Advanced Analysis of Algorithms - Final (Solutions)

L. Kovalchick
LCSEE,
West Virginia University,
Morgantown, WV
{lynn@csee.wvu.edu}

1 Problems

1. Let $\mathbf{A}[1..n]$ be an array of n distinct numbers. If $i < j$ and $A[i] > A[j]$, then pair (i, j) is called an inversion of $\mathbf{A}$. Design an algorithm that takes as input such an array and outputs the *number* of inversions of the array. Your algorithm should run in time $O(n \cdot \log n)$ in the worst case. (*Hint: Merge-Sort.*)

   Solution:
   Algorithm (1.1) represents a divide-and-conquer algorithm for the inversion-pair problem; note that the algorithm also sorts the input array $\mathbf{A}$.

```
Function NUMBER-INVERSION-PAIRS(A, low, high)
 1: if (low < high) then
 2:    mid = (low+high)/2
 3:    c_l = NUMBER-INVERSION-PAIRS(A, low, mid)
 4:    c_u =NUMBER-INVERSION-PAIRS(A, mid + 1, high)
 5:    c_m =MERGE-INVERSION-PAIRS(A, low, mid, high)
 6:    return(c_l + c_u + c_m)
 7: else
 8:    return(0)
 9: end if
```

Algorithm 1.1: Algorithm for determining the number of inversion pairs in the sub-array $\{A[low], A[low + 1], \ldots, A[high]\}$.

1.1 Correctness

We prove the correctness of Algorithm (1.1), by using induction on the number of elements in the array $\mathbf{A}$. Note that the number of elements in $\{A[low], A[low + 1], \ldots, A[high]\}$ is $(high - low + 1)$ and is denoted by $|\mathbf{A}|$.

(a) Observe that the number of inversion pairs in $\mathbf{A}$ is 0, when $|\mathbf{A}| \leq 1$. When $\mathbf{A}$ has at most one element, $(high \leq low)$ and Step (8) of Algorithm (1.1) is executed. It follows that Algorithm (1.1) works correctly when $|\mathbf{A}| \leq 1$.

(b) Assume that Algorithm (1.1) returns the correct number of inversion pairs when $|\mathbf{A}| \leq k$, for all $k = 2, 3, \ldots, (n - 1)$ and also sorts $\mathbf{A}$. Now consider the case, when $|\mathbf{A}| = n$. Step (2) of Algorithm (1.1) breaks up the array into 2 sub-arrays. Let $\mathbf{A_l}$ denote the sub-array $\{A[low], A[low + 1], \ldots, A[mid]\}$ and $\mathbf{A_h}$ denote the sub-array $\{A[mid + 1], A[mid + 2], \ldots, A[high]\}$. Clearly $|\mathbf{A_l}| < n$ and $|\mathbf{A_h}| < n$.

Function MERGE-INVERSION-PAIRS($\mathbf{A}, low, mid, high$)

1: {We assume that the sub-arrays $\{A[low], A[low+1], \ldots, A[mid]\}$ and $\{A[mid+1], A[mid+2], \ldots, A[high]\}$ have been sorted.}
2: $p = low$; $q = (mid+1)$; $index = low$; $c_m = 0$.
3: **while** $(p \leq mid)$ **and** $(q \leq high)$ **do**
4: **if** $(A[p] \leq A[q])$ **then**
5: $C[index] = A[p]$
6: {$A[p]$ does not form an inversion pair with an element of $\mathbf{A_h}$.}
7: $p++$
8: **else**
9: $C[index] = A[q]$
10: {$A[q]$ forms an inversion pair with each of the elements $\{A[p], A[p+1], \ldots, A[mid]\}$}
11: $q++$
12: $c_m += (mid - p + 1)$
13: **end if**
14: $index++$
15: **end while**
16: **if** $(p > mid)$ **then**
17: {All the elements in $\mathbf{A_l}$ have been processed; copy the elements in $\mathbf{A_h}$ into $\mathbf{C}$.}
18: **for** $(i = q$ **to** $high)$ **do**
19: $C[index] = A[i]$
20: $index++$
21: **end for**
22: **else**
23: {All the elements in $\mathbf{A_h}$ have been processed; copy the elements in $\mathbf{A_l}$ into $\mathbf{C}$.}
24: **for** $(i = p$ **to** $mid)$ **do**
25: $C[index] = A[i]$
26: $index++$
27: **end for**
28: **end if**
29: {Now copy the elements in $\mathbf{C}$ back into $\mathbf{A}$.}
30: **for** $(i = low$ **to** $high)$ **do**
31: $A[i] = C[i]$
32: **end for**
33: $\{\{A[low], A[low+1], \ldots, A[high]\}$ is now in sorted order}
34: **return**(c_m)

