CS29002 Algorithms Laboratory

Assignment No: 4

Last date of submission: 17–August–2016

You are given two arrays A and B storing m and n integers. Both A and B are unsorted. An AB-sum is a sum of the form $A[i]+B[j]$ for some (valid) indices i and j. You are also given a positive integer k satisfying $k \leqslant m/\log m$ and $k \leqslant n/\log n$. Your task is to find out the k smallest AB-sums. Notice that it is not important which i and which j correspond to an AB-sum. In particular, if you permute A and B, these indices change, but the k smallest AB-sums remain the same. However, if the same AB-sum is realized by two or more different index pairs (for the same permutations of A and B), these sums are to be reported separately.

A naive idea is to store the mn AB-sums in an array S, sort S, and report the k smallest elements in the sorted S. This requires $O(mn\log(mn))$ time and $O(mn)$ space. Using min-heaps, you can do much better than this.

Algorithm 1

Convert S to a min-heap. For k times, print the minimum followed by deleting the minimum. This improves the running time to $O(mn+k\log(mn))$ which is $O(mn)$. However, the space requirement remains the same.

Write a function *buildheap*(S,l) to convert an array S of l integers to a min-heap. Write another function *extractmin*(S,l) to delete and return the minimum element from a heap S of size l. Finally, write a function *method1*(A,B,m,n,k) to print the k smallest AB-sums by calling the above two functions.

Algorithm 2

Now, we design an algorithm which runs in $O(m+n)$ time and uses only $O(k)$ extra space. We first convert A and B to two min-heaps. Assume that indexing in the arrays A and B is one-based, and $A[i]$ and $B[j]$ refer to the indices i and j after these arrays have been converted to min-heaps. Let $a_1,a_2,\ldots,a_k$ be the k smallest elements in A, and $b_1,b_2,\ldots,b_k$ the k smallest elements in B. Since A and B are not sorted (they are only converted to min-heaps), we cannot guarantee $a_i = A[i]$ or $b_j = B[j]$ for $i,j \in \{1,2,\ldots,k\}$. By k minimum extractions, we can obtain these $2k$ values a_i and b_j, but we will not do that. Indeed, the k smallest AB-sums are the k smallest elements of $T = \{a_i+b_j \mid 1 \leqslant i \leqslant k,\ 1 \leqslant j \leqslant k,\ i+j \leqslant k+1\}$. The size of T is $\Theta(k^2)$. Given the bounds on k, we cannot explicitly construct T.

Let us use a priority queue Q for storing pairs (i,j) of indices satisfying $1 \leqslant i \leqslant m$ and $1 \leqslant j \leqslant n$. Q is ordered (heap ordering) with respect to the sum $A[i]+B[j]$. Given i and j, we can easily compute $A[i]+B[j]$, so this sum need not be explicitly stored in Q. Since A and B are min-heaps, we have $a_1 = A[1]$ and $b_1 = B[1]$, and $a_1+b_1 = A[1]+B[1]$ is the smallest AB-sum. So we first insert $(1,1)$ to Q. Then, we enter a loop which runs until k sums are printed. Let (i,j) be the minimum (with respect to $A[i]+B[j]$) stored in Q. We print $A[i]+B[j]$ (conditionally; see below), and delete (i,j) from Q. We then insert the four index pairs $(2i,j)$, $(2i+1,j)$, $(i,2j)$, and $(i,2j+1)$ in Q (check, before insertion, that $2i,2i+1 \leqslant m$ and $2j,2j+1 \leqslant n$).

In order to convert A and B to min-heaps, use the same function *buildheap* written for Algorithm 1. Write a function *insertQ*$(Q,l,(i,j))$ to insert a pair (i,j) of indices to Q of size l. Also write a function *extractQ*(Q,l) to delete and return the pair corresponding to the minimum stored in Q of size l. You may define a data type *indexpair* comprising two integer elements. Finally, write a function *method2*(A,B,m,n,k) to print the k smallest AB-sums using Algorithm 2.

This function should assume that the k smallest AB-sums are distinct from one another (see Algorithm 3 why this assumption is necessary). This means that you remember the last sum printed, and do not print the same sum multiple times. Stop when you have printed exactly k distinct sums. When you do not print a duplicate sum for an index pair (i,j), do not also insert the indices $(2i,j)$, $(2i+1,j)$, $(i,2j)$, and $(i,2j+1)$ to Q (but delete (i,j) from Q anyway). This will guarantee that at most $8k+1$ index pairs are inserted in Q, so its size remains $O(k)$, that is, insertion and deletion in Q can be done in $O(\log k)$ time, and the total effort associated with managing Q is $O(k\log k)$ which is $O(m+n)$. Make sure that the priority queue Q has enough memory to store $8k+1$ index pairs.

