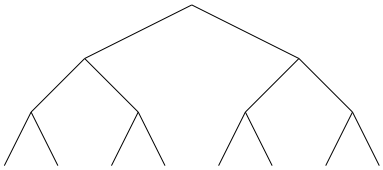
Asia-Pacific Informatics Olympiad
Solutions for 2007

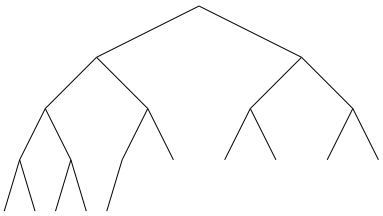
Problem 1
Mobiles

Description of Algorithm

We can define a *complete* binary tree to be one that has all leaf nodes at the same depth d. The figure below depicts a complete binary tree with depth 3.



We can likewise define a *partially complete* binary tree to be one in which all leaf nodes are at depth d or depth $d-1$, and where all leaf nodes at depth d form a contiguous block at the leftmost end of the tree. One such example is shown below with depth 4. Note that all complete binary trees are also considered to be partially complete.



This task asks for the smallest number of operations required to transform a given binary tree into a partially complete binary tree. The only operation available is to swap the left and right children of a node. To solve this task, we define the following recursive functions:

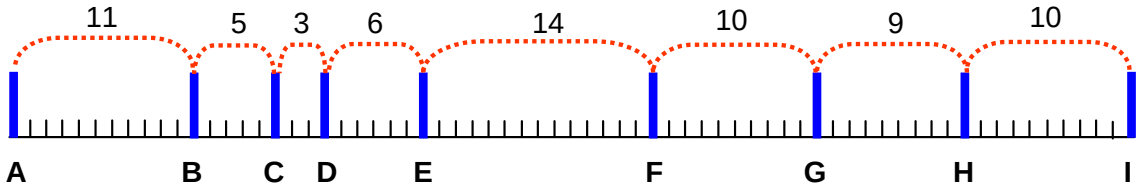
- $h(t)$, which returns the height of a subtree, t.
- $c(t)$, which returns the fewest number of swaps required to make the subtree t complete. This will be 0 if the given subtree is already complete, or ∞ otherwise (since if a tree is not already complete, it can never be made complete). To calculate this, we simply check whether the subtree's two immediate children are complete subtrees and whether they have the same height.
- $p(t)$, which returns the fewest number of swaps required to make the given subtree partially complete, or ∞ if this is not possible. Let l and r denote the left and right subtrees of t. We calculate $p(t)$ by taking the minimum of the following:
  - The result of $c(t)$;
  - If $h(l) = h(r) + 1$, then consider $p(l) + c(r)$;
  - If $h(l) = h(r) - 1$, then consider $c(l) + p(r) + 1$;
  - If $h(l) = h(r)$, then consider both $c(l) + p(r)$ and $p(l) + c(r) + 1$.

Each of these functions is calculated recursively for each subtree, working upwards from the leaves to the root of the tree. The final solution has a running time of $O(n)$, where n is the number of nodes in the tree.

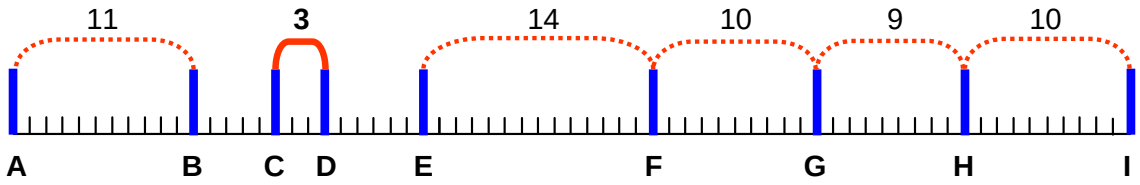
Problem 2
Backup

Description of Algorithm

The first observation is that the optimal solution will only ever connect pairs of offices that are directly next to each other. For instance, we might connect E with D or F, but we would never connect E with C or G.



