Pit Mining

Due: Wednesday, 11:59pm

Dilbert the Digger wants to mine ore (or treasure) so as to maximize his net profits. The mining region is divided into blocks, and for simplicity we assume a one dimensional layout. From a preliminary geological survey, Dilbert estimates that removing block i will produce a net profit (value of ore minus processing cost) of certain million dollars. Because of environmental restrictions, Dilbert is required to mine for ore in only one contiguous set of blocks. Which blocks should he choose?

In other words, given a sequence of N integers (possibly negative), determine the maximum possible sum of any consecutive sequence. The problem is easy if all the numbers are positive: choose the entire sequence. The difficulty is when there are negative integers: should you take a negative integer in the hope that nearby positive integers will compensate for it?

For the input below, the maximum possible sum is 104, achieved by taking blocks 2 through 6. Note that the sum will always be at least 0, since Dilbert can always choose not to dig at all.

block	0	1	2	3	4	5	6	7	8	9	10
net profit	12	-34	40	6	-10	56	12	-1	-15	10	4

Applications: In real world applications, mine operators need to solve a three dimensional version of the problem. Moreover, there are further geological restrictions, e.g., you can't uncover a region 100 feet underground without first uncovering some of its surrounding regions. The two-dimensional version of this problem also arises in image processing, where the goal is to find the "maximum likelihood estimate" of a certain kind of pattern.

Part 1: **brute force.** Your first step is to design a brute force algorithm to solve the problem. Your program should read the integers using scanf(). You will probably find it convenient to store them in an array a[]. Now, for every possible pair of integers p and q such that 0 <= p < q < N, compute the sum a[p] + ... + a[q] from scratch. Output the maximum value over all possible pairs. Also, output a pair of indices p and q that achieve this value. (If all the values are negative, your program should output the value 0 and set both indices to -1.) Note that you can use this program to test if your Part 2 algorithm works right.

Part 2: **a divide-and-conquer algorithm.** The algorithm in Part 2 is far superior to that in Part 1. In this part, you will design an even higher performance algorithm, using the divide-and-conquer paradigm. The basic idea is as follows: to solve a problem of size N, divide the problem up into two problems of size roughly N/2; solve each subproblem (recursively, of course!); then figure out a way to combine their solution to determine a solution to the original problem. After dividing up the original problem at the middle block m, you will have two smaller subsequences, say b and c.

b	m	c

Recursively find the maximum contiguous sum within subsequences b and c. Call these subsequences bmax and cmax.

	bmax		m		cmax	

If the maximum contiguous sum in the original problem falls entirely in b or entirely in c, then you have solved the problem. The remaining case is if the maximum contiguous sum contains the single block m (and maybe also some adjacent blocks). Denote the maximum sequence containing m by dmax. Thus, the divide-and-conquer scheme is: compute bmax and cmax recursively; compute dmax directly; then return the best of the three.

b		dmax		c

It is possible to compute dmax in linear time by computing (i) the largest sequence whose right endpoint is m-1 and (ii) the largest sequence whose left endpoint is m+1. An appropriate gluing together of (i) and (ii) with m yields dmax.

This essentially describes the complete algorithm, modulo a few details, like the base case. This code will be more subtle, so be sure to carefully plan out your program.

Challenge for the bored. Design an even faster algorithm. Either use your cleverness and develop an alternate strategy, or perform code tuning on the divide-and-conquer algorithm. For example, recursive programs can usually be sped up by not taking the recursion all the way down to the lowest level, e.g., in mergesort it is advantageous to switch over to insertion sort when the number of elements is less than 16 (instead of 1 in the pure mergesort algorithm). Or you can unrecursify your program - in this case, eliminating instructions from the innermost loop usually produces the best results.

This assignment was developed by Robert Sedgewick and Kevin Wayne, and is an adaption of an assignment by Jon Bentley.