

H2 Databricks VO

H3 Coding Part

H5 coding1: cache implementation

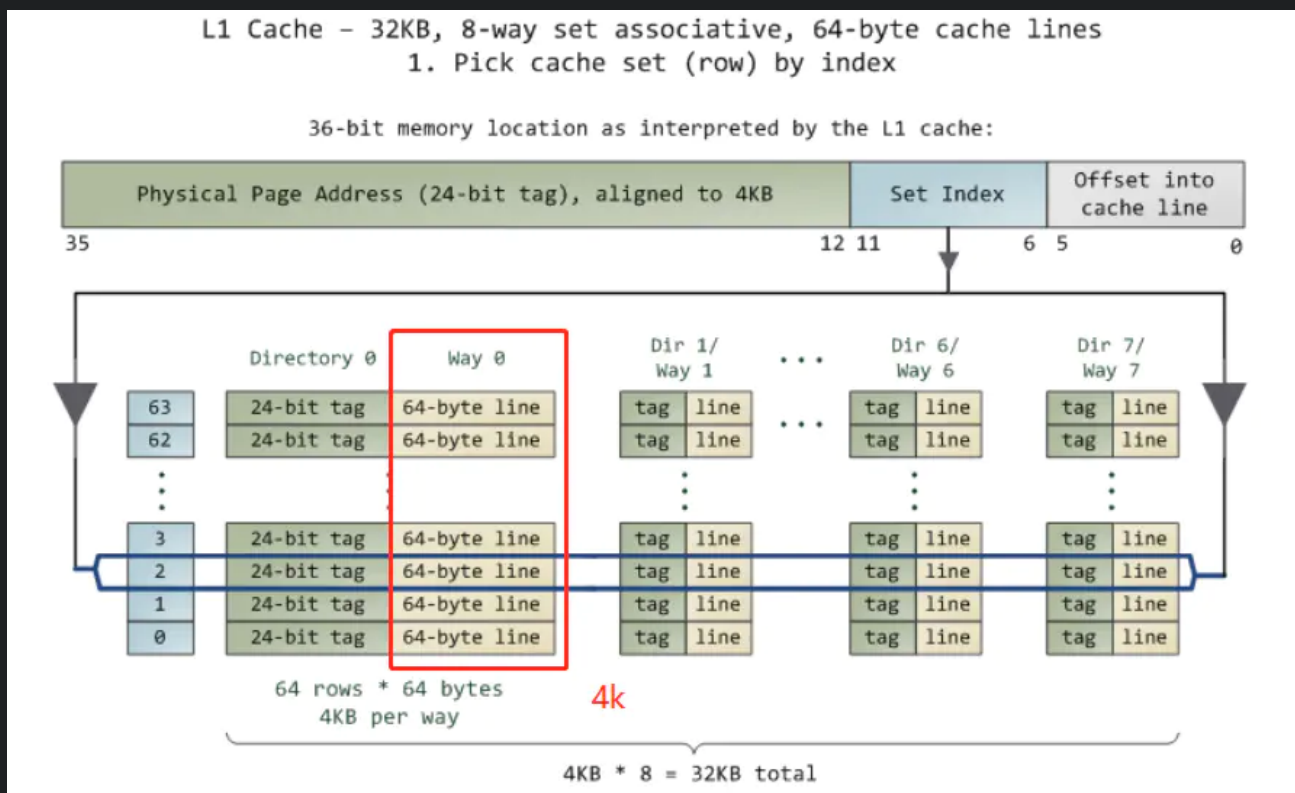
从简单的direct map cache开始，你需要存储cache的tag值和value。然后读到的时候再检测。同时你需要一个init variable来看cache是否被init，和dirty bit来看cache是否被写，如果被写才需要evict的时候写入RAM。

然后其实init和dirty bit都可以用bit operation encode到tag里面，然后你需要实现一些bit or，bit add，mask之类的东西。

然后再实现一个fully associative cache，唯一的区别就是，现在每个cache line都可以存到任意entry。但实现到这一步的时候你需要注意tag不再可以encode init & dirty bit。！ 为你需要整个的地址来当tag。

