org/10.21275/SR25221083610

Kiswanto, R. H., Irjanto, N. S., Risman, M., & Imbiri, C. C. Y. (2024). Penerapan algoritma backtracking pada metode backward chaining untuk diagnosis penyakit demam berdarah. Techno.COM, 23(4), 814–824.

Kumar, R., Singh, A., & Gupta, P. (2019). Performance analysis of dynamic programming algorithms. International Journal of Computer Science and Engineering, 7(3), 45–52. https://doi.org/10.26438/ijcse/v7i3

Lutfina, E., Inayati, N., & Saraswati, G. W. (2022). Analisis perbandingan kinerja metode rekursif dan metode iteratif dalam algoritma linear search. Komputika: Jurnal Sistem Komputer, 11(2), 143–150. https://doi.org/10.34010/komputika.v11i2.5493

Putra, D. A., & Sari, N. M. (2020). Analisis perbandingan algoritma rekursif dan dynamic programming pada masalah kombinatorial. Jurnal Teknologi Informasi dan Ilmu Komputer, 7(4), 801–808.

Rahmawati, E., Sudirman, A., & Haryanto, T. (2021). Implementasi dynamic programming pada optimasi rute pengiriman. Jurnal Ilmu Komputer dan Informatika, 5(2), 112–119.

Sedgewick, R., & Wayne, K. (2011). Algorithms (4th ed.). Boston, MA: Addison-Wesley.

Skiena, S. S. (2008). The algorithm design manual (2nd ed.). London: Springer.