Algorithm 1.2: The Merge Procedure.

Hence, we can apply the inductive hypothesis to conclude that the number of inversion pairs computed in Steps (3) and (4) are correct, i.e., c_l stores the correct number of inversion pairs in $\mathbf{A_l}$ and c_h stores the correct number of inversion pairs in $\mathbf{A_h}$. Further $\mathbf{A_l}$ and $\mathbf{A_h}$ are sorted. We now analyze the MERGE-INVERSION-PAIRS() procedure. An element of $\mathbf{A_l}$ that is moved into $\mathbf{C}$, does not form inversion pairs with any of the elements in $\{A_h[q], A_h[q+1], \ldots, A_h[high]\}$. However, an element of $\mathbf{A_h}$ that is moved into C forms inversion pairs with all the elements in $\mathbf{A_l}$ that have not been processed, i.e., $\{A_l[p], A_l[p+1], \ldots, A_l[mid]\}$. Since we are only concerned with the number of inversion pairs, we update the inversion pair count by $(mid - p + 1)$. An inversion pair (i, j) in $\{A[low], A[low+1], \ldots, A[high]\}$ must have one of the following forms:

i. $i < j$, $A[i] > A[j]$, and $i, j \in \{low, (low+1), \ldots, mid\}$ - This case is recursively handled (Step (3) of Algorithm (1.1));

ii. $i < j$, $A[i] > A[j]$, $i, j \in \{(mid+1), (mid+2), \ldots, high\}$ - This case is also recursively handled (Step (4) of Algorithm (1.1));

iii. $i < j$, $A[i] > A[j]$, $i \in \{low, (low+1), \ldots, mid\}$ - and $j \in \{(mid+1), (mid+2), \ldots, high\}$ - This case is handled by the MERGE-INVERSION-PAIRS() procedure.

Thus, we have accounted for all the inversion pairs in $\{A[low], A[low+1], \ldots, A[high]\}$, thereby proving that if Algorithm (1.1) is correct when $|\mathbf{A}| = k$, $k = 2, 3, \ldots, (n-1)$, then it must be correct for $|\mathbf{A}| = n$. Applying the principle of mathematical induction, we conclude that Algorithm (1.1) is correct.

1.2 Analysis

Let $T(n)$ denote the running time of Algorithm (1.1) on an array of n elements. It is not hard to see that (exactly as in MERGE-SORT())

$$\begin{aligned} T(n) &= 1, && \text{if } n = 1 \\ &= 2 \cdot T(\frac{n}{2}) + \Theta(n), && \text{otherwise} \end{aligned}$$

It follows that $T(n) = \Theta(n \cdot \log n)$, as per the Master Theorem. □

2. Let P be a convex polygon in 2-dimensional space, having n vertices. A triangulation of P is an addition of diagonals connecting the vertices of P, so that each interior face is a triangle. The weight of a triangulation is the sum of the lengths of the diagonals. Assuming that we can compute lengths, add, and compare them in constant time, give an efficient algorithm for calculating the minimum weight triangulation of P.
(*Hint: Dynamic Programming.*)

Solution: Observe that any optimal triangulation (indeed, any triangulation) of P consists of a "first" diagonal, which partitions the polygon into 2 parts; these parts are also polygons which need to be triangulated (See Figure (1)).