再实现一个set associative cache，就是再把cache分成不同的set。

补充知识：



l1 l2 not share, l3 share by all cpus

cache line: unit for cache

每条cache line 都有两个标志位

valid bit:表示cache line中数据是否有效（例如：1代表有效；0代表无效）。当系统刚启动时，cache中的数据都应该是无效的。

dirty bit：表示cache line里面的数据是否和下一级存储一致。=0非dirty,和下一级存储一致，=1 dirty,和下一级存储不一致。

X86: 64 set(row) * 8 way(column) -> 512 cell, 每个cell 64 byte, L1 cache共32kb

24-bit tag: 记录这一行的哪一个cell存储了对应地址的数据

当CPU访问一个内存的时候，通过内存中间的6bits定位在哪个set,再通过24bits定位响应的cacheline。类似Hash table 一样，显示O(1)搜索，然后进入冲突里搜索

确定了在哪个set后，在硬件的支持下，并行也就是同时在8个way里去寻找tag。

找到则cache命中。（注意这里，**cache只要命中了，就是命中64字节**）

否则转移到L2 cache, 若还没命中到L3 中寻找。L2 L3的原理和L1一样的，只不过会大一些比如64KB，16ways。

总共有4096个row，每个way 有256Kb，总大小4M。18bits for tags, and 12bits for set index.

H6 一致性

现代一个物理CPU一般都会有多个物理core，每个物理core在程序运行时可以支持一个并发，利用超线程技术可以支持两个并发，每个物理core都拥有自己的L1、L2 cache，一个物理CPU上所有的物理core共享一个L3 cache。因为每个core都有自己的cache，所以一个cache line可能被映射到多个core的cache中，这就会有cache不一致的问题，如果这个时候其中一个core修改了cache line，那就会有多个cache line不一致的问题。

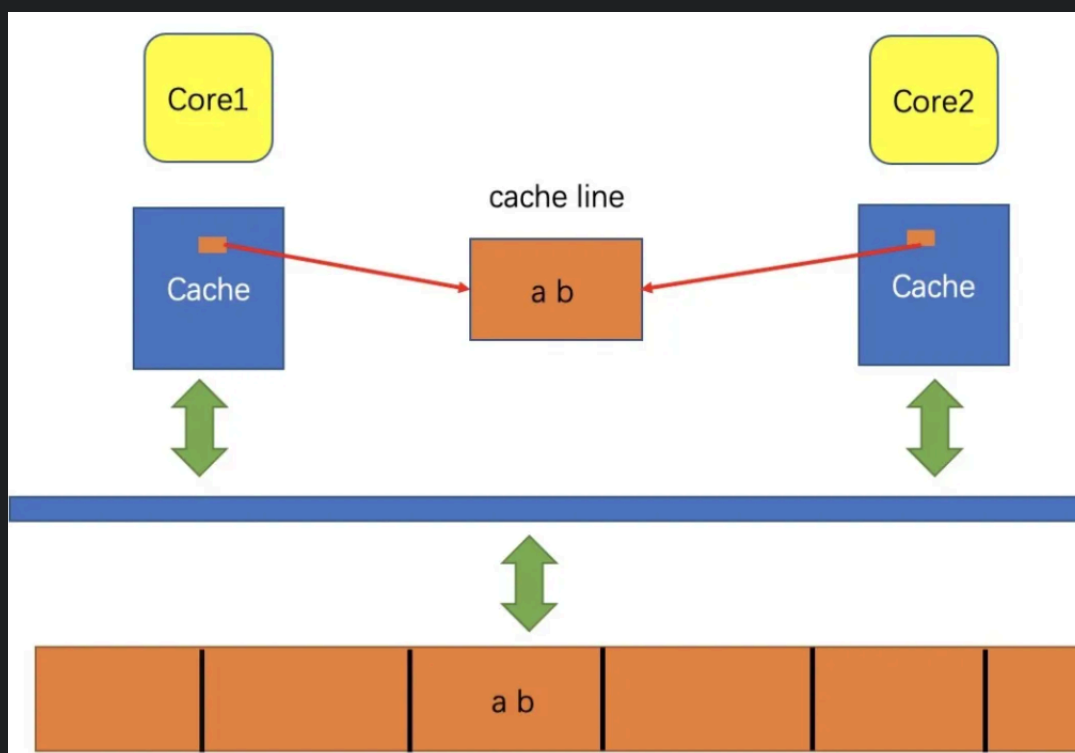
MESI（Modify+Exclusive+Shared+Invalid）cache一致性协议，这个是Intel的协议，不同的厂商有不同的协议，这里是给了缓存行四种状态：

