Introduction to Algorithms September 24, 2003
Massachusetts Institute of Technology 6.046J/18.410J
Professors Shafi Goldwasser and Silvio Micali Handout 9

Problem Set 2 Solutions

Problem 2-1. Goldilocks and the n bears.

> *Once upon a time, there was a little girl named Goldilocks. She went for a walk in the forest. Pretty soon, she came upon a house. She knocked and, when no one answered, she walked right in. At the table in the kitchen, there were three bowls of porridge. Goldilocks was hungry. She tasted the porridge from the first bowl. "This porridge is too hot!" she exclaimed. So, she tasted the porridge from the second bowl. "This porridge is too cold," she said. So, she tasted the last bowl of porridge. "Ahhh, this porridge is just right," she said happily and she ate it all up.*

In an unfortunate accident at a laboratory in Building 68, a little girl named Goldilocks wandered into a human cloning machine. As a result, there are now n little Goldilocks (Goldilocks$_1$, Goldilocks$_2$, ...) wandering around the halls of MIT. Because cloning remains an imperfect technology, each Goldilock$_i$ has a *temperature preference* t_i distinct from that of all the other Goldilocks.

Luckily, you also have managed to find n bowls of porridge, where bowl j is kept at temperature b_j. It is a great stroke of fortune that for each Goldilocks$_i$, there is exactly one bowl j so that $b_j = t_i$, and for each bowl j there is exactly one Goldilocks$_i$ whose temperature preference $t_i = b_j$. That is, the two sets $\{b_j : 1 \le j \le n\}$ and $\{t_i : 1 \le i \le n\}$ are equal. Your job is to match Goldilocks$_i$ to a bowl j of porridge such that $b_j = t_i$.

When you give bowl j to Goldilocks$_i$, she says:

$$\begin{cases} \text{"This porridge is too hot!"} & \text{if } t_i < b_j \\ \text{"This porridge is too cold!"} & \text{if } t_i > b_j \\ \text{"This porridge is just right!"} & \text{if } t_i = b_j. \end{cases}$$

Call any one such tasting a *trial*. You may only use trials to get information about temperatures of the bowls or temperature preferences of the Goldilocks. *You may **not** directly compare the temperatures of two bowls or the temperature preferences of two Goldilocks.*

(a) Give a randomized algorithm for which the expected number of trials is $O(n \log n)$.

> **Solution:** We give an algorithm MATCH-TEMPERATURES(G, B) which takes two sets $G, B \subseteq \{1, \ldots, n\}$ as input, representing Goldilocks and bowls, respectively, to be matched. We will only call the procedure with inputs that actually possess a matching, i.e., that satisfy the precondition $G = B$. The output of the algorithm will consist of n distinct pairs (i, j), where Goldilocks$_i$ has a temperature preference t_i that matches the temperature b_j of the bowl j.

MATCH-TEMPERATURES(G, B)
1 **if** the sets G and B are empty
2 **then return**
3 **if** the sets G and B contain only one element each (say $G = \{g\}$ and $B = \{b\}$)
4 **then** Output "(g, b)"
5 **return**
6 Choose a random Goldilocks$_i$ from G.
7 Compare Goldilocks$_i$ to every element of B
8 $B_< \leftarrow$ bowls in B for which Goldilocks$_i$ says "too cold!"
9 $B_> \leftarrow$ bowls in B for which Goldilocks$_i$ says "too hot!"
10 $j \leftarrow$ the one bowl in B with $b_j = t_i$
11 Compare bowl j to every element of Goldilocks$_i$
12 $G_< \leftarrow$ Goldilocks in G who say "too hot!" when they taste bowl j
13 $G_> \leftarrow$ Goldilocks in G who say "too cold!" when they taste bowl j
14 Output "(Goldilocks$_i$, b)"
15 MATCH-TEMPERATURES$(G_<, B_<)$
16 MATCH-TEMPERATURES$(G_>, B_>)$

Correctness can be seen as follows. If we pick any Goldilocks$_i$ in line 7 then there will be a matching among the Goldilocks and bowls cooler than t_i (which are in the sets $G_<$ and $B_<$), and likewise between Goldilocks and bowls hotter than t_i (which are in $G_>$ and $B_>$). Termination is also easy to see: since $|G_<| + |G_>| < |G|$ in every recursion step, the size of the first parameter reduces with every recursive call. It eventually must reach 0 or 1, in which case the recursion terminates. Correctness is obvious in these base cases.

