



**UNIVERSITAS ISLAM NEGERI
SYARIF HIDAYATULLAH JAKARTA**

ALGORITMA DAN STRUKTUR DATA

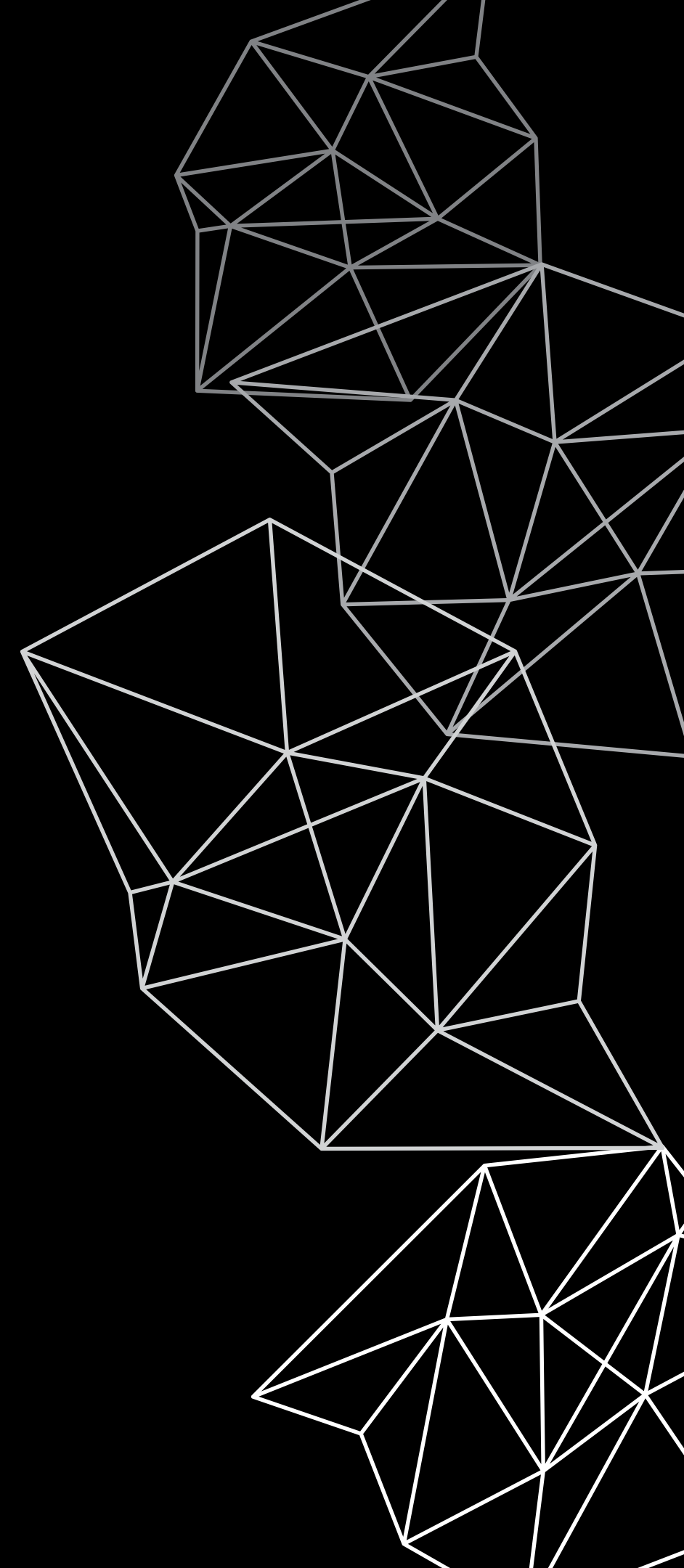
RECURSION

DOSEN PENGAMPU:

MOHAMAD IRVAN SEPTIAR MUSTI, S.SI, M.SI

DISUSUN OLEH :

SYAHRUL AKBAR RAMDHANI (11230940000027)





DAFTAR ISI

4.1 ILLUSTRATIVE EXAMPLES

- 4.1.1 The Factorial Function
- 4.1.2 Drawing an English Ruler
- 4.1.3 Binary Search
- 4.1.4 File Systems

4.2 ANALYZING RECURSIVE ALGORITHMS

4.3 RECURSION RUN AMOK

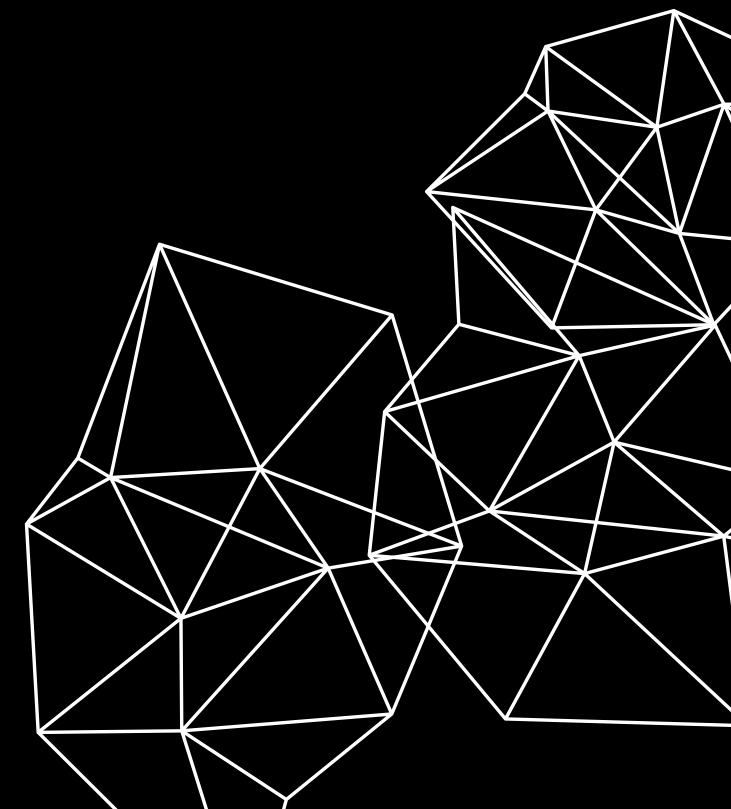
- 4.3.1 Maximum Recursive Depth
in Python

4.4 FURTHER EXAMPLES OF RECURSION

- 4.4.1 Linear Recursion
- 4.4.2 Binary Recursion
- 4.4.3 Multiple Recursion

4.5 DESIGNING RECURSIVE ALGORITHMS

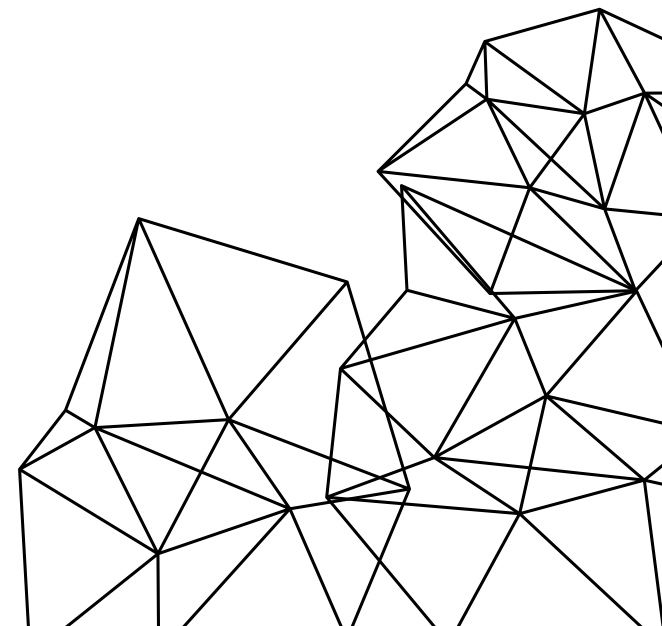
4.6 ELIMINATING TAIL RECURSION

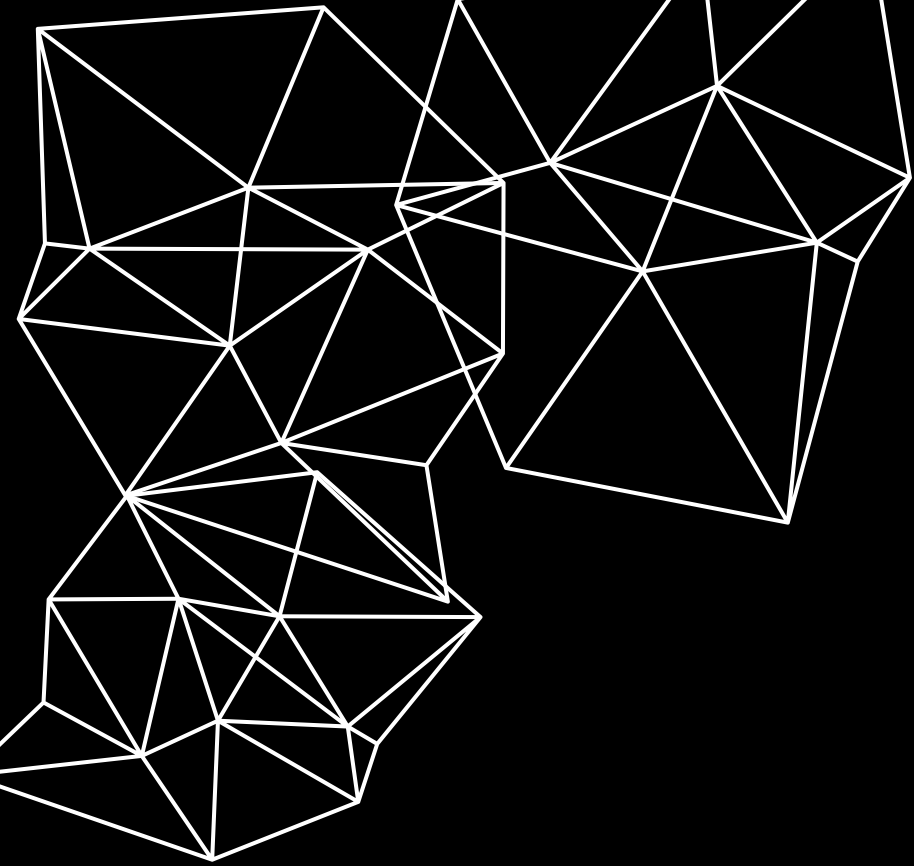




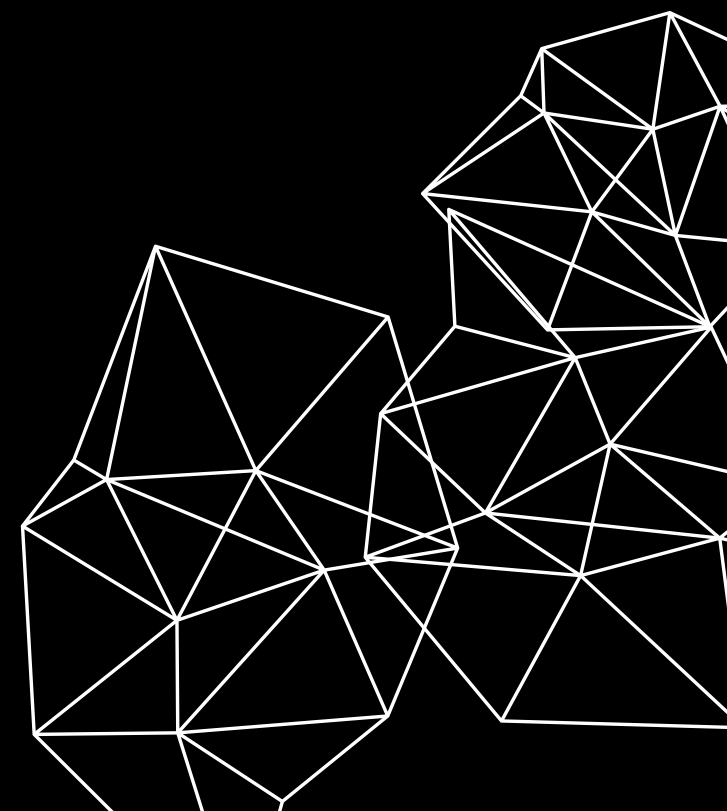
PENDAHULUAN

Rekursi adalah teknik pemrograman di mana sebuah fungsi memanggil dirinya sendiri untuk menyelesaikan masalah melalui pendekatan berbasis submasalah. Konsep ini mencerminkan pola berulang yang juga ditemukan di alam dan seni, seperti fraktal atau boneka Matryoshka. Rekursi memainkan peran penting dalam struktur data dan algoritma, karena menyederhanakan pemrosesan masalah kompleks secara elegan. Contoh umum penggunaannya meliputi: perhitungan faktorial, representasi penggaris Inggris, pencarian biner, dan navigasi struktur sistem berkas yang bersifat hierarkis dan rekursif.





4.1 ILLUSTRATIVE EXAMPLES





4.1.1 THE FACTORIAL FUNCTION

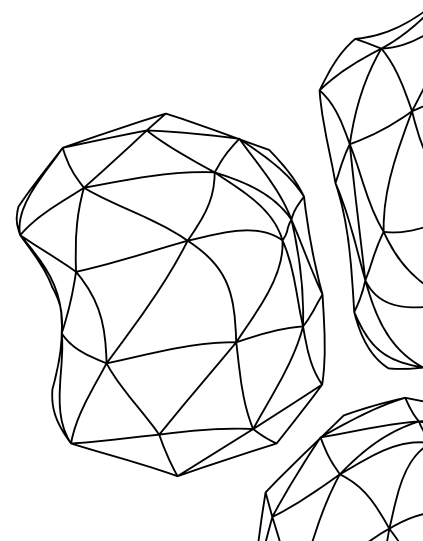
Faktorial adalah contoh klasik untuk menjelaskan konsep rekursi. Faktorial dari bilangan bulat positif n , ditulis $n!$, merupakan hasil perkalian semua bilangan bulat dari 1 hingga n . Secara konvensional, $0!$ didefinisikan sebagai 1.

