

Chapter 4 - Writing Structured Programs

Jianzhang Zhang

Alibaba Business School
Hangzhou Normal University

April 14, 2022

1 Algorithm Design

- Divide-and-Conquer
- Recursion
- Space-Time Tradeoffs
- Dynamic Programming

Please refer to chapter 4 of *Natural Language Processing with Python* and related lectures in [my programming basics course](#) for the following topics:

- Back to the Basics
- Sequences
- Questions of Style
- Functions: The Foundation of Structured Programming
- Doing More with Functions
- Program Development

Table of Contents

1 Algorithm Design

- Divide-and-Conquer
- Recursion
- Space-Time Tradeoffs
- Dynamic Programming

Table of Contents

1 Algorithm Design

- Divide-and-Conquer
- Recursion
- Space-Time Tradeoffs
- Dynamic Programming

Divide-and-Conquer

The basic idea is **dividing** a problem of size n into two problems of size $n/2$, **solving** these problems, and **combining** their results into a solution of the original problem.

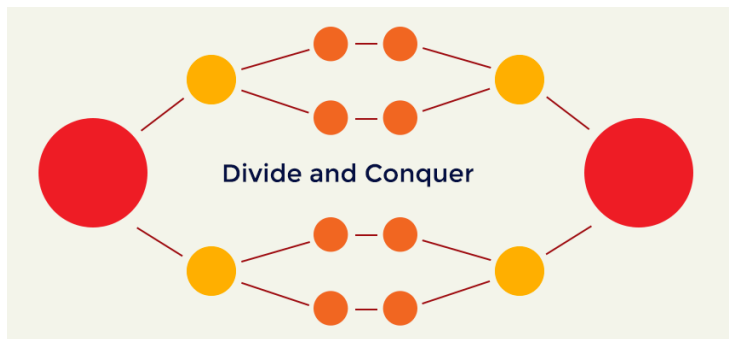


Figure 1: Divide-and-Conquer Paradigm

Merge sort algorithm a typical application of divide and conquer paradigm.

Merge Sort

Divide: 自顶向下，递归地二等分列表，直到不可分为止，即每个子列表只包含一个元素。

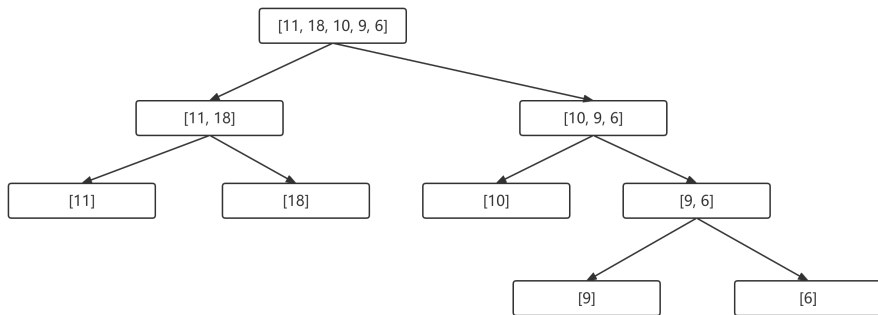


Figure 2: Dividing

```
def merge_sort(arr):  
    # dividing  
    if len(arr) == 1:  
        return arr  
    else:  
        length = len(arr)  
        left_arr = arr[:length//2]  
        right_arr = arr[length//2:]  
        print('split', arr, '--->', left_arr, right_arr)  
        # sorting and merging  
        return sort_list(merge_sort(left_arr), merge_sort(right_arr))
```

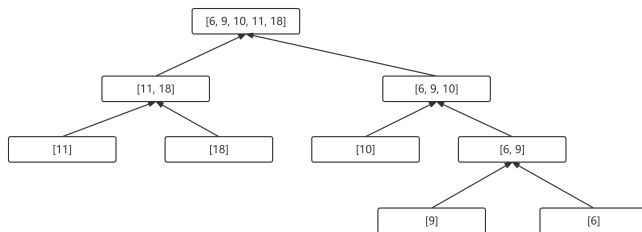



Figure 3: Sorting & Merging

- ❶ 对比第一个元素， $11 < 6$ ，6 放入结果列表中，结果列表为 [6]；
- ❷ 两个子列表变为 [11, 18] 和 [9, 10]；
- ❸ 对比第一个元素 $11 > 9$ ，9 放入结果列表中，结果列表为 [6, 9]；
- ❹ 两个子列表变为 [11, 18] 和 [10]；
- ❺ 对比第一个元素 $11 > 10$ ，10 放入结果列表中，结果列表为 [6, 9, 10]；
- ❻ 两个子列表变为 [11, 18] 和 []；
- ❼ 第一个子列表有序，第二个子列表为空，将第一个子列表加入结果列表，结果列表为 [6, 9, 10, 11, 18]。