We cannot just greedily select the smallest distances as the offices to connect. This is because each time we select a link as part of our solution, we cause the links either side to become unavailable:

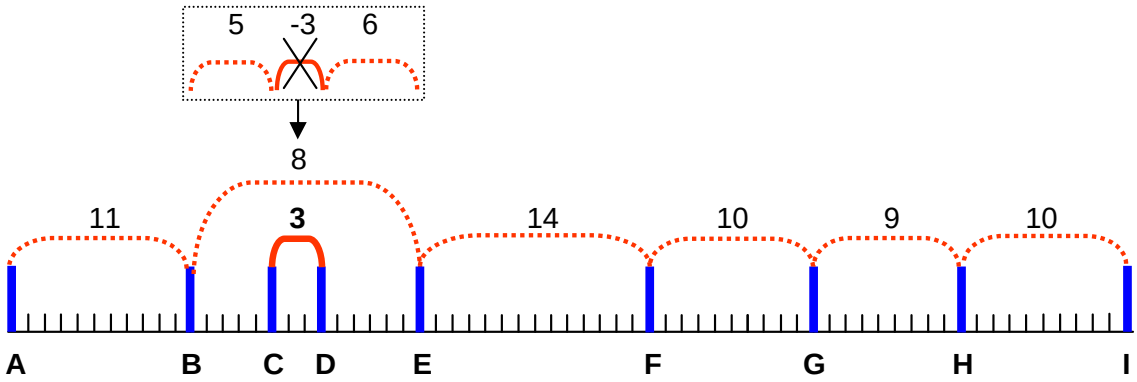


In this example, suppose we are trying to place $k = 2$ links. Greedily choosing the link $(C, D, 3)$ forces us to then use the link $(G, H, 9)$, when a solution involving $(B, C, 5)$ and $(D, E, 6)$ would be better. This style of greedy solution is therefore incorrect.

There exists a correct and fairly straightforward dynamic programming solution, which moves from the left to right across the offices, keeping track of the optimal answer which uses i links and whether or not the previous office has been linked to. This solution has a time complexity of $O(nk)$, which unfortunately is not fast enough for the given time limit.

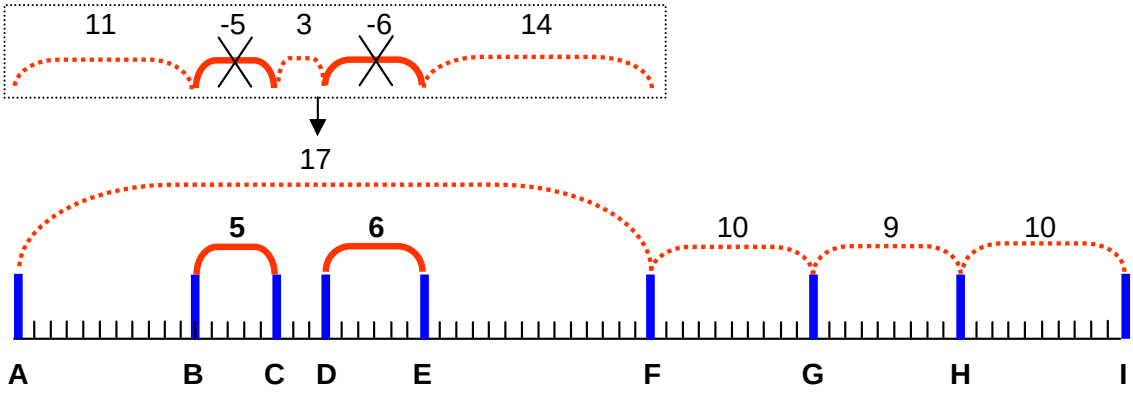
A faster solution can be found by fixing the greedy solution. In our example, the problem with the greedy solution is that after selecting $(C, D, 3)$, the two adjacent links $(B, C, 5)$ and $(D, E, 6)$ can no longer be used. The only sensible way to use them would be to remove the link $(C, D, 3)$, and add *both* $(B, C, 5)$ and $(D, E, 6)$ to replace it. The reason we must replace $(C, D, 3)$ with both adjacent links is because any solution involving only one of these adjacent links would be worse than the original solution using $(C, D, 3)$.

To allow the greedy algorithm to consider this possibility, we create a new *virtual* link, illustrated on the following page: $(B, E, 8)$, which represents the option of removing link $(C, D, 3)$ and replacing it with both links $(B, C, 5)$ and $(D, E, 6)$. If we use this possibility, we end up with two links with a total cost of $5 + 6 = 11$ instead of one link with a cost of 3, so we consider the virtual link to have a cost of $5 + 6 - 3 = 8$. Using this new virtual link increases the total number of real links by one, so it can be manipulated like any other link, and compared with "normal" links when selecting which link the greedy algorithm should add next.



