



ALGORITMA DAN STRUKTUR DATA

RECURSION

PENGULANGAN DALAM PYTHON

Dani hidayat
11220940000014

4A





REKURSI

DEFINISI

Rekursi adalah teknik dimana suatu fungsi membuat satu atau lebih panggilan ke dirinya sendiri selama eksekusi. Ada banyak contoh rekursi dalam seni dan alam. Misalnya, pola fraktal bersifat rekursif secara alami. Contoh fisik rekursi yang digunakan dalam seni adalah boneka Matryoshka Rusia. Setiap boneka terbuat dari kayu padat dan berlubang yang di dalamnya berisi boneka Matryoshka lain di dalamnya.

CONTOH ILUSTRASI

PADA PENGGUNAAN REKURSI

1

Fungsi Faktorial

adalah fungsi matematika klasik yang memiliki definisi rekursif alami.

2

English Ruler

memiliki pola rekursif yang merupakan contoh sederhana dari fraktal struktur.

3

Pencarian Biner

adalah salah satu algoritma komputer yang paling penting. Hal ini memungkinkan kita menemukan nilai yang diinginkan secara efisien dalam kumpulan data dengan miliaran entri.

4

Sistem File

memiliki struktur rekursif di mana direktori dapat disarangkan secara acak di dalam direktori lain. Algoritme rekursif banyak digunakan untuk mengeksplorasi dan mengelola sistem file ini.

1. FUNGSI FAKTORIAL

Fungsi faktorial merupakan konsep matematika sederhana yang bisa diwujudkan dalam program komputer menggunakan rekursi. Faktorial dari bilangan bulat positif n , dilambangkan dengan $n!$

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot (n-1) \cdot (n-2) \cdots 3 \cdot 2 \cdot 1 & \text{if } n \geq 1. \end{cases}$$

Definisi rekursif ini dapat diformalkan sebagai

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot (n-1)! & \text{if } n \geq 1. \end{cases}$$

IMPLEMENTASI PADA PYTHON

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


1. FUNGSI FAKTORIAL

Fungsi faktorial merupakan konsep matematika sederhana yang bisa diwujudkan dalam program komputer menggunakan rekursi. Faktorial dari bilangan bulat positif n , dilambangkan dengan $n!$

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot (n-1) \cdot (n-2) \cdots 3 \cdot 2 \cdot 1 & \text{if } n \geq 1. \end{cases}$$

Definisi rekursif ini dapat diformalkan sebagai

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot (n-1)! & \text{if } n \geq 1. \end{cases}$$

IMPLEMENTASI PADA PYTHON

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


2. MENGGAMBAR PENGGARIS INGGRIS

Penggunaan rekursi dalam menggambar penggaris inggris menjadi lebih relevan. Ide dasarnya adalah untuk membuat garis dengan label numerik pada setiap inci, dengan panjang centang yang berkurang secara proporsional.

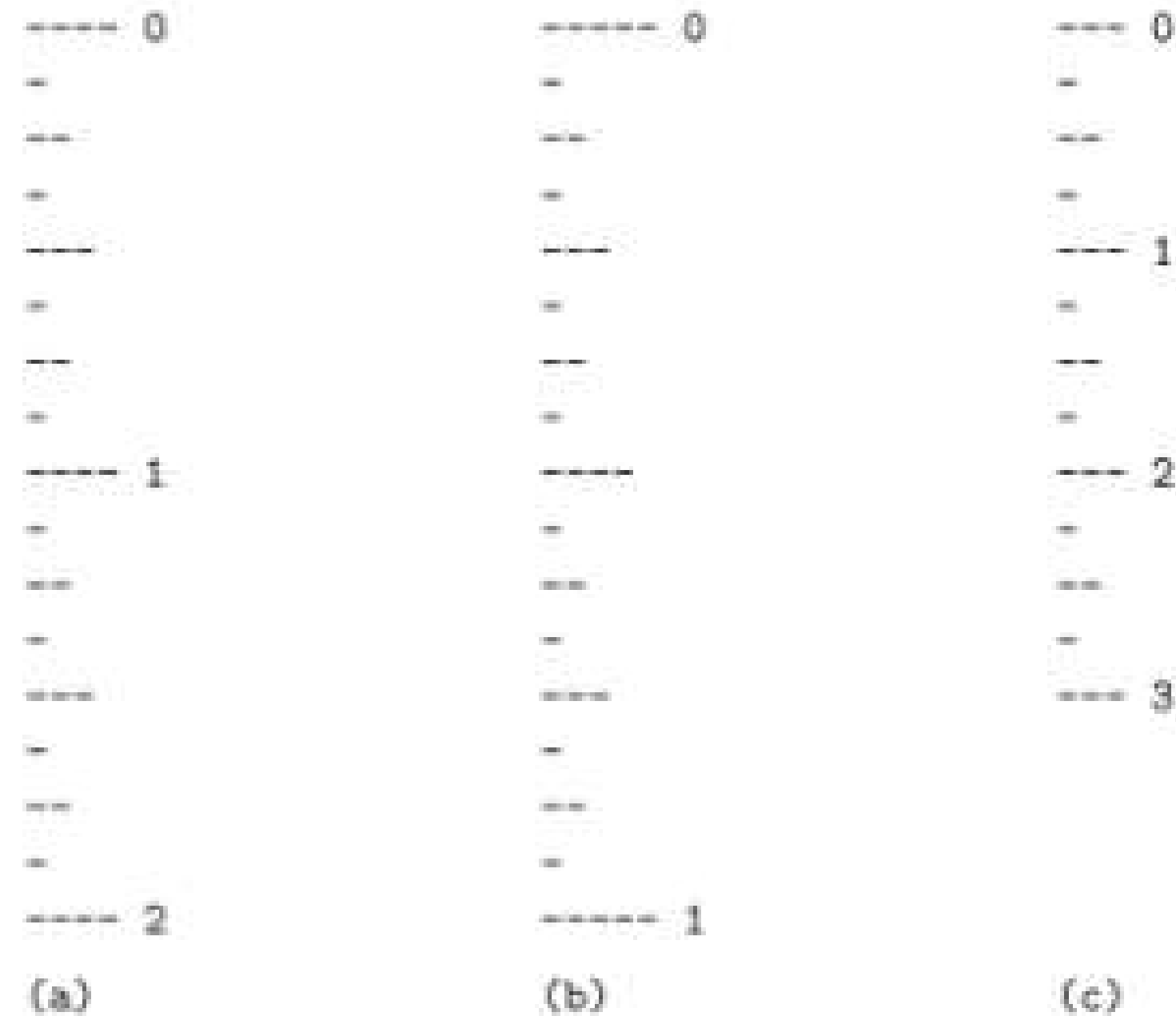


Figure 4.2: Three sample outputs of an English ruler drawing: (a) a 2-inch ruler with major tick length 4; (b) a 1-inch ruler with major tick length 5; (c) a 3-inch ruler with major tick length 3.