已排序子列表 [11, 18] 和 [6, 9, 10] 的排序合并过程如前所述，实现代码如下：

```
def sort_list(left, right):  
    l, r = deepcopy(left), deepcopy(right)  
    result = []  
    while len(left) > 0 and len(right) > 0:  
        if left[0] < right[0]:  
            result.append(left.pop(0))  
        else:  
            result.append(right.pop(0))  
    result += left  
    result += right  
    print('merge', l, r, '--->', result)  
    return result
```

```
merge_sort([1,4,2])
```

```
'''
```

```
split [1, 4, 2] ---> [1] [4, 2]
```

```
split [4, 2] ---> [4] [2]
```

```
merge [4] [2] ---> [2, 4]
```

```
merge [1] [2, 4] ---> [1, 2, 4]
```

```
'''
```

```
merge_sort([11, 18, 10, 9, 6])
```

```
'''
```

```
split [11, 18, 10, 9, 6] ---> [11, 18] [10, 9, 6]
```

```
split [11, 18] ---> [11] [18]
```

```
merge [11] [18] ---> [11, 18]
```

```
split [10, 9, 6] ---> [10] [9, 6]
```

```
split [9, 6] ---> [9] [6]
```

```
merge [9] [6] ---> [6, 9]
```

```
merge [10] [6, 9] ---> [6, 9, 10]
```

```
merge [11, 18] [6, 9, 10] ---> [6, 9, 10, 11, 18]
```

```
'''
```

Table of Contents

1 Algorithm Design

- Divide-and-Conquer
- Recursion
- Space-Time Tradeoffs
- Dynamic Programming

Recursion

Example: count the size of a (sub)tree rooted at a given node.

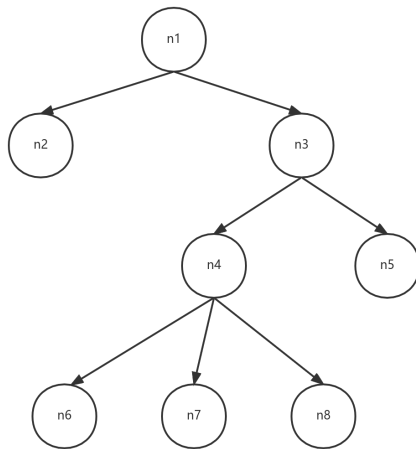


Figure 4: The size of the tree rooted at n1 is 8

采用字典结构存储上图中的树

```
n1 = {  
    'n2': {},  
    'n3': {  
        'n4': {  
            'n6': {},  
            'n7': {},  
            'n8': {}  
        },  
        'n5': {}  
    }  
}
```

两种基本情况：

1. 叶子节点（没有子节点），如，`'n5': {}`
2. 非叶子节点（有子节点），如，`'n4': {'n6': {}, 'n7': {}, 'n8': {}}`

```
# 递归计数
def tree_nodes_count_recursive(tree_dict):
    return 1 + sum(tree_nodes_count(v) for v in tree_dict.values())

# 逐层遍历计数
def tree_nodes_count_loop(tree_dict):
    trees = [tree_dict]
    total = 0
    while trees:
        total += len(trees)
        trees = [value for tree in trees for value in tree.values()]
    return total

nn = {'n1':{'n2':{'n3':{}}}}
tree_nodes_count_recursive(nn) # 4
tree_nodes_count_loop(nn) # 4
tree_nodes_count_recursive(n1) # 8
tree_nodes_count_loop(n1) # 8
```

将上述代码中的 `values()` 替换为 `hyponyms()` 即可用来计算WordNet中以同义词集 s 为根节点的子树节点数。

```
def recursive_size(s):  
    return 1 + sum(size1(child) for child in s.hyponyms())  
  
def loop_size(s):  
    layer = [s] # The first layer is the synset itself  
    total = 0  
    # it computes the next layer by finding the hyponyms of  
    ↪ everything in the last layer  
    while layer:  
        total += len(layer)  
        layer = [h for c in layer for h in c.hyponyms()]  
    return total  
  
from nltk.corpus import wordnet as wn  
dog = wn.synset('dog.n.01')  
recursive_size(dog), loop_size(dog) # 190
```


Example: a **letter trie** is a data structure that can be used for indexing a lexicon.

