【华为OD机考 统一考试机试C卷】堆内存申请（C++ Java JavaScript Python C语言）

华为OD机考:统一考试 C卷 + D卷 + B卷 +A卷

目前在考C卷，经过两个月的收集整理，C卷真题已基本整理完毕
抽到原题的概率为2/3到3/3，也就是最少抽到两道原题。请注意：大家刷完C卷真题，最好要把B卷的真题刷一下，因为C卷的部分真题来自B卷。
另外订阅专栏还可以联系笔者开通在线 OJ 进行刷题，提高刷题效率。
真题目录：华为OD机考机试 真题目录（C卷 + D卷 + B卷 + A卷） + 考点说明
专栏：2023华为OD机试(B卷+C卷+D卷)（C++JavaJSPy）
华为OD面试真题精选：华为OD面试真题精选
在线OJ：点击立即刷题，模拟真实机考环境

题目描述

有一个总空间为100字节的堆，现要从中新申请一块内存，内存分配原则为：优先紧接着前一块已使用内存，分配空间足够且最接近申请大小的空闲内存。

输入描述

第1行是1个整数，表示期望申请的内存字节数

第2到第N行是用空格分割的两个整数，表示当前已分配的内存的情况，每一行表示一块已分配的连续内存空间，每行的第1和第2个整数分别表示偏移地址和内存块大小，如：

0 1
3 2

表示 0 偏移地址开始的 1 个字节和 3 偏移地址开始的 2 个字节已被分配，其余内存空闲。

输出描述

若申请成功，输出申请到内存的偏移；

若申请失败，输出 -1

备注：

1. 若输入信息不合法或无效，则申请失败
2. 若没有足够的空间供分配，则申请失败
3. 堆内存信息有区域重叠或有非法值等都是无效输入

用例

输入	**1** **0 1** **3 2**
输出	1
说明	堆中已使用的两块内存是偏移从0开始的1字节和偏移从3开始的2字节，空闲的两块内存是偏移从1开始2个字节和偏移从5开始95字节，根据分配原

解题思路

1. **内存块排序**:
   - 将 `usedMemory` 列表中的内存块按起始地址进行排序。这样可以更容易地找到连续的空闲内存区域。
2. **寻找最佳匹配内存块**:
   - 程序初始化一个变量 `start` （起始地址）为0， `bestFitStart` （最佳匹配起始地址）为-1，以及 `minSizeDiff` （最小大小差异）为最大整数值。

     然后，它遍历已排序的内存块列表。对于每个内存块，程序执行以下操作：
     - 检查内存块的合法性（起始地址是否有效，大小是否合理）。
     - 计算当前可用的空闲空间（ `freeSpace` ）。
     - 如果这个空闲空间足够大并且比之前找到的空间更接近申请大小，则更新 `bestFitStart` 和 `minSizeDiff` 。
3. **检查最后的空闲空间**:
   - 程序还需要检查列表中最后一个内存块后面的空间。如果那里有足够的空间，且这个空间的大小比之前找到的任何空间更接近申请大小，则更新 `bestFitStart` 。

C++

```
#include <iostream>
#include <vector>
```

```cpp
#include <algorithm>
#include <limits>

using namespace std;

int main() {
    int mallocSize; // 需要分配的内存大小
    cin >> mallocSize; // 从标准输入读取内存大小

    // 用于存储已被使用的内存块的向量，每个内存块由一对整数表示（起始地址，大小）
    vector<pair<int, int> > usedMemory;

    int start, size;
    while (cin >> start >> size) { // 循环读取已使用内存块的起始地址和大小
        usedMemory.push_back(make_pair(start, size)); // 将读取的内存块添加到向量中
    }

    // 如果分配的内存大小不合理，则输出-1并结束程序
    if (mallocSize <= 0 || mallocSize > 100) {
        cout << -1 << endl;
        return 0;
    }

    // 对已使用的内存块按起始地址进行排序
    sort(usedMemory.begin(), usedMemory.end());

    start = 0; // 初始化用于搜索空闲内存的起始地址
    int bestFitStart = -1; // 存储最佳匹配的内存块起始地址
    int minSizeDiff = numeric_limits<int>::max(); // 最小大小差异，初始化为int的最大值

    // 遍历所有已使用的内存块
    for (size_t i = 0; i < usedMemory.size(); ++i) {
        int blockStart = usedMemory[i].first; // 内存块的起始地址
        int blockSize = usedMemory[i].second; // 内存块的大小

        // 检查内存块是否有效
        if (blockStart < start || blockSize <= 0 || blockStart + blockSize > 100) {
            cout << -1 << endl;
            return 0;
        }

