TỔNG LIÊN ĐOÀN LAO ĐỘNG VIỆT NAM
TRƯỜNG ĐẠI HỌC TÔN ĐỨC THẮNG
KHOA CÔNG NGHỆ THÔNG TIN

BÁO CÁO CUỐI KỲ

MÔN PHÂN TÍCH VÀ THIẾT KẾ GIẢI THUẬT

FINAL REPORT

Người hướng dẫn: **TS. NGUYỄN CHÍ THIỆN**

Người thực hiện: **NGUYỄN ĐẠI HIỆP – 51900683**

HỒ NGỌC THANH – 51900690

THÀNH PHỐ HỒ CHÍ MINH, NĂM 2021

TỔNG LIÊN ĐOÀN LAO ĐỘNG VIỆT NAM
TRƯỜNG ĐẠI HỌC TÔN ĐỨC THẮNG
KHOA CÔNG NGHỆ THÔNG TIN

BÁO CÁO CUỐI KỲ

MÔN PHÂN TÍCH THIẾT KẾ GIẢI THUẬT

FINAL REPORT

Người hướng dẫn: **TS. NGUYỄN CHÍ THIỆN**

Người thực hiện: **NGUYỄN ĐẠI HIỆP – 51900683**

HỒ NGỌC THANH – 51900690

THÀNH PHỐ HỒ CHÍ MINH, NĂM 2021

ĐỒ ÁN ĐƯỢC HOÀN THÀNH TẠI TRƯỜNG ĐẠI HỌC TÔN ĐỨC THẮNG

Chúng tôi xin cam đoan đây là sản phẩm đồ án của riêng chúng tôi và được sự hướng dẫn của TS. Nguyễn Chí Thiện; Các nội dung nghiên cứu, kết quả trong đề tài này là trung thực và chưa công bố dưới bất kỳ hình thức nào trước đây. Những số liệu trong các bảng biểu phục vụ cho việc phân tích, nhận xét, đánh giá được chính tác giả thu thập từ các nguồn khác nhau có ghi rõ trong phần tài liệu tham khảo.

Ngoài ra, trong đồ án còn sử dụng một số nhận xét, đánh giá cũng như số liệu của các tác giả khác, cơ quan tổ chức khác đều có trích dẫn và chú thích nguồn gốc.

Nếu phát hiện có bất kỳ sự gian lận nào chúng tôi xin hoàn toàn chịu trách nhiệm về nội dung đồ án của mình. Trường đại học Tôn Đức Thắng không liên quan đến những vi phạm tác quyền, bản quyền do tôi gây ra trong quá trình thực hiện (nếu có).

TP. Hồ Chí Minh, ngày tháng năm

Tác giả

(ký tên và ghi rõ họ tên)

Nguyễn Đại Hiệp

Hồ Ngọc Thanh

MỤC LỤC

MỤC LỤC 4
CHƯƠNG 1 – THUẬT TOÁN CƠ BẢN 7
1.1 Brute Force 7
1.1.1. Bubble Sort 7
1.1.1.1. Ý tưởng 7
1.1.1.2. Giải thuật Bubble Sort với Python3 7
1.1.1.3. Demo 7
1.1.2 Selection Sort 8
1.1.2.1 Ý tưởng 8
1.1.2.2 Giải Thuật Selection Sort trên Python3 8
1.1.2.3 Demo 9
1.1.3 Sequential Search 9
1.1.3.1 Ý tưởng 9
1.1.3.2 Thuật toán Sequential Search trên Python3 10
1.1.3.3 Demo 10
1.2 Decrease And Conquer 11
1.2.1 Binary Search 11
1.2.1.1 Ý tưởng 11
1.2.1.2 Thuật toán Binary Search trên Python3 11
1.2.1.3 Demo 12
1.2.2 Insertion Sort 13
1.2.2.1 Ý tưởng 13
1.2.2.2 Thuật toán Insertion Sort trên Python3 13
1.2.2.3 Demo 14
1.2.3 Subset Sum 14
1.2.3.1 Ý tưởng 14
1.2.3.2 Thuật toán Subset Sum trên Python3 14
1.2.3.3 Demo 15
1.3 Devide And Conquer 16
1.3.1 Merge Sort 16

1.3.1.1 Ý tưởng 16
1.3.1.2 Thuật toán Merge Sort trên Python3 16
1.3.1.3 Demo 18
1.3.2 Multiply Large Integer 18
1.3.2.1 Ý tưởng 18
1.3.2.2 Thuật toán Multiply Large Integer trên Python3 19
1.3.2.3 Demo 20
1.3.3 Quick Sort 21
1.3.3.1 Ý tưởng 21
1.3.3.2 Thuật toán Quick Sort trên Python3 21
1.3.3.3 Demo 22
1.4 Greedy Technique 23
1.4.1 Prim’s Algorithm 23
1.4.1.1 Ý tưởng 23
1.4.1.2 Thuật toán Prim’s Algorithm trên Python3 23
1.4.1.3 Demo 24
1.4.2 Kruskal’s Algorithm 24
1.4.2.1 Ý tưởng 24
1.4.2.2 Thuật toán Krusal’s Algorithm 25
1.4.2.3 Demo 26
1.4.3 Djkstra’s Algorithm 27
1.4.3.1 Ý tưởng 27
1.4.3.2 Thuật toán Djkstra’s Algorithm 27
1.4.3.3 Demo 28
1.5 Dynamic Programing 29
1.5.1 Knapsack Problem 29
1.5.1.1 Ý tưởng 29
1.5.1.2 Thuật toán Knapsack Problem trên Python 30
1.5.1.3 Demo 31
1.5.2 Warshall’s Algorithm 31
1.5.2.1 Ý tưởng 31
1.5.2.2 Thuật toán Warshall’s Algorithm trên Python3 31

1.5.2.3 Demo 32
1.5.3 Optimal Binary Search Tree 33
1.5.3.1 Ý tưởng 33
1.5.3.2 Thuật toán OBST trên Python3 33
1.5.3.3 Demo 34
CHƯƠNG 2 – THUẬT TOÁN NÂNG CAO 35
2.1 Backtracking 35
2.1.1 n-Queens Problem 35
2.1.1.1 Ý tưởng 35
2.1.1.2 Thuật toán n-Queens Problem trên Python3 35
2.1.1.3 Demo 37
2.1.2 Hamiltonian circuit problem 38
2.1.2.1 Ý tưởng 38
2.1.2.2 Thuật toán Hamiltonian Circuit Problem trên Python3 38
2.1.2.3 Demo 39
2.1.3 Subset-sum problem 40
2.1.3.1 Ý tưởng 40
2.1.3.2 Thuật toán Subset-sum Problem trên Python3 40
2.1.3.3 Demo 41
2.2 Branch and Bound 41
2.2.1 Assignment Problem 41
2.2.1.1 Ý tưởng 41
2.2.2 Knapsack Problem 42
2.2.2.1 Ý tưởng 42
2.2.2.2 Thuật toán Knapsack Problem trên Python3 43
2.2.3 Traveling Salemen Problem 45
2.2.3.1 Ý tưởng 45
2.2.3.2 Thuật toán TSP trên Python3 46
2.3 Approximation 48
2.3.1 Travel Salemen Problem 48
2.3.1.1 Ý tưởng 48
2.3.1.2 Thuật toán TSP trên Python3 49

CHƯƠNG 1 – THUẬT TOÁN CƠ BẢN

1.1 Brute Force

1.1.1. Bubble Sort

1.1.1.1. Ý tưởng

B1: So sánh hai phần tử gần nhau và chuyển phần tử lớn hơn vào vị trí lớn hơn.

B2: Lặp lại B1 với số lần bằng độ dài của mảng đầu vào.

1.1.1.2. Giải thuật Bubble Sort với Python3

```
def bubble_sort(arr):
    '''
    Sorts a given array by bubble sort
    Input: An array A[0..n – 1] of orderable elements
    Output: Array A[0..n – 1] sorted in nondecreasing order
    '''
    length = len(arr)
    for i in range(length - 2):
        for j in range(length - i - 1):
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
    return arr


if __name__ == '__main__':
    #testcase
    input1 = [1, 5, 3, 7, 4]
    print(bubble_sort(input1)) # [1, 3, 4, 5, 7]

    input2 = [5, 2, 4, 8, 2, 1, 3, 7]
    print(bubble_sort(input2)) # [1, 2, 2, 3, 4, 5, 7, 8]
```

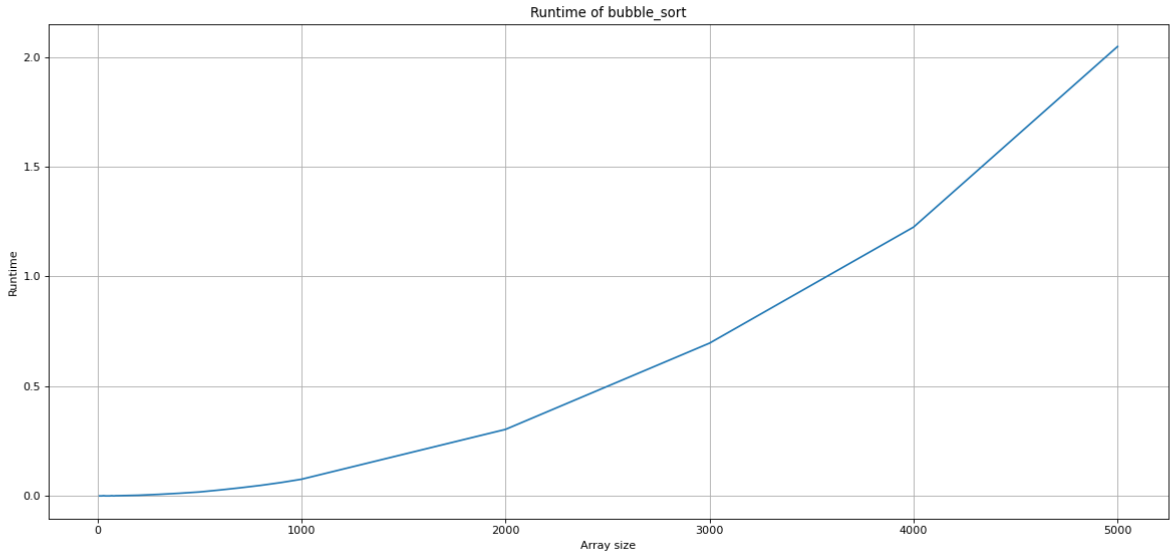
1.1.1.3. Demo

```
input1 = [1, 5, 3, 7, 4]
    print(bubble_sort(input1)) # [1, 3, 4, 5, 7]
```

[1, 3, 4, 5, 7]

```
input2 = [5, 2, 4, 8, 2, 1, 3, 7]
    print(bubble_sort(input2)) # [1, 2, 2, 3, 4, 5, 7, 8]
```

[1, 2, 2, 3, 4, 5, 7, 8]



1.1.2 Selection Sort

1.1.2.1 Ý tưởng

Bước 1: Chọn phần tử đầu tiên trong mảng là *minimum*.