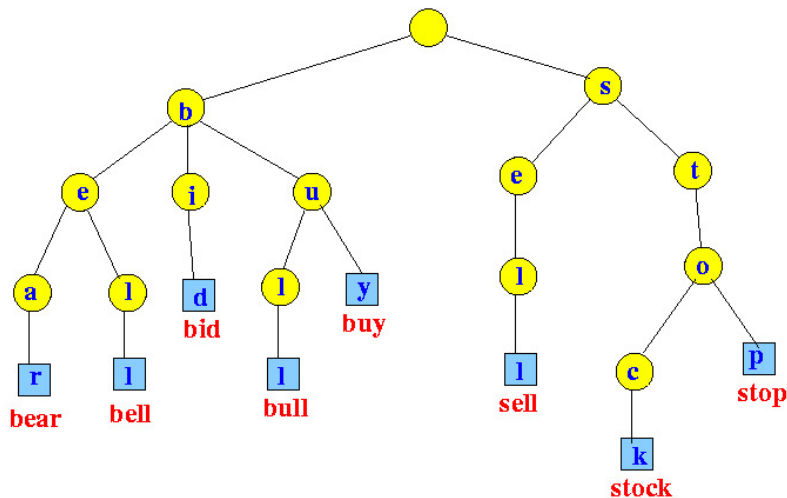


Figure 5: Letter Trie Example

Recursively build a letter trie with Python dictionaries.

```
def insert(trie, key, value):
    if key:
        first, rest = key[0], key[1:]
        if first not in trie:
            trie[first] = {}
        insert(trie[first], rest, value)
    else:
        trie['value'] = value

trie = {}
insert(trie, 'bear', '熊')
insert(trie, 'bell', '钟')
insert(trie, 'bid', '出价')
insert(trie, 'bull', '公牛')
insert(trie, 'buy', '买')
import pprint
pprint.pprint(trie, width=40)
```

Table of Contents

1 Algorithm Design

- Divide-and-Conquer
- Recursion
- Space-Time Tradeoffs
- Dynamic Programming

Space-Time Tradeoffs

We can sometimes significantly speed up the execution of a program by **building an auxiliary data structure**, such as an index.

Example: implements **a simple text retrieval system** for the Movie Reviews Corpus by indexing the document collection.

```
import re

def raw(file):
    contents = open(file).read()
    contents = re.sub(r'<.*?>', ' ', contents)
    contents = re.sub('\s+', ' ', contents) # 将空白字符替换为空格
    return contents

def snippet(doc, term):
    text = ' '*30 + raw(doc) + ' '*30
    pos = text.index(term)
    return text[pos-30:pos+30] # 上下文窗口为左右30个字符
```

```
import nltk

# 对movie reviews构建索引
files = nltk.corpus.movie_reviews.abspaths()
print("Building Index...")
idx = nltk.Index((w, f) for f in files for w in raw(f).split())

# 使用构建的索引进行查询
query = ''
while query != "quit":
    query = input("query> ")
    if query in idx:
        for doc in idx[query]:
            print(snippet(doc, query))
    else:
        print("Not found")
```

Example: replace the tokens of a corpus with integer identifiers.

- 1 Create a vocabulary `dict` for the corpus, in which each word is stored once;
- 2 Each document is preprocessed to become a list of integers;
- 3 Any language models can now work with integers.

```
def preprocess(tagged_corpus):  
    # 使用集合存储词表和标签表  
    words = set()  
    tags = set()  
    for sent in tagged_corpus:  
        for word, tag in sent:  
            words.add(word)  
            tags.add(tag)  
    word2idx = dict((w, i) for (i, w) in enumerate(words))  
    tag2idx = dict((t, i) for (i, t) in enumerate(tags))  
    return word2idx, tag2idx
```

```
from nltk.corpus import brown

tagged_brown = brown.tagged_sents()
word2idx, tag2idx = preprocess(tagged_brown)
word_list = [(word2idx[w] for (w, _) in sent) for sent in
    ↪ tagged_brown[:20]]
pos_list = [(tag2idx[t] for (_, t) in sent) for sent in
    ↪ tagged_brown[:20]]
# [20312, 40664, 53115, 43127, ... ]
print(word_list[3])
# [99, 188, 353, 62, 218, ... ]
print(pos_list[3])
```

If you need to process an input text to **check that all words are in an existing vocabulary**, the vocabulary should be stored as **a set, not a list**.

The elements of a set are **automatically indexed**, so testing membership of a large set will be **much faster** than testing membership of the corresponding list.

Table of Contents

1 Algorithm Design

- Divide-and-Conquer
- Recursion
- Space-Time Tradeoffs
- Dynamic Programming

Basic idea: Dynamic programming is a general technique for designing algorithms and used when a problem contains **overlapping sub-problems**. Instead of computing solutions to these sub-problems repeatedly, we simply **store them in a lookup table**.