Algorithm 3

Let us lift the assumption that the k smallest AB-sums are distinct. Algorithm 2 now faces a subtle problem. Let $A[1]+B[1]=50$, $A[1]+B[2]=A[2]+B[1]=60$, $A[2]+B[2]=70$, and $A[3]+B[2]=80$ be the five smallest AB-sums. So the printing should be $50,60,60,70,80,\ldots$, but Algorithm 2 prints $50,60,70,80,\ldots$. Two 60's must be printed, since this sum is realized by two different (i,j) pairs. If you do not remove the duplicates during printing, you get another faulty output $50,60,60,70,70,80,\ldots$. The second printing of 70 is because of the fact that the pair $(2,2)$ was inserted twice in Q, once after printing $A[1]+B[2]$ (so $(2,2)$ is now $(2i,j)$), and a second time after printing $A[2]+B[1]$ (so $(2,2)$ is now $(i,2j)$). You should print a sum multiple times if and only if it is realized by multiple index pairs, but not because of the insertion of the same pair multiple times in the queue.

Propose a remedy of this problem. You have to avoid duplicate insertions of the same index pairs. In order to maintain the overall running time at $O(m+n)$, the effort to avoid each duplicate insertion must take at most $O(\log k)$ time. A search in Q may take $\Theta(k)$ time (heaps are not suitable for efficient searching of a general element). Maintaining a height-balanced tree on inserted index pairs is too much overhead (at least from a coding perspective). So make the assumption that each AB-sum can be realized by only a constant number of index pairs, and suggest and implement a simpler solution. You may use $O(k)$ extra space (in addition to the space needed by Q).

Write a function *method3*(A,B,m,n,k) to print the k smallest AB-sums using Algorithm 3. Since A and B are already converted to min-heaps by *method2*(), you do not need to call *buildheap*() again from *method3*().

The *main*() function:

- Read m, the elements of A, n, the elements of B, and k from the user. Assume that the user enters such elements of A and B that there are no repetitions in the k smallest AB-sums.
- Print the k smallest AB-sums by calling *method1*(A,B,m,n,k). Then, print the k smallest AB-sums by calling *method2*(A,B,m,n,k).
- The user then re-enters m and n elements of A and B (m,n as above), this time without the restriction of no duplicates in the k smallest AB-sums. The value of k also remains the same as before.
- Print the k smallest AB-sums by calling *method1*(A,B,m,n,k). Then, print the k smallest AB-sums by calling *method3*(A,B,m,n,k).

Sample output

```
m = 64

+++ Array A:
 647 225 200 820 789 338  72 274 407 577 306 167 928  40 417  86 751 384 697 144
 137 823 241 986 665 468 225 121 372 143  86 737  86   4 557 874 341 628 148 748
 923 173 633 852 212  50 656 681 153 353 824   8 176 783 993 559 970 936 399  61
 797 203 797 882

n = 60

+++ Array B:
 206 354 757 547 700 623  14 623 514 646 194 444 414 849 125 566 202 948 292  96
 732 285 374 702 940 772 762 737 974 559 620 898 631  96 445 331 437 177 672 951
 822 866 395 955 715 520 240 636 187 532 731 637 535 823 339 475 314 819 931   7

k = 10

+++ Method 1:
  11  15  18  22  47  54  57  64  68  75

+++ Method 2:
      11 = A[ 1] + B[ 1] =    4 +    7
      15 = A[ 3] + B[ 1] =    8 +    7
      18 = A[ 1] + B[ 3] =    4 +   14
      22 = A[ 3] + B[ 3] =    8 +   14
      47 = A[ 7] + B[ 1] =   40 +    7
      54 = A[ 7] + B[ 3] =   40 +   14
      57 = A[ 2] + B[ 1] =  50 +   7
      64 = A[ 2] + B[ 3] =  50 +  14
      68 = A[15] + B[ 1] =  61 +   7
      75 = A[15] + B[ 3] =  61 +  14

+++ Re-reading arrays

+++ Array A:
 304 530 283  99 580 567 424 995 790 467 919 332 907  40 812 503 540 613 895 512
 320 550 174 859 780 702 492 196 894 261  16 916 510 298  15 808 864 438 804 373
 623 442 704 249 481 516 751  21 847 646 251 167 196 425 744 695 845 236 609 458
 216 624 374 725

+++ Array B:
 640 107 252 223 263 774 595 885 934 300 852 416 534 603 155 382 968 125 267 882
 268  12 296 113 247 904 570 462 247 662 905 886 487 876 827 749 650 142 353 585
 160 205 719 693 526 592 793 494 435  61  96 702  72 391 814  37 296 103 499 261

+++ Method 1:
  27  28  33  52  52  53  58  76  77  77

+++ Method 3:
      27 = A[ 1] + B[ 1] =  15 +  12
      28 = A[ 3] + B[ 1] =  16 +  12
      33 = A[ 6] + B[ 1] =  21 +  12
      52 = A[ 7] + B[ 1] =  40 +  12
      52 = A[ 1] + B[ 3] =  15 +  37
      53 = A[ 3] + B[ 3] =  16 +  37
      58 = A[ 6] + B[ 3] =  21 +  37
      76 = A[ 1] + B[ 6] =  15 +  61
      77 = A[ 3] + B[ 6] =  16 +  61
      77 = A[ 7] + B[ 3] =  40 +  37
```

Submit a single C/C++ source file. Do not use global/static variables.
Do not invoke heap functions from C++ STL.