What about the running time? The random pick of Goldilocks$_i$ in line 7 is what determines the sets $G_<$ and $G_>$ into which the input is split. The sizes of these sets can be any pair from $\{(i, n-1-i) \mid 0 \le i \le n-1\}$, and every split is equally likely, i.e. has probability $1/n$. The expected runtime $E(n)$ for an input of n Goldilocks/bowls can therefore be recursively expressed as

$$E(n) = \frac{1}{n}\sum_{i=0}^{n-1}(E(i) + E(n-1-i)) + cn$$

for some constant c. For the cn-term just observe that we have one Goldilocks$_i$ try all bowls, and one bowl b tried by every Goldilocks, for a total of $2n$ trials. The solution to the recurrence is $E(n) = O(n \log n)$, as discussed in class when deriving the running time for randomized quicksort. We replicate the analysis here for completeness:

$$\begin{aligned} E(n) &= \frac{1}{n}\sum_{i=0}^{n-1}(E(i)+E(n-1-i))+cn \\ &= \frac{1}{n}\sum_{i=0}^{n-1}2E(i)+cn \\ &= \frac{2}{n}\sum_{i=0}^{n-1}E(i)+cn \end{aligned}$$

We can multiply both sides by n obtaining:

$$n \cdot E(n) = 2\sum_{i=0}^{n-1}E(i)+cn^2 \quad (1)$$

Thus, we also have:

$$(n-1)\cdot E(n-1) = 2\sum_{i=0}^{n-2}E(i)+c(n-1)^2 \quad (2)$$

If we subtract (2) from (1), then we have:

$$n \cdot E(n) - (n-1)\cdot E(n-1) = 2E(n-1)+2cn-c$$

We can drop the insignificant c on the right and rearrange the terms. Then we have:

$$\begin{aligned} n \cdot E(n) &= (n+1)E(n-1)+2cn \Rightarrow \\ \frac{E(n)}{n+1} &= \frac{E(n-1)}{n}+\frac{2c}{n+1} \end{aligned}$$

Now we have a formula for $E(n)$ in terms of $E(n-1)$. Thus, we obtain:

$$\frac{E(n)}{n+1} = 2c\sum_{i=3}^{n+1}\frac{1}{i}+\frac{E(1)}{2}$$

Thus, we have $\frac{E(n)}{n+1} = O(\lg n) \Rightarrow E(n) = O(n \lg n)$.

(b) Suppose that when you give Goldilocks$_i$ bowl j with $b_j \neq t_i$, she spits out a spoonful and says "Yuck!" (without indicating whether it's too hot or too cold). Give the fastest algorithm you can to match Goldilocks to bowls in this scenario.

Solution: First let Goldilocks$_i$ try every bowl (bowl 1, bowl 2, ..., in order) until she says "just right!" (We are guaranteed that one bowl will have $b_j = t_1$, so this will eventually happen.) We can just set that pair aside. Then pick Goldilocks$_2$, and let her try all remaining bowls. Again she will match some bowl, and we can set that pair aside. Continue this way until every Goldilocks has been matched.

How many trials does this take? Certainly we have to complete at most n trials to find the first match, $n-1$ trials to find the second match, etc. So the number of trials is at most

$$n + (n-1) + (n-2) + \cdots + 2 + 1 = \sum_{k=1}^{n} k = O(n^2).$$

On the other hand, if Goldilocks$_1$ likes bowl n, and Goldilocks$_2$ likes bowl $n-1$, etc., then we will do n trials to find the first match, and $n-1$ to find the second, and so on. In total this is a quadratic number ($\sum_{i=0}^{n} n - i$) of trials, so there exists an "bad" instance requiring a quadratic number of comparisons. Thus the algorithm requires $\Omega(n^2)$ trials in the worst case. Therefore it needs $\Theta(n^2)$ trials, since we have matching $O()$ and $\Omega()$ bounds.

Problem 2-2. Hitting the target.

Give an $O(n)$ algorithm for the following problem: Given a target integer T and an array A of n integers such that $0 \leq A[i] \leq 65536$ for every $1 \leq i \leq n$, determine if there exists a pair $i, j \in \{1, \ldots n\}$ such that $A[i] + A[j] = T$.

Solution: Use COUNTING-SORT or RADIX-SORT to sort the array A. (These methods apply since the elements we are trying to sort are integers in a bounded range.) Then the following code will find a match if one exists:

```
FIND-TARGET-PAIR(A)  ▷ A must be sorted
i ← 1
j ← n
while j ≥ i
   if A[i] + A[j] = 65536
        return "yes"
   else if A[i] + A[j] < 65536
        i ← i + 1
   else j ← j − 1
```

For correctness, it's obvious that the above code only returns "yes" if $A[i] + A[j] = 65536$. For the converse, suppose that $A[i^*] + A[j^*] = 65536$ for $i^* \leq j^*$. (For convenience, let i^* be the minimum index such that $A[i^*] + A[j^*] = 65536$, and let j^* be the maximum index so that $A[i^*] + A[j^*] = 65536$. This implies that $A[i] < A[i^*]$ and $A[j] > A[j^*]$ for all $i < i^*$ and $j > j^*$, since the array is sorted.) Now suppose that i is incremented to i^* before j is decremented to j^*. (The other case is analogous.) Then we know that $j > j^*$, $i = i^*$, and thus that $A[j] + A[i] > A[j^*] + A[i^*] = 65536$. Therefore we decrement j in the next iteration of the while loop. We will continue to decrement j until we reach $j = j^*$, at which point we will return "yes."