Definisi formal:

- $n! = 1$, jika $n = 0$ (kasus dasar)
- $n! = n \times (n - 1)!$, jika $n \geq 1$ (kasus rekursif)

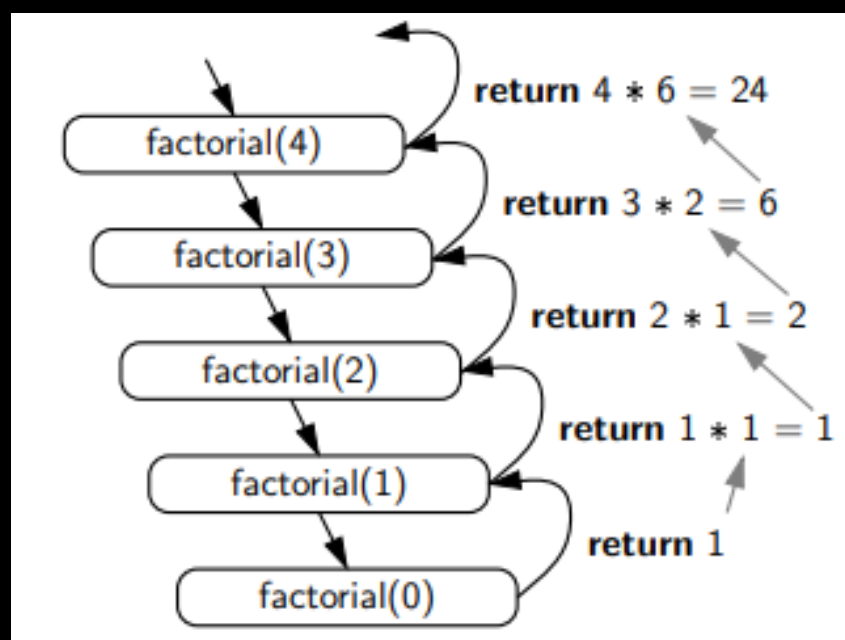
Sebagai contoh, $5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$.

Faktorial sering digunakan dalam menghitung permutasi, yaitu jumlah cara menyusun n objek berbeda. Misalnya, untuk tiga huruf $\{a, b, c\}$, terdapat $3! = 6$ susunan berbeda: abc, acb, bac, bca, cab, cba.



IMPLEMENTASI REKURSIF DARI FUNGSI FAKTORIAL

```
1 def factorial(n):  
2     if n == 0:  
3         return 1  
4     else:  
5         return n * factorial(n-1)
```



Rekursi bukan sekadar notasi matematika, tetapi juga teknik penting dalam pemrograman. Sebagai contoh, kita dapat menggunakan rekursi untuk mengimplementasikan fungsi faktorial dalam Python.

Fungsi rekursif tidak menggunakan perulangan eksplisit (for atau while). Sebaliknya, pengulangan terjadi melalui pemanggilan fungsi itu sendiri. Setiap pemanggilan membawa nilai argumen yang lebih kecil, hingga mencapai kasus dasar di mana rekursi berhenti.

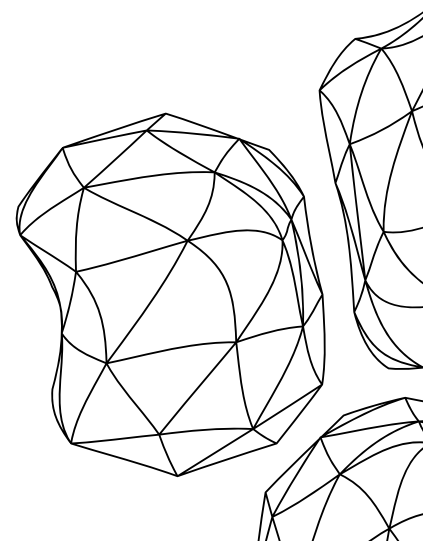
Untuk memahami bagaimana fungsi rekursif bekerja, digunakan jejak rekursi, yang memperlihatkan setiap pemanggilan fungsi dan pengembalian nilainya. Setiap pemanggilan baru digambarkan sebagai panah ke bawah, dan setiap pengembalian nilai sebagai panah ke atas.



4.1.2 DRAWING AN ENGLISH RULER

Menghitung faktorial memang bisa dilakukan secara iteratif menggunakan loop, sehingga rekursi tidak selalu diperlukan. Namun, ada banyak kasus kompleks di mana rekursi menawarkan solusi yang lebih sederhana dan elegan — salah satunya adalah pembuatan penggaris Inggris (English ruler).

Penggaris Inggris adalah contoh pola fraktal sederhana. Struktur garisnya dibentuk oleh pola berulang:

- Garis utama (major tick) tiap 1 inci, biasanya dengan label angka (0, 1, 2, ...).
 - Garis minor (minor tick) di antara garis utama, seperti untuk $1/2$, $1/4$, $1/8$ inci.
 - Semakin kecil fraksi, semakin pendek garisnya.
 - Pola garis-garis ini bersifat rekursif: setiap interval di antara dua garis utama mengikuti pola yang serupa dalam skala lebih kecil..
- 

PENDEKATAN REKURSIF UNTUK MENGGAMBAR PENGGARIS

```
1 def draw_line(tick_length, tick_label=''):
2     """Draw one line with given tick length (followed by optional label)."""
3     line = '-' * tick_length
4     if tick_label:
5         line += ' ' + tick_label
6     print(line)
7
8 def draw_interval(center_length):
9     """Draw tick interval based upon a central tick length."""
10    if center_length > 0:           # stop when length drops to 0
11        draw_interval(center_length - 1)   # recursively draw top ticks
12        draw_line(center_length)           # draw center tick
13        draw_interval(center_length - 1)   # recursively draw bottom ticks
14
15 def draw_ruler(num_inches, major_length):
16     """Draw English ruler with given number of inches, major tick length."""
17     draw_line(major_length, '0')         # draw inch 0 line
18     for j in range(1, 1 + num_inches):
19         draw_interval(major_length - 1)   # draw interior ticks for inch
20         draw_line(major_length, str(j))   # draw inch j line and label
```

Secara umum, interval dengan garis tengah sepanjang L (dengan $L \geq 1$) terdiri dari:

Interval dengan garis tengah $L - 1$

Satu garis tengah sepanjang L

Interval dengan garis tengah $L - 1$

Meskipun bisa dibuat secara iteratif, menggunakan rekursi jauh lebih mudah.

Implementasi terdiri dari tiga fungsi:

`draw_ruler`: fungsi utama yang mengatur pembuatan seluruh penggaris.

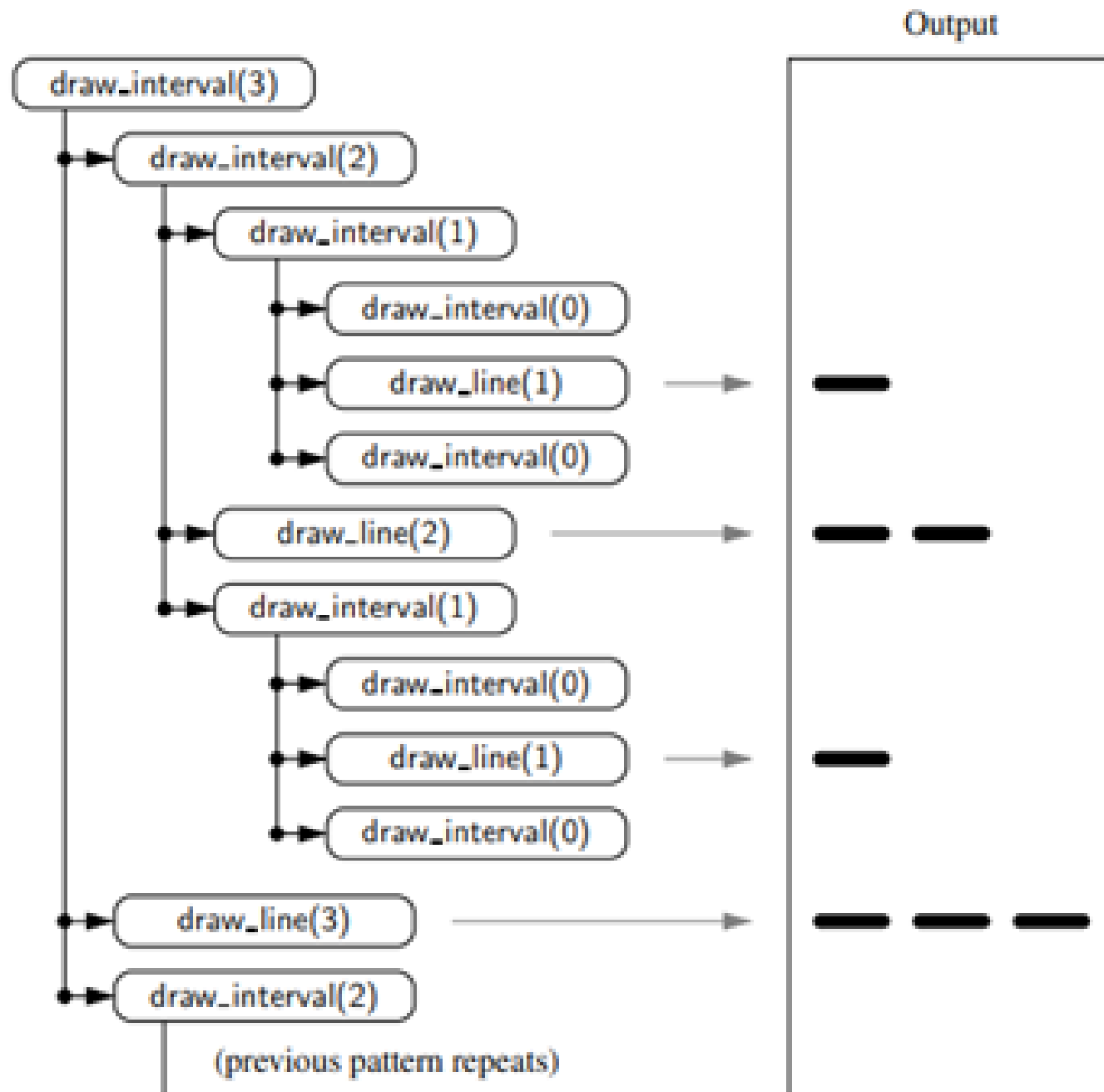
`draw_line`: menggambar satu garis (tick) dengan panjang tertentu dan label opsional.

`draw_interval`: fungsi rekursif yang menggambar garis-garis kecil di antara dua garis utama, berdasarkan panjang garis tengah.

Basis kasusnya adalah ketika $L = 0$, yang berarti tidak menggambar apa pun.

Untuk $L \geq 1$, `draw_interval` dipanggil secara rekursif dua kali (atas dan bawah), dan `draw_line` dipanggil di tengah.

MENGILUSTRASIKAN GAMBAR PENGGARIS MENGUNAKAN JEJAK REKURSI



Eksekusi fungsi rekursif `draw_interval` dapat divisualisasikan dengan penelusuran ulang, namun jejaknya lebih kompleks dibanding contoh faktorial karena ada dua pemanggilan rekursif. Jejak ini dapat digambarkan seperti outline dokumen.

Penjelasan Alur Rekursi

- `draw_interval(L)` memanggil dirinya dua kali:
- Pertama untuk bagian atas (sebelum `draw_line(L)`)
- Kedua untuk bagian bawah (setelah `draw_line(L)`)
- Di antara dua pemanggilan tersebut, digambar satu garis tengah dengan panjang `L`.
- Ketika mencapai `draw_interval(0)`, tidak ada yang digambar (basis kasus).

Perbandingan dengan Rekursi Faktorial

- Rekursi faktorial (`factorial(n)`) hanya memiliki satu pemanggilan rekursif.
- `draw_interval(L)` lebih kompleks karena memiliki dua cabang rekursif (mirip struktur pohon biner).
- Hasilnya adalah pola simetris, dengan garis-garis yang lebih pendek diapit oleh garis yang lebih panjang di tengah.
-

4.1.3 BINARY SEARCH

Binary search adalah algoritma efisien untuk mencari nilai dalam urutan data yang telah terurut, dengan kompleksitas waktu $O(\log n)$. Berbeda dari sequential search yang memeriksa elemen satu per satu ($O(n)$), binary search memanfaatkan sifat data yang terurut dengan membagi ruang pencarian menjadi dua pada setiap langkah.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
2	4	5	7	8	9	12	14	17	19	22	25	27	28	33	37

Langkah-langkah pencarian biner:

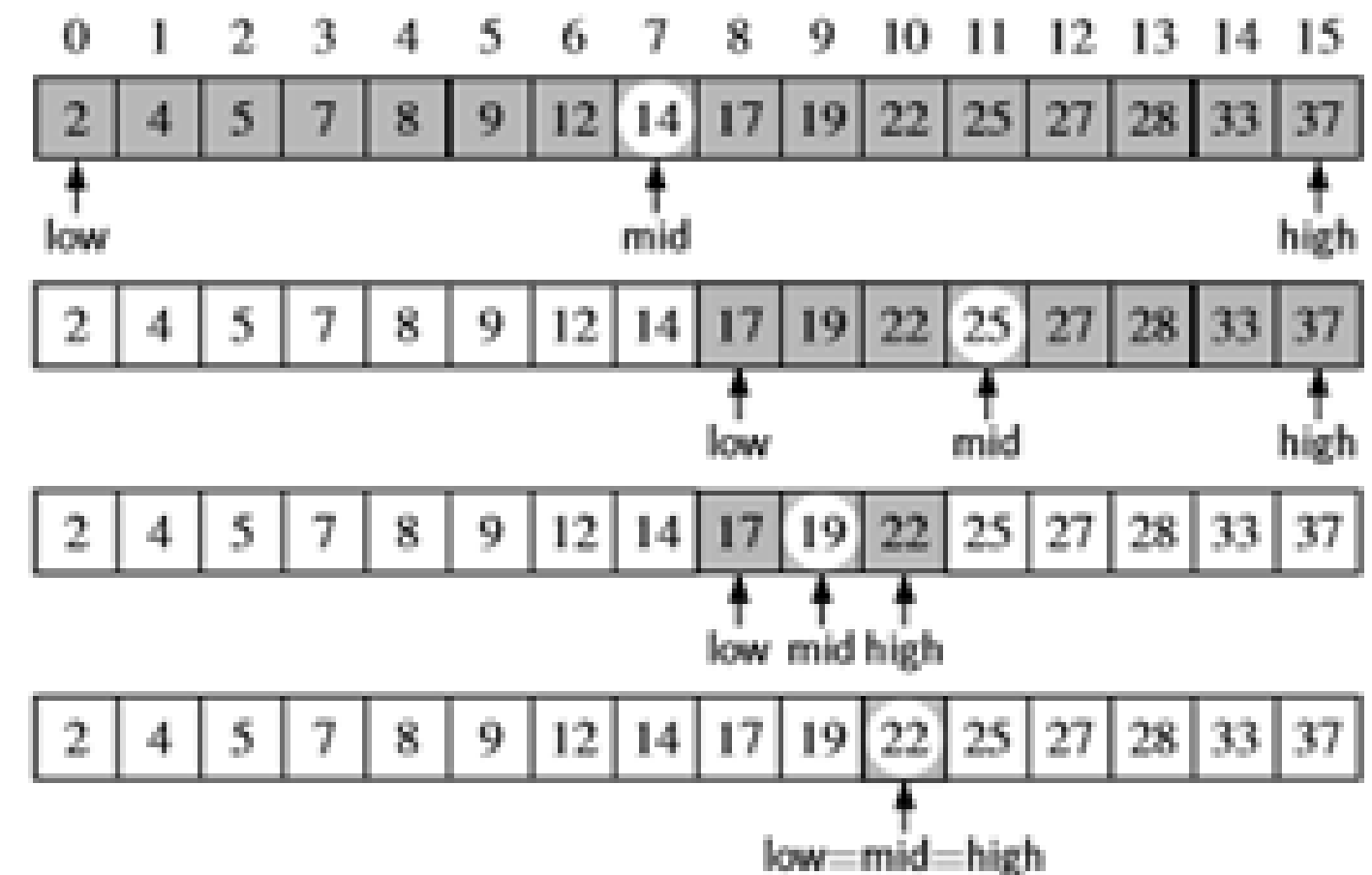
1. Tetapkan dua parameter: $low = 0$ dan $high = n-1$.
2. Hitung indeks tengah: $mid = (low + high) // 2$.
3. Bandingkan $data[mid]$ dengan target:
 - Jika sama, pencarian berhasil.
 - Jika $target < data[mid]$, lanjutkan pencarian di bagian kiri (low hingga $mid-1$).
 - Jika $target > data[mid]$, lanjutkan pencarian di bagian kanan ($mid+1$ hingga $high$).
4. Jika $low > high$, artinya target tidak ditemukan

Proses ini dilakukan secara rekursif atau iteratif hingga target ditemukan atau rentang pencarian habis. Binary search sangat efektif untuk data terurut yang dapat diakses melalui indeks.

IMPLEMENTASI DALAM PYTHON

Pencarian biner memiliki kompleksitas waktu $O(\log n)$, yang jauh lebih cepat dibanding pencarian sekuensial. Misalnya, untuk satu miliar elemen, $\log n$ hanya sekitar 30, menunjukkan efisiensi yang sangat tinggi. Berikut contoh implementasi pada Python:

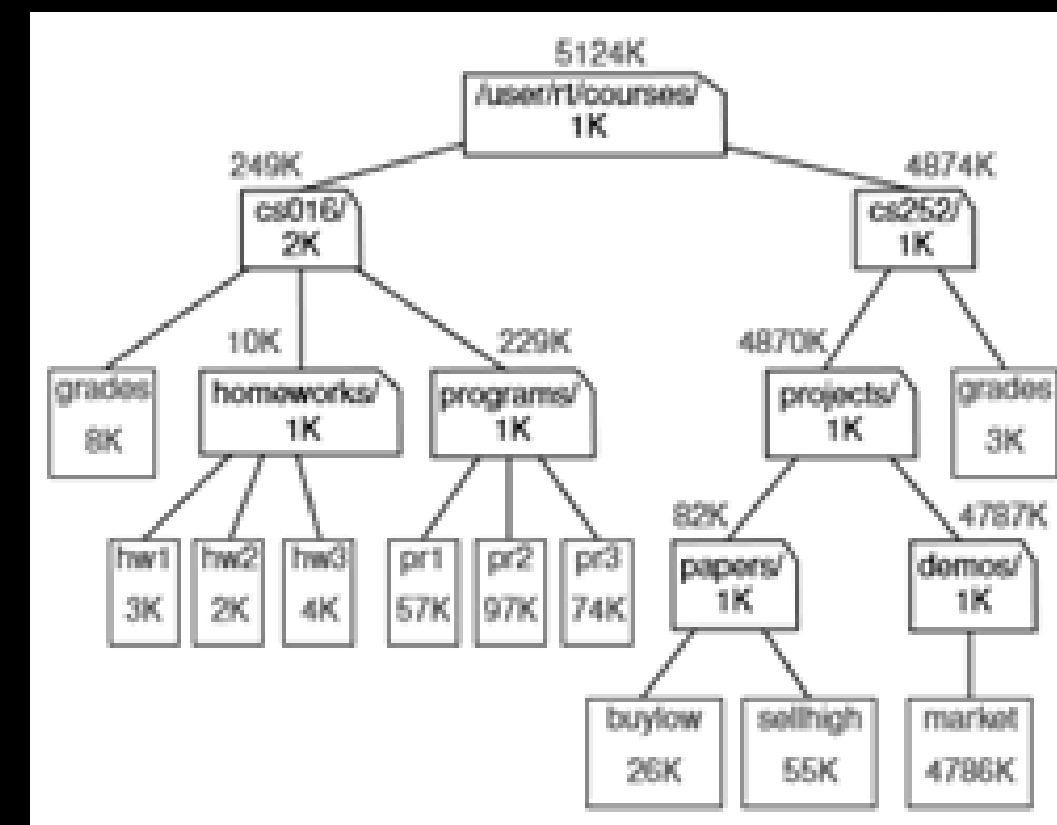
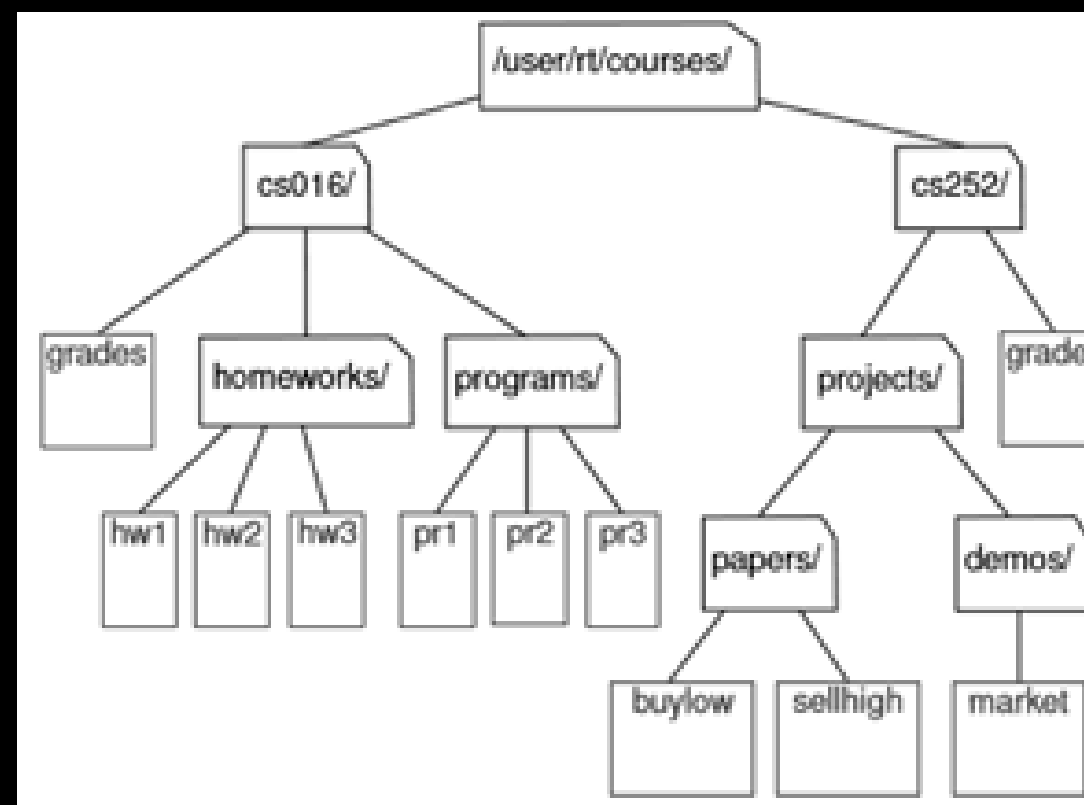
```
1 def binary_search(data, target, low, high):
2     """Return True if target is found in indicated portion of a Python list.
3
4     The search only considers the portion from data[low] to data[high] inclusive.
5     """
6     if low > high:
7         return False                # interval is empty; no match
8     else:
9         mid = (low + high) // 2
10        if target == data[mid]:      # found a match
11            return True
12        elif target < data[mid]:
13            # recur on the portion left of the middle
14            return binary_search(data, target, low, mid - 1)
15        else:
16            # recur on the portion right of the middle
17            return binary_search(data, target, mid + 1, high)
```



4.1.4 FILE SYSTEMS

Sistem operasi modern mengelola sistem file secara rekursif, di mana sebuah direktori (folder) dapat berisi berkas maupun subdirektori, yang pada gilirannya juga dapat berisi berkas dan direktori lainnya, tanpa batas kedalaman tertentu. Struktur ini mencerminkan hierarki pohon, dengan direktori tingkat atas sebagai akar, dan setiap cabangnya berpotensi mengandung struktur serupa secara berulang.

Untuk keperluan analisis ruang penyimpanan, sistem dapat menghitung ruang disk kumulatif yang digunakan oleh sebuah direktori dengan menjumlahkan ruang dari semua berkas dan subdirektori di dalamnya, termasuk yang bersarang secara rekursif. Informasi ini penting untuk pemantauan dan manajemen kapasitas disk.





PERHITUNGAN PENGGUNAAN DISK SECARA REKURSIF

Penggunaan disk kumulatif dalam sistem berkas dihitung dengan menjumlahkan ukuran langsung suatu entri (file atau direktori) dan total penggunaan disk dari seluruh isi di dalamnya jika entri tersebut adalah direktori. Proses ini dilakukan secara rekursif.

Contoh: Direktori cs016 berukuran total 249K, hasil dari penjumlahan ukuran langsung (2K) dan ukuran semua subdirektornya (grades, homeworks, programs).

Algoritma DiskUsage(path):

- Hitung ukuran langsung dari path.
- Jika path adalah direktori, lakukan pemanggilan DiskUsage pada setiap isi di dalamnya dan tambahkan ke total.
- Kembalikan total ukuran.

Algorithm DiskUsage(path):

Input: A string designating a path to a file-system entry

Output: The cumulative disk space used by that entry and any nested entries

total = size(path) {immediate disk space used by the entry}

if path represents a directory then

for each child entry stored within directory path do

total = total + DiskUsage(child) {recursive call}

return total



IMPLEMENTASI PADA PYTHON

Modul os dalam Python untuk Perhitungan Ruang Disk
Python menyediakan modul os yang sangat berguna untuk berinteraksi dengan sistem file. Dalam konteks perhitungan ruang disk secara rekursif, hanya beberapa fungsi kunci yang diperlukan:

- `os.path.getsize(path)`
- Mengembalikan ukuran (dalam byte) dari file atau direktori pada path tertentu.
- `os.path.isdir(path)`
- Mengembalikan True jika path adalah direktori, False jika sebaliknya.
- `os.listdir(path)`
- Mengembalikan daftar semua nama entri (file atau direktori) yang ada dalam direktori pada path.
- `os.path.join(path, filename)`
- Menggabungkan path direktori dan nama file menjadi path lengkap, sesuai dengan sistem operasi yang digunakan.

Fungsi-fungsi ini memungkinkan pengembangan algoritma rekursif yang efisien untuk menavigasi dan menghitung penggunaan disk dari seluruh struktur sistem file.

```
1 import os
2
3 def disk_usage(path):
4     """ Return the number of bytes used by a file/folder and any descendents. """
5     total = os.path.getsize(path) # account for direct usage
6     if os.path.isdir(path): # if this is a directory,
7         for filename in os.listdir(path): # then for each child:
8             childpath = os.path.join(path, filename) # compose full path to child
9             total += disk_usage(childpath) # add child's usage to total
10
11     print ('{0:<7}'.format(total), path) # descriptive output (optional)
12     return total # return the grand total
```



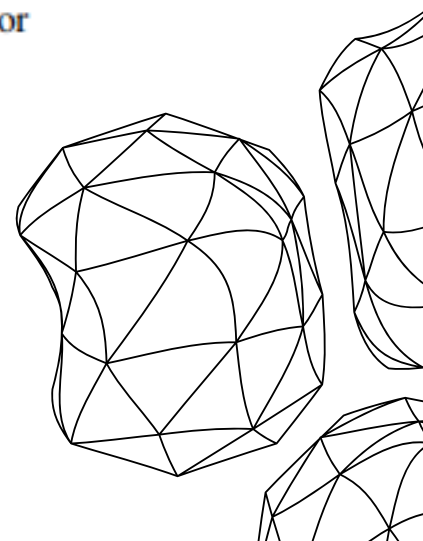

JEJAK REKURSI DALAM PENGGUNAAN DISK

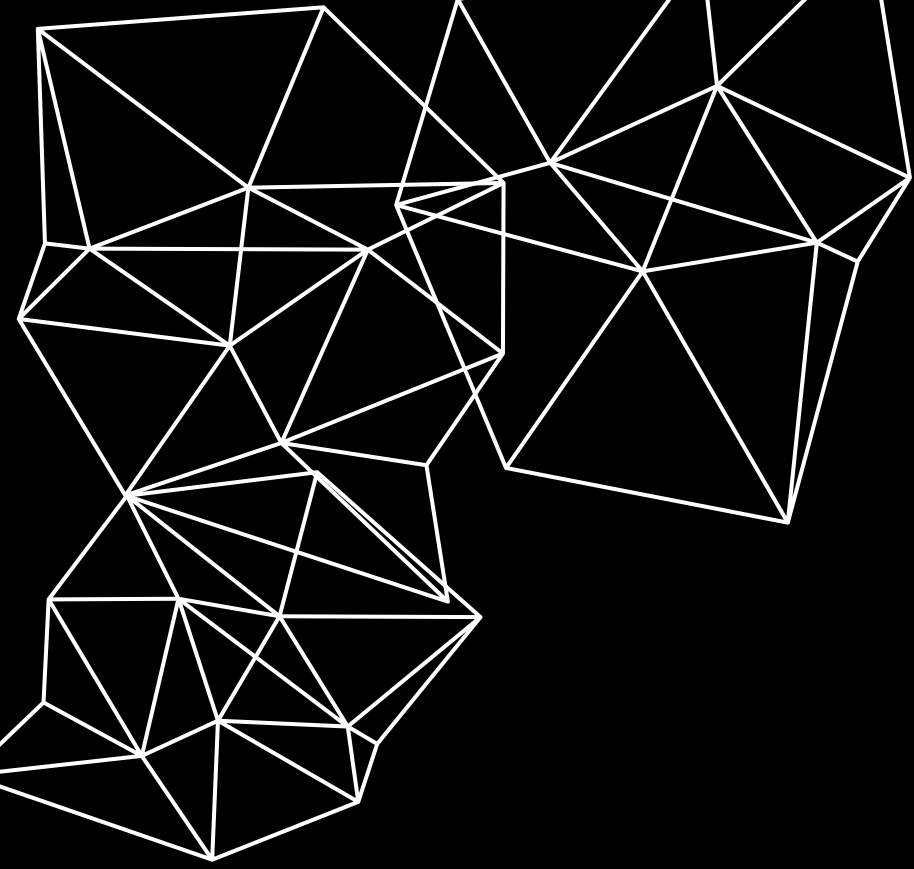
Dalam implementasi perhitungan penggunaan disk dengan rekursi, pernyataan print sengaja dimasukkan untuk menghasilkan jejak rekursi yang menyerupai output dari utilitas Unix/Linux klasik `du`. Fungsi ini melaporkan jumlah ruang disk yang digunakan oleh sebuah direktori dan semua kontennya, dengan opsi `verbose` yang menunjukkan rincian penggunaan ruang.

