# Theory of Algorithms. Homework 2

Peter Racioppo

**1.) Consider two sums $X = x_1 + ... + x_n$ and $Y = y_1 + ... + y_m$. Give an algorithm that finds indices *i* and *j* such that swapping $x_i$ and $y_j$ make the two sums equal. Analyze your algorithm.**

We want $X - x_i + y_j = Y - y_j + x_i \Rightarrow \frac{1}{2}(X - Y) = x_i - y_j$. Suppose, without loss of generality, that $n \geq m$. Computing $\frac{1}{2}(X - Y)$ takes $O(n)$ time since we have to look at each element to compute a sum, so this is a lower bound on the run time of any possible algorithm. Sort *X* and *Y* from smallest to largest and store the sorted elements in vectors $\mathbf{X} = [\min(X), ..., \max(X)]$ and $\mathbf{Y} = [\min(Y), ..., \max(Y)]$. This takes $O(n \log n)$ time. The most straightforward thing to do now is compute $x_i - y_j$ for all *nm* combinations of *i* and *j*, each computation costing constant time. That would take $O(n^2)$ time in total. But we can do better with a divide-and-conquer approach. Define $\frac{1}{2}(X - Y) = A$. Go to the middle element of **X** and compute $x_i - y_j$ for each element of **Y**. (If **X** has an odd number of elements you can go to the middle one; otherwise, there will be two middle elements, and you can just have a rule that you pick the upper or lower of these.) If $x_i - y_j < A$ for every *x*,*y* pair corresponding to this element of **X**, we need to move in the direction of increasing **X** elements, since this is the only way to increase *A*. Otherwise, we need to go in the direction of decreasing **X** elements. Using this rule, we'll do a binary search on the elements of **X**. Likewise, we can do a binary search on the elements of **Y**. Namely, if we're at element *k* of **X**, we compute $x_k - y_j$ for the middle element of **Y**. If $x_k - y_j < A$, we move toward lower valued elements of **Y**, and otherwise we move toward higher valued elements. Thus, the total run time is $O(\log n \log m) = O(\log^2 n)$. The whole algorithm is then $O(n \log n)$, since the largest contribution comes from sorting.

**2.) Consider distinct items $x_1, ..., x_n$ with positive weights $w_1, ..., w_n$ such that $\sum_{i=1}^{n} w_i = 1$. The weighted median is the item $x_k$ that satisfies $\sum_{x_i < x_k} w_i < 0.5$ and $\sum_{x_j > x_k} w_j \leq 0.5$.**

**a) Show how to compute the weighted median of *n* items in worst-case time $O(n \log n)$ using sorting.**

First, sort the $w_i$ by increasing value and arrange the corresponding $x_i$ in the same order in worst-case $O(n \log n)$ time. Define arrays **W** and **X** to hold these values. Then, sum the elements of **W**, starting at the first element and moving right, checking in constant time at each step whether $\sum w_i \geq 0.5$ and stopping as soon as this is the case, and also incrementing a counter that records the number of steps taken, in a total of $O(n)$ time. The *k*th element of **W**, that is the weight corresponding to the weighted median, is the second to last element we examined, so that *k* is the number of steps we took. The weighted median $x_k$ is the *k*th element of **X**. Sorting is the bottleneck here, so the total algorithm is $O(n \log n)$.

**b.) Show how to compute the weighted median in worst-case time $O(n)$ using a linear time median-finding algorithm as a black box.**