The running time is obviously linear, because we do only a constant amount of work before incrementing i or decrementing j, and we can only do so $O(n)$ times before $i > j$.

Problem 2-3. Beating the sorting lower bound?

(a) You are a spammer. You have an array A with n entries, each containing information about a person. Each person is from one of k different families, and there are exactly n/k members of each family in the array. (You may assume that each family has a unique last name.) You would like to sort the n entries in alphabetical order according to ⟨*last name, first name*⟩, using a comparison-based sorting algorithm. You know how to sort using $\Theta(n \log n)$ comparisons, but since A is truly massive in size, you would like to sort the data faster. So you hire King Arthur Anderson Consulting to address your problem.

The Anderson Consultant claims that he can sort A using $o(n \log n)$ comparisons! He says he can do this because the array is partially sorted in the following sense:

(i) Entries $A[1, \ldots, n/k]$ correspond to the members of one of the families, entries in $A[n/k + 1, \ldots, 2n/k]$ correspond to the members of another family, etc. In other words, entries in $A[(i-1)n/k + 1, \ldots, i \cdot n/k]$ for $i, 1 \leq i \leq k$ correspond to all the n/k members of one of the k famillies.

(ii) The families are *already* in alphabetical order according to last name. In other words, the last name of the family $A[1, \ldots, n/k]$ comes alphabetically before the last name of the family $A[n/k+1, \ldots, 2n/k]$, etc. In general, the last name of the family $A[(i-1)n/k + 1, \ldots, i \cdot n/k]$ comes alphabetically before the last name of the family $A[i \cdot n/k + 1, \ldots, (1+1)n/k]$ for all $i, 1 \leq i < k$.

You verify that (i) and (ii) are indeed correct. Should you believe his claim for all values of k? If not, for what values of k is it possible to sort A using $o(n \log n)$ comparisons? Prove your answer.

Solution: You shouldn't believe him in general.

For example, consider $k = 2$. The first $n/2$ elements (family #1) can be in any of $(n/2)!$ orders, and the last $n/2$ elements (family #2) can be in any of $(n/2)!$ orders. Therefore, by the same argument as given in class for the lower bound for comparison-based sorting, the number of leaves in the decision tree is $[(n/2)!]^2$. And thus the height of the tree must be at least $\log[(n/2)!]^2 = 2\log(n/2)! = \Omega(n \log n)$.

The generalization of this argument says that the depth of the decision tree must be $\Omega(\log[(n/k)!]^k) = \Omega(k \cdot (n/k) \cdot \log(n/k)) = \Omega(n \log(n/k))$. In order for $n \log(n/k)$ to grow slower than $n \log n$, we need $\log(n/k) = o(\log n)$. In other words, we need

$$\begin{aligned} \log(n/k) &= o(\log n) \\ \log n - \log k &= o(\log n) \\ \log k &= \log n - o(\log n) \\ \log k &= \log n(1 - o(1)) \\ k &= n^{1-o(1)} \end{aligned}$$

in order for this lower bound to be asympotically slower than $n \log n$.

In the case that $k = n^{1-o(1)}$, we can indeed achieve $o(n \log n)$: we sort each group using, say, MergeSort, and then, by the guarantee about the order of the groups, the entire array is sorted. The total time taken per group is $O(n/k \log(n/k))$, and there are k groups. So the total time we take is $k \cdot O((n/k) \log(n/k)) = c \cdot n \log(n/k) = c \cdot n \cdot o(\log n)) = o(n \log n)$.

(b) Anderson goes bankrupt, and loses all of your data. You have a scrambled backup copy of A, so you can no longer assume that (i) and (ii) hold. You hire another consultant from ConsultingAgency.com to sort the data. She invents new algorithms that are not comparison based. She claims that she can sort your data in $o(n)$ time. Should you believe her? Prove your answer.

Solution: No. Any algorithm to sort a list (comparison-based or not) must scan all n integers in the input–you can't sort what you haven't read–which requires $\Omega(n)$ time.

(c) After proceedings in bankruptcy court, you recover your partially sorted array. The ConsultingAgency.com consultant says that she can also use her algorithms on this partially sorted array for any values of k. For what values of k should you believe her claim? Prove your answer.

Solution: For $k = n$, the array is in fact already sorted! But for any $k < n$, each family the input might be unsorted. Thus to sort any one family, we must use $\Omega(n/k)$ steps simply to read that section of the input. In total, this requires $\Omega(n)$ steps.