Output yang dihasilkan oleh fungsi rekursif ini mengikuti urutan penyelesaian panggilan rekursif. Setiap panggilan rekursif menghitung dan melaporkan total ruang disk untuk entri direktori setelah menghitung total ruang disk dari entri-entri di dalamnya. Sebagai contoh, total ruang disk untuk `/user/rt/courses/cs016` hanya dapat dihitung setelah ruang disk untuk sub-direktori `grades`, `homeworks`, dan `programs` dihitung dan selesai. Dengan demikian, algoritma rekursif menghitung total ruang disk secara kumulatif dari entri terdalam hingga entri yang lebih besar.

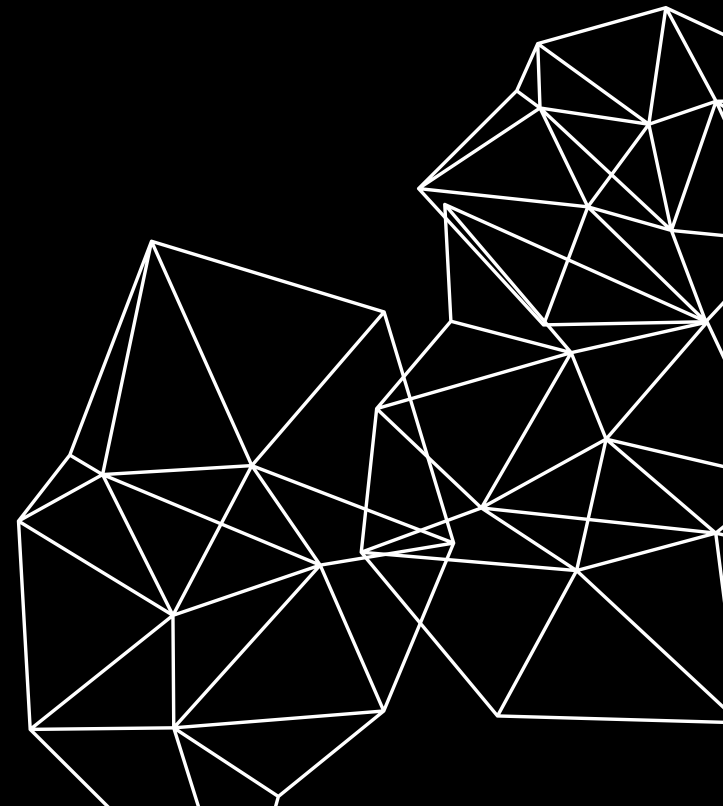
```
8      /user/rt/courses/cs016/grades
3      /user/rt/courses/cs016/homeworks/hw1
2      /user/rt/courses/cs016/homeworks/hw2
4      /user/rt/courses/cs016/homeworks/hw3
10     /user/rt/courses/cs016/homeworks
57     /user/rt/courses/cs016/programs/pr1
97     /user/rt/courses/cs016/programs/pr2
74     /user/rt/courses/cs016/programs/pr3
229    /user/rt/courses/cs016/programs
249    /user/rt/courses/cs016
26     /user/rt/courses/cs252/projects/papers/buylow
55     /user/rt/courses/cs252/projects/papers/sellhigh
82     /user/rt/courses/cs252/projects/papers
4786   /user/rt/courses/cs252/projects/demos/market
4787   /user/rt/courses/cs252/projects/demos
4870   /user/rt/courses/cs252/projects
3      /user/rt/courses/cs252/grades
4874   /user/rt/courses/cs252
5124   /user/rt/courses/
```

Figure 4.8: A report of the disk usage for the file system shown in Figure 4.7, as generated by the Unix/Linux utility `du` (with command-line options `-ak`), or equivalently by our `disk_usage` function from Code Fragment 4.5.





4.2 ANALYZING RECURSIVE ALGORITHMS



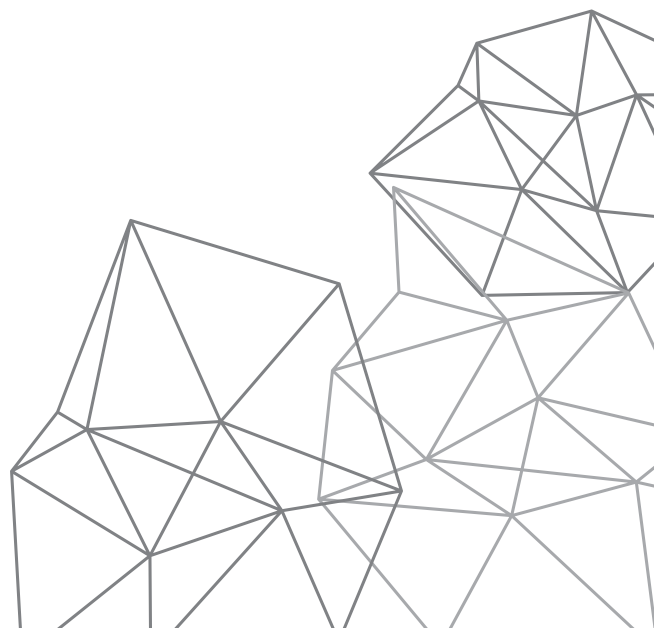


KOMPUTASI FAKTORIAL

Fungsi faktorial rekursif memiliki karakteristik yang mudah dianalisis. Untuk input sebesar n , fungsi ini melakukan tepat $n+1$ pemanggilan (aktivasi), mulai dari n dan berlanjut hingga mencapai kasus dasar (0). Setiap aktivasi membutuhkan waktu konstan $O(1)$, karena hanya melibatkan:

- Satu operasi perbandingan untuk memeriksa kondisi dasar
- Satu operasi perkalian
- Satu pemanggilan rekursif

Dengan demikian, total waktu eksekusi bersifat linear terhadap input, menghasilkan kompleksitas waktu $O(n)$. Pola eksekusi ini membentuk deret pemanggilan bersarang sebanyak $n+1$ lapisan, di mana setiap lapisan menyumbang operasi konstan sebelum perhitungan selesai secara beruntun dari lapisan terdalam.





MENGGAMBAR PENGGARIS INGGRIS

Pada algoritma penggaris Inggris, pemanggilan fungsi `draw_interval(c)` menghasilkan sejumlah garis output yang mengikuti pola pertumbuhan eksponensial. Secara matematis, untuk setiap nilai $c \geq 0$, fungsi ini menghasilkan tepat $2^c - 1$ garis output.

Kasus dasar ($c = 0$):

Pada $c = 0$, tidak ada garis yang digambar ($2^0 - 1 = 0$).

Kasus rekursif ($c > 0$):

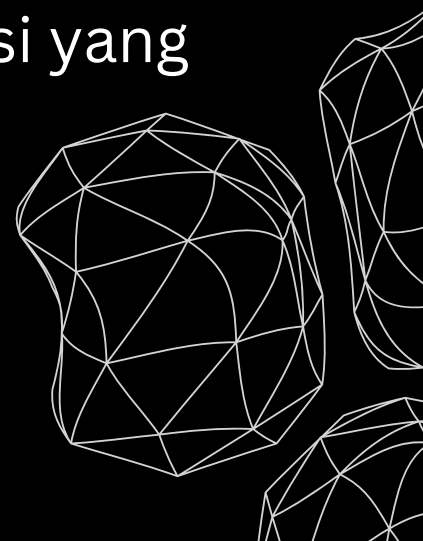
Setiap pemanggilan `draw_interval(c)` memicu dua pemanggilan rekursif ke `draw_interval(c-1)` serta satu pemanggilan ke `draw_line`. Ini menciptakan hubungan rekursif, di mana jumlah garis total dihitung dengan rumus $2^c - 1$, yang menunjukkan pertumbuhan eksponensial.

Bukti formal:

Bukti melalui induksi matematika menunjukkan bahwa jumlah garis yang digambar mengikuti rumus:

$$1 + 2 \times (2^{(c-1)} - 1) = 2^c - 1$$

Analisis ini menunjukkan kompleksitas output yang tumbuh eksponensial seiring dengan peningkatan nilai c . Selain menjelaskan algoritma penggaris Inggris, analisis ini juga memperkenalkan konsep dasar persamaan rekurensi yang berguna untuk menganalisis algoritma rekursif lainnya.



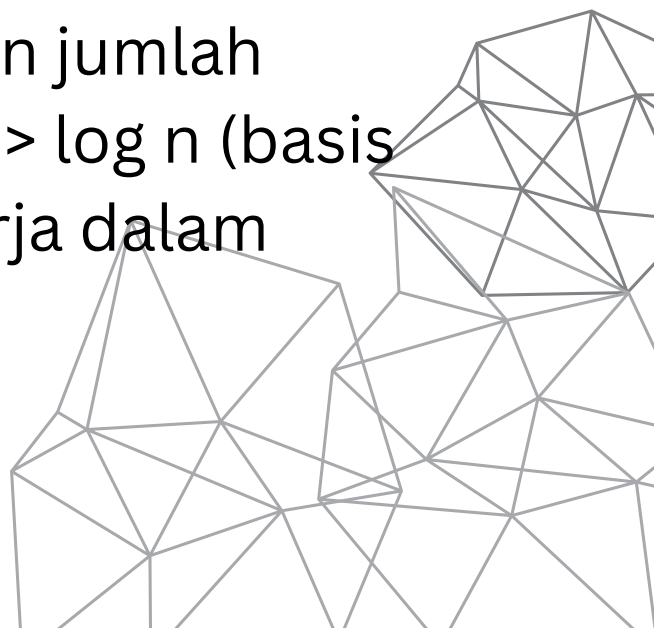


MELAKUKAN PENCARIAN BINER

Algoritma binary search memiliki efisiensi waktu yang tinggi karena setiap pemanggilan rekursif hanya melakukan operasi primitif dalam waktu konstan. Kompleksitas waktu algoritma ini adalah $O(\log n)$, yang tergantung pada jumlah pemanggilan rekursif, tidak lebih dari $\text{floor}(\log n) + 1$ untuk deret berukuran n elemen. Pembuktian Kompleksitas Waktu: Binary search mengurangi jumlah elemen yang perlu diperiksa setidaknya menjadi separuh pada setiap langkah rekursif. Setelah menghitung $\text{mid} = \text{floor}((\text{low} + \text{high}) / 2)$, jumlah elemen yang tersisa pada salah satu dari dua bagian adalah kurang dari separuh dari jumlah elemen sebelumnya. Hal ini dijelaskan melalui dua kondisi:

1. Elemen yang berada di sebelah kiri mid berjumlah $(\text{mid} - 1) - \text{low} + 1 \leq (\text{high} - \text{low} + 1) / 2$.
2. Elemen yang berada di sebelah kanan mid berjumlah $\text{high} - (\text{mid} + 1) + 1 \leq (\text{high} - \text{low} + 1) / 2$.

Pada kasus terburuk (pencarian gagal), proses berhenti ketika tidak ada kandidat yang tersisa, dan jumlah maksimum pemanggilan rekursif r adalah bilangan bulat terkecil dimana $n/(2^r) < 1$, yang berarti $r > \log n$ (basis 2), atau secara ekuivalen $r = \text{floor}(\log n) + 1$. Ini menunjukkan bahwa algoritma binary search bekerja dalam waktu $O(\log n)$, menjadikannya sangat efisien untuk pencarian dalam data yang sudah terurut.





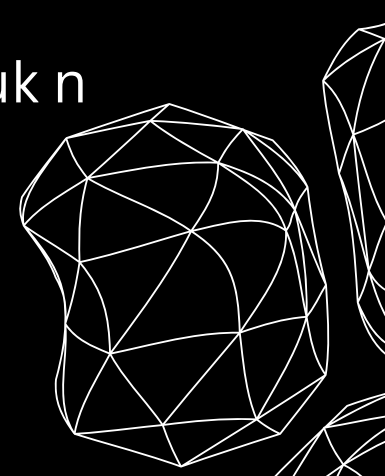
KOMPUTASI PENGGUNAAN RUANG DISK

Algoritma rekursif untuk menghitung penggunaan ruang disk di sistem file adalah contoh aplikasi rekursi pada struktur data hierarkis. Setiap pemanggilan fungsi memproses satu entri (file atau direktori) dan secara rekursif menangani entri-entri dalam direktori jika ada. Meskipun pada awalnya tampak memiliki kompleksitas kuadratik $O(n^2)$ karena banyaknya pemanggilan rekursif dan pemrosesan entri, analisis lebih lanjut menunjukkan bahwa total operasi sebenarnya bersifat linear $O(n)$.