The key observation is that in any optimal triangulation, once the first diagonal is chosen for a polygon, the 2 sub-polygons (sub-problems) that result must also be triangulated optimally; otherwise, we can combine optimal solutions to the sub-problems and get a better solution to the initial problem, thereby contradicting the optimality of the initial triangulation. In other words, the principle of optimality applies and we can use Dynamic Programming.

The Dynamic Program is based on the following sequence of decisions that need to be made:

(a) Make a decision on the first diagonal, creating 2 sub-polygons.

(b) Make a decision on the first diagonals of the created sub-polygons, thereby creating 4 sub-polygons and so on.

(c) The recursion bottoms out, when the created sub-polygon is a triangle; in this case the sub-polygon is already a triangulation, with cost of triangulation 0.

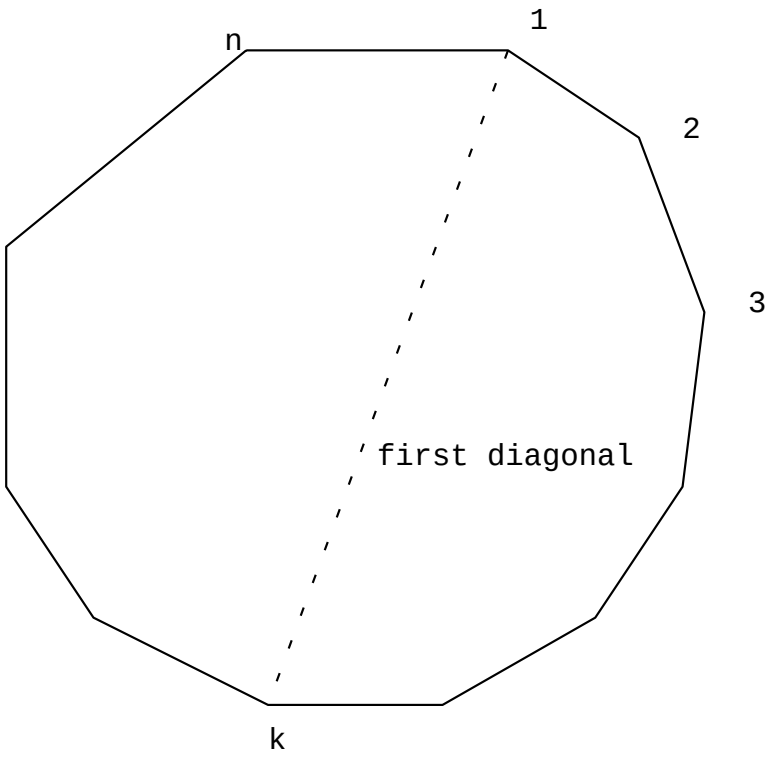

Figure 1: Optimal triangulation

Observe that given a polygon on k points, there are at most $O(k^2)$ choices for the first diagonal. Accordingly, we use the bottom-up procedure, described by Algorithm (1.3).

Function OPTIMAL-TRIANGULATE(P)
1: **for** ($k = 4$ **to** n) **do**
2: Compute the optimal triangulations for all k-gons of the initial polygon, formed by $(k-1)$ edges of the original polygon and one chord, connecting the k^{th} vertex to the first vertex.
3: **end for**

Algorithm 1.3: Algorithm for optimal polygon triangulation

Note that there are at most n, k-gons, for any $k = 4, 5, \ldots, n$. (Why?) Thus, we begin by computing the optimal triangulations of all the 4-gons, then use these results to compute the triangulations of all the 5-gons and so on. The crucial observation is that when we wish to calculate the optimal triangulation of a given k-gon, the optimal triangulation of all v-gons $v < k$, have already been calculated. Accordingly, computing the optimal triangulation of a k-gon, consists of the following steps:

(a) For each of the $O(k^2)$ pairs of vertices between which the first diagonal can be drawn,
   i. Look up the costs of the optimal triangulations of the resulting sub-polygons, say s_1 and s_2.
   ii. Compute the cost of this triangulation, by summing up the cost of the diagonal, s_1 and s_2.

(b) Take the minimum cost over all these $O(k^2)$ first diagonals.