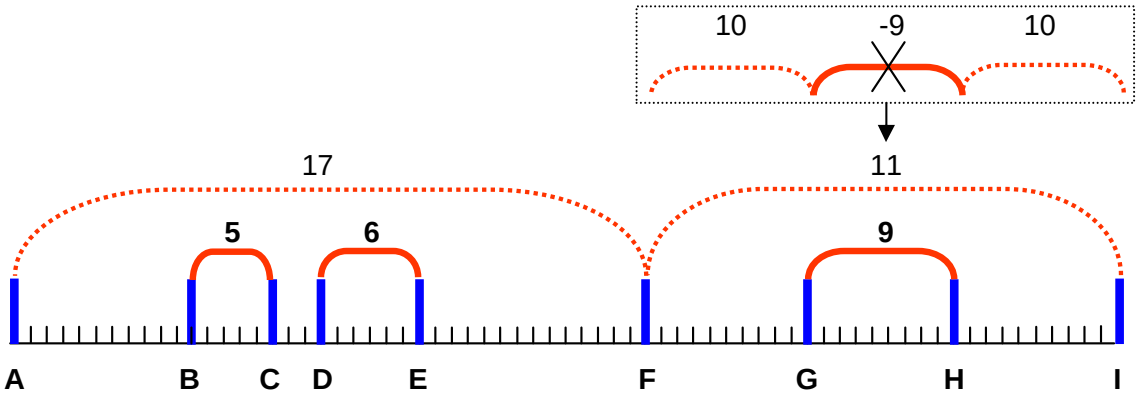
Consider again the previous example, where we have greedily chosen the first link $(C, D, 3)$ and we are deciding what to do next. With our new algorithm, the virtual link $(B, E, 8)$ is the best choice. Once we select $(B, E, 8)$ we cannot use adjacent links $(A, B, 11)$ or $(E, F, 14)$ any more, so we create yet another new virtual link in case we wish to undo this decision later on.

But what should this new virtual link look like? The only way to add an extra link inside the range A–F without using the current selections $(B, C, 5)$ and $(D, E, 6)$ is to replace them with links $(A, B, 11)$, $(C, D, 3)$ and $(E, F, 14)$ instead. This operation has a total cost of $11-5+3-6+14 = 17$, so our new virtual link is $(A, F, 17)$.



The cost of the new virtual link can also be expressed more directly: It is the cost of the two links at the end points $(A, B, 11)$ and $(E, F, 14)$, minus the cost of the (virtual) link $(B, E, 8)$ that sits between them: $11 - 8 + 14 = 17$.

Suppose we are now searching for $k = 3$ links. We have already greedily chosen the real link $(C, D, 3)$ and the virtual link $(B, E, 8)$. The next best link to select is now $(G, H, 9)$. After selecting it, we create a new virtual link between F and I with a total cost of $10 - 9 + 10 = 11$: $(F, I, 11)$.



We notice that every time we select a link, destroy the invalid links on either side, and create a new virtual link that spans them, what we are always left with is a sequence of potential links, where the end point of one link is always the starting point of the next (for illustration, see the sequences of dotted lines on the previous three diagrams). We can therefore describe such a sequence as:

$$(P_1, P_2, C_1)\ (P_2, P_3, C_2)\ \ldots\ (P_i, P_{i+1}, C_i)\ \ldots\ ,$$

where P_i and P_{i+1} are the end points of link i in the sequence, and C_i is its cost.

When our greedy algorithm chooses a link (P_k, P_{k+1}, C_k) to use, we must update the sequence as follows:

- Invalidate links (P_{k-1}, P_k, C_{k-1}) and $(P_{k+1}, P_{k+2}, C_{k+1})$ if they exist;
- If both of those links did exist, then create a new virtual link $(P_{k-1}, P_{k+2}, (C_{k-1} - C_k + C_{k+1}))$.

Using this system, selecting the remaining link with the smallest cost is always the best way to obtain a solution with one more link than before.

We can implement the overall algorithm by maintaining a heap of available links. Every time we pop the link with smallest cost off the heap, we also need to remove the two adjacent links in the current sequence, and if both exist (i.e., we are not at the leftmost or rightmost end of the sequence), we create a new virtual link as described above and add it to the heap. As each of these operations is $O(\log n)$, the total running time of this algorithm is $O(k \log n)$.

