پاسخ سوال اول :

```
# https://www.geeksforgeeks.org/minimum-sum-product-two-arrays/
# :)))
# Python program to find
# minimum sum of product
# of two arrays with k
# operations allowed on
# first array.

# Function to find the minimum product
def min_product(a,b,n,k):

    diff = 0
    res = 0
    for i in range(n):

        # Find product of current
        # elements and update result.
        pro = a[i] * b[i]
        res = res + pro

        # If both product and
        # b[i] are negative,
        # we must increase value
        # of a[i] to minimize result.
        if (pro < 0 and b[i] < 0):
            temp = (a[i] + 2 * k) * b[i]

        # If both product and
        # a[i] are negative,
        # we must decrease value
        # of a[i] to minimize result.
        elif (pro < 0 and a[i] < 0):
            temp = (a[i] - 2 * k) * b[i]

        # Similar to above two cases
        # for positive product.
        elif (pro > 0 and a[i] < 0):
```

```
            temp = (a[i] + 2 * k) * b[i]
        elif (pro > 0 and a[i] > 0):
            temp = (a[i] - 2 * k) * b[i]

        # Check if current difference
        # becomes higher
        # than the maximum difference so far.
        d = abs(pro - temp)

        if (d > diff):
            diff = d
    return res - diff
```

پاسخ سوال دوم :

رویکرد:

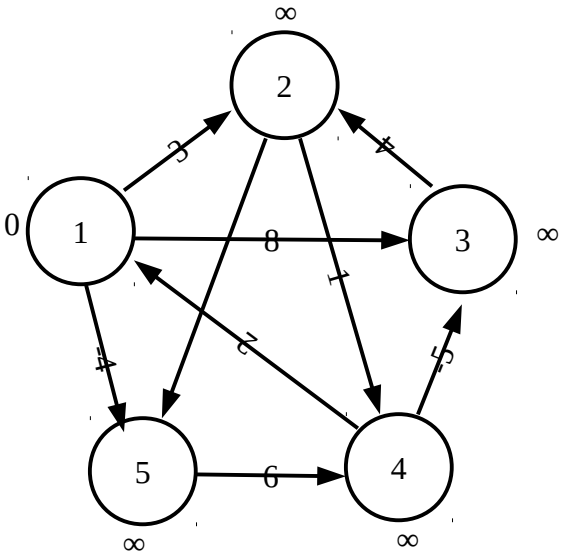
- عدم تعادل ۱ ها و ۰ ها در LSB اعداد داده شده را میتوان در بین 2^{n-1} عدد صحیح مشاهده کرد. از آنجا که یک عدد از دست رفته است، یا یک ۰ یا یک ۱ از LSB گم شده است. اگر شماره ای که از دست رفته است، ۰0 باشد، تعداد ۱ ها بیشتر یا برابر با تعداد ۰ ها خواهد بود. اگر LSB گمشده ۱ باشد، تعدا ۱ ها کمتر از تعداد ۰ ها است.
- از مرحله قبل، می توان به راحتی LSB عدد گمشده را تعیین کرد.
- پس از تعیین LSB عدد گم شده ، تمام اعداد را که دارای LSB متفاوت از عدد گمشده است، صرف نظر کنید، به عنوان مثال اگر عدد گمشده 0 = LSB باشد، سپس تمام اعداد را با 1 = LSB و برعکس حذف کنید.
- ادامه روند را از مرحله یک به طور کامل ادامه دهید و برای LSB بعدی مجددا متوقف شوید.
- فرایند فوق را ادامه دهید تا تمام بیت های عدد گم شده پیدا شوند.

پیچیدگی زمانی $O(2^n)$

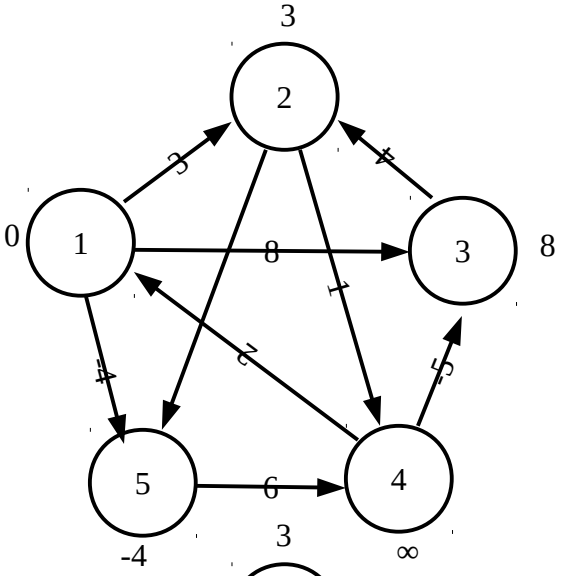
پیچیدگی حافظه $O(n)$ >> چون فقط یه حافظه به طول n برای حفظ عدد گم شده داریم