Pendekatan Rekursif untuk Menggambar Penggaris

Pola penggaris Inggris adalah contoh sederhana dari fraktal, yaitu bentuk yang memiliki struktur rekursif pada berbagai tingkat perbesaran. Pertimbangkan aturan dengan panjang detak utama 5. Secara umum, interval dengan panjang tik tengah $L \geq 1$ terdiri dari interval dengan panjang centang pusat $L-1$, sebuah centang tunggal dengan panjang L , interval dengan panjang centang pusat $L-1$

Implementasi terdiri dari tiga fungsi:

- draw_ruler (fungsi utama), mengelola konstruksi seluruh penggaris
- draw_line (fungsi utilitas), menggambar satu tanda centang
- draw_interval (fungsi rekursif), menggambar urutan tanda centang kecil dalam suatu interval


```

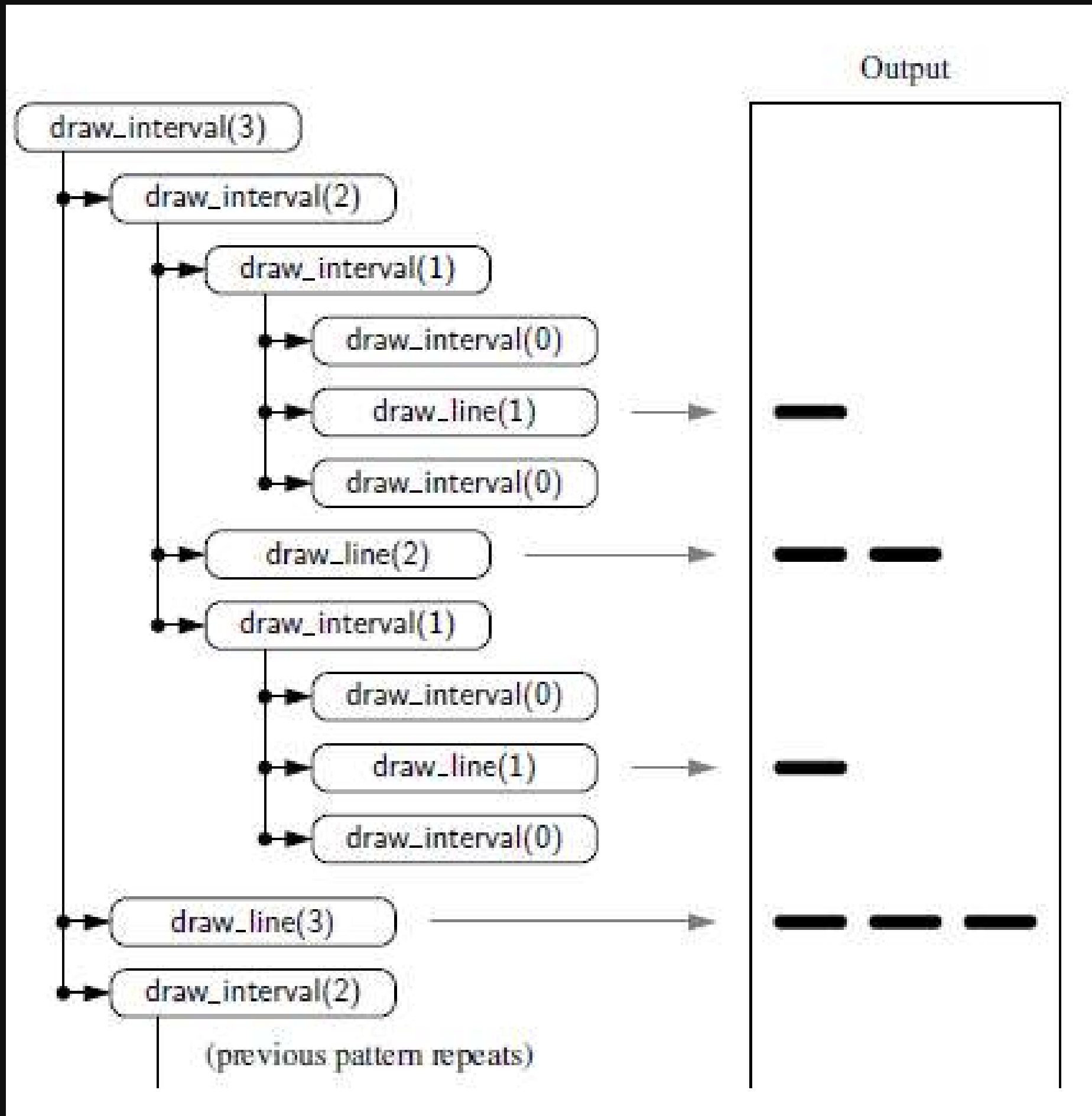
1 def draw_line(tick_length, tick_label=' '):
2     """Draw one line with given tick length (followed by optional label)."""
3     line = '-' * tick_length
4     if tick_label:
5         line += ' ' + tick_label
6     print(line)
7
8 def draw_interval(center_length):
9     """Draw tick interval based 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

```

Code Fragment 4.2: A recursive implementation of a function that draws a ruler.

- `draw_line` menggambar satu tanda centang dengan panjang `tick_length` dan label `tick_label`
- `draw_interval` menggambar urutan tanda centang minor dengan panjang tick tengah `center_length`
- `draw_ruler` menggambar penggaris dengan `num_inches` inci dan panjang tick mayor `major_length`

Mengilustrasikan Gambar Penggaris Menggunakan Jejak Rekursi



Gambar di samping menunjukkan jejak rekursi parsial untuk pemanggilan fungsi `draw_interval(3)`. Jejak ini lebih rumit dibandingkan contoh faktorial karena setiap pemanggilan menghasilkan dua pemanggilan rekursif. Setiap kotak mewakili satu pemanggilan fungsi `draw_interval`. Jejak rekursi membantu memahami eksekusi fungsi rekursif.

3. PENCARIAN BINER

Pencarian Biner adalah algoritma rekursif klasik yang digunakan untuk menemukan nilai target secara efisien dalam urutan terurut dari n elemen. Ini adalah salah satu algoritma komputer yang paling penting, dan ini adalah alasan mengapa kita sering menyimpan data dalam urutan terurut.

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

Figure 4.4: Values stored in sorted order within an indexable sequence, such as a Python list. The numbers at top are the indices.

Ketika data tidak diurutkan, algoritma pencarian sederhana adalah sequential search, di mana setiap elemen diurutkan satu per satu sampai target ditemukan. Ini memerlukan waktu $O(n)$ karena setiap elemen harus diperiksa.

Algoritma ini memanfaatkan dua parameter, low dan high, untuk menunjukkan batas interval pencarian saat ini. Jika target tidak ditemukan, pencarian berhenti ketika interval menjadi kosong. Namun, dalam kasus data yang terurut, pencarian biner dapat memberikan efisiensi $O(\log n)$ karena kemampuannya untuk membagi interval pencarian menjadi setengah setiap langkahnya.

Terdapat tiga kasus:

- Jika $\text{target} == \text{data}[\text{mid}]$, maka kita telah menemukan item yang dicari, dan pencarian berakhir dengan sukses.
- Jika $\text{target} < \text{data}[\text{mid}]$, maka kita mengulang pada paruh pertama dari urutan, yaitu pada interval indeks low ke $\text{mid} - 1$.
- Jika $\text{target} > \text{data}[\text{mid}]$, maka kita mengulang pada paruh kedua dari urutan, yaitu pada interval indeks dari $\text{mid} + 1$ ke high

Pencarian gagal jika $\text{low} > \text{high}$, karena interval $[\text{low}, \text{high}]$ kosong.

Berikut implementasi Python pada Code Fragment 4.3, dan ilustrasi eksekusi algoritma pada gambar 4.5. Ketika pencarian berurutan berjalan dalam waktu $O(n)$, pencarian biner yang lebih efisien berjalan dalam waktu $O(\log n)$.

```
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]:
11            return True                # found a match
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)
```

Code Fragment 4.3: An implementation of the binary search algorithm.