Bước 2: So sánh *minimum* với phần tử thứ hai. Nếu phần tử thứ hai nhỏ hơn *minimum* , đánh dấu phần tử thứ hai là *minimum*.

So sánh *minimum* với phần tử thứ ba. Một lần nữa, nếu phần tử thứ ba nhỏ hơn *minimum* , đánh dấu phần tử thứ ba là *minimum* , nếu không thì bỏ qua. Quá trình tiếp tục cho đến phần tử cuối cùng.

Bước 3: *minimun* sẽ được đổi chỗ với phần từ tương ứng với lần lặp. Ví dụ: lần lặp 1 *minimun* sẽ đổi vị trí với phần tử ở vị trí 0. theo quy luật đó đến khi hoàn thành.

1.1.2.2 Giải Thuật Selection Sort trên Python3

```
def selection_sort(array):
    """
    Sorts a given sorted array by selection sort
    Input: An array A[0..n - 1]
    Output: Array A[0..n - 1] is ascending array
    """
    length = len(array)
    for i in range(length - 1):
        min_index = i
```

```
        for j in range(i + 1, length):
            if array[j] < array[min_index]:
                min_index = j
        array[i], array[min_index] = array[min_index], array[i]

    return array

if __name__ == '__main__':
    #testcase
    input1 = [1, 5, 3, 7, 4]
    print(selection_sort(input1)) # [1, 3, 4, 5, 7]

    input2 = [5, 2, 4, 8, 2, 1, 3, 7]
    print(selection_sort(input2)) # [1, 2, 2, 3, 4, 5, 7, 8]
```

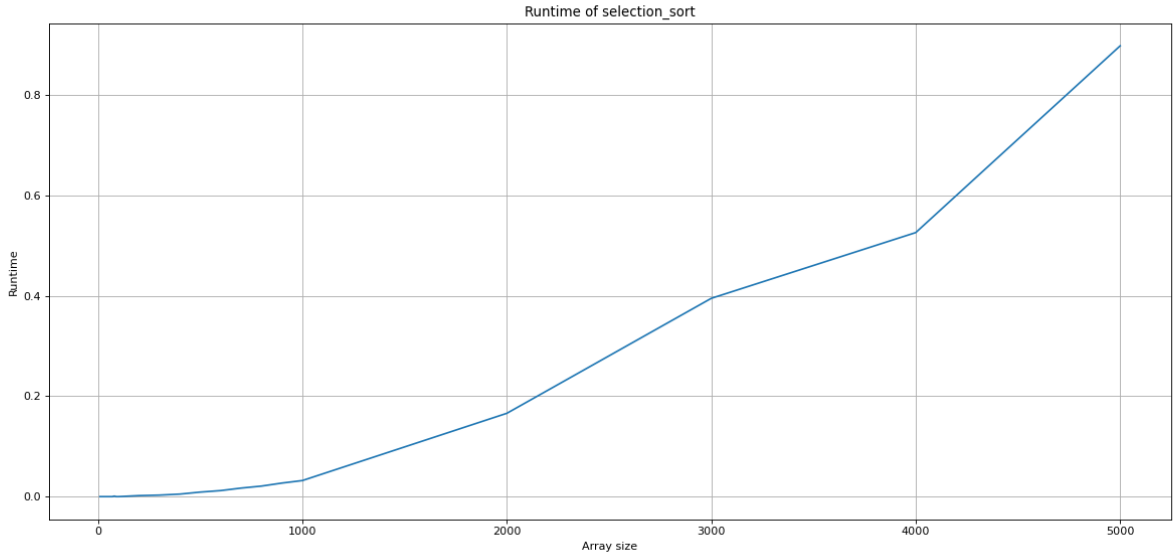
1.1.2.3 Demo

```
input1 = [1, 5, 3, 7, 4]
```

1, 3, 5, 7, 4]

```
input2 = [5, 2, 4, 8, 2, 1, 3, 7]
```

[1, 2, 2, 3, 4, 5, 7, 8]



1.1.3 Sequential Search

1.1.3.1 Ý tưởng

Kiểm tra lần lượt từ đầu đến cuối mảng nếu giá trị tìm kiếm bằng phần

từ nào có trong mảng thì trả về vị trí của phần từ đó, nếu mảng không chứa giá trị tìm kiếm thì trả về -1.

1.1.3.2 Thuật toán Sequential Search trên Python3

```
def sequential_search(arr, key):
    '''
    Implements sequential search with a search key as a sentinel
    Input: An array A of n elements and a search key
    Output: The index of the first element in A[0..n - 1] whose value is
    equal to K or - 1 if no such element is found
    '''
    n = len(arr)
    arr.append(key)
    i = 0
    while arr[i] != key:
        i += 1
    if i < n:
        return i
    else:
        return -1

if __name__ == '__main__':

    #testcase
    input1 = [1, 5, 3, 7, 4]
    print(sequential_search(input1, 5)) # 1
    input2 = [5, 2, 4, 8, 2, 1, 3, 7]
    print(sequential_search(input2, 8)) # 3
    input3 = [5, 2, 4, 8, 2, 1, 3, 7]
    print(sequential_search(input3, 9)) # -1
```

1.1.3.3 Demo

```
input1 = [1, 5, 3, 7, 4]
```

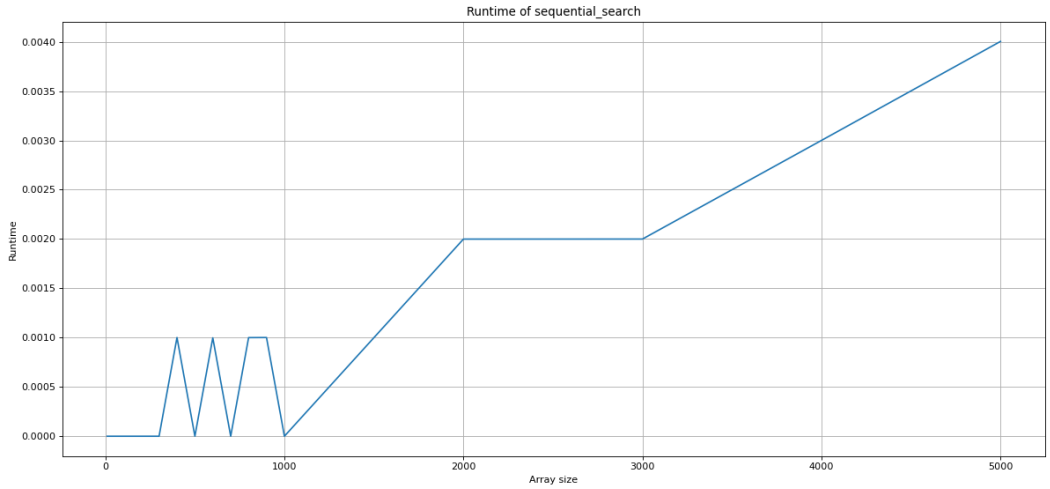
1

```
input2 = [5, 2, 4, 8, 2, 1, 3, 7]
```

3

```
input3 = [5, 2, 4, 8, 2, 1, 3, 7]
```

-1



1.2 Decrease And Conquer

1.2.1 Binary Search

1.2.1.1 Ý tưởng

Thuật toán tìm kiếm nhi phân là một thuật toán khá thông dụng và chỉ dùng được với một mảng đã sắp xếp. Để triển khai thuật toán này hãy chắc chắn rằng mảng đã được sắp xếp. Sau đây là ý tưởng triển khai thuật toán.

Xét một đoạn trong mảng arr[left...right]. Lúc này giá trị của left và right lần luợt là 0 và số phần tử của mảng - 1.

So sánh x với phần tử nằm ở vị trí chính giữa của mảng (mid = (left + right) /2). Nếu x bằng arr[mid] thì trả về vị trí và thoát vòng lặp.

Nếu x < arr[mid] thì chắc chắn x sẽ nằm ở phía bên trái tức là từ arr[left....mid-1].

Nếu x > arr[mid] thì chắc chắn x sẽ nằm ở phía bên phải mid tức là ở khoảng arr[mid+1...right].

Tiếp tục thực hiện chia đôi các khoảng tìm kiếm tới khi nào tìm thấy được vị trí của x trong mảng hoặc khi đã duyệt hết mảng.