```

```
        // 计算当前内存块和上一个内存块之间的空闲空间
        int freeSpace = blockStart - start;
        // 如果找到足够的空闲空间且空间差异比之前找到的更小，则更新最佳匹配的起始地址和最小大小差异
        if (mallocSize <= freeSpace && (freeSpace - mallocSize) < minSizeDiff) {
            bestFitStart = start;
            minSizeDiff = freeSpace - mallocSize;
        }

        // 更新搜索的起始地址为当前内存块的结束地址
        start = blockStart + blockSize;
    }

    // 检查最后一个内存块后是否有足够的空闲空间
    if (100 - start >= mallocSize && (100 - start - mallocSize) < minSizeDiff) {
        bestFitStart = start;
    }

    // 输出最佳匹配的起始地址
    cout << bestFitStart << endl;

    return 0;
}
```

Java

```
import java.util.ArrayList;
import java.util.List;
import java.util.Scanner;
import java.util.Arrays;

public class Main {
    public static void main(String[] args) {
        // 创建一个扫描器来读取用户输入
        Scanner sc = new Scanner(System.in);

        // 读取第一行输入，这是我们要分配的内存大小
        int mallocSize = Integer.parseInt(sc.nextLine());
        // 创建一个列表来存储已使用的内存块
        List<int[]> usedMemory = new ArrayList<>();
```

```
        // 循环读取后续的输入行，每行代表一个已使用的内存块
        while (sc.hasNextLine()) {
            String line = sc.nextLine();
            // 如果读取到空行，结束输入
            if (line.isEmpty()) {
                break;
            }
            // 将输入行分割成字符串数组，然后转换成整数数组
            int[] memoryBlock = Arrays.stream(line.split(" "))
                                      .mapToInt(Integer::parseInt)
                                      .toArray();
            // 将这个内存块添加到已使用的内存列表中
            usedMemory.add(memoryBlock);
        }

        // 如果要分配的内存大小不在合法范围内，输出-1并结束程序
        if (mallocSize <= 0 || mallocSize > 100) {
            System.out.println(-1);
            return;
        }

        // 按照内存块的起始地址对已使用的内存列表进行排序
        usedMemory.sort((a, b) -> a[0] - b[0]);

        // 初始化起始地址为0
        int start = 0;
        // 初始化最佳适配的起始地址为-1
        int bestFitStart = -1;
        // 初始化最小大小差为最大整数
        int minSizeDiff = Integer.MAX_VALUE;

        // 遍历已使用的内存列表
        for (int[] block : usedMemory) {
            // 获取内存块的起始地址和大小
            int blockStart = block[0];
            int blockSize = block[1];

            // 如果内存块的起始地址小于当前的起始地址，或者内存块的大小小于等于0，或者内存块的结束地址大于100，输出-1并结束程序
            if (blockStart < start || blockSize <= 0 || blockStart + blockSize > 100) {
```

```
                System.out.println(-1);
                return;
            }

            // 计算当前的起始地址和内存块的起始地址之间的空闲空间
            int freeSpace = blockStart - start;
            // 如果空闲空间大于等于要分配的内存大小，并且空闲空间和要分配的内存大小的差小于当前的最小大小差，更新最佳适配的起始地址和最小大小差
            if (mallocSize <= freeSpace && (freeSpace - mallocSize) < minSizeDiff) {
                bestFitStart = start;
                minSizeDiff = freeSpace - mallocSize;
            }

            // 更新当前的起始地址为内存块的结束地址
            start = blockStart + blockSize;
        }

        // 检查最后一个内存块之后的空闲空间，如果空闲空间大于等于要分配的内存大小，并且空闲空间和要分配的内存大小的差小于当前的最小大小差，更新最佳适配的起始地址
        if (100 - start >= mallocSize && (100 - start - mallocSize) < minSizeDiff) {
            bestFitStart = start;
        }

        // 输出最佳适配的起始地址
        System.out.println(bestFitStart);
    }
}
```