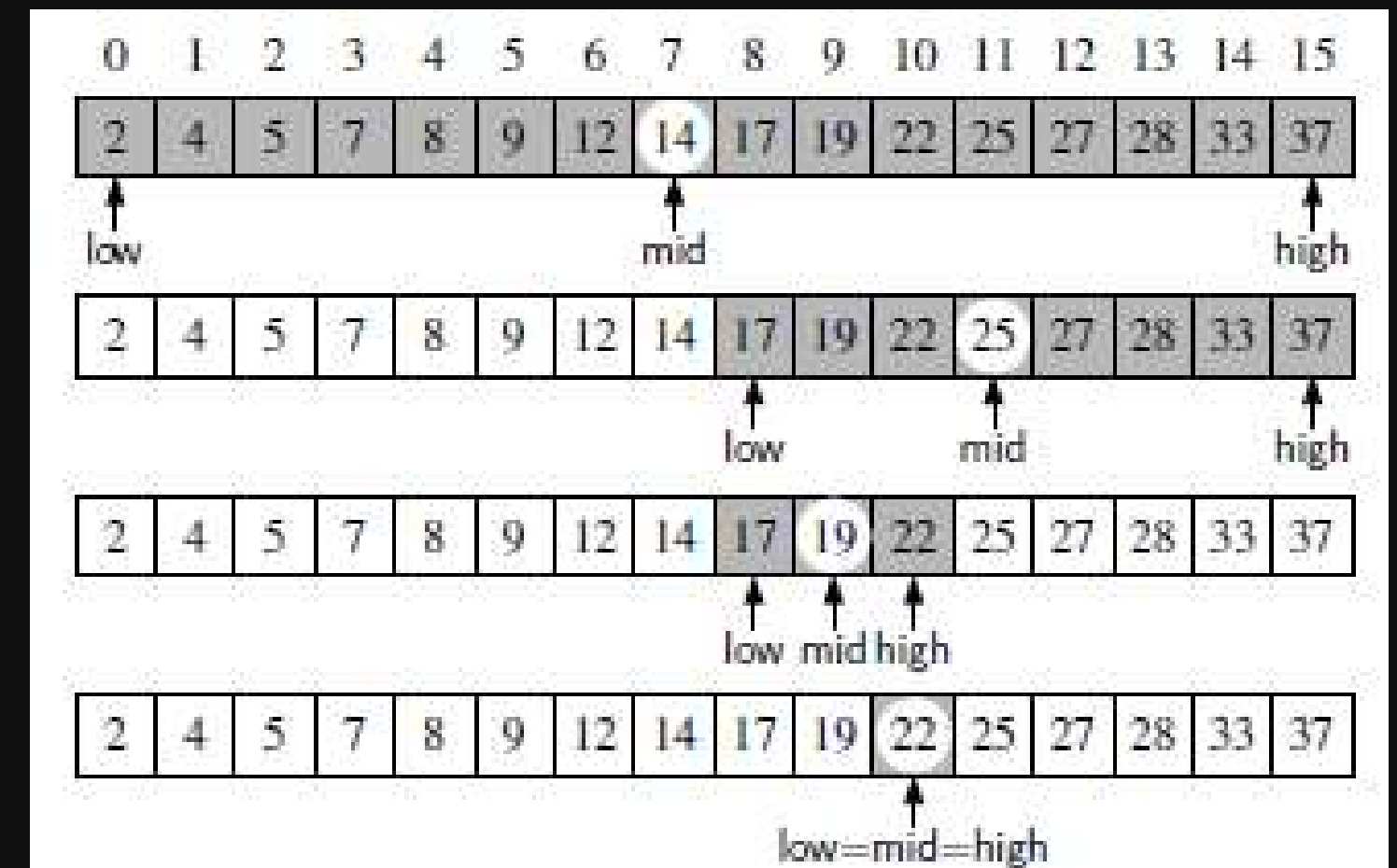
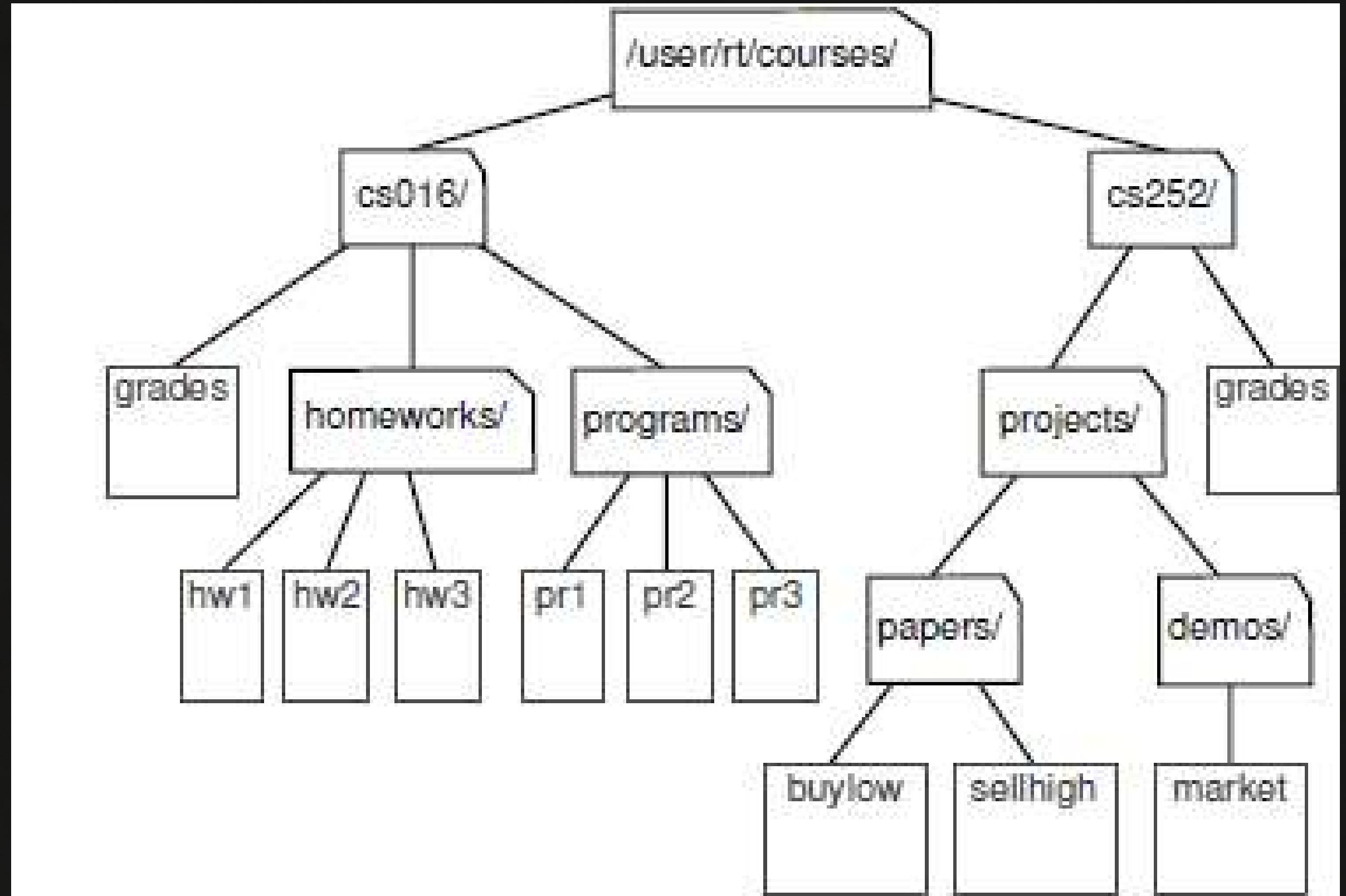


Figure 4.5: Example of a binary search for target value 22.

3. PENCARIAN BINER

Sistem operasi modern mendefinisikan file-system direktori (folder) dengan cara rekursif. Sistem operasi memungkinkan direktori untuk dibuat dengan susunan yang tidak terbatas (selama ada cukup ruang dalam memori), meskipun harus ada beberapa direktori dasar yang hanya berisi file.



Pada file-system. kita dapat menyalin dan menghapus direktori, ini diimplementasikan dengan algoritma rekursif. Gambar tersebut menunjukkan diagram sistem file dengan anotasi tambahan untuk menjelaskan jumlah ruang disk yang digunakan. Ada dua jenis penggunaan ruang disk:

- Ruang disk langsung: Ruang disk yang digunakan oleh file atau direktori itu sendiri.
- Ruang disk kumulatif: Ruang disk yang digunakan oleh file atau direktori dan semua file dan direktori di dalamnya.

Ruang disk kumulatif untuk sebuah entri dapat dihitung dengan algoritma rekursif sederhana. Ini sama dengan ruang disk langsung yang digunakan oleh entri tersebut ditambah jumlah penggunaan ruang disk kumulatif dari semua entri yang disimpan langsung di dalam entri tersebut. Contoh, ruang disk kumulatif untuk cs016 adalah 249K merupakan kumulatif dari 2K itu sendiri, 8K dalam grades, 10K dalam homeworks, dan 229K dalam programs. Pseudo-code untuk algoritma ini diberikan pada Code Fragment 4.4.

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

Code Fragment 4.4: An algorithm for computing the cumulative disk space usage nested at a file-system entry. Function size returns the immediate disk space of an entry.