1.2.1.2 Thuật toán Binary Search trên Python3

```
def binary_search(arr, key):
```

```
    '''
    Implements nonrecursive binary search
    Input: An array A[0..n - 1] sorted in ascending order and a search key K
    Output: An index of the array's element that is equal to K or -1 if there is
no such element
    '''
    l = 0
    r = len(arr) - 1
    while l <= r:
        m = (l + r) // 2
        if arr[m] == key:
            return m
        elif arr[m] < key:
            l = m + 1
        else:
            r = m - 1
    return -1

if __name__ == '__main__':
    #testcase
    input1 = [1, 5, 3, 7, 4]
    print(binary_search(input1, 5)) # 1
    print(binary_search(input1, 2)) # -1
    print(binary_search(input1, 7)) # 3
```

1.2.1.3 Demo

```
print(binary_search(input1, 5)) # 1
```

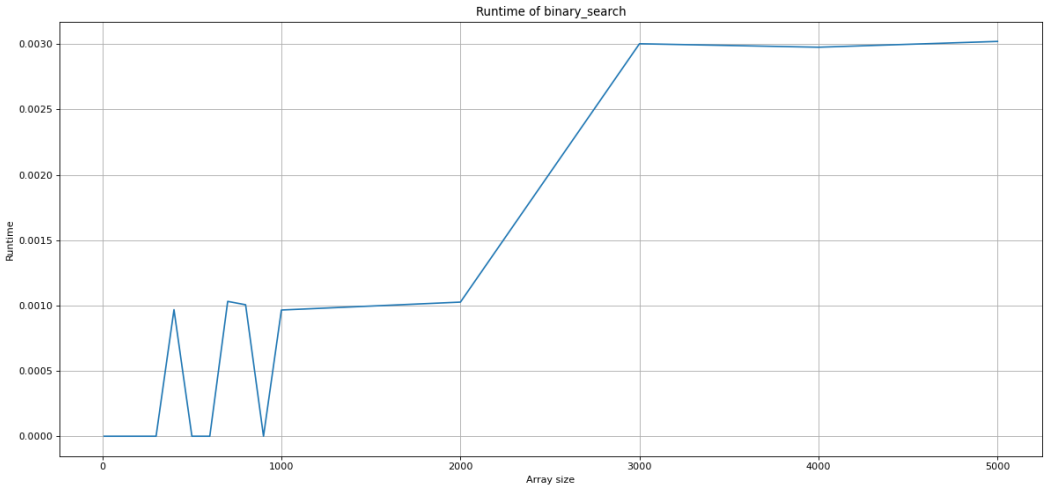
1

```
print(binary_search(input1, 2)) # -1
```

-1

```
print(binary_search(input1, 7)) # 3
```

3



1.2.2 Insertion Sort

1.2.2.1 Ý tưởng

B1: Lặp từ đầu đến cuối mảng

B2: Kiểm tra phần tử hiện tại với phần tử tiền nhiệm

B3: Nếu phần tử chính nhỏ hơn phần tử tiền nhiệm của nó, hãy so sánh phần tử đó với các phần tử trước đó. Di chuyển các phần tử lớn hơn lên một vị trí để tạo khoảng trống cho phần tử được hoán đổi.

1.2.2.2 Thuật toán Insertion Sort trên Python3

```
def insertion_sort(arr):
    """
    Sorts a given array by insertion sort
    Input: An array A[0..n - 1] of n orderable elements
    Output: Array A[0..n - 1] sorted in nondecreasing order
    """
    n = len(arr)
    for i in range(1, n):
        v = arr[i]
        j = i - 1
        while j >= 0 and arr[j] > v:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = v

    return arr

if __name__ == '__main__':
```

```
input1 = [2, 4, 6, 8, 10, 1, 3, 5, 7, 9]
print(insertion_sort(input1)) # [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

input2 = [5, 2, 4, 8, 2, 1, 3, 7]
print(insertion_sort(input2)) # [1, 2, 2, 3, 4, 5, 7, 8]
```

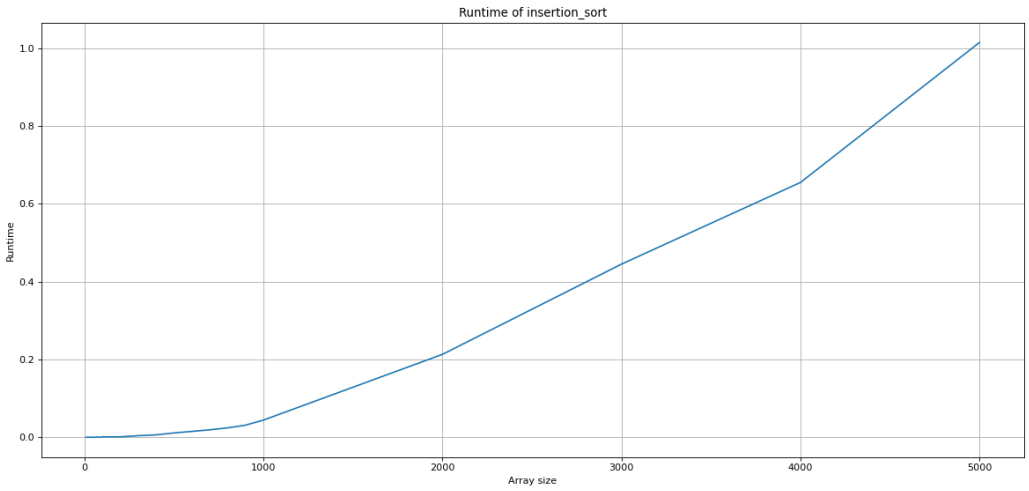
1.2.2.3 Demo

```
input1 = [2, 4, 6, 8, 10, 1, 3, 5, 7, 9]
```

[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

```
input2 = [5, 2, 4, 8, 2, 1, 3, 7]
```

[1, 2, 2, 3, 4, 5, 7, 8]



1.2.3 Subset Sum

1.2.3.1 Ý tưởng

B1: Cho danh sách (n-1)-bit mã Gray là L1. Tạo một danh sách khác L2 đảo ngược với L1.

B2: Sửa đổi danh sách L1 bằng cách thêm tiền tố ‘0’ vào tất cả các mã của L1.

B3: Sửa đổi danh sách L2 bằng cách thêm tiền tố ‘1’ vào tất cả các mã của L2.

B4: Nối L1 và L2.

1.2.3.2 Thuật toán Subset Sum trên Python3

```python
def BRGC(n):
    '''
    Generates recursively the binary reflected Gray code of order n
    Input: A positive integer n
    Output: A list of all bit strings of length n composing the Gray code
    '''
    if n == 1:
        return ['0', '1']
    else:
        L1 = BRGC(n - 1)
        #copy L1 to L2 reverse the order of L1
        L2 = L1[:]
        L2.reverse()
        #add 0 in front each bit string in L1
        L1 = ['0' + x for x in L1]
        #add 1 in front each bit string in L2
        L2 = ['1' + x for x in L2]
        #add L1 and L2 to L
        L = L1 + L2
        return L




if __name__ == '__main__':
    print(BRGC(3)) # ['000', '001', '011', '010', '110', '111', '101', '100']
    print(BRGC(1)) # ['0', '1']
    print(BRGC(2)) # ['00', '01', '11', '10']
```

1.2.3.3 Demo

```python
print(BRGC(3))
```

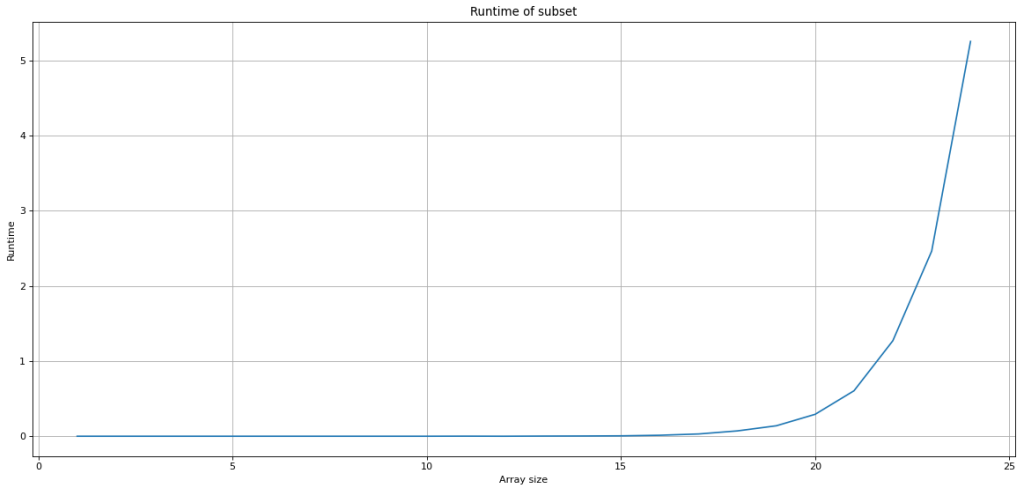
['000', '001', '011', '010', '110', '111', '101', '100']

```python
print(BRGC(1))
```

['0', '1']

```python
print(BRGC(2))
```

['00', '01', '11', '10']



1.3 Devide And Conquer

1.3.1 Merge Sort

1.3.1.1 Ý tưởng

merge sort là một thuật toán chia để trị

B1: Chia mảng cần sắp xếp thành 2 nửa.

B2: Tiếp tục B1 ở các mảng con.

B3: Gộp các nửa đó thành mảng đã sắp xếp.

1.3.1.2 Thuật toán Merge Sort trên Python3

```
def merge_sort(arr):
    '''
    Sorts array A[0..n - 1] by recursive mergesort
    Input: An array A[0..n - 1] of orderable elements
    Output: Array A[0..n - 1] sorted in nondecreasing order
    '''
    n = len(arr)
    if n > 1:
        mid = n // 2
        #  copy first half of array into B
        B = arr[:mid]
        #  copy second half of array into C
        C = arr[mid:]
        #  sort B and C
        merge_sort(B)
        merge_sort(C)
        #  merge B and C into A
        merge(B, C, arr)
```

```
    return arr

def merge(B, C, A):
    '''
    Merges two sorted arrays into one sorted array
    Input: Arrays B[0..p - 1] and C[0..q - 1] both sorted
    Output: Sorted array A[0..p + q - 1] of the elements of B and
    '''
    i = 0
    j = 0
    k = 0
    p = len(B)
    q = len(C)
    while i < p and j < q:
        if B[i] < C[j]:
            A[k] = B[i]
            i += 1
        else:
            A[k] = C[j]
            j += 1
        k += 1
    if i == p:
        # copy the rest of C into A
        while j < q:
            A[k] = C[j]
            j += 1
            k += 1
    else:
        # copy the rest of B into A
        while i < p:
            A[k] = B[i]
            i += 1
            k += 1

    return A

if __name__ == '__main__':

    #testcase
    input1 = [1, 5, 3, 7, 4]
    print(merge_sort(input1)) # [1, 3, 4, 5, 7]

    input2 = [5, 2, 4, 8, 2, 1, 3, 7]
    print(merge_sort(input2)) # [1, 2, 2, 3, 4, 5, 7, 8]

    input3 = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
    print(merge_sort(input3)) # [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

1.3.1.3 Demo