1. Modified--被修改了，尚未同步到内存中，其他core如果想读写这块cache的内容，必须要等到cache写回内存中之后再重新映射
2. Exclusive--cache和内存中的数据一致，但是没有其他core读到这个cache line
3. Shared--cache和内存中的数据一致，并且也有其他core读到这个cache line

4. Invalid--cache已经失效了，一般是其他core对cache做了修改，将cache line状态修改为Modified导致的

这些协议是怎么生效的，如上图中x被改了之后他给自己标记成Modified，然后数据写回内存后通知其他核，给他们的这个缓存行状态改成Invalid,意思就是告诉他们我改过了，你们这个都无效了，如果需要用到最新数据，重新去内存中取。

H5 伪共享False Sharing



MESI虽然解决了cache一致性的问题，但是引入了一个cache伪共享的问题。首先来解释一下什么是伪共享。

因为cache line的最小单位是64字节，所以在load内存的时候很有可能会load到你需要的部分，如下图所示。core1需要修改变量a，core2需要修改变量b，但是a和b正好在一个cache line中，根据MESI协议，core1先发起操作，此时没有其他cache访问该cache line，core1将cache line修改为Exclusive状态，修改完成后修改为Modified，这个时候core2发起指令修改变量b，因为MESI协议的限制，所以需要等core1将cache line写会内存才能读，这样导致了两个core互相竞争一个cache line，彼此相互影响，变成了串程序，降低了并发性，这就是cache line伪共享，虽然不同并发访问的是不同的变量，但是因为变量在一个cache line中，无法实现真正的并发访问。

using align to store just a in one cache line -> classic tradeoff between time and space. We can do more in real business scenario.

H5 coding2: revenue

coding是祖传面试题，revenue。题目大意是databricks有一堆customer，每个customer有对应的revenue。需要支持三个API：

`int insert(int revenue)` /* Insert a new customer to the system, with the given revenue. Return the new customer ID /

`int insert(int revenue, int referrer)` / Insert a new customer to the system, with the given revenue, and revenue also added to the referrer. Return the new customer ID /

`vector get_lowest_k_by_total_revenue(int k, int min_total_revenue)` / Get the k customers with the lowest revenue but have revenue above the min_total_revenue. Return the vector of customer IDs that satisfy the condition */

实现思路：

1. write-optimized 如果大部分时间都是insert的话，`get_lowest_k_by_total_revenue`跑的慢不慢根本无所谓。所以你可以直接用一个array，然后`get_lowest_k_by_total_revenue`的时候遍历全部。
2. read-optimized 如果大部分时间我们都想读，也就是`get_lowest_k_by_total_revenue`。那我们最好maintain一个sorted array。然后每次get的时候binary search for the starting point。

follow-up:

在讲述完上面的思路之后，面试官让实现了第二种，需要当场跑test cases。在test cases过完之后，也是祖传follow-up，如果我们想算indirect-referer的revenue怎么办。比如 2 refer 1, 1 refer 0, 然后我们想算0的overall revenue。输入还包括一个depth，当depth=0就是只算自己，=1就是revenue再加上direct referer。

同样是两种思路：

3. write-optimized 给每个customer记录一个children list，然后只有在get revenue的时候再去用 DFS/BFS 算total revenue
4. read-optimized 每个customer都maintain一个total revenue list, total

revenue list每个element对应不同的depth。这个需要你每次insert的时候maintain这个结构，但你在get revenue的时候就可以直接读，不用计算。