Pemahaman dan Pembelajaran:

- Algoritma ini menunjukkan bahwa meskipun direktori dapat berisi banyak entri, setiap entri hanya diproses satu kali, yang menjamin kompleksitas waktu linear.
- Proses rekursif ini mendistribusikan beban komputasi secara merata, sebuah fenomena yang dikenal sebagai amortisasi.
- Dengan pendekatan ini, dapat dibuktikan bahwa total iterasi adalah $n-1$, menghasilkan kompleksitas waktu keseluruhan $O(n)$.

Analisis algoritma penghitungan disk usage ini memberikan beberapa wawasan penting dalam ilmu komputer. Pertama, algoritma ini merupakan contoh konkret dari teknik amortisasi, di mana analisis kumulatif memberikan hasil yang lebih akurat daripada asumsi kasus terburuk untuk setiap langkah. Kedua, struktur sistem file secara alami membentuk pohon (tree structure), menjadikan algoritma ini sebagai implementasi spesifik dari tree traversal. Dalam konteks yang lebih luas, pemahaman ini mengarah pada generalisasi bahwa semua algoritma tree traversal memiliki kompleksitas waktu $O(n)$ untuk n node.





4.3 RECURSION RUN AMOK



EFISIENSI REKURSI: KASUS FUNGSI UNIQUE3

Rekursi adalah alat yang sangat berguna dalam pemrograman, tetapi jika tidak diterapkan dengan bijak, dapat menyebabkan ketidakefisienan yang signifikan. Salah satu contoh masalah ini adalah fungsi `unique3` yang digunakan untuk memeriksa keunikan elemen dalam sebuah rangkaian. Meskipun logika rekursif yang digunakan dalam fungsi ini benar — yaitu memastikan bahwa bagian pertama dan terakhir dari rangkaian bersifat unik, dan elemen pertama tidak sama dengan elemen terakhir — implementasi rekursif ini sangat tidak efisien.

Analisis Masalah:

- Setiap pemanggilan untuk ukuran n menghasilkan dua pemanggilan rekursif untuk ukuran $n-1$, yang kemudian berkembang menjadi empat pemanggilan untuk $n-2$, delapan untuk $n-3$, dan seterusnya, secara eksponensial.
- Hal ini menciptakan total pemanggilan fungsi yang membentuk deret geometri: $1 + 2 + 4 + \dots + 2^{(n-1)}$, yang hasilnya setara dengan $2^n - 1$.
- Kompleksitas waktu yang dihasilkan adalah $O(2^n)$, yang sangat tinggi dan tidak efisien, terutama jika dibandingkan dengan solusi yang lebih optimal.

Penyebab Ketidakefisienan: Masalah utama dalam pendekatan rekursif ini bukanlah penggunaan rekursi itu sendiri, melainkan pola rekursi yang berulang dan redundan. Fungsi tersebut melakukan pemeriksaan yang sama berulang kali, yang mengarah pada pertumbuhan pemanggilan yang eksponensial.

Alternatif yang Lebih Efisien: Sebagai perbandingan, solusi yang lebih efisien dapat diterapkan menggunakan pendekatan iteratif atau rekursif dengan algoritma yang memiliki kompleksitas waktu $O(n)$ atau $O(n \log n)$. Dengan menggunakan struktur data yang lebih efisien seperti set atau memanfaatkan teknik pembagian dan penaklukan (divide and conquer), kita dapat mengurangi jumlah pemanggilan dan mencapai hasil yang lebih cepat.

```
1 def unique3(S, start, stop):
2     """Return True if there are no duplicate elements in slice S[start:stop]."""
3     if stop - start <= 1: return True          # at most one item
4     elif not unique(S, start, stop-1): return False # first part has duplicate
5     elif not unique(S, start+1, stop): return False # second part has duplicate
6     else: return S[start] != S[stop-1]        # do first and last differ?
```

Code Fragment 4.6: Recursive `unique3` for testing element uniqueness.



REKURSI YANG TIDAK EFISIEN UNTUK MENGHITUNG BILANGAN FIBONACCI

Fungsi Fibonacci dapat didefinisikan secara rekursif sebagai berikut:

$$\begin{aligned} F_0 &= 0 \\ F_1 &= 1 \\ F_n &= F_{n-2} + F_{n-1} \quad \text{for } n > 1. \end{aligned}$$

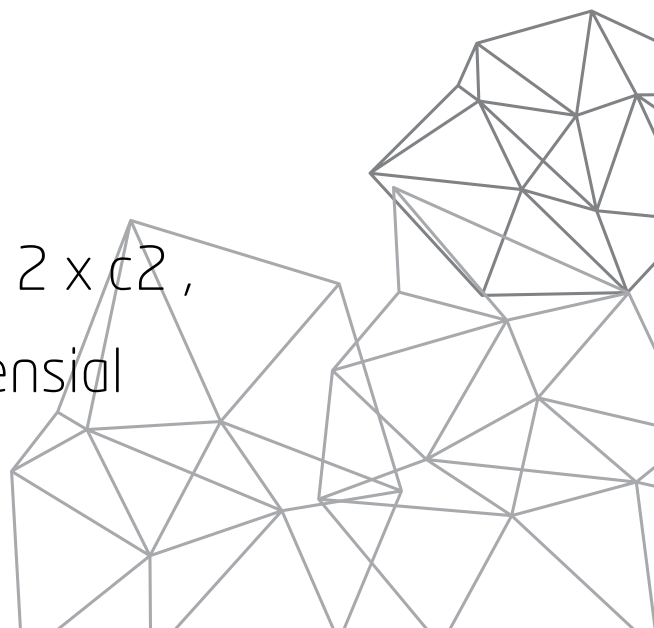
Ironisnya, implementasi langsung dari definisi ini menghasilkan fungsi yang sangat tidak efisien, seperti pada Code berikut:

```
1 def bad_fibonacci(n):
2     """ Return the nth Fibonacci number."""
3     if n <= 1:
4         return n
5     else:
6         return bad_fibonacci(n-2) + bad_fibonacci(n-1)
```

Sayangnya, pendekatan ini menyebabkan jumlah pemanggilan rekursif yang eksponensial. Misalkan c_n menyatakan jumlah pemanggilan fungsi dalam eksekusi `bad_fibonacci(n)`. Berikut adalah perhitungan untuk beberapa nilai c_n :

$c_0 = 1$	$c_5 = 1 + c_3 + c_4 = 1 + 5 + 9 = 15$
$c_1 = 1$	$c_6 = 1 + c_4 + c_5 = 1 + 9 + 15 = 25$
$c_2 = 1 + c_0 + c_1 = 1 + 1 + 1 = 3$	$c_7 = 1 + c_5 + c_6 = 1 + 15 + 25 = 41$
$c_3 = 1 + c_1 + c_2 = 1 + 1 + 3 = 5$	$c_8 = 1 + c_6 + c_7 = 1 + 25 + 41 = 67$
$c_4 = 1 + c_2 + c_3 = 1 + 3 + 5 = 9$	

Pola ini menunjukkan bahwa jumlah pemanggilan lebih dari dua kali lipat setiap dua indeks berturut-turut. Misalnya, $c_4 > 2 \times c_2$, $c_5 > 2 \times c_3$, dan seterusnya. Dengan demikian, $c_n > 2^{\lceil n/2 \rceil}$, yang berarti `bad_fibonacci(n)` memerlukan waktu eksponensial terhadap n .



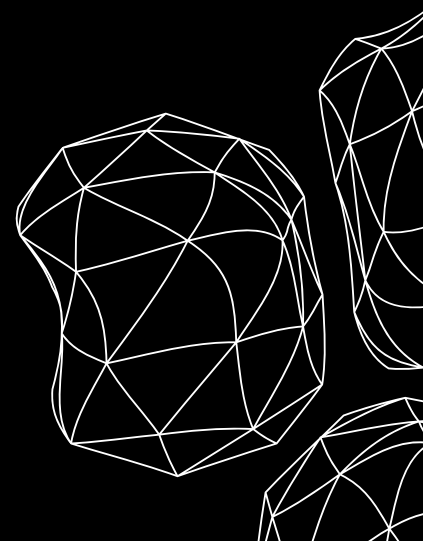


REKURSI YANG EFISIEN UNTUK MENGHITUNG BILANGAN FIBONACCI

Kita sering tergoda menggunakan pendekatan rekursif yang tidak efisien untuk menghitung bilangan Fibonacci karena rumus $F_n = F_{n-2} + F_{n-1}$ terlihat sederhana. Namun, masalah muncul saat F_{n-1} juga memanggil F_{n-2} secara terpisah, sehingga terjadi perhitungan berulang yang sama. Bahkan, F_{n-3} akan dihitung ulang berkali-kali, menyebabkan efek bola salju yang menghasilkan waktu eksponensial $O(2^n)$ pada fungsi `bad_fibonacci`. Solusi yang lebih baik adalah mengubah pendekatan rekursif sehingga setiap pemanggilan hanya melakukan satu rekursi, bukan dua. Caranya adalah dengan mengembalikan sepasang bilangan Fibonacci berurutan (F_n, F_{n-1}) , di mana $F_{-1} = 0$. Meskipun terkesan lebih rumit, pendekatan ini menghindari perhitungan ulang dengan memanfaatkan nilai yang sudah diperoleh dari rekursi sebelumnya. Implementasinya ditunjukkan pada Code Fragment 4.8:

```
1 def good_fibonacci(n):
2     """ Return pair of Fibonacci numbers, F(n) and F(n-1). """
3     if n <= 1:
4         return (n, 0)
5     else:
6         (a, b) = good_fibonacci(n-1)
7         return (a+b, a)
```

Dalam hal efisiensi, perbedaan antara rekursi buruk (`bad_fibonacci`) dan rekursi baik (`good_fibonacci`) sangat mencolok. Fungsi `good_fibonacci` hanya memerlukan waktu linear $O(n)$ karena setiap rekursi mengurangi nilai n sebanyak 1, menghasilkan total n pemanggilan. Setiap langkah non-rekursif hanya membutuhkan waktu konstan, sehingga keseluruhan proses berjalan jauh lebih cepat dibandingkan pendekatan eksponensial.





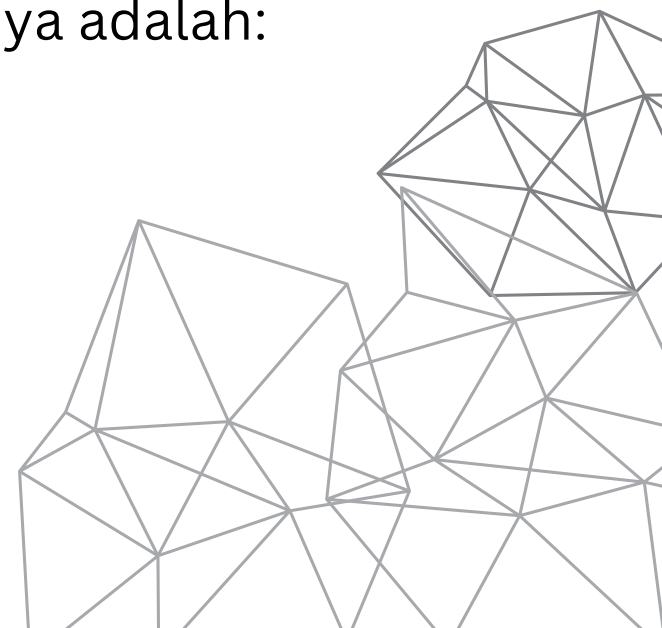
4.3.1 KEDALAMAN MAKSIMUM REKURSI PADA PYTHON

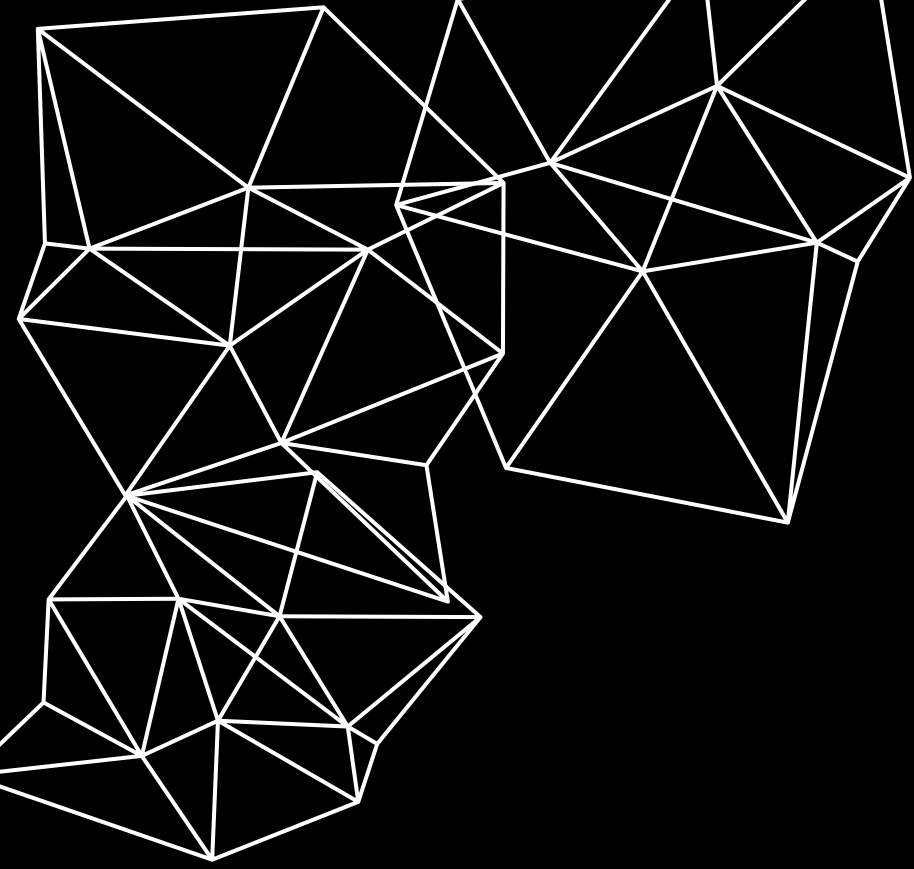
Salah satu risiko utama dalam penggunaan rekursi adalah rekursi tak terbatas, yakni kondisi di mana fungsi terus memanggil dirinya sendiri tanpa pernah mencapai base case. Hal ini menyebabkan konsumsi sumber daya yang ekstrem karena setiap pemanggilan menciptakan activation record baru di memori. Contoh ekstremnya adalah fungsi `fib(n)` yang memanggil dirinya sendiri tanpa syarat penghentian. Namun, kesalahan semacam ini juga bisa muncul dalam bentuk yang lebih halus—misalnya dalam implementasi binary search—jika indeks batas rekursif tidak dikurangi dengan benar, sehingga menghasilkan pemanggilan dengan rentang yang sama secara berulang.