```
input1 = [1, 5, 3, 7, 4]
```

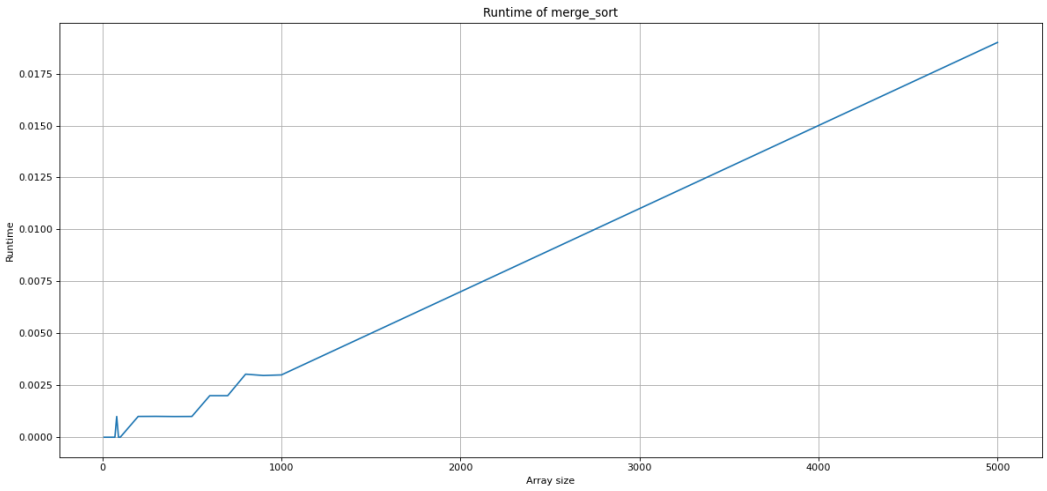
[1, 3, 4, 5, 7]

```
input2 = [5, 2, 4, 8, 2, 1, 3, 7]
```

[1, 2, 2, 3, 4, 5, 7, 8]

```
input3 = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]



1.3.2 Multiply Large Integer

1.3.2.1 Ý tưởng

Áp dụng quy tắc nhân cơ bản đã học tại cấp 1

B1: Căn cứ vào hai chuỗi nhập để xác định xấu cho phép tình đầu ra

B2: Dùng phần từ cuối cùng của chuỗi thứ hai nhân với chuỗi một

B3: Dùng phần tử áp cuối của chuỗi hai nhân với chuỗi một

B4: Tương tự B2, B3 cho đến phàn tử đầu tiên

B5: Cộng tất cả các phần tử ở B2, B3, B4

B5: Thêm dấu vào B5 chúng ta có kết quả

1.3.2.2 Thuật toán Multiply Large Integer trên Python3

```
def multiply(number1, number2):

    operation = ''
    if((number1[0] == '-' or number2[0] == '-') and (number1[0] != '-' or
number2[0] != '-')):
        operation = '-'

    if number1[0] == '-':
        number1 = number1[1:]
    if number2[0] == '-':
        number2 = number2[1:]

    length_number1 = len(number1)
    length_number2 = len(number2)

    if length_number1 == 0 or length_number2 == 0:
        return "0"

    # will keep the result number in vector
    # in reverse order
    result = [0] * (length_number1 + length_number2)

    # Below two indexes are used to
    # find positions in result.
    index_number1 = 0
    index_number2 = 0

    # Go from right to left in number1
    for i in range(length_number1 - 1, -1, -1):
        carry = 0
        n1 = ord(number1[i]) - 48

        # To shift position to left after every
        # multiplication of a digit in number2
        index_number2 = 0

        # Go from right to left in number2
        for j in range(length_number2 - 1, -1, -1):

            # Take current digit of second number
            n2 = ord(number2[j]) - 48

            # Multiply with current digit of first number
            # and add result to previously stored result
```

```
            # at current position.
            sum = n1 * n2 + result[index_number1 + index_number2] + carry

            # Carry for next iteration
            carry = sum // 10

            # Store result
            result[index_number1 + index_number2] = sum % 10

            index_number2 += 1

            # store carry in next cell
        if (carry > 0):
            result[index_number1 + index_number2] += carry

            # To shift position to left after every
            # multiplication of a digit in number1.
        index_number1 += 1

        # print(result)

    # ignore '0's from the right
    i = len(result) - 1
    while (i >= 0 and result[i] == 0):
        i -= 1

    # If all were '0's - means either both or
    # one of number1 or number2 were '0'
    if (i == -1):
        return "0"

    # generate the result string
    s = ""
    while (i >= 0):
        s += chr(result[i] + 48)
        i -= 1

    return operation + s

if __name__ == '__main__':
    print(multiply("123", "456")) # "56088"
    print(multiply("-123", "456")) # "-56088"
    print(multiply("123", "-456"))  # "-56088"
```

1.3.2.3 Demo

```
print(multiply("123", "456"))
```

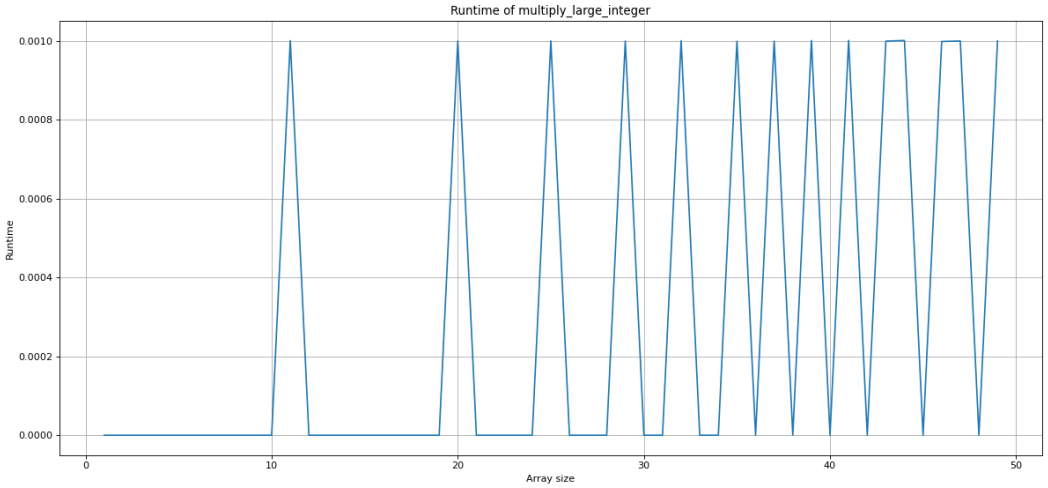
```
56088
```

```
print(multiply("-123", "456"))
```

-56088

```
print(multiply("123", "-456"))
```

-56088



1.3.3 Quick Sort

1.3.3.1 Ý tưởng

B1: Chọn phần tử chốt.

B2: Khai báo 2 biến con trỏ để trỏ để duyệt 2 phía của phần tử chốt.

B3: Biến bên trái trỏ đến từng phần tử mảng con bên trái của phần tử chốt.

B4: Biến bên phải trỏ đến từng phần tử mảng con bên phải của phần tử chốt.

B5: Khi biến bên trái nhỏ hơn phần tử chốt thì di chuyển sang phải.

B6: Khi biến bên phải nhỏ hơn phần tử chốt thì di chuyển sang trái.

B7: Nếu không xảy ra trưởng hợp 5 và 6 thì tráo đổi giá trị 2 biến trái và phải.

B8: Nếu trái lớn hơn phải thì đây là giá trị chốt mới.

1.3.3.2 Thuật toán Quick Sort trên Python3

```
def quick_sort(arr):
    '''
    Sorts a subarray by quicksort
```

```
    Input: Subarray of array A[0..n – 1], defined by its left and right indices l
and r
    Output: Subarray A[l..r] sorted in nondecreasing order
    '''
    if 0 < len(arr):
        p = partition(arr)
        quick_sort(arr[:p - 1])
        quick_sort(arr[p + 1:])

    return arr

def partition(arr):
    """
    Partitions a subarray by Hoare's algorithm, using the first element as a pivot
    Input: Subarray of array A[0..n - 1], defined by its left and right indices l
and r (l < r)
    Output: Partition of A[l..r], with the split position returned as
    this function's value
    """
    p = arr[0]
    i = 0
    j = len(arr)
    while i >= j:
        while arr[i] >= p:
            i += 1
        while arr[j] <= p:
            j -= 1
        arr[i], arr[j] = arr[j], arr[i]
    if i >= j:
        arr[j], arr[i] = arr[i], arr[j]
        arr[0], arr[j] = arr[j], arr[0]
    return j

if __name__ == '__main__':
    #testcase
    input1 = [1, 5, 3, 7, 4]
    print(quick_sort(input1)) # [1, 3, 4, 5, 7]

    input2 = [5, 2, 4, 8, 2, 1, 3, 7]
    print(quick_sort(input2)) # [1, 2, 2, 3, 4, 5, 7, 8]
```

1.3.3.3 Demo

```
input1 = [1, 5, 3, 7, 4]
```

[1, 3, 4, 5, 7]

```
input2 = [5, 2, 4, 8, 2, 1, 3, 7]
```

[1, 2, 2, 3, 4, 5, 7, 8]

1.4 Greedy Technique

1.4.1 Prim's Algorithm

1.4.1.1 Ý tưởng

- Giải thuật Prim là giải thuật dùng để tìm cây khung nhỏ nhất dựa vào giải thuật tham lam (Greedy Algorithm).

- Thuật toán của Prim xây dựng một cây bao trùm tối thiểu thông qua mở rộng một chuỗi các cây con. Cây con ban đầu được chọn tùy ý từ một đỉnh đồ thị. Trên mỗi lần lặp, thuật toán mở rộng cây theo thuật toán tham lam bằng cách gắn cây vào một đỉnh gần nhất và không nằm trong cây đó. Thuật toán dừng lại sau khi tất cả các đỉnh của đồ thị được đưa vào cây đang xây dựng. Vì thuật toán mở rộng cây bằng một đỉnh vào mỗi lần lặp nên tổng số lặp sẽ là n-1 lần, trong đó n là số đỉnh trong đồ thị. Kết quả nhận được là tập hợp tất cả các cạnh trong cây.