Apply the median-finding algorithm to find the median of the $w_i$ in $O(n)$ time. Compare every element of **W** to the median and sort **W** into two subsets consisting of the elements smaller than and greater than the median, in $O(n)$ time (each of the $n$ comparisons takes constant time). Sum the left subset in $O(n)$ time. If the sum of the left subset is less than or equal to 0.5, $w_k$ is to the right of the median. Otherwise, it is to the left of the median. Subtract the sum of the left subset from the set that we're looking for, namely 0.5. We'll continue in this way, using the median function to perform a binary search. For example, if the left subset summed to 0.3, we would next divide the right subset into two still smaller sets; if the leftmost of these two subsets summed to less than 0.5–0.3 = 0.2, we would continue stepping right. Both our median search and our sum functions take $O(m)$ time, where $m$ is the number of elements in each subset, which falls by a factor of two at each step. Thus, the total run time of the algorithm is $O\left(\sum_{i=0}^{a}\left(\frac{1}{2}\right)^i n\right) = O\left(\left(2-2^{-a}\right)n\right)$, where $a = \log n$. But $n \le \left(2-2^{-\log n}\right)n \le 2n$, so the run time of this algorithm is $O(n)$.

**3.) Consider a set of *n* intervals [$a_i$,$b_i$] that cover the unit interval. That is, [0,1] is contained in the union of intervals.**

**a.) Describe an algorithm that computes a minimum subset of the intervals that also covers [0,1].**

Scan left to right and add all left endpoints to an array **L**. In addition, create an array **R** that holds the right endpoints in the order in which their left endpoints appear in **L**. Add the first element of **R** to an array **U**, which will hold all the elements of a minimal subset for the interval between 0 and its greatest value. Now move through **L**, removing each element of **L** as you come to it and incrementing a counter at each step, and stopping when an element is encountered which is greater than the greatest element in **U**. Say that $k$ such steps through **L** have been made. Find the maximum of the $k$ first elements of **R** and also remove these elements from **R**. This maximum right endpoint corresponds to the line segment we want, so we add it to **U**. Continue this procedure, using the rightmost endpoint in **U** as our reference right endpoint, until 1, the value of the end of the interval, is added to **U**. Note that degeneracy does not pose a problem. If some number of left endpoints overlap the greatest element in **U**, this is handled in exactly the same way as if they came before it. If some number of right endpoints in a $k$-sized subset of **R** overlap, so that there are multiple maxima, we can just arbitrarily choose one of these segments.

**b.) Prove the correctness and analyze the running time of your algorithm.**

Initializing **L** should take $O(n)$ time, and then we should need $O(n \log n)$ to sort the right endpoints into **R**. Each time we move through **L**, we're making a constant time comparison of two numbers $k$ times, where $k$ is the number of elements in the relevant subinterval, so each sub-operation is $O(k)$. We then have to find a maximum from $k$ elements in **R**, which should probably take $O(k)$ time (and certainly no more than $O(k \log k)$). But there are no more than $n$–1 elements that will be looked at, so the total run times of these operations is $O(\sum k) = O(n)$.

The set $U$ is minimal because we've guaranteed that there is always exactly one pair of intervals that is overlapping at each endpoint (other than the first and last), so the set covers the whole interval and removing any one segment will leave a gap between its preceding and succeeding segments. Also, the procedure is completely general. There is always at least one left point to the left of every right point, and the maximum right endpoint of the set of corresponding line segments must always be to the right of the previous right endpoint, since the full set is guaranteed to cover [1,0]. So our procedure always produces a minimal set.

**4.) Given a convex polygon *P*, divide it into triangular regions with minimum total perimeter.**

**(i) How many diagonals does *P* contain in total?**

Each vertex in $P$ forms a diagonal with every other vertex, except the two adjacent to it. Since an $n$-sided polygon, has $n$-vertices, it will thus have $n$–3 diagonals for each vertex, or $n(n–3)$ in total. However, if directionality doesn't matter, this procedure will have double counted the diagonals, since each diagonal contains two vertices, so in fact there are $n(n–3)/2$ diagonals in $P$.

