Problem Set 9 Solutions

Reading: Chapters 32.1–32.2, 30.1–30.2, 34.1–34.2, 35.1

Both exercises and problems should be solved, but *only the problems* should be turned in. Exercises are intended to help you master the course material. Even though you should not turn in the exercise solutions, you are responsible for material covered in the exercises.

Mark the top of each sheet with your name, the course number, the problem number, your recitation section, the date and the names of any students with whom you collaborated.

Three-hole punch your paper on submissions.

You will often be called upon to "give an algorithm" to solve a certain problem. Your write-up should take the form of a short essay. A topic paragraph should summarize the problem you are solving and what your results are. The body of the essay should provide the following:

1. A description of the algorithm in English and, if helpful, pseudocode.
2. At least one worked example or diagram to show more precisely how your algorithm works.
3. A proof (or indication) of the correctness of the algorithm.
4. An analysis of the running time of the algorithm.

Remember, your goal is to communicate. Full credit will be given only to correct algorithms that are *which are described clearly*. Convoluted and obtuse descriptions will receive low marks.

Exercise 9-1. On-line String Matching

Recall that in an on-line algorithm, the input is generated as the algorithm is running. The idea is to solve the problem efficiently before seeing all the input. You can't scan forward to look at 'future' input, but you can store all input seen so far, or some computation on it.

(a) In this setting, the text $T[1 \ldots n]$ is being broadcast on the network, one letter at a time, in the order $T[1], T[2], \ldots$. You are interested in checking if the text seen so far contains a pattern P, where P has length m. Every time you see the next letter of the text T, you want to check if the text seen so far contains P.

Design an algorithm that solves this problem efficiently. Your algorithm should use no more than $\Theta(m)$ time on preprocessing P. In addition it should do only constant amount of work per letter received. Your algorithm can be randomized, with constant probability of correctness.

Solution:

For both parts we are going to use Karp-Rabin (KR) algorithm seen in the class (the finite state machine / Knuth-Morris-Pratt algorithms could be used as well). As in the class, we will assume that the alphabet is $\{0, 1\}$ (the algorithm can be easily modified to handle arbitrary alphabet). Recall that KR used $\log^{O(1)} n$ time to find a random prime q, and $O(m)$ time to "hash" the pattern P; we will refer to these two steps as "preprocessing".

After the preprocessing, the KR algorithm computes the hash values for the m-length substrings of T in an incremental way. Whenever a new symbol $T[i]$ is given, KR computes the hash value of the substring $T[i-m+1 \ldots i]$ in constant time. Thus, KR works in our setting without any modifications.

(b) Now say that you have the same pattern P, but the text $T[1 \ldots n]$ is being broadcast in reverse. That is, in the order $T[n], T[n-1], \ldots$ Modify your algorithm so that it still detects the occurrence of P in the text $T[i \ldots n]$ immediately (i.e., in constant time) after the letter $T[i]$ is seen.

Solution:

For an array $A[1 \ldots n]$, let A^R be the "reverse" of A, i.e., $A^R[i] = A[n-i+1]$. Searching for an occurrence of P in T is equivalent to searching for an occurrence of P^R in T^R; thus, we can focus on the latter task. Since the symbols of T are given in the reverse order, it means that the symbols of T^R are given in the proper order (i.e., $T^R[1], T^R[2] \ldots$). Thus, we can find the occurrences of P^R in T^R by using the algorithm from the part (a).

Exercise 9-2. Some Summations

Assume you are given two sets $A, B \subset \{0 \ldots m\}$. Your goal is to compute the set $C = \{x + y : x \in A, y \in B\}$. Note that the set of values in C could be in the range $0 \ldots 2m$.

Your solution should run in time $O(m \log m)$ (the sizes of $|A|$ and $|B|$ do not matter).

Example:

$$\begin{aligned} A &= \{1, 4\} \\ B &= \{1, 3\} \\ \\ C &= \{2, 4, 5, 7\} \end{aligned}$$

Solution:

The key realization is that when two polynomials multiply, their exponents are added in every possible pairing. So we take our set A, and do the following: Turn the set into a polynomial of maximum degree m. The coefficient of x^0 is the number of times 0 appears in the set. The coefficient of x^1 is the number of times 1 appears in the set, and on. Do the same with B. The example would then become:

Example:

$$\begin{aligned} A &= x^4 + x^1 \\ B &= x^3 + x^1 \\ \\ C &= x^7 + x^5 + x^4 + x^2 \end{aligned}$$

$C = A * B$. Recall that the coefficient of x^i in C is given by:

$$c_i = \sum_{j=0}^{i} a_j b_{i-j}$$

Note that in this case, all terms of the sum are non-negative. We can compute C in $O(m \log m)$ time using the FFT.