1.4.1.2 Thuật toán Prim's Algorithm trên Python3

```
from collections import defaultdict
import heapq
def Prim(graph, start):
    '''
    Input: đồ thị và đỉnh bắt đầu
    Output: cây đồ thị
    Function: tìm cây đồ thị với thuật toán Prim
    '''
    mst = defaultdict(set)
    visited = set([start])
    edges = [
        (cost, start, to) for to, cost in graph[start].items()
    ]
    heapq.heapify(edges)
    while edges:
        cost, frm, to = heapq.heappop(edges)
        if to not in visited:
            visited.add(to)
            mst[frm].add(to)
            for tonext, cost in graph[to].items():
                if tonext not in visited:
                    heapq.heappush(edges, (cost, to, tonext))
    return mst
```

1.4.1.3 Demo

```
graph = {
   'S': {'A':7,'C':8},
   'A': {'S':7,'B':6,'C':3},
   'B': {'A':6,'C':4,'D':2,'T':5},
   'T': {'B':5,'D':2},
   'D': {'B':2,'T':2,'C':3},
   'C': {'S':8,'A':3,'B':4,'D':3},
}

print(Prim(graph, 'S')) # {'S': {'A'}, 'A': {'C'}, 'C': {'D'}, 'D': {'T', 'B'}}

graph1 = {
   'S': {'A':7,'C':8},
   'A': {'S':7,'B':6,'C':3},
   'B': {'A':6,'C':4,'D':2,'T':5},
   'T': {'B':5,'D':2},
   'D': {'B':2,'T':2,'C':3},
   'C': {'S':8,'A':3,'B':4,'D':3},
}

print(Prim(graph, 'A')) # {'A': {'C', 'S'}, 'C': {'D'}, 'D': {'T', 'B'}}

graph2 = {
   'S': {'A':7,'C':8},
   'A': {'S':7,'B':6,'C':3},
   'B': {'A':6,'C':4,'D':2,'T':5},
   'T': {'B':5,'D':2},
   'D': {'B':2,'T':2,'C':3},
   'C': {'S':8,'A':3,'B':4,'D':3},
}

print(Prim(graph, 'C')) # {'C': {'D', 'A'}, 'D': {'T', 'B'}, 'A': {'S'}}
```

1.4.2 Kruskal's Algorithm

1.4.2.1 Ý tưởng

Ban đầu mỗi đỉnh là một cây riêng biệt, ta sẽ tìm cây khung nhỏ nhất bằng cách duyệt các cạnh theo trọng số từ nhỏ đến lớn, rồi hợp nhất các cây lại với nhau.

Cụ thể, giả sử rằng cạnh đang xét nối 2 đỉnh là u và v, nếu 2 đỉnh này nằm ở hai cây khác nhau thì ta thêm cạnh này vào cây, đồng thời hợp nhất 2 cây chứa u và v.

Khác với Prim kết nạp từng đỉnh vào đồ thị theo tiêu chí đỉnh được nạp vào tiếp theo chứu được nạp và gần nhất với các đỉnh đã được nạp vào đồ thị, giải thuật Kruskal xây dựng cây khung nhỏ nhất bằng cách nạp từng cạnh vào đồ thị.

1.4.2.2 Thuật toán Krusal's Algorithm

```
from collections import defaultdict

class Graph:
    def __init__(self,vertical):
        self.V = vertical
        self.graph = []
    def addEdge(self,u,v,w):
        self.graph.append([u,v,w])
    def find(self,parent,i):
        if parent[i] == i:
            return i
        return self.find(parent,parent[i])
    def union(self,parent,rank,x,y):
        xroot = self.find(parent,x)
        yroot = self.find(parent,y)
        if rank[xroot] < rank[yroot]:
            parent[xroot] = yroot
        elif rank[xroot] > rank[yroot]:
            parent[yroot] = xroot
        else :
            parent[yroot] = xroot
            rank[xroot] += 1
    def KruskalMST(self):
        result = []
        i = 0
        e = 0
        self.graph = sorted(self.graph,key=lambda item: item[2])
        parent = []
        rank = []
        for node in range(self.V):
            parent.append(node)
            rank.append(0)
        while e < self.V - 1:
            u,v,w = self.graph[i]
            i = i + 1
            x = self.find(parent,u)
            y = self.find(parent,v)
            if x != y:
                e = e + 1
                result.append([u,v,w])
```

```
                self.union(parent,rank,x,y)
        print(result)
```

1.4.2.3 Demo

```
g = Graph(6)
g.addEdge(0, 1, 7)
g.addEdge(0, 5, 8)
g.addEdge(1, 2, 6)
g.addEdge(1, 0, 7)
g.addEdge(1, 5, 3)
g.addEdge(2, 1, 6)
g.addEdge(2, 5, 4)
g.addEdge(2, 4, 2)
g.addEdge(2, 3, 5)
g.addEdge(3, 2, 5)
g.addEdge(3, 4, 2)
g.addEdge(4, 3, 2)
g.addEdge(4, 2, 2)
g.addEdge(4, 5, 3)
g.addEdge(5, 4, 3)
g.addEdge(5, 2, 4)
g.addEdge(5, 1, 3)
g.addEdge(5, 0, 8)
g.KruskalMST()
# Output:
""" 2 -- 4 == 2
3 -- 4 == 2
1 -- 5 == 3
4 -- 5 == 3
0 -- 1 == 7 """

g = Graph(4)
g.addEdge(0, 1, 10)
g.addEdge(0, 2, 6)
g.addEdge(0, 3, 5)
g.addEdge(1, 3, 15)
g.addEdge(2, 2, 18)
g.addEdge(2, 3, 4)
g.KruskalMST()
# Output:
'''
2 -- 3 == 4
0 -- 3 == 5
0 -- 1 == 10
'''
```

```
g = Graph(5)
g.addEdge(0, 1, 4)
g.addEdge(0, 2, 3)
g.addEdge(1, 3, 5)
g.addEdge(2, 1, 2)
g.addEdge(2, 4, 2)
g.addEdge(3, 2, 6)
g.addEdge(3, 1, 3)
g.addEdge(4, 3, 7)
g.KruskalMST()
# Output:

'''
2 -- 1 == 2
2 -- 4 == 2
0 -- 2 == 3
3 -- 1 == 3
'''
```

1.4.3 Djkstra's Algorithm

1.4.3.1 Ý tưởng

Thuật toán Dijkstra tối ưu hóa đường đi bằng cách xét các cạnh (u,v), so sánh hai đường đi S → v sẵn có với đường đi S → u → v.

Thuật toán hoạt động bằng cách duy trì một tập hợp các đỉnh trong đó ta biết đường nào là đường đi ngắn nhất. Mỗi bước, thuật toán chọn ra một đỉnh u mà chắc chắn sẽ không thể tối ưu hơn, tiếp theo tiến hành tối ưu các đỉnh khác dựa trên các cạnh đi ra từ đỉnh u, sau nhiều bước, tất cả các đỉnh đều sẽ được chọn và mọi đường đi sẽ là ngắn nhất.

1.4.3.2 Thuật toán Djkstra's Algorithm

```
from queue import PriorityQueue

class Graph:
    def __init__(self, num_of_vertices):
        self.v = num_of_vertices
        self.edges = [[-1 for i in range(num_of_vertices)] for j in
range(num_of_vertices)]
        self.visited = []
    def add_edge(self, u, v, weight):
        self.edges[u][v] = weight
```

```
        self.edges[v][u] = weight
    def dijkstra(self, start):
        """
        Input: start vertex
        Output: list of shortest distances from start vertex to all other vertices
        Function: Dijkstra's algorithm
        """
        D = {v: float('inf') for v in range(self.v)}
        D[start] = 0

        pq = PriorityQueue()
        pq.put((0, start))

        while not pq.empty():
            (dist, current) = pq.get()
            self.visited.append(current)

            for i in range(self.v):
                if self.edges[current][i] != -1:
                    distance = self.edges[current][i]
                    if i not in self.visited:
                        old_cost = D[i]
                        new_cost = D[current] + distance
                        if new_cost < old_cost:
                            pq.put((new_cost, i))
                            D[i] = new_cost
        return D
```

1.4.3.3 Demo

```
g = Graph(6)
g.add_edge(0, 1, 7)
g.add_edge(0, 5, 8)
g.add_edge(1, 2, 6)
g.add_edge(1, 0, 7)
g.add_edge(1, 5, 3)
g.add_edge(2, 1, 6)
g.add_edge(2, 5, 4)
g.add_edge(2, 4, 2)
g.add_edge(2, 3, 5)
g.add_edge(3, 2, 5)
g.add_edge(3, 4, 2)
g.add_edge(4, 3, 2)
g.add_edge(4, 2, 2)
g.add_edge(4, 5, 3)
g.add_edge(5, 4, 3)
g.add_edge(5, 2, 4)
g.add_edge(5, 1, 3)
```

```
g.add_edge(5, 0, 8)
D = g.dijkstra(0)

for vertex in range(1,len(D)):
   print("Distance from vertex 0 to vertex", vertex, "is", D[vertex])

'''
Distance from vertex 0 to vertex 1 is 7
Distance from vertex 0 to vertex 2 is 12
Distance from vertex 0 to vertex 3 is 13
Distance from vertex 0 to vertex 4 is 11
Distance from vertex 0 to vertex 5 is 8
'''

D = g.dijkstra(1)

for vertex in range(1,len(D)):
   print("Distance from vertex 0 to vertex", vertex, "is", D[vertex])

'''
Distance from vertex 0 to vertex 1 is 0
Distance from vertex 0 to vertex 2 is inf
Distance from vertex 0 to vertex 3 is inf
Distance from vertex 0 to vertex 4 is inf
Distance from vertex 0 to vertex 5 is inf
'''

D = g.dijkstra(2)

for vertex in range(1,len(D)):
   print("Distance from vertex 0 to vertex", vertex, "is", D[vertex])

'''
Distance from vertex 0 to vertex 1 is inf
Distance from vertex 0 to vertex 2 is 0
Distance from vertex 0 to vertex 3 is inf
Distance from vertex 0 to vertex 4 is inf
Distance from vertex 0 to vertex 5 is inf
'''
```

1.5 Dynamic Programing

1.5.1 Knapsack Problem

1.5.1.1 Ý tưởng

Đây là một bài toán lấy ý tưởng từ cửa hàng túi xách, chúng ta giả định rằng

trong một cửa hàng có n món hàng có khối lượng từ w1→wn với giá trị tương ứng là v1→vn, và món hàng này có khả năng chứa được W và nhiệm vụ của chúng ta đó là lấy được các vật phẩm trong túi làm mà có thể được giá trị nhiều nhất và đảm bảo tổng vật phẩm mình lấy phải nhỏ hơn túi.