- os.path.getsize(path)

Mengembalikan penggunaan disk langsung (diukur dalam byte) untuk file atau direktori yang diidentifikasi oleh jalur string (e.g., /user/rt/courses).

- os.path.isdir(path)

Kembalikan True jika entri yang ditunjuk oleh jalur string adalah sebuah direktori; False jika tidak.

- os.listdir(path)

Mengembalikan daftar string yang merupakan nama semua entri dalam direktori yang ditunjuk oleh jalur string. Dalam contoh sistem file kita, jika parameternya adalah /user/rt/courses, ini akan mengembalikan list [cs016 , cs252].

- os.path.join(path, filename)

Buat string jalur dan string nama file menggunakan pemisah sistem operasi yang sesuai di antara keduanya (misalnya, karakter / untuk sistem Unix/Linux, dan karakter \ untuk Windows). Kembalikan string yang mewakili jalur lengkap ke file.

Implementasi Pada Python

```
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
```

Code Fragment 4.5: A recursive function for reporting disk usage of a file system.

Jejak Rekursi

```
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.



MENGANALISIS

ALGORITMA

REKURSIF



MENGHITUNG FAKTORIAL

Analisis efisiensi fungsi untuk menghitung faktorial relatif mudah dilakukan. Untuk menghitung faktorial(n), terdapat total $n + 1$ aktivasi, karena parameter berkurang dari n pada panggilan pertama, menjadi $n - 1$ pada panggilan kedua, hingga mencapai kasus dasar dengan parameter 0. Dengan demikian, kompleksitas waktu algoritma ini adalah $O(n)$, yang menyiratkan bahwa jumlah operasi yang dilakukan meningkat secara linear dengan nilai n . Semakin besar nilai n , semakin banyak operasi yang diperlukan, namun setiap operasi tetap sederhana.

MENGGAMBAR PENGGARIS INGGRIS

Untuk menganalisis aplikasi penggaris bahasa Inggris, kita mempertimbangkan jumlah total garis keluaran dari pemanggilan awal ke `draw_interval(c)`, dimana c adalah panjang tengah. Setiap baris output didasarkan pada pemanggilan ke `draw_line`, dan setiap panggilan rekursif ke `draw_interval` dengan parameter yang bukan nol menghasilkan satu panggilan langsung ke `draw_line`. Kita menggunakan intuisi bahwa pemanggilan `draw_interval(c)` untuk $c > 0$ menghasilkan dua pemanggilan `draw_interval(c-1)` dan satu pemanggilan `draw_line`. Hal ini membentuk dasar untuk membuktikan klaim bahwa jumlah garis yang dihasilkan adalah $2^c - 1$.

Bukti dilakukan menggunakan induksi matematika. Kasus dasarnya adalah ketika $c = 0$, di mana tidak ada keluaran yang dihasilkan. Secara umum, jumlah garis yang dicetak oleh `draw_interval(c)` adalah $2^c - 1$, sesuai dengan klaim yang dibuktikan dengan menggunakan induksi. Dengan demikian, analisis ini menyediakan alat matematis yang ketat untuk memahami jumlah garis yang dihasilkan oleh algoritma tersebut, dan hal ini berguna dalam menganalisis kinerja algoritma rekursif secara keseluruhan.

MELAKUKAN **PENCARIAN** BINER

Mempertimbangkan waktu berjalan dari algoritma pencarian biner, kami mengamati bahwa sejumlah operasi primitif dieksekusi pada setiap pemanggilan rekursif metode pencarian biner. Oleh karena itu, waktu berjalan sebanding dengan jumlah pemanggilan rekursif yang dilakukan. Kami akan menunjukkan bahwa paling banyak $\lceil \log n + 1 \rceil$ panggilan rekursif dilakukan selama pencarian biner dari sebuah barisan yang memiliki n elemen, yang mengarah kepada klaim berikut ini.

Proposition 4.2: Algoritma pencarian biner berjalan dalam waktu $O(\log n)$ untuk urutan terurut dengan n elemen.

Justification: Untuk membuktikan klaim ini, sebuah fakta penting adalah bahwa pada setiap pemanggilan rekursif, jumlah entri kandidat yang masih harus dicari diberikan oleh nilai $\text{high} - \text{low} + 1$.

Selain itu, jumlah kandidat yang tersisa berkurang setidaknya setengahnya dengan setiap pemanggilan rekursif. Secara khusus, dari definisi pertengahan (mid), jumlah kandidat yang tersisa adalah

$$(mid - 1) - low + 1 = \left\lfloor \frac{low + high}{2} \right\rfloor - low \leq \frac{high - low + 1}{2}$$

or

$$high - (mid + 1) + 1 = high - \left\lfloor \frac{low + high}{2} \right\rfloor \leq \frac{high - low + 1}{2}$$

Pada awalnya, jumlah kandidat adalah n ; setelah pemanggilan pertama dalam pencarian biner, jumlah kandidat paling banyak adalah $n/2$; setelah pemanggilan kedua, jumlah kandidat paling banyak adalah $n/4$; dan seterusnya. Secara umum, setelah pemanggilan ke- j dalam pencarian biner, jumlah entri kandidat yang tersisa paling banyak adalah $n/2^j$. Pada kasus terburuk (pencarian yang gagal), pemanggilan rekursif akan berhenti ketika tidak ada lagi entri kandidat.

Dengan kata lain (mengingat bahwa kita menghilangkan basis logaritma jika basisnya adalah 2), $r > \log n$. Dengan demikian, kita memiliki $r = \lceil \log n + 1 \rceil$, yang mengimplikasikan bahwa pencarian biner berjalan dalam waktu $O(\log n)$.

MENGHITUNG **PENGGUNAAN** RUANG **DISK**

Algoritma penggunaan disk secara rekursif menghitung total penggunaan ruang disk untuk setiap entri dalam sistem file. Dengan mengasumsikan n sebagai jumlah entri dalam sistem file, setiap entri memicu satu pemanggilan rekursif. Meskipun jumlah panggilan rekursifnya dapat mencapai $O(n)$, namun dengan mengamati bahwa setiap entri hanya dihitung sekali, total operasi yang dilakukan tetap berbanding lurus dengan jumlah entri, yaitu $O(n)$. Analisis ini menggunakan teknik rekursi dan amortisasi untuk memahami struktur sistem file sebagai pohon, memberikan wawasan yang lebih dalam tentang kinerja algoritma penggunaan disk dalam konteks keseluruhan sistem file.



RECURSION

RUN AMOK



Pada bagian ini, akan meninjau “masalah keunikan elemen”. Kasus dasarnya adalah ketika $n = 1$, di mana elemen-elemennya pasti unik. Namun, untuk $n \geq 2$, elemen-elemennya unik jika dan hanya jika $n - 1$ elemen pertama unik, $n - 1$ item terakhir unik, dan elemen pertama dan terakhir berbeda. Implementasi rekursif dari konsep ini diberikan dalam Code Fragment 4.6, diberi nama `unique3` untuk membedakannya dari `unique1` dan `unique2`.

```
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.

Penggunaan rekursi dalam kasus ini sangat tidak efisien. Bagian nonrekursif dari setiap panggilan memiliki waktu $O(1)$, sehingga keseluruhan waktu berjalan sebanding dengan jumlah total pemanggilan rekursif. Jika $n = 1$, waktu berjalan dari `unique3` adalah $O(1)$, karena tidak ada panggilan rekursif. Namun, dalam kasus umum, satu panggilan ke `unique3` dengan masalah berukuran n dapat menghasilkan dua panggilan rekursif pada masalah berukuran $n - 1$, yang pada gilirannya dapat menghasilkan empat panggilan dengan rentang ukuran $n - 2$, dan seterusnya. Dalam kasus terburuk, jumlah total pemanggilan fungsi diberikan oleh penjumlahan geometri, yang setara dengan $2^n - 1$. Oleh karena itu, waktu berjalan dari `unique3` adalah $O(2^n)$, yang sangat tidak efisien untuk menyelesaikan masalah keunikan elemen. Ketidakefisienan ini tidak berasal dari penggunaan rekursi itu sendiri, tetapi dari cara rekursi tersebut diimplementasikan.