✨ treemap

```
from sortedcontainers import SortedList
class Revenue:
    def __init__(self):
        self.id_cnt = 0
        # id -> revenue
        self.revenue_dict = {}
        # sorted list of (revenue, id)
        self.sorted_revenue_list = SortedList()

    def insert(self, revenue: int, referrer: int = None):
        """
        Insert a new customer to the system, with the
        given revenue,
        and revenue also added to the referrer. Return
        the new customer ID
        """
        self.sorted_revenue_list.add((revenue,
self.id_cnt))
        self.revenue_dict[self.id_cnt] = revenue
        self.id_cnt += 1

        if referrer is not None and referrer in
self.revenue_dict:
            new_revenue = self.revenue_dict[referrer] +
revenue

        self.sorted_revenue_list.remove((self.revenue_dict[refer
rer], referrer))
```

```

        self.revenue_dict[referrer] += revenue
        self.sorted_revenue_list.add((new_revenue,
referrer))

    return self.id_cnt - 1

    def get_lowest_k_by_total_revenue(self, k: int,
min_total_revenue: int):
        '''
            Get the k customers with the lowest revenue but
            have revenue above the min_total_revenue.
            Return the vector of customer IDs that satisfy
            the condition
        '''
        idx =
self.sorted_revenue_list.bisect_right((min_total_revenue,
0))

        res = []
        for i in range(idx, idx+k):
            if i >= len(self.sorted_revenue_list):
                break
            res.append(self.sorted_revenue_list[i][1])
        return res

# re = Revenue()
# print(re.insert(10))
# print(re.insert(20, 0))
# print(re.insert(40, 1))
# print(re.insert(100, 10))
# print(re.get_lowest_k_by_total_revenue(1, 35)) # ->
(2)

```

```
# print(re.get_lowest_k_by_total_revenue(2, 35)) # ->
(1, 2)

# sl = SortedList([10, 11, 13, 14])
# print(sl.bisect_left(12))
```

H5 Coding3: kv store, get/put有timestamp, 要求能统计过去5分钟的qps

```
import time
class KeyValueStore(object):
    def __init__(self):
        self.key_value = {}
        self.last_time_put = 0
        self.last_time_get = 0
        self.last_idx_put = 0
        self.last_idx_get = 0
        self.ring_buffer_put = [0] * 7
        self.ring_buffer_get = [0] * 7

    def clear(self):
        self.ring_buffer_put = [0] * 7
        self.ring_buffer_get = [0] * 7

    # Writes the given key-value pair to the map
    def put(self, key, val):
        self.key_value[key] = val
        cur_time = time.time()
        # print(cur_time)
        dif = int(cur_time - self.last_time_put)
        # print(dif)
        if dif >= 7:
            self.put_qps = 0
```

```

        self.clear()
        self.last_idx_put = 0
    else:
        for i in range(dif - 1):
            idx = (self.last_idx_put + i + 1) % 7
            self.ring_buffer_put[idx] = 0
        self.last_idx_put = (self.last_idx_put + dif)
% 7

        # print(self.last_idx_put)

    self.ring_buffer_put[self.last_idx_put] += 1
    self.last_time_put = cur_time
    print(self.ring_buffer_put)

    # Returns the average number of put calls per second
    over the past 5 minutes
    def measure_put_load(self):
        cur_time = time.time()
        # print(cur_time)
        dif = int(cur_time - self.last_time_put)
        # print(dif)
        if dif >= 7:
            return 0
        res = 0
        if dif == 0:
            res +=
self.ring_buffer_put[self.last_idx_put]
        for i in range(dif - 1):
            idx = (self.last_idx_put + 1 + i) % 7
            self.ring_buffer_put[idx] = 0
        print(self.ring_buffer_put)
        return sum(self.ring_buffer_put)
obj = KeyValueStore()