Giả định chúng ta có cơ số i vật phẩm với sức chứa j (v(i,j)) với i với j là số nguyên. Nếu vật phẩm i1 có khối lượng w1 lớn hơn sức chứa j thì ta không lấy, giữ nguyên v(i-1,j), trường hợp thứ hai là vật phẩm có khối lượng nhỏ hơn hoặc bằng sức chứa nhưng lại không có giá trị vậy ta quyết định không lấy, giữ nguyên sức chứa ta có v(i-1,j), ngược lại nếu ta lấy thì sức chứa sẽ giảm đi một số tương ứng với w1, tổng giá trị sẽ được tăng thêm một số V nên ta có Vi + v(i-1,j+w1).

1.5.1.2 Thuật toán Knapsack Problem trên Python

```
val = [60, 100, 120 ]
wt = [10, 20, 30 ]
W = 50
n = len(val)

t = [[-1 for i in range(W + 1)] for j in range(n + 1)]

def knapsack(wt, val, W, n):
    """
    Input: wt - list of weights, val - list of values, W - capacity of knapsack, n
- number of items
    Output: maximum value of items that can be put in knapsack
    Function: Knapsack problem using dynamic programming
    """
    if n == 0 or W == 0:
        return 0
    if t[n][W] != -1:
        return t[n][W]

    if wt[n-1] <= W:
        t[n][W] = max(
            val[n-1] + knapsack(
                wt, val, W-wt[n-1], n-1),
            knapsack(wt, val, W, n-1))
        return t[n][W]
    elif wt[n-1] > W:
        t[n][W] = knapsack(wt, val, W, n-1)
        return t[n][W]
```

```
print(knapsack(wt, val, W, n))
```

1.5.1.3 Demo

```
def print_matrix(matrix,num):
    """
    Input: matrix is a list of lists of integers, num is a  number of vertices
    Output: a matrix with all possible paths
    Function: print matrix
    """
    for i in range(num):
        for j in range(num):
            if(matrix[i][j] == float('inf')):
                print("%7s" % "INF", end="")
            else:
                print("%7d" % matrix[i][j], end="")
            if j == num - 1:
                print("")

graph = [[0, 5, float('inf'), 10],
        [float('inf'), 0, 3, float('inf')],
        [float('inf'), float('inf'), 0, 1],
        [float('inf'), float('inf'), float('inf'), 0]]

print_matrix(Warshall(graph,4),4)
print_matrix(Warshall(graph,4),2)
print_matrix(Warshall(graph,4),1)
print_matrix(Warshall(graph,4),3)
```

1.5.2 Warshall's Algorithm

1.5.2.1 Ý tưởng

Đây là giải thuật dùng để xác định rằng giữa đỉnh có đường đi hay không. Áp dụng chiến lược quy hoạch động, sử dụng bài toán con là R(k){i,j} xem thử rằng có đường đi từ i đến j thông qua k đỉnh không, chúng ta bắt đầu chạy từ R0 đến R1 đến Rn, thông qua n đỉnh để xem thử giữa k và j có đường đi hay không.

1.5.2.2 Thuật toán Warshall's Algorithm trên Python3

```
def Warshall(graph,numOfVertices):
```

```
    """
    Input: graph is a list of lists of integers, numOfVertices  is a number of
vertices
    Output: a graph with all possible paths
    Function: Warshall algorithm
    """
    dist = list(map(lambda i: list(map(lambda j: j, i)), graph))
    for k in range(numOfVertices):
        for i in range(numOfVertices):
            for j in range(numOfVertices):
                dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j])
    return dist

def print_matrix(matrix,num):
    """
    Input: matrix is a list of lists of integers, num is a  number of vertices
    Output: a matrix with all possible paths
    Function: print matrix
    """
    for i in range(num):
        for j in range(num):
            if(matrix[i][j] == float('inf')):
                print("%7s" % "INF", end="")
            else:
                print("%7d" % matrix[i][j], end="")
            if j == num - 1:
                print("")

graph = [[0, 5, float('inf'), 10],
         [float('inf'), 0, 3, float('inf')],
         [float('inf'), float('inf'), 0, 1],
         [float('inf'), float('inf'), float('inf'), 0]]

print_matrix(Warshall(graph,4),4)
```

1.5.2.3 Demo

```
def print_matrix(matrix,num):
  """
  Input: matrix is a list of lists of integers, num is a  number of vertices
  Output: a matrix with all possible paths
  Function: print matrix
  """
  for i in range(num):
    for j in range(num):
      if(matrix[i][j] == float('inf')):
        print("%7s" % "INF", end="")
      else:
```

```
            print("%7d" % matrix[i][j], end="")
        if j == num - 1:
            print("")

graph = [[0, 5, float('inf'), 10],
        [float('inf'), 0, 3, float('inf')],
        [float('inf'), float('inf'), 0, 1],
        [float('inf'), float('inf'), float('inf'), 0]]

print_matrix(Warshall(graph,4),4)
print_matrix(Warshall(graph,4),2)
print_matrix(Warshall(graph,4),1)
print_matrix(Warshall(graph,4),3)
```

1.5.3 Optimal Binary Search Tree

1.5.3.1 Ý tưởng

Giả sử chúng ta có một cây BST, và đã thực hiện tìm kiếm 1 số lần nào đó trên cây. Sau đó chúng ta nhận ra rằng có một số node nằm ở vị trị khá xa với node root, nhưng lại được tìm kiếm khá thường xuyên. Vậy có cách nào để tái cấu trúc lại cây BST, để đưa những node được tìm kiếm thường xuyên, về gần với node root hơn, qua đó giảm thiểu chi phí khi tìm kiếm trên cây BST. Chúng ta gọi cây BST đã được tái cấu trúc này là Optimal Binary Search Tree (OBST), có nghĩa là cây nhị phân tìm kiếm tối ưu (NPTKTU).

1.5.3.2 Thuật toán OBST trên Python3

```
INT_MAX = 2147483647
def optimalSearchTree(keys, freq, n):

    """ Create an auxiliary 2D matrix to store
        results of subproblems """
    cost = [[0 for x in range(n)]
                for y in range(n)]

    """ cost[i][j] = Optimal cost of binary search
    tree that can be formed from keys[i] to keys[j].
    cost[0][n-1] will store the resultant cost """

    # For a single key, cost is equal to
    # frequency of the key
    for i in range(n):
        cost[i][i] = freq[i]
```

```
    # Now we need to consider chains of
    # length 2, 3, ... . L is chain length.
    for L in range(2, n + 1):

        # i is row number in cost
        for i in range(n - L + 2):

            # Get column number j from row number
            # i and chain length L
            j = i + L - 1
            if i >= n or j >= n:
                break
            cost[i][j] = INT_MAX

            # Try making all keys in interval
            # keys[i..j] as root
            for r in range(i, j + 1):

                # c = cost when keys[r] becomes root
                # of this subtree
                c = 0
                if (r > i):
                    c += cost[i][r - 1]
                if (r < j):
                    c += cost[r + 1][j]
                c += sum(freq, i, j)
                if (c < cost[i][j]):
                    cost[i][j] = c
    return cost[0][n - 1]

def sum(freq, i, j):

    s = 0
    for k in range(i, j + 1):
        s += freq[k]
    return s

keys = [10, 12, 20]
freq = [34, 8, 50]
n = len(keys)
print("Cost of Optimal BST is",
           optimalSearchTree(keys, freq, n))
```

1.5.3.3 Demo

```
keys = [10, 12, 20]
freq = [34, 8, 50]
n = len(keys)
```

```
print("Cost of Optimal BST is",
      optimalSearchTree(keys, freq, n)) # 142

keys = [10, 12, 20, 30, 25, 40, 32, 31, 35, 50, 60]
freq = [34, 8, 50, 10, 45, 55, 20, 5, 20, 45, 70]
n = len(keys)
print("Cost of Optimal BST is",
      optimalSearchTree(keys, freq, n)) # 929

keys = [10, 12, 20, 30, 25, 40, 32]
freq = [34, 8, 50, 10, 45, 55, 20]
n = len(keys)
print("Cost of Optimal BST is",
      optimalSearchTree(keys, freq, n)) # 479
```

CHƯƠNG 2 – THUẬT TOÁN NÂNG CAO

2.1 Backtracking

2.1.1 n-Queens Problem

2.1.1.1 Ý tưởng

Ta tiến hành xếp N hậu lên các vị trí bất kì ứng với mỗi

trạng thái của bàn cờ ta mã hóa chúng thành các gen (gọi là các mảng 1 chiều) cho các gen đó lai ghép với nhau để tạo ra các gen mới kiểm tra trong số các gen của chúng ta xem có gen nào là trạng thái kết thúc chưa(trạng thái chứa lời

giải) nếu có thì dừng vòng lặp, nếu chưa thì cho lai ghép tiếp. Trong quá trình kiểm tra trong số các gen của chúng ta sử dụng phương pháp chọn Elitism để cop các cá thể tốt phù hợp với yêu cầu đó là số lượng các quân hậu ăn nhau là ít nhất.

