拉姆刚开始学习英文单词，对单词排序很感兴趣。

如果给拉姆一组单词，他能够迅速确定是否可以将这些单词排列在一个列表中，使得该列表中任何单词的首字母与前一单词的为字母相同。

你能编写一个程序来帮助拉姆进行判断吗？

这题如果是直接按顺序就很简单，判断下当前单词开头是不是等于前一个单词结尾就行了。但是如果是可以调换顺序，就变成了一个一笔画的问题，判断是否存在欧拉通路

将单词建成一副图，取出每个单词的第一个字母和最后一个字母，连一条有向边。然后根据欧拉通路的性质，判断下每个点的入度出度（奇数度的点不能大于 2） 最后 bfs 判断图是否是连通图

#include<iostream> #include<string> #include<algorithm> #include<string.h> #include<queue> using namespace std; char str[300];

int g[30][30]; int In[30];

int Out[30]; int num[30];

void init()

{

memset(g, 0, sizeof(g));

memset(In, 0, sizeof(In));

memset(Out, 0, sizeof(Out));

memset(num, 0, sizeof(num));

}

bool bfs(int s,int n)

{

queue <int> q;

q.push(s); int mark[30];

memset(mark, 0, sizeof(mark)); while (!q.empty())

{

int front = q.front(); mark[front] = 1; q.pop();

for (int i = 0; i < 30; i++)

{

if (g[front][i] && mark[i] == 0)

{

g[front][i] = 0;

* 1. ush(i);

}

}

}

int ha = 0;

for (int i = 0; i < 30 ;i++) if (mark[i]) ha++;

if (ha==n) return true; return false;

}

int main()

{

int n,s;

//freopen("data.txt", "r", stdin); while (cin >> n)

{

init();

bool temp = true;

for (int i = 0; i<n; i++)

{

cin >> str;

int len = strlen(str); Out[str[0] - 'a']++;

In[str[len - 1] - 'a']++;

g[str[0] - 'a'][str[len - 1] - 'a'] = 1;

g[str[len - 1] - 'a'][str[0] - 'a'] = 1;

if (num[str[0] - 'a'] == 0) num[str[0] - 'a'] = 1;

if (num[str[len - 1] - 'a'] == 0) num[str[len - 1] - 'a'] = 1; s = str[0] - 'a';

}

int sum1 = 0; int sum2 = 0;

for (int i = 0; i < 30; i++)

{

if ((In[i] - Out[i]) >=1) sum1++;

if ((In[i] - Out[i]) <= -1) sum2++;

if (abs(In[i] - Out[i])>1) temp = false;

}

if (sum1 >= 2 || sum2 >= 2) temp = false; int ha = 0;

for (int i = 0; i < 30; i++)

{

if (num[i] == 1) ha++;

}

temp = temp & bfs(s,ha);

if (temp) cout << "Yes" << endl; else cout << "No" << endl;



}

}

在计算机中，页式虚拟存储器实现的一个难点是设计页面调度（置换）算法。其中一种实现方式是 FIFO 算法。

FIFO 算法根据页面进入内存的时间先后选择淘汰页面，先进入内存的页面先淘汰，后进入内存的后淘汰。

假设 Cache 的大小为 2,有 5 个页面请求，分别为 2 1 2 3 1，则 Cache 的状态转换为：

(2)->(2,1)->(2,1)->(1,3)->(1,3)，其中第 1,2,4 次缺页，总缺页次数为 3。

现在给出Cache 的大小 n 和m 个页面请求，请算出缺页数。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32 | #include <iostream>  #include <vector>    bool vecfind(std::vector<int>& vec, int num) {  int len = vec.size();  for(int i = 0;i < len;++i){  if(num == vec[i])  return true;  }  return false;  }    int main() {  int n, m;  while(std::cin >> n >> m) {  std::vector<int> vecCache;  std::vector<int> vecPage(m);  int cnt = 0;  for(int i = 0;i < m;++i) {  std::cin >> vecPage[i];  }  for(int i = 0;i < m;++i) {  if(vecfind(vecCache, vecPage[i])) {  continue;  }  else {  if(vecCache.size() < n) {  vecCache.push\_back(vecPage[i]);  }  else{  vecCache.erase(vecCache.begin());  vecCache.push\_back(vecPage[i]); |

|  |  |
| --- | --- |
| 33  34  35  36  37  38  39  40  41  42  43 | }  ++cnt;  }  }    std::cout << cnt << std::endl;  }      return 0;  } |

短作业优先（SJF, Shortest Job First）又称为“短进程优先”SPN(Shortest Process Next)； 是对FCFS 算法的改进，其目标是减少平均周转时间。

短作业优先调度算法基于这样一种思想： 运行时间短的优先调度；

如果运行时间相同则调度最先发起请求的进程。

PS:本题题面描述有误，但原题如此，不宜修改，实际优先级如下:

* + 1. 接到任务的时间；
    2. 如果接收时间相同则调度 运行时间最短的任务。

等待时间：一个进程从发起请求到开始执行的时间间隔。

现在有 n 个进程请求 cpu，每个进程用一个二元组表示：(p,q),p 代表该进程发起请求的时间，p 代表需要占用 cpu 的时间。

请计算 n 个进程的平均等待时间。

这题目确实有错，最后按照下面的优先级能提交正确： 调度最先发起请求的进程优先调度；

若请求同时发出，则 运行时间短的优先调度；

感觉这种调度方式难度要比题意中的要简单一些，不过思路的话应该差不多。下面是提交的正确的代码，很简单了：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | int main(){  int n;  while (cin >> n){  vector<int> rts;  vector<int> cts;  for (int i = 0; i < n; i++){  int r, t;  cin >> r >> t;  rts.push\_back(r);  cts.push\_back(t); |



|  |  |
| --- | --- |
| 11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36 | }    int cur\_time = 1;  double wait\_time = 0;    while (!rts.empty()){  // 确定要执行进程  int cur\_process = 0;  for (int i = 0; i < rts.size(); i++) {  if (rts[i] < rts[cur\_process] || (rts[i] == rts[cur\_process] && cts[i] < cts[cur\_process])) // 对应两个优先级  cur\_process = i;  }  if (cur\_time > rts[cur\_process]){  wait\_time = wait\_time + (cur\_time - rts[cur\_process]);  cur\_time = cur\_time + cts[cur\_process];  }  else{  cur\_time = rts[cur\_process] + cts[cur\_process];  }  rts.erase(rts.begin() + cur\_process);  cts.erase(cts.begin() + cur\_process);  }    printf("%.4f\n", wait\_time / n);  }  } |