The total time taken by the above strategy is computed as follows: For a given k, there are at most $O(n)$ distinct k-gons. For a given k-gon, there are at most $O(k^2)$ first diagonal choices. Corresponding to each of these choices, the optimal triangulation can be computed in $O(1)$ time. Thus for a given k, the total time spent is at most $O(n \cdot k^2)$. Thus, the total time taken by the algorithm is at most $\sum_{k=4}^{n} O(n \cdot k^2) = O(n^4)$.

The space requirement for the algorithm is $O(n^2)$, since for each k, we need $O(n)$ space, to store the costs of the optimal triangulations of the n, k-gons corresponding to that k.

For a slightly more efficient (and less intuitive!) approach in terms of running time, see [CLR92]. □

3. Let $\mathcal{T} = \{T_1, T_2, \ldots, T_n\}$ denote a collection of n tasks. Task T_i has start time s_i and finish time f_i, with $s_i \leq f_i$, i.e., task T_i *must* start at time s_i and it will finish at time f_i. Two tasks T_i and T_j are non-conflicting, if either $f_i \leq s_j$ or $f_j \leq s_i$. The tasks in $\mathcal{T}$ are to be assigned to machines, so that the resultant schedule is non-conflicting. Design an algorithm that schedules the tasks in $\mathcal{T}$ using the *fewest* number of machines. Clearly, I can obtain a non-conflicting schedule, by assigning each job to a different machine! (*Hint: Use a greedy strategy that sorts the tasks in $\mathcal{T}$ by their start times.*)

   Solution:
   We will assume that the tasks in $\mathcal{T}$ are sorted in order by increasing start times. Observe that if they were not sorted, then we could sort them into this order in time $O(n \cdot \log n)$, breaking ties arbitrarily. Algorithm (1.4) represents a greedy algorithm for this problem.

Function TASK-SCHEDULE(A set $\mathcal{T}$ of tasks, such that each task has a start time s_i and a finish time f_i)

```
1:  m = 0
2:  while (T ≠ ∅) do
3:     Remove from T the task i with smallest start time s_i
4:     if (there is a machine j with no task conflicting with task i) then
5:        schedule task i on machine j
6:     else
7:        m = m + 1
8:        schedule task i on machine m
9:     end if
10: end while
11: return(m)
```

Algorithm 1.4: A greedy algorithm for the task scheduling problem.

1.3 Correctness

We must prove that Algorithm (1.4) always produces the optimal solution.

Theorem 1.1 *Given a set of n tasks specified by their start and finish times, Algorithm (1.4) produces a schedule of the tasks with the minimum number of machines.*

Proof: Suppose that Algorithm (1.4) does not work; that is, it finds a non-conflicting schedule using k machines. Now, suppose that there is another non-conflicting schedule that uses only $(k-1)$ machines. Let k be the last machine allocated by our algorithm, and let i be the first task scheduled on k. By the structure of the algorithm, when we scheduled i, each of the machines 1 through $(k-1)$ contained tasks that conflict with i. Since they conflict with i and because we consider tasks ordered by their start times, all the tasks currently conflicting with task i must have start times less than or equal to s_i and have finish times after s_i. In other words, these tasks not only conflict with task i, they all conflict with each other, which implies it is impossible for us to schedule all the tasks in $\mathcal{T}$ using only $(k-1)$ machines. Therefore, k is the minimum number of machines needed to schedule all the tasks in $\mathcal{T}$. □
□

4. In class, we showed that the MAX2SAT problem is `NP-complete`. What is the complexity of the MAX1SAT problem? Either design a polynomial time algorithm for MAX1SAT or show that the problem is `NP-complete`.