Example: 在梵文中，短音节 S 的占一个长度单位，长音节 L 占两个长度单位，找出所有可能的长短音节组合方式，使得组合之后的结果长度为 n 。例如 $V_4 = \{LL, SSL, SLS, LSS, SSSS\}$ ， V_4 可以进一步划分为两个子集合：

$$V_4 =$$

LL, LSS

i.e. L prefixed to each item of $V_2 = L, SS$

$SSL, SLS, SSSS$

i.e. S prefixed to each item of $V_3 = SL, LS, SSS$

下面提供四种实现方式：

```
# 1. 递归实现
def virahanka1(n):
    # 第一种基本情况
    if n == 0:
        return [""]
    # 第二种基本情况
    elif n == 1:
        return ["S"]
    else:
        s = ["S" + prosody for prosody in virahanka1(n-1)]
        l = ["L" + prosody for prosody in virahanka1(n-2)]
        return s + l

virahanka1(4)
# ['SSSSS', 'SSSL', 'SSLS', 'SLSS', 'SLL', 'LSSS', 'LSL', 'LLS']
virahanka1(5)
```

思考：上述实现有什么显著缺陷？

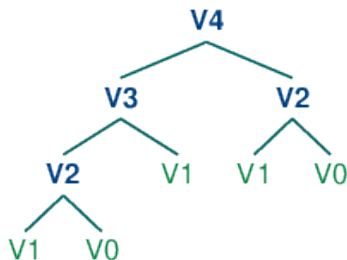


Figure 6: Call Structure of the Recursion Solution

第一种递归解法存在**重复计算**，例如，对于 V_2 ，计算 V_{20} 时，需要计算 4181 次，计算 V_{40} 时，需要计算 63245986 次。

思考： V_2 的计算次数是如何得出的（ V_2 的计算次数与 n 的存在怎样的关系）？

改进第一种递归解法：将子问题的计算结果保存到一个查找表中。

2. 自底向上的动态规划

```
def virahanka2(n):
```

```
    # V0, V1
```

```
    lookup = [[""], ["S"]]
```

```
    # n>=2才会执行下面的循环语句
```

```
    for i in range(n-1):
```

```
        s = ["S" + prosody for prosody in lookup[i+1]]
```

```
        l = ["L" + prosody for prosody in lookup[i]]
```

```
        lookup.append(s + l)
```

```
    return lookup[n]
```

$n = 3$ 时上述函数的执行过程如下:

$i = 0$, "S" + lookup[1], ---> V2

$i = 0$, "L" + lookup[0], ---> V2

lookup = [[""], ["S"], V2]

$i = 1$, "S" + lookup[2], ---> V3

$i = 1$, "L" + lookup[1], ---> V3

lookup = [[""], ["S"], V2, V3]

3. 自顶向下的动态规划

```
def virahanka3(n, lookup={0:[""], 1:["S"]}):
```

```
    # n>=2下面语句才会执行
```

```
    if n not in lookup:
```

```
        s = ["S" + prosody for prosody in virahanka3(n-1)]
```

```
        l = ["L" + prosody for prosody in virahanka3(n-2)]
```

```
        lookup[n] = s + l
```

```
    return lookup[n]
```

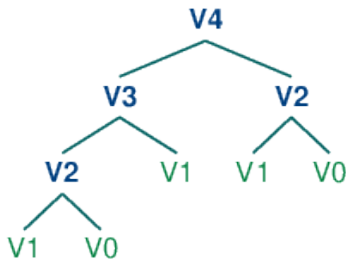


Figure 7: Call Structure of the Above Solution

- 上述自顶向下的递归调用过程虽然与第一种方法一样，但是由于存在一个查找表保存中间计算结果，因此，**不存在重复计算**；
- 在某些应用中，**自底向上**的动态规划可能会计算一些对**求解主问题没有用的子问题**，因而会造成一些资源浪费；
- **自顶向下**的动态规划可以避免这种不必要的资源浪费，因为其首先将主问题逐步分解，**仅计算对求解主问题有用的子问题**。

```
# 4. Python memoize decorator
from nltk import memoize
@memoize
def virahanka4(n):
    if n == 0:
        return [""]
    elif n == 1:
        return ["S"]
    else:
        s = ["S" + prosody for prosody in virahanka4(n-1)]
        l = ["L" + prosody for prosody in virahanka4(n-2)]
    return s + l
```

- 第四种方法使用了 Python 的 `memoize` 装饰器;
- 该装饰器存储函数调用的结果及对应的参数;
- 当该函数使用同样的参数被调用时, 不再重复计算, 直接返回存储的结果。

Please refer to chapter 4 of *Natural Language Processing with Python* for the following topics:

- Matplotlib
- csv
- NetworkX
- Numpy
- Other Python Libraries

THE END