javaScript

```
// 引入readline模块以从标准输入读取数据
const readline = require('readline');

// 创建readline接口
const rl = readline.createInterface({
  input: process.stdin,  // 将标准输入流作为输入源
  output: process.stdout // 将标准输出流作为输出源
});

let mallocSize; // 需要分配的内存大小
let usedMemory = []; // 用于存储已使用内存块的数组

```

```
// 当新的行被接收到时触发
rl.on('line', (line) => {
  if (!mallocSize) { // 如果还未读取到mallocSize
    mallocSize = parseInt(line); // 解析并设置需要分配的内存大小
    // 如果内存大小不合理，则输出-1并结束程序
    if (mallocSize <= 0 || mallocSize > 100) {
      console.log(-1);
      process.exit(0);
    }
  } else { // 如果已读取到mallocSize，则读取内存块
    const memoryBlock = line.split(' ').map(Number); // 将行分割并转换为数字数组
    usedMemory.push(memoryBlock); // 将内存块添加到usedMemory数组中
  }
});

// 当输入流被关闭时触发
rl.on('close', () => {
  // 对已使用的内存块按起始地址进行排序
  usedMemory.sort((a, b) => a[0] - b[0]);

  let start = 0; // 初始化用于搜索空闲内存的起始地址
  let bestFitStart = -1; // 存储最佳匹配的内存块起始地址
  let minSizeDiff = Number.MAX_SAFE_INTEGER; // 最小大小差异，初始化为最大安全整数值

  // 遍历所有已使用的内存块
  for (let block of usedMemory) {
    let blockStart = block[0]; // 内存块的起始地址
    let blockSize = block[1]; // 内存块的大小

    // 检查内存块是否有效
    if (blockStart < start || blockSize <= 0 || blockStart + blockSize > 100) {
      console.log(-1);
      process.exit(0);
    }

    // 计算当前内存块和上一个内存块之间的空闲空间
    let freeSpace = blockStart - start;
    // 如果找到足够的空闲空间且空间差异比之前找到的更小，则更新最佳匹配的起始地址和最小大小差异
    if (mallocSize <= freeSpace && (freeSpace - mallocSize) < minSizeDiff) {
      bestFitStart = start;

```

```
      minSizeDiff = freeSpace - mallocSize;
    }

    // 更新搜索的起始地址为当前内存块的结束地址
    start = blockStart + blockSize;
  }

  // 检查最后一个内存块后是否有足够的空闲空间
  if (100 - start >= mallocSize && (100 - start - mallocSize) < minSizeDiff) {
    bestFitStart = start;
  }

  // 输出最佳匹配的起始地址
  console.log(bestFitStart);
});
```

Python

```
import sys

# 读取第一行输入，这是我们要分配的内存大小
mallocSize = int(sys.stdin.readline())
# 创建一个列表来存储已使用的内存块
usedMemory = []

# 循环读取后续的输入行，每行代表一个已使用的内存块
for line in sys.stdin:
    # 将输入行分割成字符串数组，然后转换成整数数组
    memoryBlock = list(map(int, line.split()))
    # 将这个内存块添加到已使用的内存列表中
    usedMemory.append(memoryBlock)

# 如果要分配的内存大小不在合法范围内，输出-1并结束程序
if mallocSize <= 0 or mallocSize > 100:
    print(-1)
    sys.exit(0)

# 按照内存块的起始地址对已使用的内存列表进行排序
usedMemory.sort(key=lambda x: x[0])

