Fidessa

Codility test - 2016 Dec 06 *(One question in 40min)*
Given a vector of unsorted signed integers having size *N*, return maximum sum-distance with *O(N)* time complexity and *O(1)* space complexity, which is defined as :

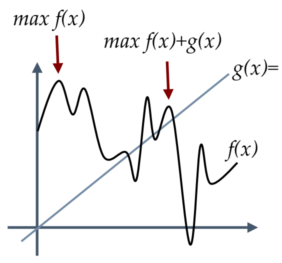
$$\max_{0\le m\le n<N} A[n]+A[m]+(n-m)$$

In general, for any monotonic increasing g(x)

we have $\arg\max_{L\le x<U} f(x)+g(x) \ge \overbrace{\arg\max_{L\le x<U} f(x)}^{M}$

and hence $\max_{L\le x<U} f(x)+g(x) = \max_{M\le x<U} f(x)+g(x)$

the proof is similar to the following.



Answer : First of all, suppose :

$$k = \arg\max_{0\le n<N} A[n]$$

and $$n' = \arg\max_{0\le n<N} A[n]+n$$

and $$m' = \arg\max_{0\le m<N} A[m]-m$$

then $A[n']+n' \ge A[k]+k$ — *since* $A[n]+n$ *is max when* $n=n'$

$n' \ge k+A[k]-A[n']$

$n' \ge k$ — *since* $A[n]$ *is max when* $n=k$ *, hence* $A[k]\ge A[n']$

then $A[m']-m' \ge A[k]-k$ — *since* $A[m]-m$ *is max when* $m=m'$

$-m' \ge -k+A[k]-A[m']$

$-m' \ge -k$ — *since* $A[n]$ *is max when* $n=k$ *, hence* $A[k]\ge A[m']$

$m' \le k \le n'$

Therefore we have :

$$\max_{0\le m\le n<N} A[n]+A[m]+(n-m) = \max_{\substack{0\le m\le k\\ k\le n<N}} A[n]+A[m]+(n-m)$$

$$= \max_{k\le n<N} A[n]+n+\max_{0\le m\le k} A[m]-m$$

As we decouple *m'* from *n'*, the objective function is broken into two parts, hence we can do it in a single scan :

```
void question(std::vector<int>& A, unsigned long size)
{
      for(unsigned long n=0; n!=size; ++n) A.push_back(rand()%201-100);
}

int solution(const std::vector<int>& A)
{
      unsigned long index;
      int maxA = std::numeric_limits<int>::min();
      int max0 = std::numeric_limits<int>::min();
      int max1 = std::numeric_limits<int>::min();

      k = 0;
      for(int n=0; n!=A.size(); ++n) { if (maxA < A[n])       { maxA = A[n]; k=n;      }}
      for(int n=k; n<=A.size(); ++n) { if (max1 < A[n]+n)     { max1 = A[n]+n;         }}
      for(int n=0; n!=k+1; ++n)      { if (max0 < A[n]-n)     { max0 = A[n]-n;         }}
      return max0 + max1;            // Note that n,m can be the same.
}
```

This question is similar to question 5 in algorithm.doc, which cannot be solved in *O(N)* time complexity, what makes the difference? Lets put the objective function of this question and that for question 5 side by side :

$$\max_{0\le m\le n<N} A[n]+A[m]+(n-m) = \max_{k\le n<N}(A[n]+n)+\max_{0\le m<k}(A[m]-m)$$ *from this question*

$$\max_{0\le m\le n<N} A[n]-A[m] \ne \max_{0\le n<N} A[n]-\min_{0\le m<N} A[m]$$ *from question 5 in algorithm.doc*

$$\max_{0\le m,n<N} A[n]-A[m] = \max_{0\le n<N} A[n]-\min_{0\le m<N} A[m]$$ *a simplied version*

The main difference is that the $n>m$ constraint in question 5 cannot be relaxed (we can't decouple *n* and *m* like what we did above), so we need to do either a *N(N-1)/2* brute force search or a *logN* divide and conquer.

Codility test - 2016 Dec 13 *Two questions in 60min*

Question 1

Given two non-negative integers *A* and *B*, combine them into a new integer *C*, such that the *n*th most significant digit in *A* becomes the *(2n-1)*th most significant digit in *C*, while the *n*th most significant digit in *B* becomes the *2n*th most significant digit in *C*, hence digits of *A* and *B* are concaternated in an alternative manner, if there are excess digits from either *A* or *B*, append them at the back of *C*.

```
int solution(int A, int B)
{
      if (A==0) return -1; // B is allowed to be zero.
      std::stringstream ss0; ss0 << A;
      std::stringstream ss1; ss1 << B;
      std::stringstream ss2;
      std::string s0 = ss0.str();
      std::string s1 = ss1.str();

      int common_size = std::min(s0,s1);
      for(int n=0; n!=common_size; ++n) ss2 << s0[n] << s1[n];
      if (s0.size() > common_size) ss2 << s0.substr(common_size);
      if (s1.size() > common_size) ss2 << s1.substr(common_size);

      int result;
      ss2 >> result;
      return result;
}
```

Question 2

The other question is about binary tree transversal. Given a binary tree and a node in the tree (not necessarily the root), a left path is defined as the path that always pick the left child node as we transverse down the tree starting from the given node and until there is no more left child node, while a right path is defined in a similar way. Length of a path is number of nodes on the path minus by *1*, i.e. excluding starting node. Find the maximum path length given a binary tree in *O(N)* time complexity and *O(1)* space complexity. Challenge of this question is that no path is allowed to sometimes turn left and sometimes turn right, all paths either stick to left or stick to right, hence the longest path may not start with the root.

Answer : We need to propagate 3 informations starting from leaf nodes to the root, hence post-order depth first search is needed. The 3 pieces of informations include : length of left path starting from current node, length of right path starting from current node, and maximum path length of subtree rooted with current node (note that : maximum path of subtree rooted with current node can be much greater than the length of both left and right path starting from current node), all 3 informations are updated recursively.

```
std::tuple<int,int,int> path_length_implementation(node<T>* root)
{
      if (!root) return std::make_tuple<int,int,int>(0,0,0);

      auto ans_lhs = path_length_implementation(root->lhs);
      auto ans_rhs = path_length_implementation(root->rhs);

      int lhs_length = 1 + std::get<0>(ans_lhs);
      int rhs_length = 1 + std::get<1>(ans_rhs);
      int max_length = std::max(std::max(lhs_length, std::get<2>(ans_lhs)),
                                std::max(rhs_length, std::get<2>(ans_rhs)));

      return std::make_tuple<int,int,int>(lhs_length, rhs_length, max_length);
}

int path_length(node<T>* root)
{
      return std::get<2>(path_length_implementation(root));
}
```