```



```
obj.put(2, 'a')
obj.put(100, 'aasdas')
time.sleep(3)
obj.put(3, 'b')
time.sleep(5)
obj.put(4, 'c')
time.sleep(3)
print(obj.measure_put_load())
```

✨ringbuffer or queue

题目：有一个map class，提供两个interface，get和put，这两个不是重点，重点要求是想求5分钟内，平均每秒钟的request数量。

初始思路：这个map需要一些global variable来记录request，然后可能会涉及multi-threading的问题，询问面试官后回复只需考虑single-thread。（但考虑更多并询问之后，个人感觉面试官对你的印象更好一点hhhh）

第一版答案：用一个queue，存request的timestamp，每次get和put的时候剔除老数据，然后计算平均每秒钟request数量。

follow-up：这个solution会存在什么问题，有什么解决方案

ans：如果request数量特别多，可能会有memory issue，解决方案->

第二版答案：用一个array, size=300，每秒钟对应一个slot，然后array是circular buffer，每个slot记录在那一秒有几个request。用circular buffer时同样涉及清除expired数据，然后再计算平均。然后在面试官的建议下再refactor code，把清除expired数据写到一个新的function，在get和put里面call。

follow-up：如果想记录更长的时间段怎么办（300个slot不够用）

ans：可以把每个bucket对应的时间变长，现在是每个bucket对应1s，之后可以变成10s或者1min

follow-up：如果想测试这个代码，你会怎么办

ans：先考虑corner case，刚好超过5分钟的case，看是否得出的答案和你想要的一致。之后可以生成简单case，人肉验证准确性。再之后可以生成random case，这个时候你可以写一个第一版答案的solution来验证准确性。！为第一版答案更加易懂，更容易人工验证其正确性。


follow-up：现在的测试是5分钟average，每个测试可能都会跑很长时间，比如跑

一天，你有什么解决方案。

ans: 可以写一个time的wrapper，让你可以自己定义时间，然后再get和put的时候获取的timestamp就是你设定的值。

H5 Coding4: cidar block, [leetcode751](#)

ip allow/deny

follow up 如果cidr描述是range怎么update (叻蔻 琦瑶 )

大家一定要看搞懂followup

我就是挂在followup了

H5 Coding5: Run Length Encoder

1 1 1 1 1 1 1 1 [1, 8]

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

有一个stream/序列，如果有8or超过8个一样的，就可以用run length encoding，比如[1,8]表示8个1，这个是个object，这里化简用了括号

如果有不一样的就要8个一组变成另一个object

要写encoder和decoder

H5 Coding6:应该是一个snapshot kv 的变种题设计一个set-like container with iterator

要求是可以达到一边iterate一边修改set elements而不影响iterator。

```
set<int> s({1,2,3});
set<int>::iterator iter = s.begin();
while (iter != s.end()) {
    if (*iter == 1) {
        s.erase(1);
        s.insert(4);
    }
    print(*iter);
    iter++;
}
```

上面的代码应该输出1 2 3但是再print一遍就是 2 3 4.

我用c++写的，用tree map 存数字和versioned history, 最后磕磕绊绊的写出来了。

也没时间讨论优化，而且还是linearly search version

H5 Coding7:惰性求值数组

lazy array，要求实现 `array.map(...).indexOf(...)`，其中map传进去一个function，indexOf返回运行了所有function之后传入值的index。要求map的操作最后再做，所以叫lazy array。For example:

```
arr = LazyArray([10, 20, 30, 40, 50])
```

```
arr.map(lambda x:x2).indexOf(40) ----> 1
```

```
arr.map(lambda x:x2).map(lambda x:x*3).indexOf(240) ----> 3
```