پاسخ سوال سوم :

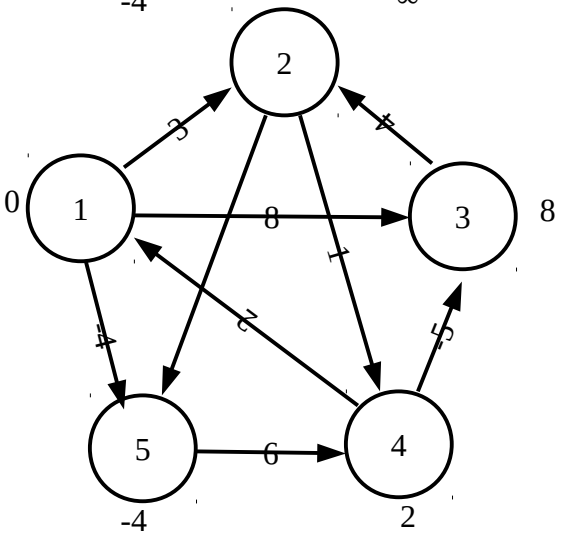
مرحله اول

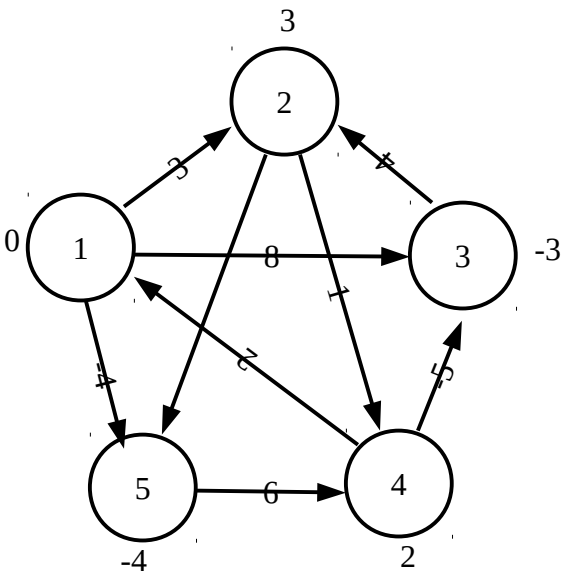


مرحله دوم



مرحله سوم





مرحله چهارم

پاسخ سوال چهارم :

Ford-Fulkerson Algorithm

1) Start with initial flow as 0.
2) While there is an augmenting path from source to sink.
Add this path-flow to flow.
3) Return flow.

چگونه الگوریتم فوق را پیاده سازی کنیم؟

ابتدا مفهوم گراف باقی مانده را تعریف کنیم که برای درک پیاده سازی مورد نیاز است.

نمودار باقیمانده از یک شبکه جریان یک گراف است که جریان احتمالی اضافی را نشان می دهد. اگر مسیری از منبع به مقصد در نمودار باقی مانده وجود دارد، پس می توان جریان را اضافه کرد. هر لبه یک گراف باقی مانده، دارای یک مقدار به نام ظرفیت باقی مانده است که برابر با ظرفیت اصلی جریان جریان منفی است. ظرفیت باقی مانده اساسا ظرفیت فعلی لبه است.

اکنون در مورد جزئیات پیاده سازی : ظرفیت باقیمانده ۰ است اگر بین دو رأس گراف باقی مانده هیچ لبه وجود نداشته باشد. ما می توانیم نمودار باقیمانده را به عنوان گراف اولیه به صورت اولیه تنظیم کنیم زیرا هیچ جریان اولیه ای وجود ندارد و در ابتدا ظرفیت باقیمانده با ظرفیت اصلی برابر است. برای پیدا کردن یک مسیر افزایشی،

می توانیم یک BFS یا DFS از گراف باقی مانده را انجام دهیم. مثلا با استفاده از BFS، می توانیم دریابیم که آیا مسیر از منبع به مقصد وجود دارد. BFS همچنین آرایه parent را ایجاد می کند. با استفاده از آرایه parent ،، ما از طریق مسیر یافت شده عبور می کنیم و با پیدا کردن حداقل ظرفیت باقی مانده در مسیر، مسیر جریان ممکن را از طریق این مسیر پیدا می کنیم. بعدا مسیر یافت شده را به جریان کلی اضافه می کنیم.