REKURSI YANG TIDAK EFISIEN UNTUK MENGHITUNG ANGKA FIBONACCI

Di Bagian 1.8, kami memperkenalkan proses untuk menghasilkan angka Fibonacci. Ironisnya, implementasi langsung berdasarkan definisi ini menghasilkan fungsi `bad_fibonacci` yang ditunjukkan pada Code Fragment 4.7.

```
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)
```

Code Fragment 4.7: Computing the n^{th} Fibonacci number using binary recursion.

Sayangnya, implementasi langsung rumus Fibonacci menghasilkan fungsi yang sangat tidak efisien. Menghitung bilangan Fibonacci ke- n dengan cara ini memerlukan jumlah pemanggilan fungsi yang eksponensial. Secara khusus, misalkan C_n menunjukkan jumlah panggilan yang dilakukan dalam eksekusi `bad_fibonacci(n)`. , maka kita dapat melihat bahwa jumlah panggilan meningkat lebih dari dua kali lipat untuk setiap dua indeks berturut-turut. Artinya, $C_n > 2^{(n/2)}$, yang berarti `bad_fibonacci(n)` melakukan sejumlah panggilan eksponensial dalam n .

REKURSI YANG EFISIEN EFISIEN UNTUK MENGHITUNG ANGKA FIBONACCI

Rekursi yang tidak efisien untuk menghitung angka Fibonacci menghasilkan waktu berjalan eksponensial karena setiap panggilan membuat dua panggilan rekursif. Ini disebabkan oleh duplikasi perhitungan nilai sebelumnya, yang menyebabkan pekerjaan yang tidak perlu dilakukan secara berulang. Kita dapat menghitung F_n dengan lebih efisien menggunakan rekursi di mana setiap pemanggilan hanya membuat satu panggilan rekursif. Untuk melakukan hal ini, kita perlu mendefinisikan ulang ekspektasi fungsi tersebut. Daripada membuat fungsi tersebut mengembalikan nilai tunggal, yaitu bilangan Fibonacci ke- n , kita mendefinisikan fungsi rekursif yang mengembalikan sepasang bilangan Fibonacci berurutan (F_n, F_{n-1}) , dengan menggunakan konvensi $F_{n-1} = 0$. Implementasi:

```
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)
```

Code Fragment 4.8: Computing the n^{th} Fibonacci number using linear recursion.

Fungsi `bad_fibonacci` menggunakan waktu eksponensial. Sementara eksekusi fungsi `good_fibonacci(n)` membutuhkan waktu $O(n)$. Setiap panggilan rekursif ke `good_fibonacci` mengurangi argumen n sebesar 1. Karena pekerjaan nonrekursif untuk setiap panggilan menggunakan waktu yang konstan, keseluruhan komputasi dijalankan dalam waktu $O(n)$.

KEDALAMAN REKURSIF MAKASIMUM

DENGAN PYTHON

Rekursi tak terbatas adalah bahaya dalam penggunaan rekursi di mana setiap panggilan rekursif membuat panggilan rekursif lainnya tanpa mencapai kasus dasar. Contoh sederhana seperti `fib(n)` yang selalu memanggil `fib(n)` dapat menyebabkan rekursi tak terbatas. Ini dapat menghabiskan sumber daya komputasi karena setiap panggilan rekursif menciptakan catatan aktivasi yang memerlukan memori tambahan.

Untuk mengatasi rekursi tak terbatas, Python membatasi jumlah maksimum aktivasi fungsi yang dapat aktif secara bersamaan. Nilai defaultnya adalah 1000. Jika batas ini tercapai, Python akan memunculkan `RuntimeError`. Namun, kita dapat mengubah batas ini menggunakan modul `sys`:

```
import sys
old = sys.getrecursionlimit( )      # perhaps 1000 is typical
sys.setrecursionlimit(1000000)      # change to allow 1 million nested calls
```




CONTOH REKURSI LEBIH LANJUT



REKURSI **LINEAR**

Rekursi linear adalah fungsi rekursif dirancang sedemikian rupa sehingga setiap pemanggilan isi menghasilkan paling banyak satu panggilan rekursif baru. . Dari rekursi yang telah kita lihat sejauh ini, penerapan fungsi faktorial (Bagian 4.1.1) dan fungsi good_fibonacci (Bagian 4.3) adalah contoh jelas dari rekursi linear. Konsekuensi dari definisi rekursi linear adalah bahwa setiap jejak rekursi akan muncul sebagai satu rangkaian panggilan

REKURSI **LINEAR**

Menjumlahkan Elemen Barisan Secara Rekursif:

Rekursi linear dapat menjadi alat yang berguna untuk memproses urutan data. Untuk menghitung jumlah suatu barisan S dari n bilangan bulat, kita dapat menggunakan pendekatan rekursi linear. Jumlah semua n bilangan bulat di S adalah 0 jika $n = 0$. Jika tidak, jumlah $n - 1$ bilangan bulat pertama di S ditambah dengan elemen terakhir di S .

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

Figure 4.9: Computing the sum of a sequence recursively, by adding the last number to the sum of the first $n - 1$.

REKURSI **LINEAR**

Algoritma `linear_sum` memiliki kompleksitas waktu $O(n)$ karena membuat $n + 1$ pemanggilan fungsi untuk input berukuran n . Hal ini karena setiap panggilan membutuhkan waktu yang konstan untuk bagian non rekursifnya. Selain itu, ruang memori yang digunakan oleh algoritma (selain urutan S) juga $O(n)$, karena setiap panggilan fungsi menambahkan $n + 1$ catatan aktivasi dalam jejak, dengan menggunakan jumlah ruang memori yang konstan.

```
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]
```

Code Fragment 4.9: Summing the elements of a sequence using linear recursion.

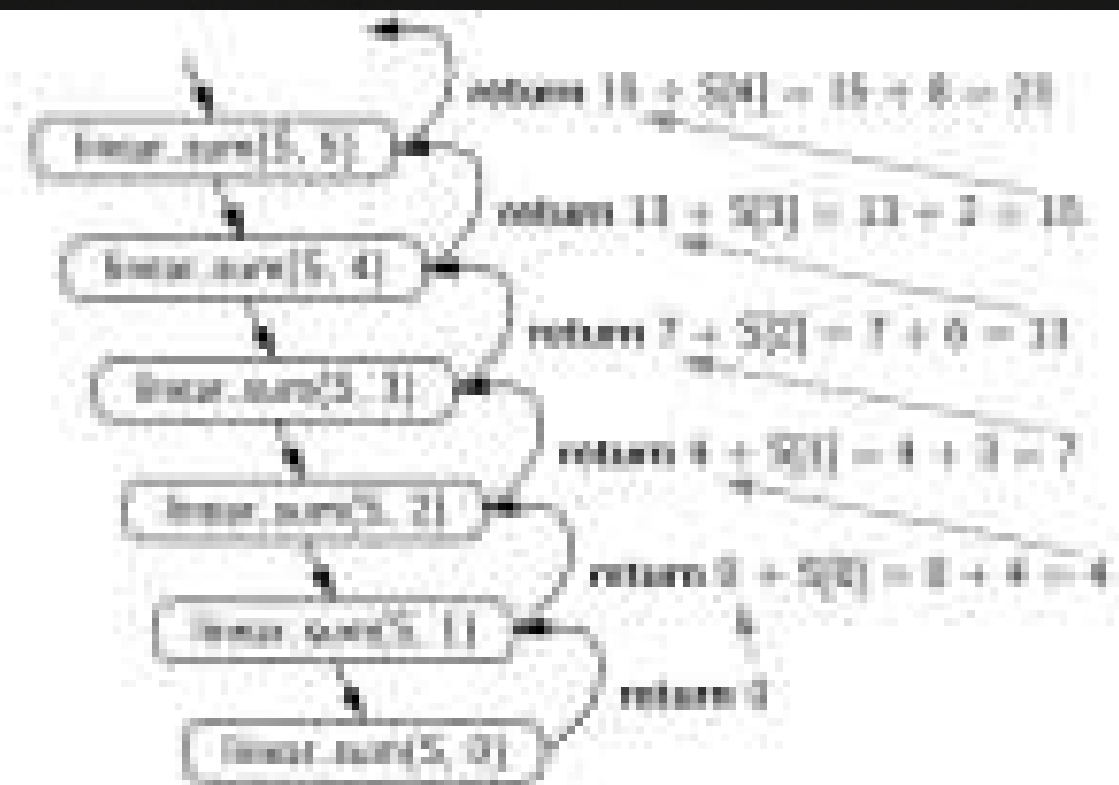


Figure 4.10: Recursion trace for an execution of `linear_sum(S, 5)` with input parameter $S = [4, 3, 8, 2, 8]$.