注意这里重新开了一个chain，上一行的map就不计算在内了

followup是优化. 考虑缓存map和indexof的执行结果

```
class LazyArray:
    def __init__(self, arr):
        self.funcs = []
        self.arr = arr

    def map(self, func):
        self.funcs.append((func, 'm'))
        return self

    def filter(self, func):
        self.funcs.append((func, 'f'))
        return self

    def indexOf(self, res):
        for i, num in enumerate(self.arr):
            for func, fType in self.funcs:
                if fType == 'm':
                    num = func(num)
```

```

        else:
            if not func(num):
                break
            if num == res:
                return i
        return -1

# arr = LazyArray([10,20,30])
# print(arr.map(lambda x: x * 2).map(lambda x: x *
3).filter(lambda x: x > 100).indexOf(180))

```

H5 Coding8: NewString

Design a class newString such that the time complexity for insertion, deletion and read is less than $O(n)$.

```

class NewString{
public:
char read(int index);
void insert(char c, int index);
void delete(int index);
}

```

```

import random

class Node(object):
    def __init__(self, key, val, level):
        self.key = key
        self.val = val
        self.forward = [None]*(level+1)

class SkipList(object):
    def __init__(self, max_lvl, P):

```

```

        self.MAXLVL = max_lvl
        self.P = P
        self.header = self.createNode(self.MAXLVL, -1,
-1)

        self.level = 0
        self.size = 0

# create new node
def createNode(self, lvl, key, val):
    n = Node(key, val, lvl)
    return n

# create random level for node
def randomLevel(self):
    lvl = 0
    while random.random() < self.P and \
        lvl < self.MAXLVL: lvl += 1
    return lvl

# insert given key in skip list
def insertElement(self, key, val):
    if key < 0 or key > self.size:
        print("fail")
        return

    # create update array and initialize it
    update = [None] * (self.MAXLVL + 1)
    current = self.header

    for i in range(self.level, -1, -1):
        while current.forward[i] and \
            current.forward[i].key < key:
            current = current.forward[i]

```

```

        update[i] = current

    current = current.forward[0]
    # Generate a random level for node
    rlevel = self.randomLevel()

    if rlevel > self.level:
        for i in range(self.level+1, rlevel+1):
            update[i] = self.header
        self.level = rlevel

    # create new node with random level generated
    n = self.createNode(rlevel, key, val)

    # insert node by rearranging references
    for i in range(rlevel+1):
        n.forward[i] = update[i].forward[i]
        update[i].forward[i] = n
    self.size += 1
    print("Successfully inserted key {}".format(key))

def deleteElement(self, key):
    # create update array and initialize it
    current = self.header
    for i in range(self.level, -1, -1):
        while current.forward[i] and \
            current.forward[i].key < key:
            current = current.forward[i]
        if current.forward[i] and
current.forward[i].key == key:

```

```

        current.forward[i] =
current.forward[i].forward[i]

    self.size -= 1
    print("Successfully deleted key {}".format(key))

def read(self, key):
    # create update array and initialize it
    current = self.header
    for i in range(self.level, -1, -1):
        while current.forward[i] and \
            current.forward[i].key < key:
            current = current.forward[i]
        if current.forward[i] and
current.forward[i].key == key:
            print("val is ", current.forward[i].val)
            return
        print("key not found")

# Display skip list level wise
def displayList(self):
    print("\n*****Skip List*****")
    head = self.header
    for lvl in range(self.level+1):
        print("Level {}: ".format(lvl), end=" ")
        node = head.forward[lvl]
        while(node != None):
            if node.forward[lvl] != None:
                print(f"{node.key}:{node.val}", end="
-> ")
            else:

```

```

        print(f"{node.key}:{node.val}",
end=" ")

        node = node.forward[lvl]
        print("")

lst = SkipList(3, 0.5)
lst.insertElement(0, 'a')
lst.insertElement(1, 'b')
lst.insertElement(2, 'c')
lst.insertElement(1, 'e')
lst.insertElement(100, 'd')
lst.insertElement(0, 'g')
lst.deleteElement(0)
lst.read(100)
lst.read(1)
lst.displayList()

```

H5 Coding9: 2d array blocks

char, x代表block, 1, 2。。之类的数字代表不同的通勤方式，每种会需要不同的时间和开销，问从起点到终点的最快的通勤方式是什么，如果有相同的返回开销最少的。下面是个栗子。