Untuk menghindari efek fatal dari rekursi tak terbatas, Python menetapkan batas maksimum kedalaman rekursi, umumnya 1000 level. Jika batas ini dilampaui, Python akan menghasilkan `RuntimeError: maximum recursion depth exceeded`. Batas ini cukup untuk kebanyakan algoritma dengan kedalaman rekursi $O(\log n)$, seperti binary search, namun bisa menjadi kendala bagi algoritma dengan rekursi linear $O(n)$.

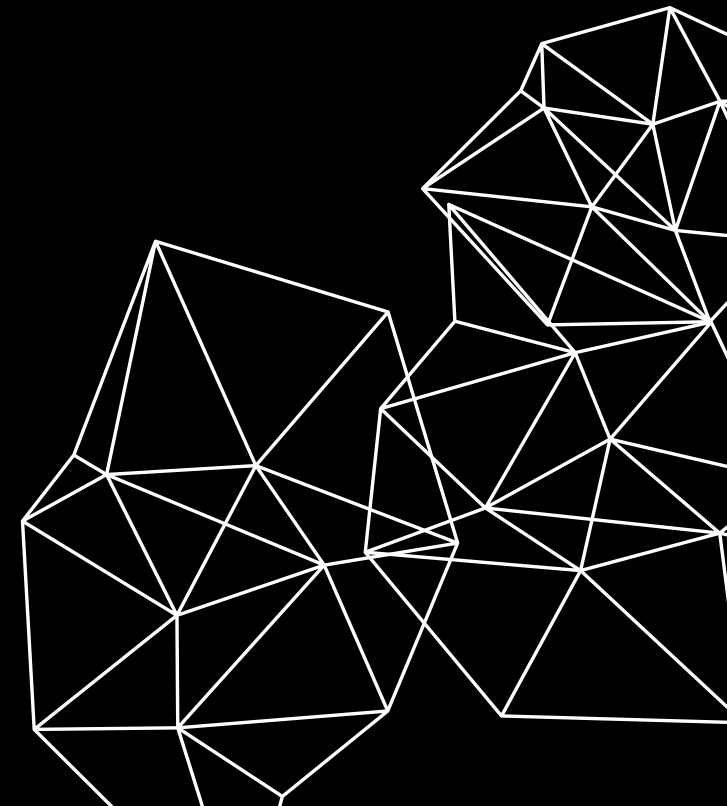
Untungnya, batas rekursi di Python dapat diubah secara dinamis menggunakan modul ``sys``. Contoh penggunaannya adalah:

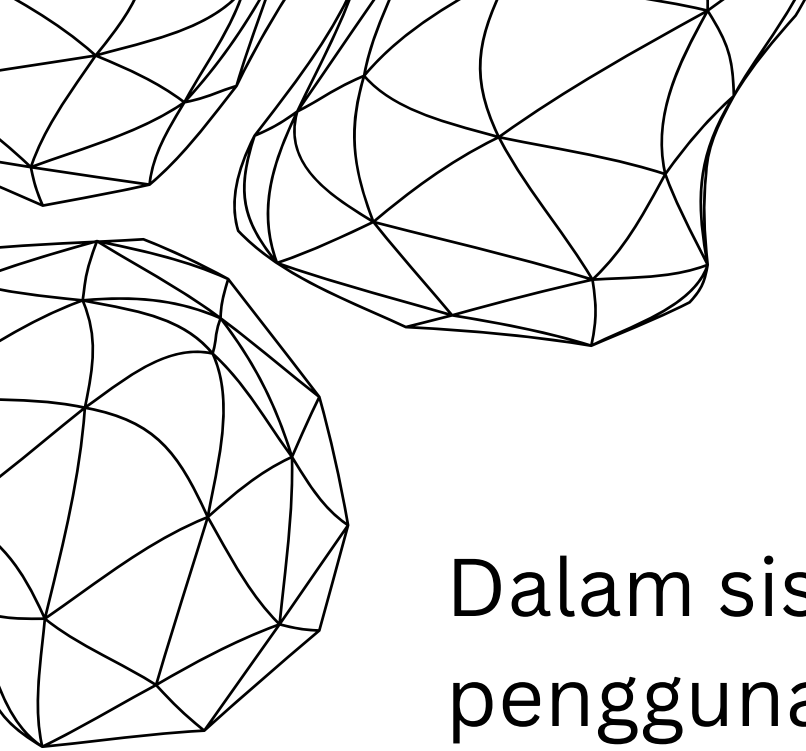
```
import sys
old = sys.getrecursionlimit() # mendapatkan batas saat ini (misalnya 1000)
sys.setrecursionlimit(1000000) # mengubah batas menjadi 1 juta pemanggilan
```





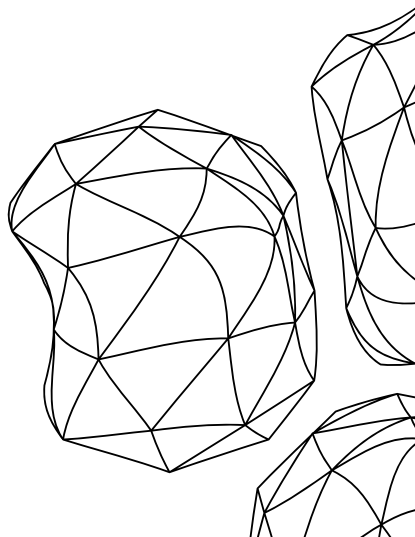
4.4 FURTHER EXAMPLES OF RECURSION





Dalam sisa bab ini, kami memberikan contoh tambahan tentang penggunaan rekursi. Kami mengatur presentasi kami dengan mempertimbangkan jumlah maksimum pemanggilan rekursif yang dapat dimulai dari dalam tubuh sebuah aktivasi.

- Jika sebuah pemanggilan rekursif dimulai dari satu pemanggilan lainnya, kita menyebutnya sebagai rekursi linier.
- Jika sebuah pemanggilan rekursif dapat memulai dua pemanggilan lainnya, kita menyebutnya sebagai rekursi biner.
- Jika sebuah pemanggilan rekursif dapat memulai tiga atau lebih pemanggilan lainnya, ini adalah rekursi multipel.

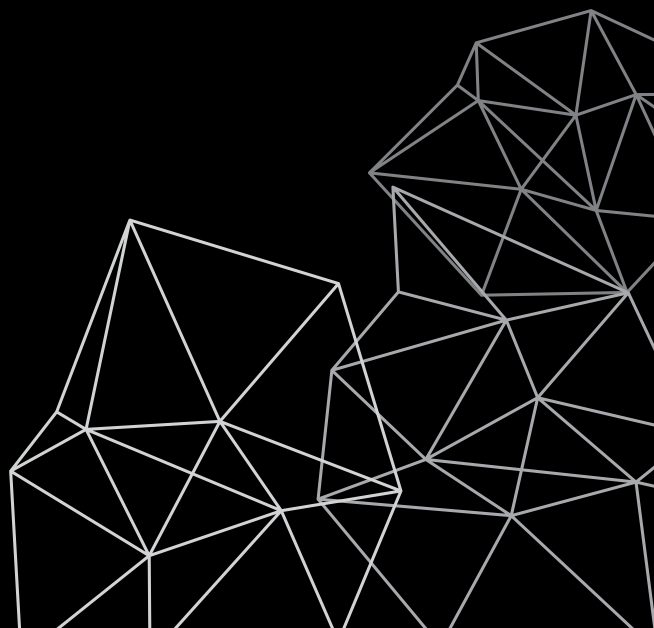


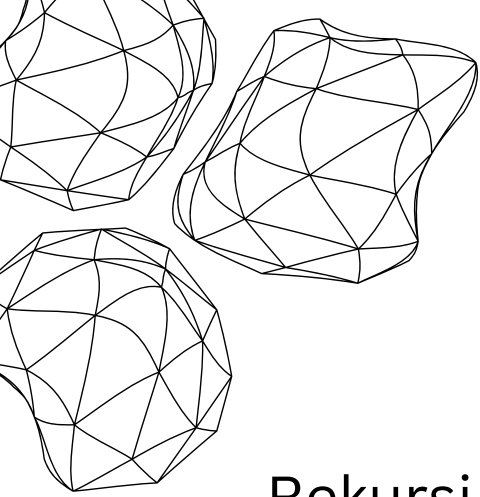


4.4.1 REKURSI LINIER

Rekursi linear terjadi ketika sebuah fungsi rekursif melakukan maksimal satu pemanggilan rekursi baru dalam setiap eksekusi. Contohnya termasuk fungsi faktorial dan implementasi Fibonacci yang efisien (`good_fibonacci`). Meskipun dinamakan "binary", algoritma binary search juga termasuk rekursi linear karena hanya satu dari dua cabang rekursif yang dieksekusi setiap kali.

Ciri khas rekursi linear adalah jejak pemanggilan berbentuk satu rangkaian lurus, seperti pada rekursi faktorial. Istilah ini merujuk pada struktur jejak rekursi, bukan analisis waktu. Misalnya, binary search tetap memiliki kompleksitas waktu $O(\log n)$ meskipun bersifat rekursi linear.





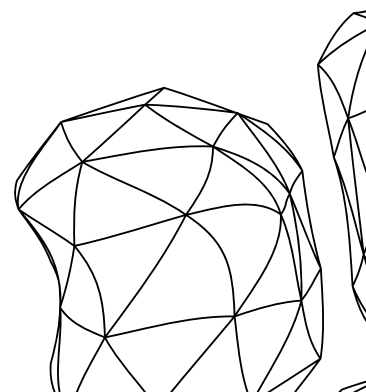
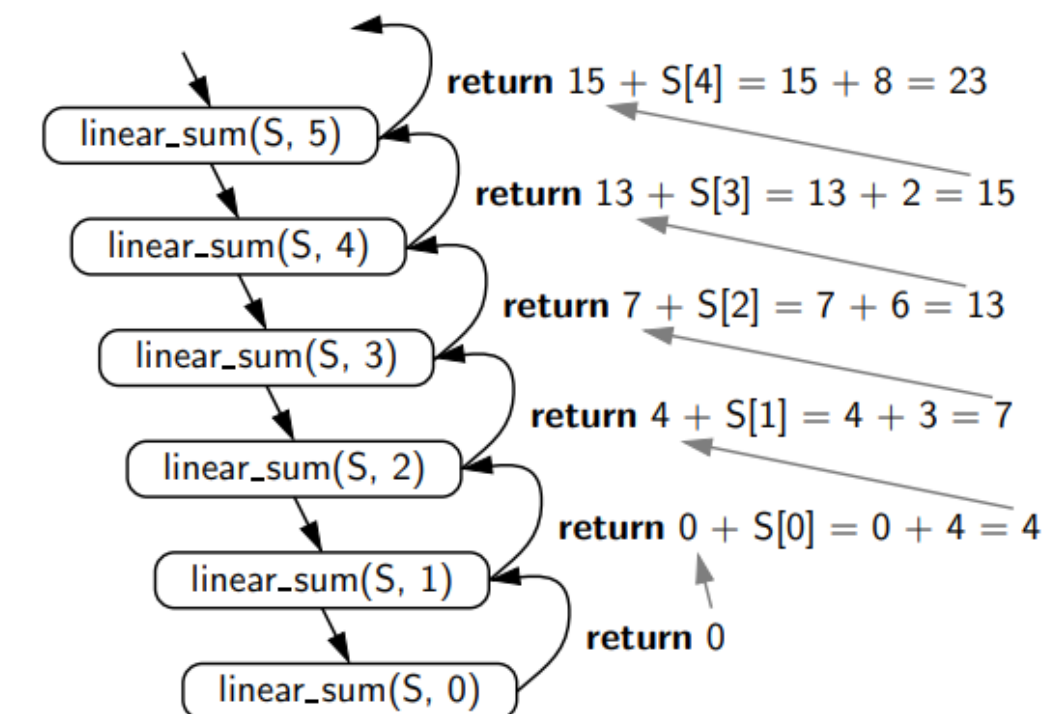
MENJUMLAHKAN ELEMEN-ELEMEN DARI SUATU BARISAN SECARA REKURSIF

Rekursi linear merupakan alat yang berguna untuk memproses rangkaian data, seperti daftar (list) dalam Python. Sebagai contoh, jika kita ingin menghitung jumlah dari suatu rangkaian bilangan bulat S dengan panjang n , masalah ini dapat diselesaikan menggunakan rekursi linear. Observasinya adalah jumlah semua n bilangan dalam S adalah 0 jika $n = 0$, dan jika tidak, jumlah tersebut adalah hasil penjumlahan dari $n-1$ bilangan pertama dalam S ditambah elemen terakhir S . (Lihat Gambar 4.9).