REKURSI **LINEAR**

Membalikkan Urutan dengan Rekursi Masalah pembalikan n elemen suatu barisan S dapat diselesaikan menggunakan rekursi linear. Ini dapat dicapai dengan menukar elemen pertama dan terakhir, lalu membalikkan elemen yang tersisa secara rekursif. Implementasi algoritma ini disajikan dalam Code Fragment 4.10, di mana panggilan pertama dilakukan sebagai kebalikan ($S, 0, \text{len}(S)$). Penjelasan langkah-langkah: Langkah pertama: Elemen pertama ($S[0]$) dan elemen terakhir ($S[n-1]$) ditukar. Langkah kedua: Algoritma dipanggil secara rekursif untuk membalikkan elemen yang tersisa ($S[1], S[2], \dots, S[n-2]$). Langkah ketiga: Setelah elemen yang tersisa dibalik, barisan S terbalik sepenuhnya.

```
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
```

Code Fragment 4.10: Reversing the elements of a sequence using linear recursion.

REKURSI LINEAR

Algoritma rekursif dalam Code Fragment 4.10 memiliki dua kasus dasar implisit:

- Saat `start == stop`, yang menandakan rentang kosong
- Saat `start == stop - 1`, yang menandakan rentang hanya memiliki satu elemen.

Argumen ini menunjukkan bahwa algoritma rekursif tersebut dijamin akan berakhir setelah total $1 + [n/2]$ panggilan rekursif. Karena setiap panggilan melibatkan jumlah pekerjaan yang konstan, keseluruhan proses berjalan dalam waktu $O(n)$

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

Figure 4.11: A trace of the recursion for reversing a sequence. The shaded portion has yet to be reversed.

REKURSI LINEAR

Algoritma Rekursif untuk Menghitung Kekuatan Sebagai contoh penggunaan rekursi linear, kita akan mempertimbangkan masalah menghitung pangkat suatu bilangan (x) ke suatu bilangan bulat non negatif, (n). Dalam kata lain, kita ingin menghitung fungsi pangkat, yang didefinisikan sebagai $\text{power}(x,n)=x^n$. Dalam pembahasan ini, kita akan meninjau dua formulasi rekursif yang berbeda untuk masalah ini yang menghasilkan algoritma dengan kinerja yang sangat berbeda. Untuk menghitung fungsi pangkat dengan menggunakan definisi alternatif yang menggunakan teknik kuadrat, perhatikan gambar di samping. Jika kita mengimplementasikan rekursi ini dengan membuat dua panggilan rekursif untuk menghitung $\text{power}(x, \lfloor n/2 \rfloor)$, jejak rekursi akan menunjukkan panggilan $O(n)$. Kita dapat melakukan operasi yang jauh lebih sedikit dengan menghitung $\text{power}(x, \lfloor n/2 \rfloor)$ sebagai hasil parsial, lalu mengalikannya dengan dirinya sendiri.

$$\text{power}(x,n) = \begin{cases} 1 & \text{if } n = 0 \\ x \cdot (\text{power}(x, \lfloor \frac{n}{2} \rfloor))^2 & \text{if } n > 0 \text{ is odd} \\ (\text{power}(x, \lfloor \frac{n}{2} \rfloor))^2 & \text{if } n > 0 \text{ is even} \end{cases}$$

```
1 def power(x, n):
2     """ Compute the value x**n fo
3     if n == 0:
4         return 1
5     else:
6         partial = power(x, n // 2)
7         result = partial * partial
8         if n % 2 == 1:
9             result *= x
10        return result
```


REKURSI LINEAR

Code Fragment 4.12: Computing the power function using repeated squaring.

To illustrate the execution of our improved algorithm, Figure 4.12 provides a recursion trace of the computation $\text{power}(2, 13)$.

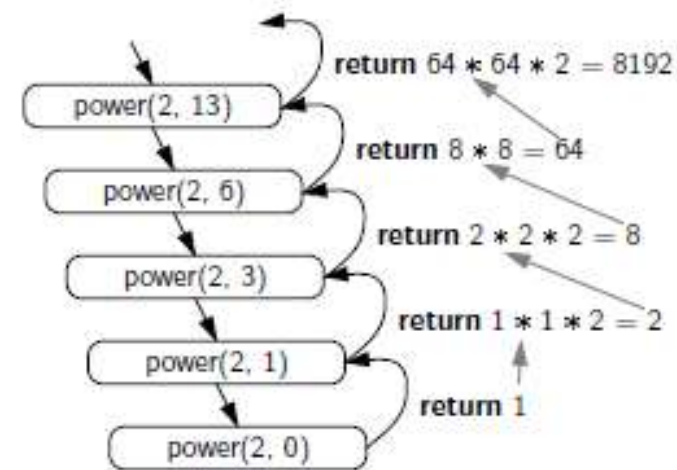


Figure 4.12: Recursion trace for an execution of $\text{power}(2, 13)$.

Penjelasan jejak rekursi:

- Baris pertama menunjukkan panggilan awal ke fungsi $\text{power}(2, 13)$.
- Fungsi power kemudian memanggil dirinya sendiri secara rekursif dengan argumen $(2, 6)$.
- Panggilan rekursif ini kemudian memanggil $\text{power}(2, 3)$.
- Proses ini berlanjut sampai mencapai kasus dasar $\text{power}(2, 0)$, yang mengembalikan nilai 1.
- Nilai 1 kemudian dikembalikan ke atas melalui level rekursi, dikalikan dengan 2 di setiap level.
- Hasil akhirnya adalah $\text{power}(2, 13) = 2^{13} = 8192$.

Gambar jejak rekursi ini membantu kita memahami cara kerja fungsi rekursif power . Kita dapat melihat

bagaimana nilai-nilai dihitung dan bagaimana panggilan rekursif dihubungkan satu sama lain.

REKURSI BINER

Rekursi biner adalah suatu fungsi yang membuat dua panggilan rekursif. Menghitung jumlah satu atau nol elemen adalah hal yang sepele. Dengan dua elemen atau lebih, kita dapat menghitung jumlah paruh pertama dan jumlah paruh kedua secara rekursif, lalu menjumlahkan jumlah tersebut. Implementasinya adalah sebagai berikut:

```
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)
```

Code Fragment 4.13: Summing the elements of a sequence using binary recursion.

REKURSI BINER

Analisis algoritma `binary_sum` dilakukan dengan mempertimbangkan kasus di mana n adalah pangkat dua, untuk kesederhanaan. Jejak rekursi dari eksekusi `binary_sum(0, 8)` ditunjukkan dalam Figure 4.13, dengan setiap kotak diberi label nilai parameter `start:stop` untuk panggilan tersebut. Karena rentang dibagi dua pada setiap panggilan rekursif, kedalaman rekursi adalah $1 + \log_2 n$. Oleh karena itu, `binary_sum` menggunakan ruang tambahan sebesar $O(\log n)$, yang merupakan peningkatan besar dibandingkan dengan ruang $O(n)$ yang digunakan oleh fungsi `linear_sum` pada Code Fragment 4.9. Namun, waktu berjalan dari `binary_sum` adalah $O(n)$, karena terdapat $2n - 1$ pemanggilan fungsi, masing-masing memerlukan waktu yang konstan.