输入：

2D Grid:

I3I3ISI2IXIXI

I3I1I1I2IXI2I

I3I1I1I2I2I2I

I3I1I1I1IDI3I

I3I3I3I3I4I

I4I4I4I4I4I4I

时间array: [3, 2, 1, 1] 表示 (通勤方式) 1, 2, 3, 4 -> (时间) 3, 2, 1, 1

开销array: [0, 1, 3, 2] 表示 (通勤方式) 1, 2, 3, 4 -> (开销) 0, 1, 3, 2

输出：2 (用2可以最快的从S到D)


```

import heapq
def getMinRoute(arr, time, cost):
    M, N = len(arr), len(arr[0])
    def findStartAndEnd(arr):
        startPos, endPos = None, None
        flag = 2
        for i in range(M):
            for j in range(N):
                if arr[i][j] == 'S':
                    startPos = (i, j)
                    flag -= 1

                if arr[i][j] == 'D':
                    endPos = (i, j)
                    flag -= 1
            if not flag:
                return startPos, endPos

    s, e = findStartAndEnd(arr)
    directions = [(0, 1), (0, -1), (1, 0), (-1, 0)]
    heap = []
    for d in directions:
        x, y = s[0] + d[0], s[1] + d[1]
        if arr[x][y] not in 'X':
            method = int(arr[x][y]) - 1
            if 0 <= x < M and 0 <= y < N:
                heapq.heappush(heap, (time[method],
cost[method], x, y, method))
    visited = set()
    # print(s, e)
    while heap:
        t, c, x, y, m = heapq.heappop(heap)

```

```

        visited.add((x, y))

    for d in directions:
        nx, ny = x + d[0], y + d[1]
        # print(nx, ny)
        if (nx, ny) == e:
            return m + 1
        if arr[nx][ny] not in 'SX':
            method = int(arr[nx][ny]) - 1
            if (nx, ny) not in visited and 0 <= nx <
M and 0 <= ny < N and method == m:
                heapq.heappush(heap, (t +
time[method], c + cost[method], nx, ny, method))
        return -1
# print(getMinRoute([
#     ['3','3','S','2','X', 'X'],
#     ['3','1','1','2','X', '2'],
#     ['3','1','1','2','2', '2'],
#     ['3','1','1','1','D', '3'],
#     ['3','3','3','3','3', '4'],
#     ['4','4','4','4','4', '4']], [3,2,1,1], [0,1,3,2]))

```

H3 System Design Part

H5 SD1: word count

1个directory里面有很多file，我们需要做的就是计算这个directory里面的word count。每隔一段时间会有一次get word count的操作，同时，会有新的file加进来。目标是尽量快的实现这个过程。

初始思路：

用一个map<string, long>来记录word count。在bootstrap的时候先扫描当前全部file，然后每次get word count的时候再扫描新的file。

优化进程：

1. 文件是sorted by timestamp，可以记录latest timestamp，在扫描新file的时候用类似binary search，直接跳到最新file的位置。
2. 每个file是独立的，可以使用multi-threading来加速，但这设计到一个concurrency write的问题。最好有fine grained的数据结构，如果没有的话可以用一个global lock解决。global lock比较coarse grained，所以尽量少call，反应到design上就是先做local word count，再把local word count merge到 global word count。
3. system会crash，可以写中间结果到file里面，然后每次读file来bootstrap。另外的方案是写append-only log，然后每隔一段时间再merge log。
4. 经常写中间结果到file会inefficient，可以记录一个new file count，只有当new file count超过一定的threshold之后才触发。
5. 如果在写中间结果的中途system crash的话，可以自定义一个file terminator，然后读中间结果的时候检测有没有这个terminator，如果有，说明中间结果是正确的写完了的，如果没有则说明这个中间结果不完整，不能使用。

H5 SD2: WebCrawler

感觉是多线程+带一些系统设计。。比较无语

H5 SD3: 一个共享编辑playlist，讨论客户端服务端的状态，api和合并冲突算法。类似git

H5 SD4: Payment system