مهم این است که ما باید ظرفیت های باقی مانده را در نمودار باقیمانده به روز کنیم. ما جریان مسیر را از تمام لبه ها در طول مسیر تفریق می کنیم و جریان مسیر را در امتداد لبه های معکوس اضافه می کنیم. ما نیاز داریم جریان مسیر را در امتداد لبه های معکوس اضافه کنیم؛ زیرا ممکن است بعدا نیاز به جریان، در جهت معکوس پیدا کنیم.

پاسخ سوال ششم :

منبع : https://en.wikipedia.org/wiki/Floyd%E2%80%93Warshall_algorithm

الگوریتم Floyd Warshall

ماتریس راه حل همانند ماتریس گراف ورودی را به عنوان قدم اول راه می اندازیم. سپس ماتریس راه حل را با در نظر گرفتن تمام رأس ها به عنوان یک رأس متوسط می پردازیم. ایده این است که به صورت یکپارچه همه رأس ها را انتخاب کرده و همه کوتاه ترین مسیرها را که شامل رأس انتخاب شده به عنوان یک رأس متوسط در کوتاهترین مسیر است به روزرسانی کنید. هنگامیکه تعداد ارقام k را به عنوان یک رأس متوسط می گذاریم، در حال حاضر رأس ها {0، 1، 2، .. k-1} را به صورت رأس های متوسط می بینیم. برای هر جفت (i، j) به ترتیب از منبع و مقصد به ترتیب، دو مورد ممکن است.

1) k یک رشته میانجی در کوتاهترین مسیر از i به j نیست. ما مقدار dist [i] [j] را همانطور که هست حفظ می کنیم.

2) k یک حلقهٔ متوسط در کوتاهترین مسیر از i به j است. مقدار [dist] [i][j] را آپدیت میکنیم با مقدار dist [i][k] + dist [k][j] اگر dist[i][j] > dist[i][k] + dist[k][j]

کاربرد های Floyd–Warshall algorithm :

- Shortest paths in directed graphs (Floyd's algorithm).

- Transitive clouser of directed graphs (Warshall's algorithm). In Warshall's original formulation of the algorithm, the graph is unweighted and represented by a Boolean adjacency matrix. Then the addition operation is replaced by logical conjunction (AND) and the minimum operation by logical disjunction (OR).
- Finding a regular expression denoting the regular language accepted by a finit automata (Kleem’s algorithm, a closely related generalization of the Floyd–Warshall algorithm)
- Inversion of real matrics (Gauss–Jordan algorithm)
- Optimal routing. In this application one is interested in finding the path with the maximum flow between two vertices. This means that, rather than taking minima as in the pseudocode above, one instead takes maxima. The edge weights represent fixed constraints on flow. Path weights represent bottlenecks; so the addition operation above is replaced by the minimum operation.
- Fast computation of Pathfinder networks.
- Widest paths/Maximum bandwidth paths
- Computing canonical form of difference bound matrices (DBMs)
- Computing the similarity between graphs

پاسخ سوال هفتم :

```
# https://www.geeksforgeeks.org/fractional-knapsack-problem/
# :)))
# Python3 program to solve
# fractional  Knapsack Problem
class ItemValue:

	"""Item Value DataClass"""
	def __init__(self, wt, val, ind):
		self.wt = wt
		self.val = val
		self.ind = ind
		self.cost = val // wt

	def __lt__(self, other):
		return self.cost < other.cost

# Greedy Approach
class FractionalKnapSack:

	"""Time Complexity O(n log n)"""
	@staticmethod
```

```
def getMaxValue(wt, val, capacity):

    """function to get maximum value """
    iVal = []
    for i in range(len(wt)):
        iVal.append(ItemValue(wt[i], val[i], i))

    # sorting items by value
    iVal.sort(reverse = True)

    totalValue = 0
    for i in iVal:
        curWt = int(i.wt)
        curVal = int(i.val)
        if capacity - curWt >= 0:
            capacity -= curWt
            totalValue += curVal
        else:
            fraction = capacity / curWt
            totalValue += curVal * fraction
            capacity = int(capacity - (curWt * fraction))
            break
    return totalValue
```

پیچیدگی زمانی : O(nlogn)

اول مرتب کردن با توجه به "سود به نسبت وزن" به زمان nlogn احتیاج دارد.

سپس ما به یک بررسی نیاز داریم تا بتوانیم حداکثر "سود به نسبت وزن" را پیدا کنیم که به زمان n احتیاج دارد.

پس در کل به زمان nlogn + n نیاز داریم یعنی نتیجتاً پیچیدگی زمانی این الگوریتم O(nlogn) **است**