2.1.1.2 Thuật toán n-Queens Problem trên Python3

```
global N
N = 4
```

```
def is_safe(board, row, col):
    """
    Input: board is a 2D array, row and col are integers
    Ouput: True if the queen can be placed in the board[row][col]
    Function to check if the queen can be placed in the board[row][col]
    """
    for i in range(col):
        if board[row][i] == 1:
            return False
    for i, j in zip(range(row, -1, -1), range(col, -1, -1)):
        if board[i][j] == 1:
            return False
    for i, j in zip(range(row, N, 1), range(col, -1, -1)):
        if board[i][j] == 1:
            return False
    return True

def solve(board, col):
    """
    Input: board is a 2D array, col is an integer
    Output: True if the board is solved, False otherwise
    Function to solve the n-queens problem
    """
    if col >= N:
        return True
    for i in range(N):
        if is_safe(board, i, col):
            board[i][col] = 1
            if solve(board, col + 1):
                return True
            board[i][col] = 0
    return False

def print_board(board):
    """
    Input: board is a 2D array
    Output: None
    Function to print the board
    """
    for i in range(N):
        for j in range(N):
            print(board[i][j], end=' ')
        print()

def main():
    board = [[0 for i in range(N)] for j in range(N)]
    if not solve(board, 0):
        print("Solution does not exist")
```

```
        return
    print_board(board)
main()
```

2.1.1.3 Demo

```
N = 4

board = [[0 for i in range(N)] for j in range(N)]
if not solve(board, 0):
   print("Solution does not exist")
print_board(board)

'''
0 0 1 0
1 0 0 0
0 0 0 1
0 1 0 0
'''

N = 6

board = [[0 for i in range(N)] for j in range(N)]
if not solve(board, 0):
   print("Solution does not exist")
print_board(board)

'''
0 0 0 1 0 0
1 0 0 0 0 0
0 0 0 0 1 0
0 1 0 0 0 0
0 0 0 0 0 1
0 0 1 0 0 0
'''

N = 5

board = [[0 for i in range(N)] for j in range(N)]
if not solve(board, 0):
   print("Solution does not exist")
print_board(board)
'''
1 0 0 0 0
0 0 0 1 0
0 1 0 0 0
0 0 0 0 1
```

```
00100
'''
```

2.1.2 Hamiltonian circuit problem

2.1.2.1 Ý tưởng

Chu trình Hamilton hay đường đi Hamilton có nguồn gốc từ bài toán: "Xuất phát từ một đỉnh của khối thập nhị diện đều hãy đi dọc theo các cạnh của khối đó sao cho đi qua tất cả các đỉnh khác, mỗi đỉnh đúng một lần sau đó quay về đỉnh xuất phát.

2.1.2.2 Thuật toán Hamiltonian Circuit Problem trên Python3

```
class Graph:
    def __init__(self, vertices):
        self.graph = [[0 for column in range(vertices)] for row in
range(vertices)]
        self.V = vertices
    def isSafe(self, v, pos, path):
        """
        Input: v is the vertex to be checked, pos is the current position in the
path, path is the current path
        Output: True if the vertex is safe to be included in the path, False
otherwise
        Function: Checks if the vertex is safe to be included in the path
        """
        if self.graph[path[pos-1]][v] == 0:
            return False
        for vertex in path:
            if vertex == v:
                return False
        return True
    def hamCycle(self, path, pos):
        """
        Input: path is the current path, pos is the current position in the path
        Output: True if the path is a Hamiltonian cycle, False otherwise
        Function: Checks if the path is a Hamiltonian cycle
        """
        if pos == self.V:
            if self.graph[path[pos-1]][path[0]] == 1:
                return True
            else:
                return False
```

```
        for v in range(1, self.V):
            if self.isSafe(v, pos, path):
                path[pos] = v
                if self.hamCycle(path, pos+1):
                    return True
                path[pos] = -1
        return False
    def hamCycleUtil(self):
        """
        Input: none
        Output: True if the graph is a Hamiltonian cycle, False otherwise
        Function: Checks if the graph is a Hamiltonian cycle
        """
        path = [-1] * self.V
        path[0] = 0
        if self.hamCycle(path, 1) == False:
            print("Solution does not exist")
            return False
        self.printSolution(path)
        return True

    def printSolution(self, path):
        """
        Input: path is the current path
        Output: none
        Function: Prints the path
        """
        for vertex in path:
            print(vertex, end=" ")
        print(path[0],"\n")

g1 = Graph(5)
g1.graph = [[0, 1, 0, 1, 0],[1, 0, 1, 1, 1],[0, 1, 0, 0, 1],[1, 1, 0, 0, 1],[0, 1,
1, 1, 0]]
g1.hamCycleUtil()
```

2.1.2.3 Demo

```
g1 = Graph(5)
g1.graph = [[0, 1, 0, 1, 0],[1, 0, 1, 1, 1],[0, 1, 0, 0, 1],[1, 1, 0, 0, 1],[0, 1, 1, 1, 0]]
g1.hamCycleUtil() #0 1 2 4 3 0

g1 = Graph(4)
g1.graph = [[0, 1, 0, 1],[1, 0, 1, 0],[0, 1, 0, 1],[1, 0, 1, 0]]
g1.hamCycleUtil() #0 1 2 3 0

g1 = Graph(6)
```

```
g1.graph = [[0, 1, 0, 0, 0, 0],[1, 0, 1, 1, 0, 0],[0, 1, 0, 0, 1, 0],[0, 1, 0, 0, 0, 1],[0, 0, 1, 0, 0, 1],[0, 0, 0, 1, 1, 0]]
g1.hamCycleUtil() #0 1 2 3 0
```

2.1.3 Subset-sum problem

2.1.3.1 Ý tưởng

Ta sẽ thiết kế thuật toán quay lui để giải bài toán này. Ý tưởng thuật toán dựa trên nhận xét sau: Xét một phần tử x thuộc X, tồn tại một dãy con có tổng bằng T nếu một trong hai điều kiện sau là đúng:

- Tồn tại một tập con của X \ {x} có tổng bằng T-x
- Tồn tại một tập con của X \ {x} có tổng bằng T

2.1.3.2 Thuật toán Subset-sum Problem trên Python3

```
def isSubsetSum(st, n, sm):
    """
    input: st: set of numbers, n: size of set, sm: sum
    output: True if there is a subset of st with sum sm, False otherwise
    function: subset sum problem
    """
    subset = ([[False for i in range(sm + 1)] for i in range(n + 1)])
    for i in range(n+1):
        subset[i][0] = True
    for i in range(1, sm + 1):
        subset[0][i] = False
    for i in range(1, n + 1):
        for j in range(1, sm + 1):
            if j < st[i - 1]:
                subset[i][j] = subset[i - 1][j]
            if j >= st[i - 1]:
                subset[i][j] = subset[i - 1][j] or subset[i - 1][j - st[i - 1]]
    return subset[n][sm]

set = [3, 34, 4, 12, 5, 2]
sum = 9
n = len(set)
if(isSubsetSum(set, n, sum)):
    print("Found a subset with given sum")
else:
    print("No subset with given sum")
```

2.1.3.3 Demo

```
set = [3, 34, 4, 12, 5, 2]
sum = 9
n = len(set)
if(isSubsetSum(set, n, sum)):
   print("Found a subset with given sum")
else:
   print("No subset with given sum")

#Found a subset with given sum
#-----------------------

set = [3, 34, 4, 12, 5, 2]
sum = 6
n = len(set)
if(isSubsetSum(set, n, sum)):
   print("Found a subset with given sum")
else:
   print("No subset with given sum")

#Found a subset with given sum
#-----------------------

set = [3, 34, 4, 12, 5, 2]
sum = 1
n = len(set)
if(isSubsetSum(set, n, sum)):
   print("Found a subset with given sum")
else:
   print("No subset with given sum")

#No subset with given sum
```

2.2 Branch and Bound

2.2.1 Assignment Problem

2.2.1.1 Ý tưởng

Quy tắc lựa chọn cho nút tiếp theo trong BFS và DFS là "Blind". tức là quy tắc lựa chọn không đưa ra bất kỳ ưu tiên nào cho một nút có cơ hội rất tốt để đưa tìm kiếm đến một nút một cách nhanh chóng. Việc tìm kiếm giải pháp tối ưu thường có thể được tăng tốc bằng cách sử dụng hàm xếp hạng “thông minh”, còn được gọi là hàm chi phí gần đúng để tránh tìm kiếm trong các cây con không chứa giải pháp tối ưu. Nó tương tự như tìm kiếm giống BFS nhưng với

một tối ưu hóa chính. Thay vì tuân theo thứ tự FIFO, chúng ta chọn một nút trực tiếp với chi phí thấp nhất. Chúng ta có thể không nhận được giải pháp tối ưu bằng cách theo dõi nút với chi phí ít hứa hẹn nhất, nhưng nó sẽ mang lại cơ hội rất tốt để tìm kiếm nhanh chóng đến một nút.

Có hai cách tiếp cận:

- Đối với mỗi công nhân, chúng tôi chọn công việc với chi phí tối thiểu từ danh sách các công việc chưa được giao (lấy mục nhập tối thiểu từ mỗi hàng).
- Đối với mỗi công việc, chúng tôi chọn một công nhân có chi phí thấp nhất cho công việc đó từ danh sách công nhân chưa được phân công (lấy mục nhập tối thiểu từ mỗi cột).

2.2.2 Knapsack Problem

2.2.2.1 Ý tưởng

Danh sách đồ vật được sắp xếp theo thứ tự giảm dần của đơn giá. Xét các đồ vật theo thứ tự này.

B1. Nút gốc biểu diễn cho trạng thái ban đầu của ba lô, ở đó ta chưa chọn vật nào. Tổng giá trị các vật được chọn bằng 0. Cận trên của nút gốc bằng trọng lượng của balo nhân với đơn giá lớn nhất.

B2. Nút gốc sẽ có các nút con tương ứng với các khả năng chọn đồ vật có đơn giá lớn nhất (phân nhánh).

Với mỗi nút con ta tính như sau:

- Tổng giá trị = Tổng giá trị (nút cha) + số đồ vật được chọn * giá trị mỗi vật
- Trọng lượng balo = trọng lượng lượng (nút cha) – số đồ vật được chọn * trọng lượng mỗi vật.

- Cận trên = tổng giá trị + trọng lượng của balo * đơn giá của vật kế tiếp.

B3. Trong các nút con, ta sẽ ưu tiên phân nhánh cho nút nào có cận trên lớn hơn trước. Các con của nút này tương ứng với khả năng chọn đồ vật có đơn giá lớn tiếp theo. Với mỗi nút, xác định lại các thông số tổng giá trị, trọng lượng balo, cận trên theo công thức đã nói trong B2.

Lặp lại B3 và có thể tìm thấy một số phương án, đối với những nút có cận trên nhỏ hơn hoặc bằng giá trị lớn nhất tạm thời của một phương án đã tìm thấy thì ta không cần phân nhánh cho nút đó nữa.

Nếu tất cả các nút đều được phân nhánh hoặc bị cắt bỏ thì phương án có giá trị lớn nhất là phương án cần tìm.