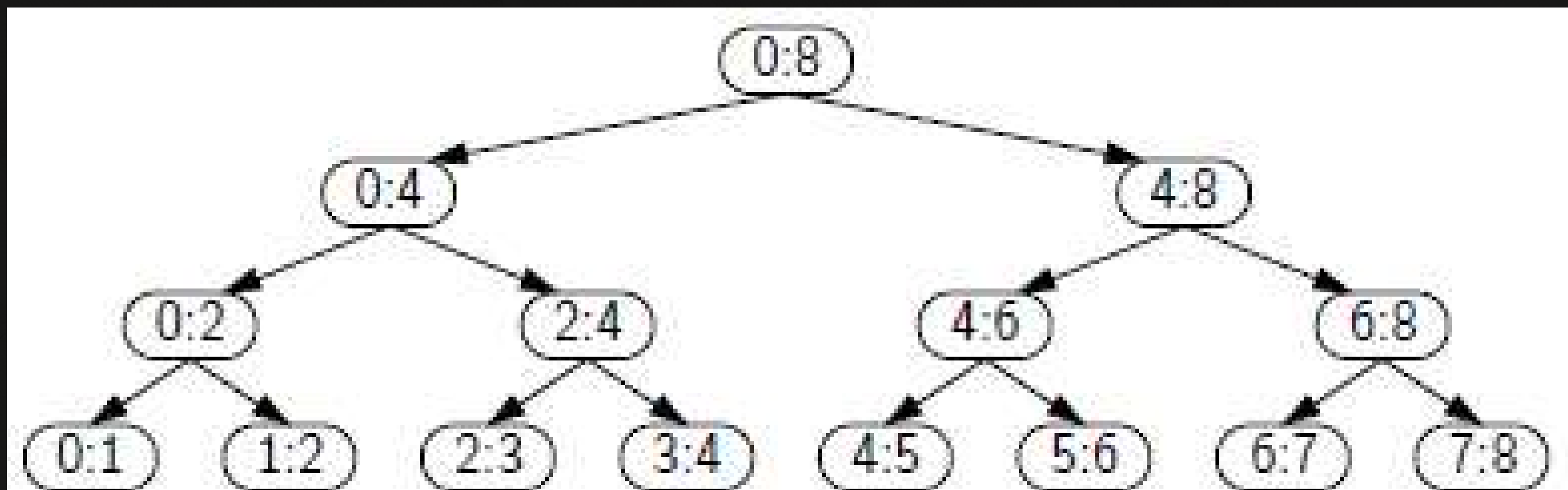
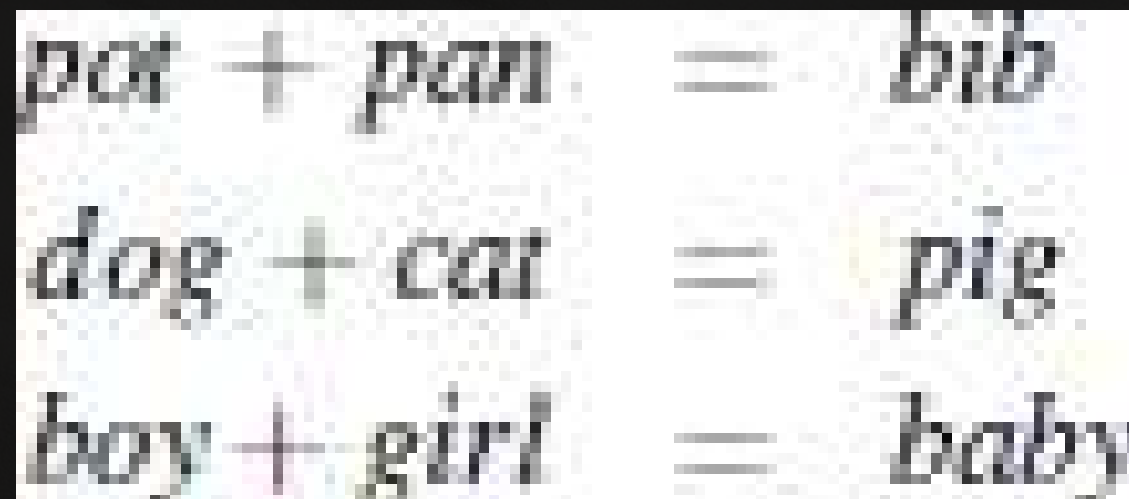


Figure 4.13: Recursion trace for the execution of `binary_sum(0, 8)`.

REKURSI BERGANDA

Rekursi berganda adalah suatu proses di mana suatu fungsi dapat membuat lebih dari dua panggilan rekursif. Contohnya yaitu tekateki penjumlahan.



pot + pan = bib
dog + cat = pig
boy + girl = baby

Untuk memecahkan teka-teki penjumlahan, kita perlu memberikan angka unik (0-9) pada setiap huruf dalam persamaan sehingga persamaan tersebut benar. Jika jumlah kemungkinan konfigurasi tidak terlalu besar, kita dapat menggunakan komputer untuk menghitung semua kemungkinan dan menguji masing-masing kemungkinan, tanpa menggunakan pengamatan manusia. Dalam hal ini, setiap huruf dalam persamaan sesuai dengan posisi tertentu dalam barisan angka $U = \{0,1,2,3,4,5,6,7,8,9\}$. Misalnya, posisi pertama melambangkan "b" , posisi kedua melambangkan "o" , posisi ketiga melambangkan "y" , dan seterusnya.

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}

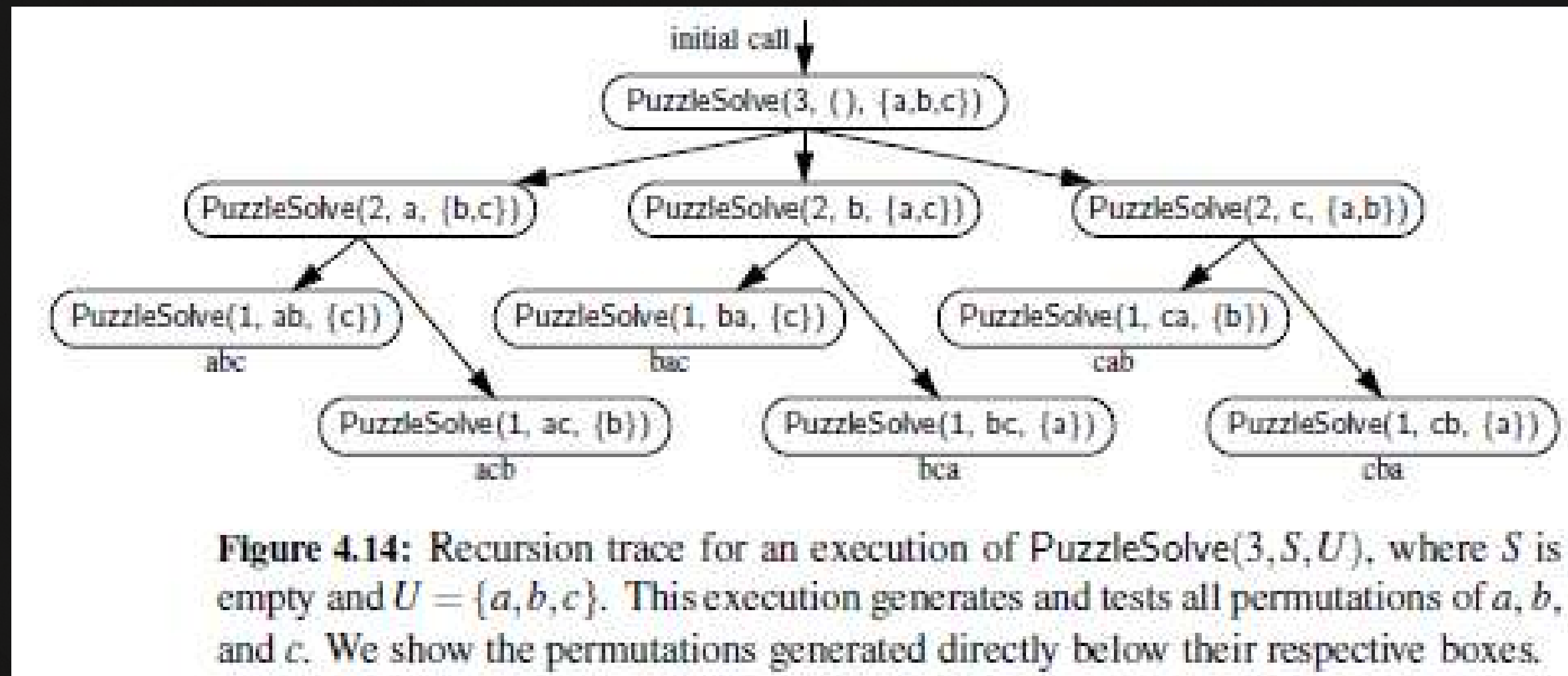
 Remove e from the end of S

 Add e back to U { e is now considered as unused}

Code Fragment 4.14: Solving a combinatorial puzzle by enumerating and testing all possible configurations.