Problem 2-4. Optimal investment strategies (with insider trading).

You are given an array A of n integers. Entry $A[i]$ is the stock price of BIM on day i. Your goal is to make the most money that you can by buying BIM once and subsequently selling once during this n-day period.

For any $r \geq \ell$, you can buy shares of BIM on day ℓ at price $A[\ell]$ and sell on day r at price $A[r]$. You need to find two indices ℓ and r such that $r \geq \ell$ and $A[r] - A[\ell]$ is maximized.

It is easy to find the best ℓ and r in time $O(n^2)$, simply by iterating over all possible pairs ℓ and r. This question asks you to give a more efficient algorithm.

(a) Give an $O(n \log n)$ algorithm to solve this problem. Your algorithm should take A as input, and produce two indices ℓ and $r \geq \ell$ such that $A[r] - A[\ell]$ is maximized.

Solution: We consider the pair $\langle \ell^*, r^* \rangle$ that maximize $v_{\ell,r}$. The simple idea of this divide-and-conquer algorithm is the following: either (1) both ℓ^* and r^* are in the range $[1, n/2]$, (2) both are in the range $[n/2, n]$, or (3) $1 \leq \ell^* < n/2 < r^* \leq n$. We'll handle the first two cases by making recursive calls on $A[1 \ldots n/2]$ and $A[n/2 \ldots n]$. What can we say in the third case?

$$\begin{aligned} v_{\ell^*,r^*} &= A[r^*] - A[\ell^*] \\ &= (A[r^*] - A[n/2]) + (A[n/2] - A[\ell^*]) \\ &= v_{\ell^*,n/2} + v_{n/2,r^*}. \end{aligned}$$

It's clear that in this case ℓ^* must maximize $v_{\ell,n/2}$—and thus minimizing $A[\ell]$—and that r^* must maximize $v_{n/2,r}$ (and thus $A[r]$). And we can find the ℓ^* and r^* that maximize these values in linear time, by iterating over all the possible choices.

FIND-LARGEST-RANGE(A, L, R)
▷ Find B where $L \leq B[0] \leq B[1] \leq R$ that maximizes $A[B[1]] - A[B[0]]$.
1 **if** $(L = R)$
2 **then** $B[0] \leftarrow L$
3 $B[1] \leftarrow L$
4 **return** B
5 $M \leftarrow \lceil (L + R)/2 \rceil$
6 $x \leftarrow$ FIND-LARGEST-RANGE(A, L, M)
7 $y \leftarrow$ FIND-LARGEST-RANGE(A, M, R)
8 $z[0] \leftarrow z \in \{L, \ldots, M\}$ that minimizes $A[z]$
9 $z[1] \leftarrow z \in \{M, \ldots, R\}$ that maximizes $A[z]$
10 **return** $w \in \{x, y, z\}$ that maximizes $A[w[1]] - A[w[0]]$.

We have that $T(n) = 2T(n/2) + cn$ since steps 6 and 7 require linear time each, and thus that $T(n) = O(n \log n)$.

(b) Optional extra credit: Give a linear time algorithm for this problem.

Solution: The solution in part (a) handles cases (1) and (2) very efficiently, but computes the minimum value in $A[L \ldots M]$ and the maximum value of $A[M \ldots R]$ by a linear time scan. Instead we can compute these values recursively as well!

```
FIND-LARGEST-RANGE-FAST(A, L, R)
▷ Find ⟨B, min, max⟩ where L ≤ B[0] ≤ B[1] ≤ R that maximizes A[B[1]] − A[B[0]]
▷ and min and max are the indices of the minimum and maximum values in A[L ... R]
▷ respectively.
 1  if (L = R)
 2     then B[0] ← L
 3     B[1] ← L
 4     return ⟨B, L, L⟩
 5  M ← ⌈(L + R)/2⌉
 6  ⟨x, l_min, l_max⟩ ← FIND-LARGEST-RANGE-FAST(A, L, M)
 7  ⟨y, r_min, r_max⟩ ← FIND-LARGEST-RANGE-FAST(A, M, R)
 8  z[0] ← l_min
 9  z[1] ← r_max
10  w ← w ∈ {x, y, z} that maximizes A[w[1]] − A[w[0]]
11  return ⟨w, min(l_min, r_min), max(l_max, r_max)⟩.
```

It is easy to show by induction that *min* and *max* correctly compute the minimum and maximum values in $A[L \ldots R]$. The correctness of the algorithm follows immediately from that fact and the proof of correctness in part (a).

By inspection, each line of the program (aside from the recursive calls) takes $O(1)$ time. Thus we have $T(n) = 2T(n/2) + c$, and that $T(n) = O(n)$.