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

Jejak rekursi dari fungsi `linear_sum` untuk contoh kecil ditunjukkan pada Gambar 4.10. Untuk masukan berukuran n , algoritma `linear_sum` melakukan $n+1$ panggilan fungsi. Oleh karena itu, waktu yang dibutuhkan adalah $O(n)$, karena setiap panggilan non-rekursif memerlukan waktu konstan. Selain itu, ruang memori yang digunakan oleh algoritma (selain rangkaian S) juga $O(n)$, karena setiap dari $n+1$ catatan aktivasi dalam jejak memerlukan ruang memori konstan pada panggilan rekursif terakhir (dengan $n=0$).

```
1 def linear_sum(S, n):
2     """Return the sum of the first n numbers of sequence S."""
3     if n == 0:
4         return 0
5     else:
6         return linear_sum(S, n-1) + S[n-1]
```



MEMBALIK URUTAN DENGAN REKURSIF

Untuk membalik urutan n elemen dalam list S menggunakan rekursi linear, kita bisa menukar elemen pertama dan terakhir, lalu secara rekursif membalik elemen-elemen di antaranya. Terdapat dua kasus dasar: (1) jika $\text{start} == \text{stop}$, artinya tidak ada elemen; (2) jika $\text{start} == \text{stop} - 1$, hanya satu elemen — keduanya tidak memerlukan tindakan. Rekursi dijamin berhenti karena setiap panggilan mengurangi jarak $\text{stop} - \text{start}$ sebesar 2, hingga mencapai kasus dasar. Total panggilan rekursif adalah $1 + \lfloor n/2 \rfloor$, dan karena setiap panggilan hanya melakukan kerja konstan, algoritma ini berjalan dalam waktu $O(n)$.

```
1 def reverse(S, start, stop):
2     """ Reverse elements in implicit slice S[start:stop]. """
3     if start < stop - 1:                # if at least 2 elements:
4         S[start], S[stop-1] = S[stop-1], S[start]    # swap first and last
5         reverse(S, start+1, stop-1)                # recur on rest
```

0	1	2	3	4	5	6
4	3	6	2	8	9	5
5	3	6	2	8	9	4
5	9	6	2	8	3	4
5	9	8	2	6	3	4
5	9	8	2	6	3	4

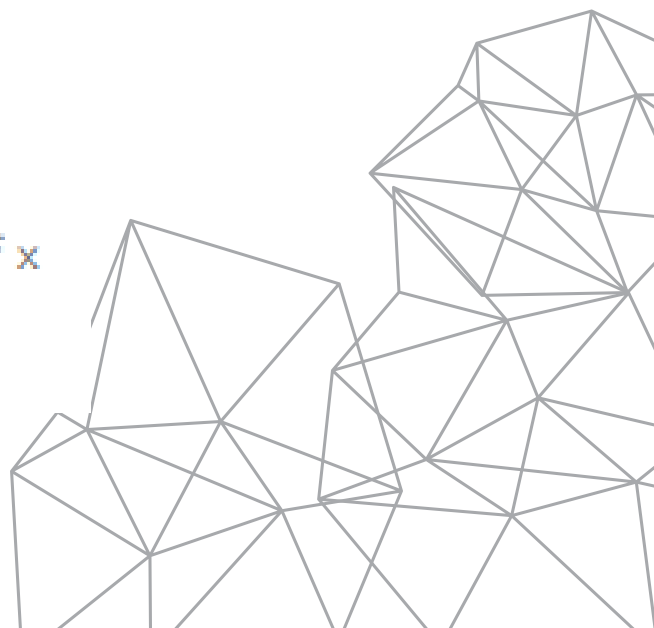


ALGORITMA REKURASIF UNTUK MENGHITUNG PANGKAT

Masalah menghitung pangkat $\text{power}(x, n)$ dapat diselesaikan secara rekursif dengan dua pendekatan berbeda. Pendekatan pertama menggunakan definisi dasar $x^n = x * x^{(n-1)}$ yang menghasilkan kompleksitas waktu dan memori sebesar $O(n)$, karena membutuhkan n pemanggilan rekursif. Sebaliknya, pendekatan yang lebih efisien adalah exponentiation by squaring, yang memanfaatkan bahwa $x^n = (x^{(n//2)})^2$ jika n genap, dan $x^n = x * (x^{(n//2)})^2$ jika n ganjil. Pendekatan ini hanya membutuhkan $O(\log n)$ pemanggilan rekursif dan memori, sehingga jauh lebih efisien untuk nilai n yang besar.

```
def power(x, n):  
    """ Compute the value x**n for integer n."""  
    if n == 0:  
        return 1  
    else:  
        return x * power(x, n-1)
```

```
1 def power(x, n):  
2     """ Compute the value x**n for integer n."""  
3     if n == 0:  
4         return 1  
5     else:  
6         partial = power(x, n // 2)           # rely on truncated division  
7         result = partial * partial  
8         if n % 2 == 1:                         # if n odd, include extra factor of x  
9             result *= x  
10        return result
```



Untuk mengilustrasikan eksekusi algoritma yang telah diperbaiki, Gambar 4.12 memberikan jejak rekursi untuk perhitungan $\text{power}(2,13)$.

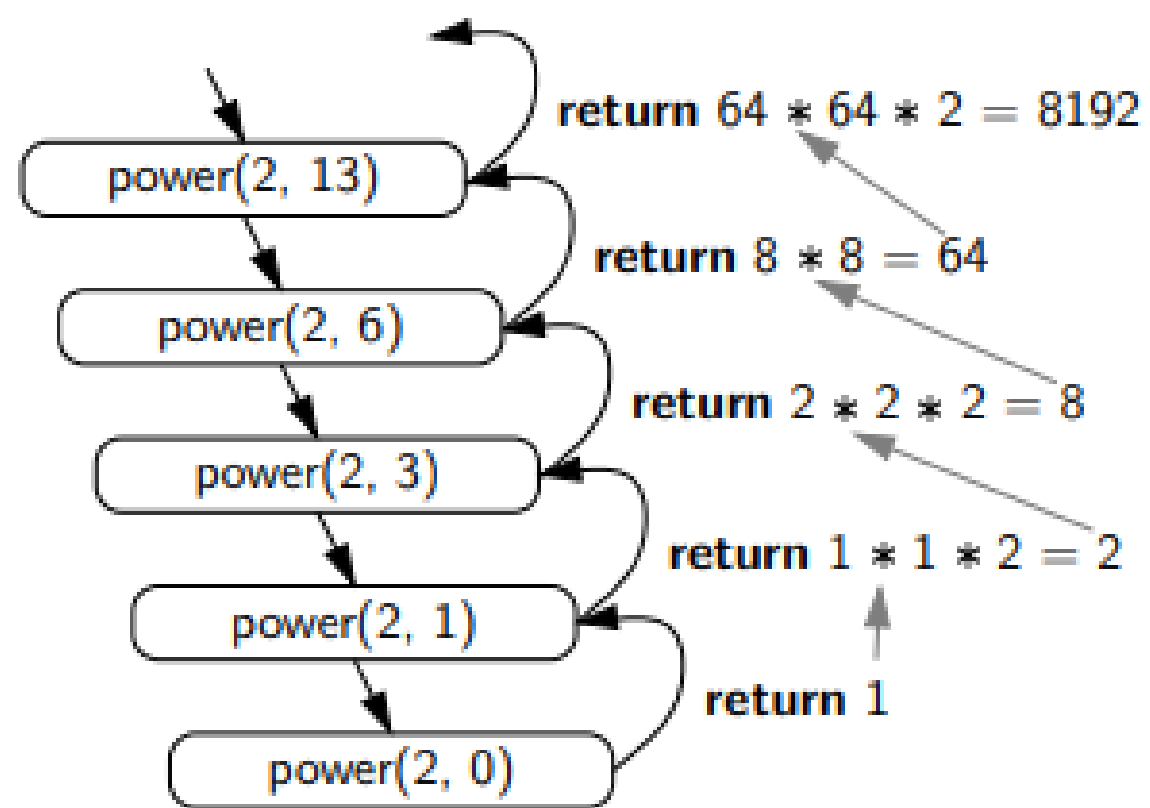


Figure 4.12: Recursion trace for an execution of $\text{power}(2, 13)$.

Pada versi yang telah direvisi, fungsi $\text{power}(x, n)$ memanfaatkan pembagian eksponen secara rekursif, sehingga setiap pemanggilan hanya memproses setengah dari nilai sebelumnya. Pola ini menghasilkan kedalaman rekursi sebesar $O(\log n)$, menjadikan total waktu eksekusi algoritma juga $O(\log n)$, karena setiap aktivasi fungsi memerlukan waktu konstan. Ini merupakan peningkatan signifikan dibandingkan versi awal dengan kompleksitas $O(n)$. Selain efisiensi waktu, versi ini juga lebih hemat memori. Jika sebelumnya dibutuhkan $O(n)$ catatan aktivasi dalam memori karena kedalaman rekursi linear, kini hanya diperlukan $O(\log n)$, sejalan dengan kedalaman rekursi yang jauh lebih dangkal. Optimasi ini menunjukkan pentingnya desain rekursif yang cermat untuk mencapai efisiensi maksimal dalam hal waktu dan ruang.

4.4.2 REKURSI BINER

Rekursi biner terjadi ketika sebuah fungsi membuat dua pemanggilan rekursif dalam satu eksekusi. Salah satu penerapannya adalah dalam fungsi `binary_sum`, yang menghitung jumlah n elemen dalam urutan S dengan membagi urutan menjadi dua bagian, menghitung jumlah masing-masing secara rekursif, lalu menjumlahkan hasilnya. Jika n merupakan pangkat dua, kedalaman rekursinya mencapai $1 + \log_2 n$, sehingga penggunaan ruang hanya $O(\log n)$, jauh lebih efisien dibandingkan versi linear yang menggunakan $O(n)$ ruang. Meski begitu, karena total jumlah pemanggilan fungsi adalah $2n - 1$ dan setiap pemanggilan membutuhkan waktu konstan, kompleksitas waktunya tetap $O(n)$.

```
1 def binary_sum(S, start, stop):
2     """Return the sum of the numbers in implicit slice S[start:stop]."""
3     if start >= stop:                # zero elements in slice
4         return 0
5     elif start == stop-1:            # one element in slice
6         return S[start]
7     else:                            # two or more elements in slice
8         mid = (start + stop) // 2
9         return binary_sum(S, start, mid) + binary_sum(S, mid, stop)
```

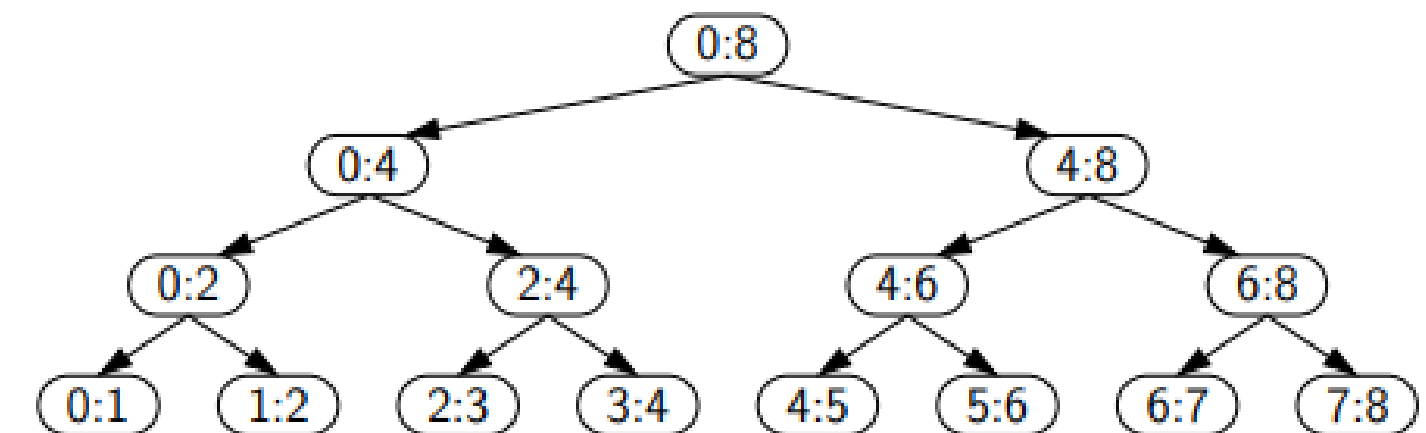


Figure 4.13: Recursion trace for the execution of `binary_sum(0, 8)`.

4.4.3 REKURSI BERGANDA

Rekursi berganda adalah bentuk rekursi di mana suatu fungsi dapat membuat lebih dari dua panggilan rekursif dalam satu eksekusi. Contohnya adalah saat menganalisis penggunaan ruang disk dalam sistem berkas, di mana fungsi memanggil dirinya sendiri untuk setiap entri dalam direktori. Rekursi berganda juga umum digunakan dalam penyelesaian teka-teki kombinatorial, seperti teka-teki penjumlahan kata contohnya: *pot+pan=bib*, *dog+cat=pig*, *boy+girl=baby*, yang mengharuskan kita menetapkan digit unik ke setiap huruf agar persamaan aritmatika valid.

Untuk menyelesaikan masalah ini secara komputasional, kita dapat menggunakan rekursi berganda untuk menghasilkan dan menguji semua kemungkinan penetapan digit ke huruf. Algoritma semacam ini bekerja dengan membangun semua urutan elemen sepanjang k tanpa pengulangan dari himpunan U (misalnya, $\{0,1,\dots,9\}$), dan menguji setiap urutan sebagai solusi potensial. Pada setiap langkah, algoritma secara rekursif memperluas urutan yang ada dengan elemen baru yang belum digunakan. Proses ini pada dasarnya menghasilkan semua permutasi dari ukuran k dari U , dan memeriksa validitas setiap permutasi terhadap teka-teki. Karena jumlah percabangan dapat sangat banyak, algoritma ini menunjukkan struktur khas dari rekursi berganda, dengan jejak rekursi yang tumbuh eksponensial tergantung pada ukuran U dan nilai k .