Proof: Observe that the MAX1SAT problem is defined as follows, given a 1SAT formula ϕ and a number k, is there an assignment to the literals $\{x_1, x_2, \ldots, x_n\}$ such that the number of satisfied clauses is greater than or equal to k? This problem is trivial since every clause of a 1SAT formula has exactly 1 literal. We are given a set of n literals where $\{x_1, x_2, \ldots, x_n\}$ and a set of m clauses; note that $m \leq 2 \cdot n$, because for each literal only x_i and x_i' can exist for $i = 1, 2, \ldots, n$. The algorithm to solve MAX1SAT is easy; we will create a counter and initialize it to 0. Observe that if a literal x_i, exists in one clause and the negation of that literal (i.e., x_i') exists in another clause, then only one of these clauses is true at any given time; so, we will increment our counter by 1 each time a pair of literals is encountered. On the other hand, if the literal x_i exists in one clause, but its negation (i.e., x_i') does not exist in any clause, then we can immediately assign x_i to true and increment our counter by 1. A similar argument holds when x_i' exists in a clause, but x_i does not exist in any clause. Observe that this problem can be solved in polynomial time (actually it is solved in linear time). □

5. The Satisfiability problem (SAT) is concerned with a finding a satisfying assignment to a conjunction of clauses. $kSAT$ is defined as the restriction of SAT in which each clause has exactly k literals. HornSAT is the restriction of SAT in which each clause is `Horn`, i.e., each clause has at most one positive literal. In class, we showed that 3SAT is `NP-complete`, whereas, 2SAT and HornSAT are decidable in polynomial time. The HornSAT⊕2SAT problem is the restriction of SAT in which each clause is either Horn or has exactly 2 literals. Argue that the HornSAT⊕2SAT problem is `NP-complete`. (*Hint: Use a reduction from 3SAT.*)

   Proof: First note that the HornSAT⊕2SAT problem is clearly in `NP`, since a Non-deterministic Turing machine can guess an assignment from $\{0,1\}^n$ and output **true** if the input instance is satisfied and **false** otherwise, i.e., the verification of instances can be carried out in polynomial time.

   We now reduce the 3SAT problem to the HornSAT⊕2SAT problem to establish its `NP-Hardness`.

   Let $\phi = C_1 \wedge C_2 \ldots C_m$ denote an instance of 3SAT on the variables $\{x_1, x_2, \ldots, x_n\}$. We transform each clause C_i into a collection of one or more clauses S_i, such that

   (a) Each clause in S_i is either a 2SAT clause or a Horn clause, and

   (b) The 3SAT formula ϕ is satisfiable if and only if the conjunction of all the clauses in S_1 through S_m is satisfiable.

   Clause C_i is transformed into S_i as follows.

   (a) C_i has 2 or more negative literals - $S_i = \{C_i\}$, i.e., nothing needs to be done, since C_i is already Horn. It is also trivially true, that an assignment to ϕ that satisfies C_i must satisfy S_i and vice versa.

   (b) C_i has exactly 2 positive literals - Let $C_i = (x_i, x_j, \bar{x_k})$. S_i is $\{(x_i, \bar{w_{i1}}, \bar{x_k}),\ (x_j, w_{i1})\}$. Consider an assignment to ϕ that satisfies C_i; it is not hard to see that we can choose w_{i1} appropriately to ensure that $(x_i, \bar{w_{i1}}, \bar{x_k}) \wedge (x_j, w_{i1})$ is satisfied. Likewise, if $(x_i, \bar{w_{i1}}, \bar{x_k}) \wedge (x_j, w_{i1})$ is satisfied by an assignment, then one of $\{x_i, x_j\}$ must be set to true or x_k must be set to false in this assignment. If this is not the case, then S_i simplifies to $(w_{i1})(\bar{w_{i1}})$, which is unsatisfiable, contradicting the hypothesis!

   (c) C_i has all 3 literals positive - Let $C_i = (x_i, x_j, x_k)$. S_i is $\{(x_i, \bar{w_{i1}}, \bar{w_{i2}}),\ (x_j, w_{i1}),\ (x_k, w_{i2})\}$. We use an argument similar to the above case that establishes that if C_i is satisfied by an assignment to ϕ, then the conjunction of the clauses in S_i is satisfiable and vice versa.

   Let ϕ' denote the formula created by taking the conjunction of the clauses in S_i, $i = 1, 2, \ldots, m$. We have thus established that ϕ is satisfiable, if and only if ϕ' is; it follows that the Horn⊕2SAT problem is `NP-complete`. □

References

[CLR92] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* MIT Press and McGraw-Hill Book Company, 6th edition, 1992.