For our example, the initial heap will be:

$$(C, D, 3)\ (B, C, 5)\ (D, E, 6)\ (G, H, 9)\ (F, G, 10)\ (H, I, 10)\ (A, B, 11)\ (E, F, 14).$$

We then choose $(C, D, 3)$, remove $(B, C, 5)$ and $(D, E, 6)$ also, and add the virtual link $(B, E, 8)$. The new heap becomes:

$$(B, E, 8)\ (G, H, 9)\ (F, G, 10)\ (H, I, 10)\ (A, B, 11)\ (E, F, 14).$$

We then choose $(B, E, 8)$, remove $(A, B, 11)$ and $(E, F, 14)$ also from the heap, and add the new virtual link $(A, F, 17)$, to obtain:

$$(G, H, 9)\ (F, G, 10)\ (H, I, 10)\ (A, F, 17),$$

and so on.

To easily determine which links are adjacent to the chosen link in the sequence, all we need to do is to maintain, for each point, the link that ends at that point and the link that starts at that point.

This problem can also be seen as a bipartite graph problem: odd points on one side, even ones on the other, and up to 2 edges for each node. Our solution can then be seen as maintaining a heap of all possible augmenting paths in this bipartite graph.

Problem 3
Zoo

Description of Algorithm

This is a special case of Max-SAT, where each clause can only contain variables which are close together (in a circular order). Specifically:

- Each animal is a boolean variable in Max-SAT. We have N variables $X_1, \ldots, X_N$ in a circular order.
- Each child is a clause in Max-SAT. If a child can see k consecutive enclosures, this means that each clause may only contain variables $X_c, X_{c+1}, \ldots, X_{c+k-1}$ for some c (where $X_{N+1} = X_1$, $X_{N+2} = X_2$ and so on). For this problem, we are given $k = 5$.

A model solution uses dynamic programming.

Linear case:

Consider a simpler problem when the enclosures $X_1, \ldots, X_N$ are placed in a linear order (i.e., X_N does not wrap back around to X_1). For each i, let $\mathcal{C}_i$ be the set of all children whose visible enclosures lie in the range $X_1, \ldots, X_i$. We iterate through $i = 1, 2, \ldots, N$, and at each stage we solve problems of the following form:

> Consider the set of children $\mathcal{C}_i$, and consider all 2^k ways in which the final k enclosures $X_{i-k+1}, X_{i-k+2}, \ldots, X_i$ may be filled or empty. For each filled/empty combination for enclosures $X_{i-k+1}, X_{i-k+2}, \ldots, X_i$, what is the maximum number of children in $\mathcal{C}_i$ who can be made happy?

We solve the problems at stage i as follows. We assume we have already computed the maximum number of happy children in $\mathcal{C}_{i-1}$ for all filled/empty combinations for enclosures $X_{i-k}, \ldots, X_{i-1}$. To calculate the new answers for stage i and a particular filled/empty combination for enclosures $X_{i-k+1}, X_{i-k+2}, \ldots, X_i$:

(i) We count how many "new" children in $\mathcal{C}_i - \mathcal{C}_{i-1}$ are made happy. We can calculate this directly from our choice of filled/empty values for $X_{i-k+1}, X_{i-k+2}, \ldots, X_i$, because every "new" child is looking at precisely these enclosures.

(ii) Assuming the "old" enclosure X_{i-k} is filled, we can look up our solutions from stage $i-1$ to find out how many children in $\mathcal{C}_{i-1}$ can be made happy with our selected filled/empty values for $X_{i-k+1}, \ldots, X_{i-1}$. Likewise, we can look up how many children in $\mathcal{C}_{i-1}$ can be made happy when the old enclosure X_{i-k} is empty. The larger of these two numbers tells us how many "old" children in $\mathcal{C}_{i-1}$ we can keep happy overall.

(iii) The number of happy "new" children from step (i) can be added to the number of happy "old" children from step (ii), giving the solution to our current problem.

The total number of problems to solve is $2^k N$, and the overall running time for the algorithm is $O(2^k(N + C))$. The space required is only $O(2^k)$, since each stage i only requires the solutions from the previous stage $i - 1$.

Circular case:

In the circular case, we need to permanently remember the states of the first $k - 1$ enclosures $X_1, \ldots, X_{k-1}$ (in additional to the "most recent" k enclosures $X_{i-k+1}, \ldots, X_i$). This allows us to correctly deal with the final children who can see back around past X_N to $X_1, \ldots, X_{k-1}$ again. The remainder of the algorithm remains more or less the same.

We therefore have 2^{2k} problems to solve at each stage, giving running time $O(2^{2k}(N + C))$ and storage space $O(2^{2k})$.