**(ii) How many diagonals does any legal set of triangles contain?**

Wherever we draw a diagonal, we will break the polygon into two smaller polygons, and each of these will continue to be broken into smaller polygons as more diagonals are drawn, until all remaining polygons are triangles. We cannot, in this process, draw a diagonal across multiple polygons, since doing so would require diagonals to cross. At any step, if we divide a polygon with $n$ sides into two polygons with $x$ and $y$ sides, we must have $x+y–2 = n$, since the joining edge between the two sub-polygons is counted as a side for each of them, but is not one of the sides of the larger polygon. Also, the number of legal diagonals in the union of the two sub-polygons is one less than the number of legal diagonals in the full polygon, since their adjoining edge is a diagonal in the larger polygon, that is $d_x+d_y+1 = d_n$. We can now prove the claim by induction on the number of sides $n$ in a convex polygon. In the base case $n = 4$, drawing a diagonal anywhere will partition the polygon into two triangles, which cannot be further subdivided, so there is exactly 1 legal diagonal. Suppose that for convex polygons with any number of sides $m \leq n-1$ the number of legal diagonals is $d_m = m–3$. This implies that $d_n = x–3+y–3+1 = x+ y–5 = n–3$, which completes the induction.

**(iii) Given a legal set of triangles, express the total cost of these triangles in terms of the perimeter of *P* and the lengths of those edges of the triangles that are diagonals of *P*.**

No two legal triangles can share an exterior edge of $P$, since doing so would require their diagonals to cross. (If two triangles share an edge, they share two vertices. If they are distinct, they don't share their third vertex, and thus one must have a vertex to the left or right of the other on the perimeter of $P$, which would require their diagonals to cross.) Also, each legal triangle must share exactly one legal diagonal with an adjacent legal triangle, since each diagonal bisects $P$ into two sub-polygons, each of which is further subdivided into triangles, so that the border of any two sub-polygons is the mutual border of two triangles.

The sum of the perimeters of the triangles will thus be the perimeter of $P$ plus twice the sum of the legal diagonals.

**(iv) Derive a recursion to compute the optimal solution. State and prove the running time of the resulting algorithm.**

By (iii), we want to minimize the sum of the lengths of the diagonals, since the perimeter of $P$ is fixed.

We'll use a greedy algorithm. At any step, the shortest diagonal must belong to one of the exterior triangles, that is the triangles that contain two exterior edges of $P$. This is the case because any other diagonal forms a polygon with more than two edges of $P$ (and no other diagonals). Since $P$ is convex, drawing an edge between one of the vertices that forms the diagonal and the vertex immediately before the other vertex that forms the diagonal will form an edge that is smaller than the diagonal, and this can be continued until the diagonal of an external triangle is reached.

So, at any step, we choose the shortest diagonal in the polygon of interest $P_i$, where $P_i$ is originally $P$, and then define a new $P_i$ as the sub-polygon that is formed by separating the exterior triangle formed by the diagonal we selected. Any diagonal that crosses the external triangle we just selected can never be part of the optimal solution. This is because any such diagonal would connect to the central vertex of the triangle we removed, but the diagonal formed by either of the side vertices of this triangle would always be shorter than this diagonal, again because $P$ is convex. Since we can always find an exterior triangle until $P_i$ has only three vertices, and since each step removes exactly one vertex from $P_i$, this procedure will gives us exactly $n$–3 vertices, exactly the number we need.

To prove that this algorithm is optimal, we need to show that at every step, it always stays ahead of any alternate algorithm. Any alternate algorithm would at each step have the same diagonals to pick from as would our algorithm, plus those we have eliminated. But we have already shown that the diagonals we have eliminated can't be part of the

solution, so at each step the alternate algorithm can do no better than considering the same set as us, and we always pick the minimal element of this set. Thus, our algorithm is optimal.

To compute the run time, note that each vertex of $P$ can form two exterior triangles, with the vertices that are two away from it around the perimeter of $P$, but this double counts the exterior triangles, so there are $n$ exterior triangles in total. Thus, in the first step, we need to check $n$ diagonals. In the next step, we need to check the same set minus the diagonal we just eliminated and plus the diagonal of the one new external triangle that's been created, and the same is true of all of the $n$–4 steps we take after the first step. Rather than performing these computations separately at each step, we can sort the sides of $P$, and at each step remove the minimum and add the new diagonal. Sorting takes $O(n \log n)$ time and placing each new diagonal at each of the $n$–4 steps after the first takes $O(\log n)$. So the full operation is $O(n \log n)$.