REKURSI BERGANDA

Pada Figure 4.14, ditunjukkan jejak rekursi dari panggilan ke `PuzzleSolve(3, S, U)`, di mana S kosong dan $U = \{a,b,c\}$. Selama eksekusi, semua permutasi dari ketiga karakter dihasilkan dan diuji. Panggilan awal membuat tiga panggilan rekursif, yang masing-masing membuat dua panggilan lagi. Jika kita menjalankan `PuzzleSolve(3, S, U)` pada himpunan U yang terdiri dari empat elemen, panggilan awal akan menghasilkan empat panggilan rekursif, yang masing-masing akan memiliki jejak seperti pada Figure 4.14.





MENDESAIN

ALGORITMA

REKURSIF



Secara umum algoritma yang menggunakan rekursi biasanya mempunyai bentuk sebagai berikut:

- **Test for base cases.** Kita mulai dengan menguji serangkaian kasus dasar (setidaknya harus ada satu). Kasus-kasus dasar ini harus didefinisikan sedemikian rupa sehingga setiap kemungkinan rangkaian panggilan rekursif pada akhirnya akan mencapai kasus dasar, dan penanganan setiap kasus dasar tidak boleh menggunakan rekursi.
- **Recur.** Jika bukan kasus dasar, kami melakukan satu atau lebih panggilan rekursif. Langkah rekursif ini mungkin melibatkan pengujian yang memutuskan beberapa kemungkinan panggilan rekursif yang akan dilakukan. Kita harus mendefinisikan setiap kemungkinan panggilan rekursif sehingga membuat kemajuan menuju kasus dasar.



PARAMETERISASI REKURSI

Parameterisasi rekursi adalah teknik merancang algoritma rekursif dengan mengubah parameter fungsi untuk memfasilitasi sub-masalah serupa. Perubahan parameterisasi ini sangat penting untuk pencarian biner. Jika kita bersikeras menggunakan tanda tangan yang lebih bersih, satu-satunya cara untuk menjalankan pencarian pada separuh daftar adalah dengan membuat instance daftar baru. Jika kita ingin menyediakan antarmuka publik yang lebih bersih, teknik standarnya adalah membuat satu fungsi untuk penggunaan publik dengan antarmuka yang lebih bersih, dan kemudian tubuhnya memanggil fungsi utilitas

non-publik yang memiliki parameter rekursif yang diinginkan.

- Pencarian biner:
- Tanda tangan alami: `binary_search(data, target)`
- Tanda tangan rekursif: `binary_search(data, target, low, high)`
- Parameter tambahan: `low` dan `high` untuk membatasi sub-list



MENGELIMINASI

REKURSI

EKOR



Rekursi ekor adalah bentuk rekursi jika panggilan rekursif apa pun yang dibuat dari satu konteks adalah operasi terakhir dalam konteks tersebut, dengan nilai kembalian dari panggilan rekursif (jika ada) segera dikembalikan oleh rekursi yang melingkupinya. Jika diperlukan, rekursi ekor harus berupa rekursi linier (karena tidak ada cara untuk melakukan panggilan rekursif kedua jika Anda harus segera mengembalikan hasil yang pertama).

Setiap rekursi ekor dapat diimplementasikan kembali secara non rekursif dengan menyertakan isi dalam satu lingkaran untuk pengulangan, dan mengganti panggilan rekursif dengan parameter baru dengan menetapkan ulang parameter yang ada ke nilai tersebut. Sebagai contoh nyata, fungsi `binary_search` kita dapat diimplementasikan kembali seperti yang ditunjukkan pada Code Fragment 4.15.

4.6. Eliminating Tail Recursion

179

```
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
```

Code Fragment 4.15: A nonrecursive implementation of binary search.

Dalam versi asli algoritma binary search, kita membuat panggilan rekursif `binary_search(data, target, low, mid-1)`. Dalam versi yang direvisi, kita hanya perlu mengganti `high = mid - 1` dan melanjutkan ke iterasi berikutnya dari loop. Kondisi kasus dasar asli `low > high` digantikan oleh kondisi loop `while low <= high`. Kita juga dapat mengembangkan implementasi nonrekursif (Code Fragment 4.16) dari metode kebalikan rekursif asli dari Code Fragment 4.10.

```
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
```


Code Fragment 4.16: Reversing the elements of a sequence using iteration.

Dalam versi baru ini, kami memperbarui nilai `start` dan `stop` selama setiap lintasan perulangan, keluar setelah kami mencapai kasus memiliki satu atau kurang elemen dalam rentang tersebut



LATIHAN REKURSI







C-4.9 Write a short recursive Python function that finds the minimum and maximum values in a sequence without using any loops.

```
1 def min_max(arr, min_val=None, max_val=None):
2     if len(arr) == 0:
3         return min_val, max_val
4     elif len(arr) == 1:
5         if min_val is None or arr[0] < min_val:
6             min_val = arr[0]
7         if max_val is None or arr[0] > max_val:
8             max_val = arr[0]
9         return min_val, max_val
10    else:
11        if min_val is None or arr[0] < min_val:
12            min_val = arr[0]
13        if max_val is None or arr[0] > max_val:
14            max_val = arr[0]
15        return min_max(arr[1:], min_val, max_val)
16
17 while True:
18     try:
19         angka = [int(a) for a in input("Masukkan angka: ").split(",")]
20         mnv, mxv = min_max(angka)
21         print('list input: ', angka, '\nnilai maksimum pada list: ', mxv, '\nnilai minimum pada list: ', mnv)
22         break
23     except Exception as e:
24         print("Input tidak valid! Silakan masukkan input yang valid.", e)
```

✓ 5.8s

```
list input: [1, 3, 4, 5, 6, 3, 7, 8, 9, 2]
nilai maksimum pada list: 9
nilai minimum pada list: 1
```






C-4.12 Give a recursive algorithm to compute the product of two positive integers, m and n , using only addition and subtraction.

```
1 ✓ def product(a, b):
2 ✓     if b == 1:
3         return a
4 ✓     else:
5         return a + product(a, b-1)
6
7 ✓ while True:
8 ✓     try:
9         m = int(input("Masukkan nilai m: "))
10        n = int(input("Masukkan nilai n: "))
11 ✓        if m <= 0 or n <= 0:
12            raise ValueError("Input harus bilangan bulat positif")
13        values = product(m, n)
14        print("m = ", m, '\nn = ', n, '\nmaka m*n = ', values)
15        break
16 ✓    except Exception as e:
17        print("Input tidak valid! Silakan masukkan input yang valid.", e)
```

m = 1
n = 2
maka m*n = 2





C-4.16 Write a short recursive Python function that takes a character string *s* and outputs its reverse. For example, the reverse of 'pots&pans' would be 'snap&stop'.

C-4.17 Write a short recursive Python function that determines if a string *s* is a palindrome, that is, it is equal to its reverse. For example, 'racecar' and 'gohangasalamiimalasagnahog' are palindromes.

C-4.16

```
1 def kebalik(s):
2     if len(s) <= 1:
3         return s
4     else:
5         return s[-1] + kebalik(s[:-1])
6
7 a = 'Pos Ruyas'
8 print(kebalik(a))
```

sayuR soP

C-4.17

```
1 def palindrom(s):
2     if len(s) <= 1:
3         return True
4     elif s[0] == s[-1]:
5         return palindrom(s[1:-1])
6     else:
7         return False
8
9 b = 'kasur rusak'
10 palindrom(b)
```

True





Link Google Colab

T4 ALGORITMA&STRUDAT Dani
Hidayat 11220940000014.ipynb





TERIMA

KASIH