For a simple argument of correctness, we observe that every coefficient in the target polynomial ($c_i x^i$) represents the number of ways we can add a number from A and from B to get to i. First, we note that all coefficients in A and B are either zero or one (no multisets were indicated). Clearly if there is no $a \in A, b \in B$ that sum to i, then for all $j \in \{0..i\}$, at least one of a_j, b_{i-j} is zero. Similarly, if $c_i = z$, then there are z pairs a_j, b_{i-j} since each such pair will multiply out to x^i, and then the sum of all of those pairs will yield $z * x^i$.

The running time of this algorithm is the time to convert A and B into polynomials. If A is a set (not a multiset) $\in \{0..m\}$ then this can be done in $O(m)$. Same is true for B. To multiply them takes $O(m \log m)$. Finally, to take the resulting polynomial and output the set C takes $O(2m) = O(m)$. The entire algorithm takes $O(m)$.

As an aside, we also note that if we allowed multisets, everything still works. If there are two ones in one set, and two threes in the other, there are indeed four different ways to get a target value of 3.

Exercise 9-3. Do Problem 35-5, on page 1051 of CLRS.

Solution:

(a) In the best case, each job has its own machine. Since a job cannot be broken down into smaller pieces, the makespan cannot be shorter than the longest job. That is, $C^*_{max} \geq \max\limits_{1 \leq k \leq n} p_k$.

(b) In the best case, the jobs are evenly distributed over the machines, such that each machine finishes at exactly the same time, C_{even}. (Starting from such a distribution, any re-allocation would clearly increase the makespan, as the maximum finish time would increase.) The makespan of an even distribution is the sum of the job lengths divided by the number of processors: $C_{even} = 1/m \sum\limits_{1 \leq k \leq n} p_k$. Since $C^*_{max} \geq C_{even}$, the claim follows.

(c) For each machine M_i, we maintain a set a_i of the jobs that are allocated to M_i, as well as a field f_i indicating the time at which M_i will become idle (under the current allocation of jobs). The GREEDY-SCHEDULE algorithm assigns each job to the machine that will become idle first. The machines are organized in a priority queue (implemented as a MIN-HEAP) using the next finish time f_i as the key for machine M_i.

```
GREEDY-SCHEDULE($J_1 \ldots J_n, p_1 \ldots p_n, M_1 \ldots M_m$)
 1  $Q \leftarrow$ empty MIN-HEAP
 2  for $i \leftarrow 1$ to $m$
 3      $f_i \leftarrow 0$      // initialize finish time
 4      $a_i \leftarrow \{\}$    // initialize job allocation
 5      INSERT(Q, $\langle M_i, a_i, f_i \rangle$), using $f_i$ as key
 6  for $j \leftarrow 1$ to $n$
 7        do $\langle M_k, a_k, f_k \rangle \leftarrow$ EXTRACT-MIN($Q$)
 8           $a_k \leftarrow a_k \cup J_j$
 9           $f_k \leftarrow f_k + p_j$
10           INSERT($Q, \langle M_k, a_k, f_k \rangle$)
11  return $\{a_1, a_2, \ldots, a_m\}$
```

This algorithm directly implements the greedy strategy. It iterates over the jobs, assigning each job to the machine that will be idle next. Each call to INSERT and EXTRACT-MIN takes $O(\lg m)$ time, as there are m machines in the heap. These calls are enclosed in two loops; the first executes m times and the second executes n times. Assuming $n \geq m$ (as otherwise the problem is trivial), the overall runtime is $O(n \lg m)$.

(d) Let J_j denote the job that has the largest completion time as scheduled by the greedy algorithm (that is, $C_j = C_{max}$). Let f'_{jk} denote the value of f_k on iteration j of the job allocation loop (line 6 of the algorithm). Then C_j can be written as follows:

$$\begin{aligned}
C_j = C_{max} &= \min_{k\in[1,m]} f'_{jk} + p_j \\
&\leq \min_{k\in[1,m]} f'_{mk} + p_j \\
&\leq 1/m \sum_{1\leq k\leq n} p_k + p_j \\
&\leq 1/m \sum_{1\leq k\leq n} p_k + \max_{1\leq k\leq n} p_k \\
&\leq C^*_{max} + C^*_{max} \\
&= 2 * C^*_{max}
\end{aligned}$$

The first line is a direct consequence of the greedy algorithm: on iteration j, the algorithm selects the minimum finish time and schedules job J_j to execute next on the corresponding machine. The second line follows because the finish times are monotonically increasing with successive iterations of the j loop. By the same argument as in part (b), the minimum finish time is maximized when all finish times are the same; thus, the average load is used as an upper bound in the third line. The fourth line recognizes that p_j is at most $\max_{1\leq k\leq k} p_k$. Finally, we substitute the results from (a) and (b) to conclude that $C_j \leq 2C^*_{max}$, thereby showing that the greedy algorithm is a 2-approximation algorithm.