```

```python
# 初始化起始地址为0
start = 0
# 初始化最佳适配的起始地址为-1
bestFitStart = -1
# 初始化最小大小差为最大整数
minSizeDiff = float('inf')

# 遍历已使用的内存列表
for block in usedMemory:
    # 获取内存块的起始地址和大小
    blockStart, blockSize = block

    # 如果内存块的起始地址小于当前的起始地址，或者内存块的大小小于等于0，或者内存块的结束地址大于100，输出-1并结束程序
    if blockStart < start or blockSize <= 0 or blockStart + blockSize > 100:
        print(-1)
        sys.exit(0)

    # 计算当前的起始地址和内存块的起始地址之间的空闲空间
    freeSpace = blockStart - start
    # 如果空闲空间大于等于要分配的内存大小，并且空闲空间和要分配的内存大小的差小于当前的最小大小差，更新最佳适配的起始地址和最小大小差
    if mallocSize <= freeSpace and (freeSpace - mallocSize) < minSizeDiff:
        bestFitStart = start
        minSizeDiff = freeSpace - mallocSize

    # 更新当前的起始地址为内存块的结束地址
    start = blockStart + blockSize

# 检查最后一个内存块之后的空闲空间，如果空闲空间大于等于要分配的内存大小，并且空闲空间和要分配的内存大小的差小于当前的最小大小差，更新最佳适配的起始地址
if 100 - start >= mallocSize and (100 - start - mallocSize) < minSizeDiff:
    bestFitStart = start

# 输出最佳适配的起始地址
print(bestFitStart)
```

C语言

```c
#include <stdio.h>
#include <stdlib.h>

// 定义一个结构体，用于表示内存块
```

```c
typedef struct {
    int start; // 内存块的起始地址
    int size;  // 内存块的大小
} MemoryBlock;

int compareMemoryBlocks(const void *a, const void *b) {
    // 用于qsort的比较函数，按内存块的起始地址排序
    MemoryBlock *blockA = (MemoryBlock *)a;
    MemoryBlock *blockB = (MemoryBlock *)b;
    return blockA->start - blockB->start;
}

int main() {
    int mallocSize; // 需要分配的内存大小
    scanf("%d", &mallocSize); // 从标准输入读取内存大小

    MemoryBlock usedMemory[100]; // 存储已分配内存块的数组
    int count = 0; // 已分配内存块的数量

    int start, size;
    while (scanf("%d %d", &start, &size) == 2) {
        // 循环读取已分配内存块的起始地址和大小
        usedMemory[count].start = start;
        usedMemory[count].size = size;
        count++;
    }

    // 如果分配的内存大小不合理，则输出-1并结束程序
    if (mallocSize <= 0 || mallocSize > 100) {
        printf("-1\n");
        return 0;
    }

    // 对已使用的内存块按起始地址进行排序
    qsort(usedMemory, count, sizeof(MemoryBlock), compareMemoryBlocks);

    int bestFitStart = -1; // 存储最佳匹配的内存块起始地址
    int minSizeDiff = 101; // 最小大小差异，初始化为一个大于最大内存的值

    start = 0; // 初始化用于搜索空闲内存的起始地址
```

```
    for (int i = 0; i < count; i++) {
        // 遍历所有已使用的内存块
        int blockStart = usedMemory[i].start;
        int blockSize = usedMemory[i].size;

        // 检查内存块是否有效
        if (blockStart < start || blockSize <= 0 || blockStart + blockSize > 100) {
            printf("-1\n");
            return 0;
        }

        // 计算当前内存块和上一个内存块之间的空闲空间
        int freeSpace = blockStart - start;
        if (mallocSize <= freeSpace && (freeSpace - mallocSize) < minSizeDiff) {
            // 如果找到足够的空闲空间且空间差异比之前找到的更小
            bestFitStart = start;
            minSizeDiff = freeSpace - mallocSize;
        }

        // 更新搜索的起始地址为当前内存块的结束地址
        start = blockStart + blockSize;
    }

    // 检查最后一个内存块后是否有足够的空闲空间
    if (100 - start >= mallocSize && (100 - start - mallocSize) < minSizeDiff) {
        bestFitStart = start;
    }

    // 输出最佳匹配的起始地址
    printf("%d\n", bestFitStart);

    return 0;
}
```

文章目录