Algorithm PuzzleSolve(k,S,U):

Input: An integer k , sequence S , and set U

Output: An enumeration of all k -length extensions to S using elements in U without repetitions

for each e in U **do**

 Add e to the end of S

 Remove e from U

 { e is now being used}

if $k == 1$ **then**

 Test whether S is a configuration that solves the puzzle

if S solves the puzzle **then**

return "Solution found: " S

else

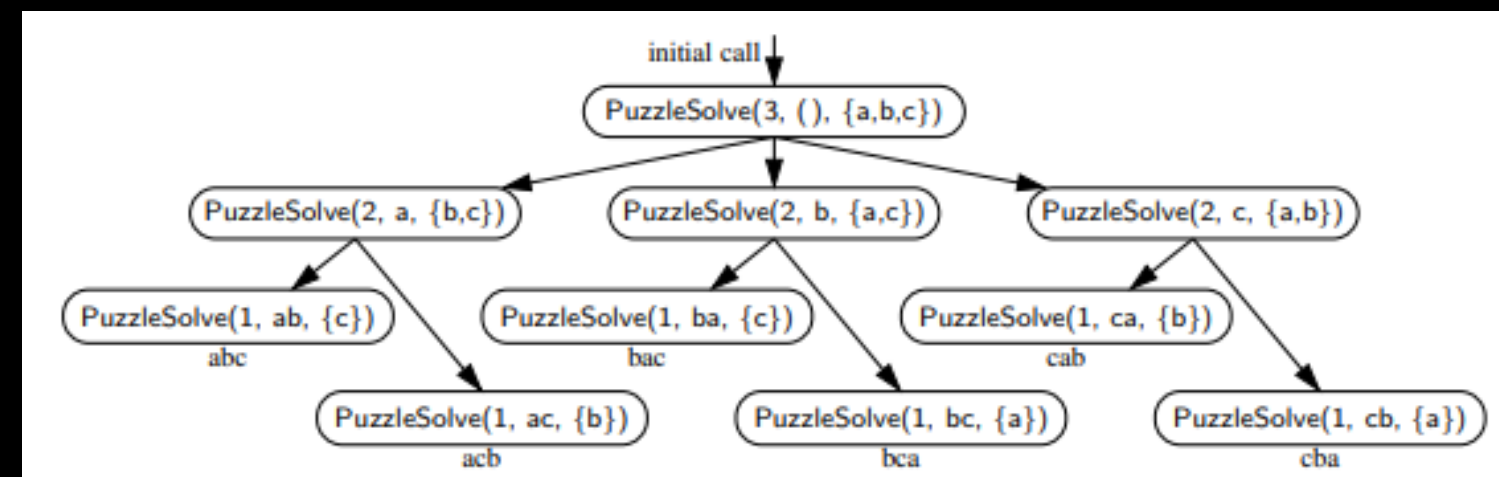
 PuzzleSolve($k-1,S,U$)

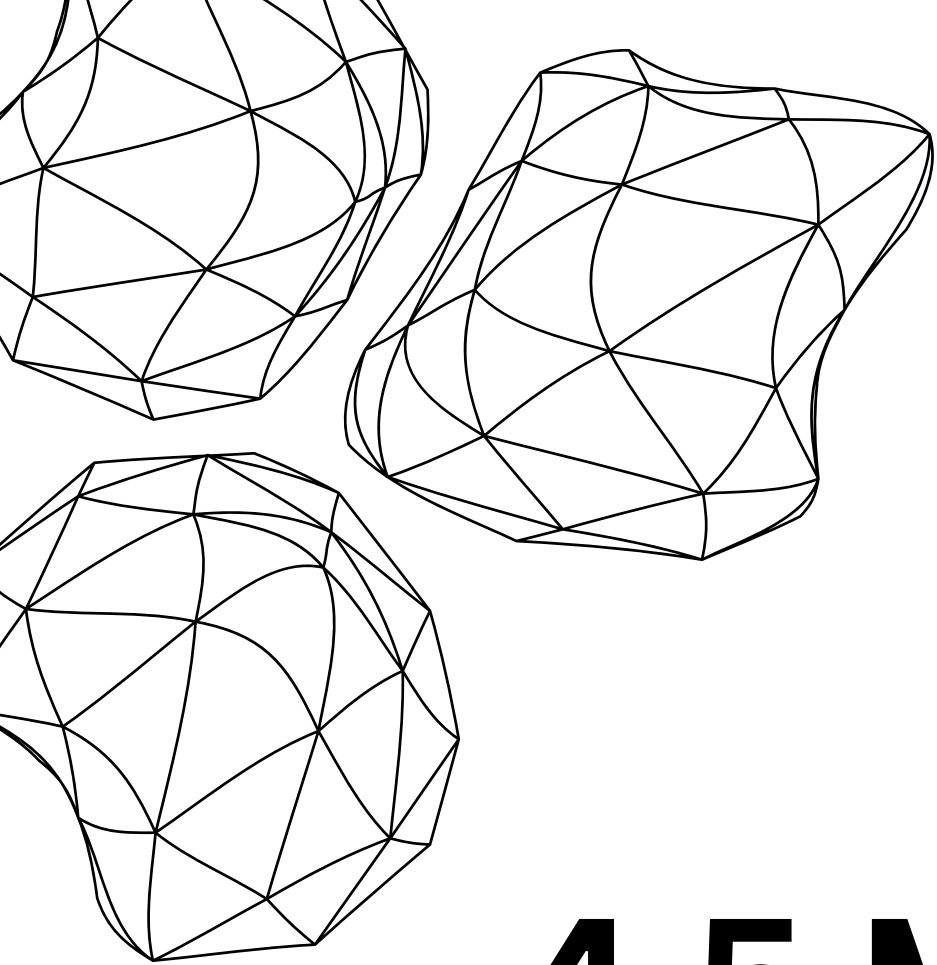
 {a recursive call}

 Remove e from the end of S

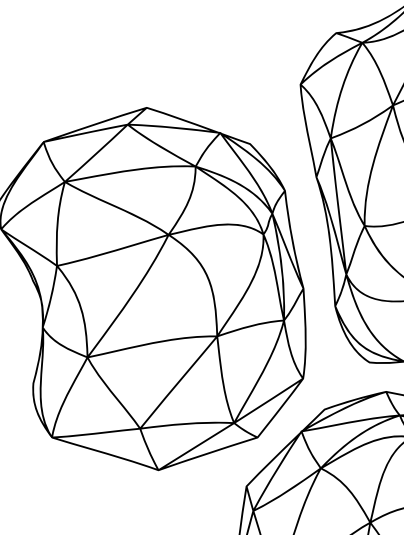
 Add e back to U

 { e is now considered as unused}





4.5 MERANCANG ALGORITMA REKURSIF



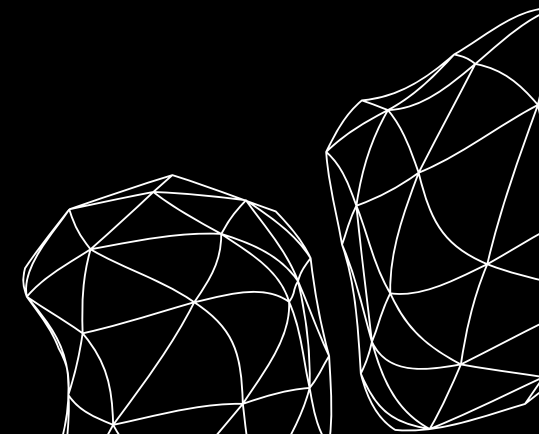


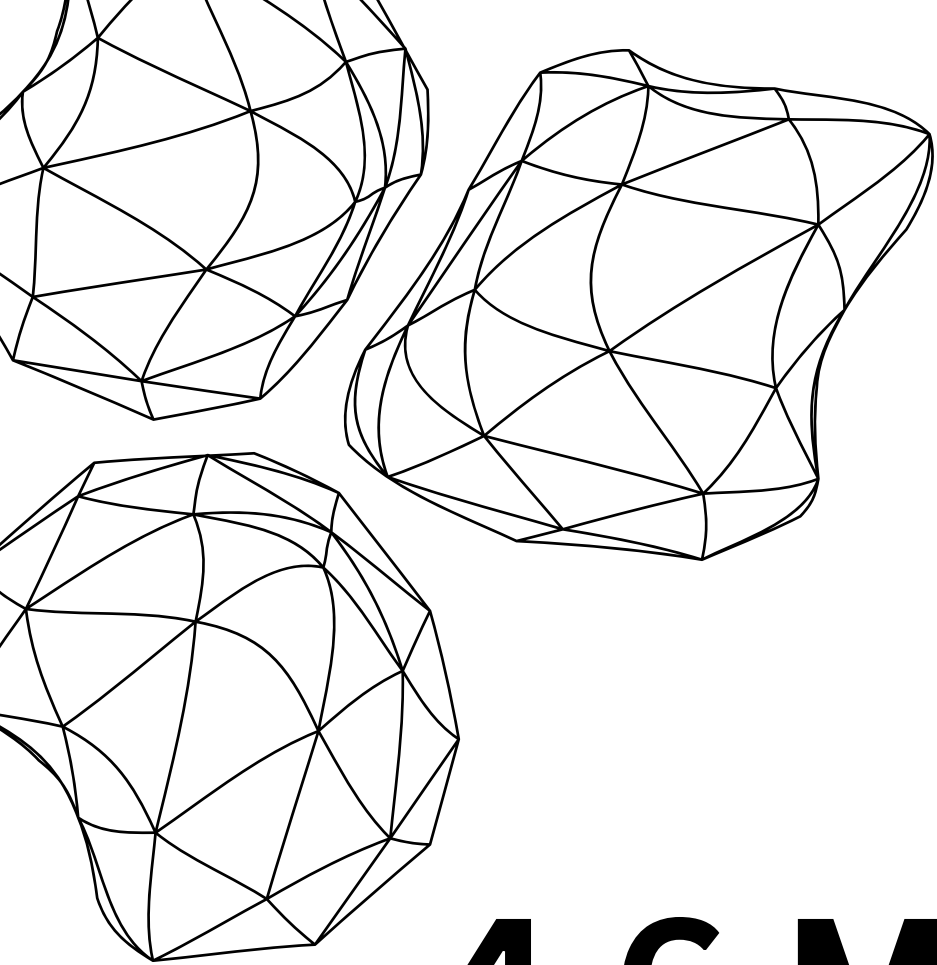
MEMPARAMETERKAN REKURSI

Algoritma rekursif umumnya terdiri dari dua komponen utama: pengujian kasus dasar dan langkah rekursif. Kasus dasar harus ditangani tanpa rekursi dan menjamin bahwa setiap rantai pemanggilan akhirnya akan berhenti. Langkah rekursif dilakukan jika kondisi dasar tidak terpenuhi, dengan memastikan bahwa setiap pemanggilan membawa kita lebih dekat ke kasus dasar.

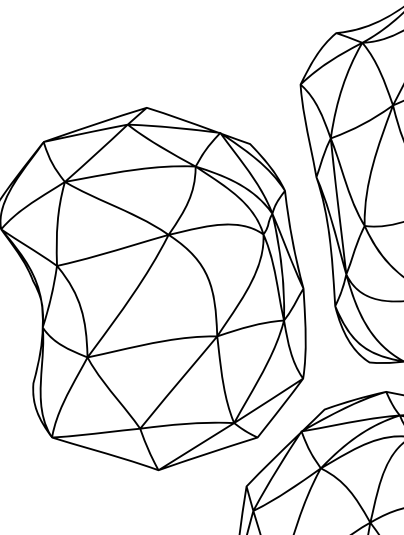
Dalam merancang algoritma rekursif, sering kali kita perlu memikirkan ulang cara mendefinisikan submasalah—dikenal sebagai reparameterisasi rekursi. Pendekatan ini memungkinkan kita mempertahankan struktur masalah yang serupa sepanjang rekursi. Contoh klasiknya adalah pencarian biner, di mana kita menambahkan parameter `low` dan `high` ke dalam fungsi untuk menunjuk batas subdaftar, alih-alih membuat salinan baru yang mahal secara komputasi. Meskipun menambahkan parameter ini membuat fungsi lebih kompleks, kita dapat menyembunyikannya dari pengguna dengan menyediakan antarmuka publik yang lebih sederhana, seperti `binary_search(data, target)`, dan menyimpan logika rekursif dalam fungsi utilitas internal.


Prinsip reparameterisasi ini juga muncul dalam berbagai algoritma lain seperti `reverse`, `linear_sum`, dan `binary_sum`, serta dalam `good_fibonacci` yang secara eksplisit mengubah struktur nilai kembalian untuk meningkatkan efisiensi. Pendekatan ini menunjukkan bahwa desain rekursif yang efektif sering kali membutuhkan fleksibilitas dalam mendefinisikan ulang masalah agar lebih mudah dipecahkan secara rekursif.





4.6 MENGHILANGKAN REKURSI EKOR





Pendekatan rekursif dalam desain algoritma menawarkan manfaat signifikan dalam hal kesederhanaan dan kejelasan, karena memungkinkan kita memanfaatkan struktur berulang dalam masalah tanpa perlu penanganan kasus yang kompleks atau loop bersarang. Namun, rekursi membawa overhead memori karena setiap panggilan fungsi membutuhkan penyimpanan jejak aktivasi, sehingga dalam situasi memori terbatas, konversi ke versi non-rekursif dapat lebih efisien.

Konversi ini biasanya dilakukan menggunakan struktur data stack untuk mensimulasikan pemanggilan rekursif, atau lebih efisien lagi, dengan memanfaatkan rekursi ekor—yakni ketika panggilan rekursif adalah instruksi terakhir dalam fungsi. Fungsi seperti `binary_search` dan `reverse` adalah contoh rekursi ekor yang dapat dengan mudah ditransformasikan ke dalam bentuk iteratif tanpa kehilangan efisiensi.

Sebaliknya, fungsi seperti `factorial` atau `linear_sum` bukanlah rekursi ekor karena mereka masih melakukan operasi tambahan setelah panggilan rekursif selesai. Meskipun demikian, algoritma-algoritma tersebut juga bisa diubah ke bentuk iteratif dengan efisiensi serupa.

Transformasi semacam ini tidak hanya mengurangi konsumsi memori stack, tetapi juga meningkatkan kinerja dalam beberapa kasus, tanpa mengorbankan kejelasan algoritma—terutama bila disertai dengan fungsi pembungkus yang menyederhanakan antarmuka pemanggilan bagi pengguna.

```
1 def binary_search_iterative(data, target):
2     """ Return True if target is found in the given Python list."""
3     low = 0
4     high = len(data)-1
5     while low <= high:
6         mid = (low + high) // 2
7         if target == data[mid]:           # found a match
8             return True
9         elif target < data[mid]:
10            high = mid - 1                # only consider values left of mid
11        else:
12            low = mid + 1                  # only consider values right of mid
13    return False                          # loop ended without success
```

```
1 def reverse_iterative(S):
2     """ Reverse elements in sequence S."""
3     start, stop = 0, len(S)
4     while start < stop - 1:
5         S[start], S[stop-1] = S[stop-1], S[start]    # swap first and last
6         start, stop = start + 1, stop - 1           # narrow the range
```



TERIMA KASIH

**DISUSUN OLEH :
SYAHRUL AKBAR RAMDHANI (11230940000027)**