**5.) Let $A[1..m]$ and $B[1..n]$ be two strings.**

**a.) Modify the dynamic programming algorithm for computing edit distance between $A$ and $B$ for the case where there are only two allowed operations: insertions and deletions of individual letters.**

Substitution is just equivalent to deleting an element from both strings, so we can just write $\alpha_{ij} = 2\delta$.

$$\mathbf{OPT}(i,j) = \min \begin{cases} \delta + \mathbf{OPT}(i-1, j) \\ \delta + \mathbf{OPT}(i, j-1) \\ \mathbf{OPT}(i-1, j-1) + \begin{cases} 0 & \text{if} \quad A(i) = A(j) \\ 2\delta & \text{if} \quad A(i) \neq A(j) \end{cases} \end{cases}$$

I believe this can be simplified to:

$$\mathbf{OPT}(i,j) = \begin{cases} \mathbf{OPT}(i-1, j-1) & \text{if} \quad A(i) = B(j) \\ \min\{\delta + \mathbf{OPT}(i-1, j), \delta + \mathbf{OPT}(i, j-1)\} & \text{if} \quad A(i) \neq B(j) \end{cases}$$

The algorithm is still $O(nm)$.

**b.) A (not necessarily contiguous) subsequence of $A$ is defined by the increasing sequence of the indices, $1 \le i_1 < i_2 < \ldots < i_k \le m$. Use dynamic programming to find the largest common subsequence of $A$ and $B$ and analyze its running time.**

There are $\sum_{i=2}^{n} \binom{n}{i} = 2^n - n - 1 = O(2^n)$ subsequences of a string with $n$ elements, so brute force will take exponential time. Define $\mathbf{M}(A(i), B(j))$ as the length of the largest subsequence that $A$ and $B$ having in common, for some strings of length $i$ and $j$, respectively. We can write a recursive algorithm:

$$\mathbf{M}(A(i), B(j)) = \max \begin{cases} \mathbf{M}(A(i-1), B(j)) \\ \mathbf{M}(A(i), B(j-1)) \\ \mathbf{M}(A(i-1), B(j-1)) + \begin{cases} 1 & \text{if} \quad A(i) = B(i) \\ 0 & \text{if} \quad A(i) \neq B(i) \end{cases} \end{cases}$$

which reduces to:

$$\mathbf{M}(A(i), B(j)) = \begin{cases} 1 + \mathbf{M}(A(i-1), B(j-1)) & \text{if} \quad A(i) = B(j) \\ \max\{\mathbf{M}(A(i-1), B(j)), \mathbf{M}(A(i), B(j-1))\} & \text{if} \quad A(i) \neq B(j) \end{cases}$$

The $ij$th element $M_{ij}$ of matrix **M** is computed from the three elements immediately beneath it and to its left, $M_{(i-1),j}$ $M_{i,(j-1)}$, and $M_{(i-1),(j-1)}$. Each element takes constant time to compute, and there are $nm$ elements, so the algorithm takes $O(nm)$ time.

**c.) What is the relationship between the edit distance defined in (a) and the longest common subsequence computed in (b)? Provide a proof.**

If we delete all of the elements from $A$ and $B$ except those that belong to the largest common sequence (LCS), the remaining sequences will be identical, so $\mathbf{OPT}(i, j) = n + m - 2\mathbf{M}(i, j)$. (We could also delete all of the elements from one of the sequences that doesn't belong to the LCS and then add to it the elements of the other sequence, requiring the same number of steps.)

Actually, we also need to prove that the right hand side here is in fact the minimal number of edit operations required. We can imagine dividing each string into sequences of elements that are in the LCS and those that are not, so that these two types of subsection are listed alternately. We can then look for intersections of these subsections between subsequences of the LCS that contain particular elements in $A$ and $B$. None of these intersections of elements not in **M** can be longer than the adjacent sequences in **M**, since otherwise they would belong to the LCS, which implies that subtracting out elements not in the LCS is indeed the fastest way to go. (I'm not sure about this; this is only a sketch of how I think it'd be done.)