2.2.2.2 Thuật toán Knapsack Problem trên Python3

```
class Priority_Queue:
    def __init__(self):
        self.pqueue = []
        self.length = 0

    def insert(self, node):
        for i in self.pqueue:
            get_bound(i)
        i = 0
        while i < len(self.pqueue):
            if self.pqueue[i].bound > node.bound:
                break
            i+=1
        self.pqueue.insert(i,node)
        self.length += 1

    def print_pqueue(self):
        for i in list(range(len(self.pqueue))):
            print ("pqueue",i, "=", self.pqueue[i].bound)

    def remove(self):
        try:
            result = self.pqueue.pop()
            self.length -= 1
        except:
            print("Priority queue is empty, cannot pop from empty list.")
        else:
            return result
```

```
class Node:
    def __init__(self, level, profit, weight):
        self.level = level
        self.profit = profit
        self.weight = weight
        self.items = []


def get_bound(node):
    """
    Input: node is a node object
    Output: node.bound is the bound of the node
    Function: knapsack bound is the maximum profit that can be obtained by
    """
    if node.weight >= W:
        return 0
    else:
        result = node.profit
        j = node.level + 1
        totweight = node.weight
        while j <= n-1 and totweight + w[j] <= W:
            totweight = totweight + w[j]
            result = result + p[j]
            j+=1
        k = j
        if k<=n-1:
            result = result + (W - totweight) * p_per_weight[k]
        return result

n = 4
W = 16
p = [40, 30, 50, 10]
w = [2, 5, 10, 5]
p_per_weight = [20, 6, 5, 2]
nodes_generated = 0
pq = Priority_Queue()

v = Node(-1, 0, 0)
nodes_generated+=1
maxprofit = 0
v.bound = get_bound(v)


pq.insert(v)

while pq.length != 0:

    v = pq.remove()
    if v.bound > maxprofit:
        u = Node(0, 0, 0)
```

```
        nodes_generated+=1
        u.level = v.level + 1
        u.profit = v.profit + p[u.level]
        u.weight = v.weight + w[u.level]
        u.items = v.items.copy()
        u.items.append(u.level)
        if u.weight <= W and u.profit > maxprofit:
            #update maxprofit
            maxprofit = u.profit
            bestitems = u.items
        u.bound = get_bound(u)
        if u.bound > maxprofit:
            pq.insert(u)
        u2 = Node(u.level, v.profit, v.weight)
        nodes_generated+=1
        u2.bound = get_bound(u2)
        u2.items = v.items.copy()
        if u2.bound > maxprofit:
            pq.insert(u2)


print("\nEND maxprofit = ", maxprofit, "nodes generated = ", nodes_generated)
print("bestitems = ", bestitems)
```

2.2.3 Traveling Salemen Problem

2.2.3.1 Ý tưởng

Người giao hàng xuất phát tại thành phố nào đó và cần đi giao hàng đến N thành phố và trở về thành phố ban đầu, mỗi thành phố chỉ qua một lần.

Khoảng cách từ thành phố này đến thành phố khác là xác định được.

Giả thiết rằng mỗi thành phố đều có đường đi đến các thành phố còn lại.

Khoảng cách giữa hai thành phố: khoảng cách địa lý/cước phí/thời gian di chuyển. Ta gọi chung là độ dài.

Tìm một chu trình sao cho tổng độ dài các cạnh là nhỏ nhất.

Ta có:

- Nút gốc: xuất phát từ thành phố nào đó.
- Nút gốc có n-1 nút con, tương ứng với các khả năng đi ra từ thành phố đầu tiên.

- Mỗi nút con bậc 1 lại có n-2 nút con ở bậc 2, tương ứng khả năng đi ra từ các thành phố bậc 1.

- Đến bậc thứ n-1: đã có n-1 cạnh, đi đến thành phố cuối cùng, quay về thành phố ban đầu sẽ còn một phương án.

Đây là bài toán tìm min nên chúng ta sẽ có công thức để tính cận dưới như sau:

- Nút gốc: cận dưới = n*độ dài của cạnh nhỏ nhất.

- Các nút khác: cận dưới = tổng giá trị từ nút gốc đến nút đó + (n-i) độ dài cạnh nhỏ nhất trong số các cạnh chưa sử dụng, với I là các cạnh đã sử dụng

- Tổng giá trị sẽ bằng tổng giá trị các cạnh từ thành phố xuất phát đến các thành phố khác.

2.2.3.2 Thuật toán TSP trên Python3

```
import math
maxsize = float('inf')

def copyToFinal(curr_path):
    final_path[:N + 1] = curr_path[:]
    final_path[N] = curr_path[0]

def firstMin(adj, i):
    min = maxsize
    for k in range(N):
        if adj[i][k] < min and i != k:
            min = adj[i][k]

    return min

def secondMin(adj, i):
    first, second = maxsize, maxsize
    for j in range(N):
        if i == j:
            continue
        if adj[i][j] <= first:
            second = first
            first = adj[i][j]

        elif(adj[i][j] <= second and
             adj[i][j] != first):
            second = adj[i][j]
```

```
    return second

def TSPRec(adj, curr_bound, curr_weight,
              level, curr_path, visited):
    global final_res

    if level == N:

        if adj[curr_path[level - 1]][curr_path[0]] != 0:

            curr_res = curr_weight + adj[curr_path[level - 1]]\
                                         [curr_path[0]]
            if curr_res < final_res:
                copyToFinal(curr_path)
                final_res = curr_res
        return

    for i in range(N):

        if (adj[curr_path[level-1]][i] != 0 and
                            visited[i] == False):
            temp = curr_bound
            curr_weight += adj[curr_path[level - 1]][i]
            if level == 1:
                curr_bound -= ((firstMin(adj, curr_path[level - 1]) +
                                firstMin(adj, i)) / 2)
            else:
                curr_bound -= ((secondMin(adj, curr_path[level - 1]) +
                                 firstMin(adj, i)) / 2)
            if curr_bound + curr_weight < final_res:
                curr_path[level] = i
                visited[i] = True

                TSPRec(adj, curr_bound, curr_weight,
                       level + 1, curr_path, visited)
            curr_weight -= adj[curr_path[level - 1]][i]
            curr_bound = temp

            visited = [False] * len(visited)
            for j in range(level):
                if curr_path[j] != -1:
                    visited[curr_path[j]] = True

def TSP(adj):
    curr_bound = 0
    curr_path = [-1] * (N + 1)
    visited = [False] * N
    for i in range(N):
        curr_bound += (firstMin(adj, i) +
```

```
                        secondMin(adj, i))

    curr_bound = math.ceil(curr_bound / 2)

    visited[0] = True
    curr_path[0] = 0
    TSPRec(adj, curr_bound, 0, 1, curr_path, visited)

adj = [[0, 10, 15, 20],
       [10, 0, 35, 25],
       [15, 35, 0, 30],
       [20, 25, 30, 0]]
N = 4
final_path = [None] * (N + 1)
visited = [False] * N
final_res = maxsize

TSP(adj)

print("Minimum cost :", final_res)
print("Path Taken : ", end = ' ')
for i in range(N + 1):
    print(final_path[i], end = ' ')
```

2.3 Approximation

2.3.1 Travel Salemen Problem

2.3.1.1 Ý tưởng

Không mất tính tổng quát ta giả sử người du lịch xuất phát từ thành phố 1. Mỗi chu trình đường đi Tp1,Tpi1,Tpi2,…Tpin,Tp1 có thể đặt tương ứng 1-1 với một hoán vị (i1,i2,…in) của 2,3,…n. Gọi C(ik – i1) là chi phí đi từ thành phố ik đến thành phố i1. Khi đó chi phí của một chi trình là tổng cách chi phí từng chặng.

Đặt f(x) = C(1-i1)+C(i1-i2)+…+(Cn-1 – in) C(n-1)

Ký hiệu D là tất cả các hoán vị của n-1 số 2,3,…n có thể phát biểu bài toán dưới dạng sau:

{f(x) => min; x thuộc D}

2.3.1.2 Thuật toán TSP trên Python3

```
from itertools   import combinations
from functools   import lru_cache as cache

def join_endpoints(endpoints, A, B):
    "Join segments [...,A] + [B,...] into one segment. Maintain `endpoints`."
    Aseg, Bseg = endpoints[A], endpoints[B]
    if Aseg[-1] is not A: Aseg.reverse()
    if Bseg[0]  is not B: Bseg.reverse()
    Aseg += Bseg
    del endpoints[A], endpoints[B]
    endpoints[Aseg[0]] = endpoints[Aseg[-1]] = Aseg
    return Aseg
def distance(A, B): return abs(A - B)
def reverse_segment_if_improvement(tour, i, j):
    "If reversing tour[i:j] would make the tour shorter, then do it."
    # Given tour [...A,B...C,D...], consider reversing B...C to get
[...A,C...B,D...]
    A, B, C, D = tour[i-1], tour[i], tour[j-1], tour[j % len(tour)]
    # Are old links (AB + CD) longer than new ones (AC + BD)? If so, reverse
segment.
    if distance(A, B) + distance(C, D) > distance(A, C) + distance(B, D):
        tour[i:j] = reversed(tour[i:j])
        return True
@cache()
def subsegments(N):
    "Return (i, j) pairs denoting tour[i:j] subsegments of a tour of length N."
    return [(i, i + length)
            for length in reversed(range(2, N))
            for i in reversed(range(N - length + 1))]
def improve_tour(tour):
    "Try to alter tour for the better by reversing segments."
    while True:
        improvements = {reverse_segment_if_improvement(tour, i, j)
                        for (i, j) in subsegments(len(tour))}
        if improvements == {None}:
            return tour

def greedy_tsp(cities):
    """Go through links, shortest first. Use a link to join segments if
possible."""
    endpoints = {C: [C] for C in cities} # A dict of {endpoint: segment}
    for (A, B) in shortest_links_first(cities):
        if A in endpoints and B in endpoints and endpoints[A] != endpoints[B]:
            new_segment = join_endpoints(endpoints, A, B)
            if len(new_segment) == len(cities):
                return new_segment

def improve_greedy_tsp(cities): return improve_tour(greedy_tsp(cities))
```

```
def shortest_links_first(cities):
    "Return all links between cities, sorted shortest first."
    return sorted(combinations(cities, 2), key=lambda link: distance(*link))
```