6.046		Dan Ports
	Problem 6-1	
No collab		Recitation #8

Part a:

We can solve the on-line string matching problem using the Rabin-Karp algorithm. We will show that no non-trivial modifications to the algorithm are required to identify matches with constant probability of correctness, using $\Theta(m + \log^c n)$ time for preprocessing and $\Theta(1)$ time per character. Since the analysis depends on the size of the alphabet Σ, we assume (without loss of generality) that the characters are the digits $\{0, 1, 2, \ldots, d-1\}$, with $d = |\Sigma|$. We also assume that the alphabet Σ, its size d, the length of the string n, and the pattern P and its length m are known in advance.

As part of the preprocessing, the algorithm must compute a suitable prime number q. We assume the correctness of the prime-determining algorithm given in Lecture 16, which finds q in time polynomial in $\log(n)$. The other preprocessing required is computation of the hash value of the pattern, which requires $\Theta(m)$ time. So the total preprocessing time required is $\Theta(m + \log^c n)$, for some constant c.

We keep track of the number i of characters already received; this counter is initialized to zero and incremented by one each time a character is received, requiring constant time. We also store the incoming characters in an array $T[1 \ldots n]$. Before the Rabin-Karp algorithm can begin identifying matches, it must receive the first m characters of the text. Thus, while $i < m$, we build the initial hash value t_0 using the recurrence $t_0 \leftarrow (dt_0 + T[i]) \bmod q$ (t_0 is initialized to 0 when the algorithm begins). This is taken directly from CLRS's Rabin-Karp-Matcher procedure; it requires constant time for each received character.

After the first m characters are received and the initial hash value t_0 is computed, Rabin-Karp can begin to check for matches with each following character. When the ith character is received, we let $s = n - i$. We compute the hash value t_s using the relation $t_s \leftarrow (d(t_s - T[s+1]d^{m-1} + T[i] \bmod q$ (assuming $s \neq 0$; if $s = 0$, we just use the value t_0 computed in the previous paragraph's procedure). Note that this computation requires constant time, and uses only the ith and $s+1$th elements of T; it depends only on values already received. If the previously computed hash value t_s equals the hash value p of the pattern, we identify the shift s as a possible location of a match and output it. We repeat this process for each incoming character until the end of the input string.

This algorithm requires $\Theta(m + \log^c n)$ preprocessing time, and constant time for processing each received character. Since it requires only trivial modifications to the Rabin-Karp procedure as presented on CLRS p. 914 or in the lecture notes, the proof of correctness follows from the correctness proof given there. The output is correct except in the case of a spurious hit; we know from the lecture notes that for each s the probability of such a spurious hit is small: it is $\frac{1}{2n}$.

Part b:

If the text is being broadcast in reverse, we simply reverse the pattern and apply the same algorithm. This provides a solution with the same running time and correctness characteristics.

To justify that reversing the pattern gives the correct results when the text is broadcast in reverse, we note that there is an occurrence of P in T if and only if $T[i \ \ldots \ i+m-1] = P[1 \ \ldots \ m]$ for some i. If we let P' and T' be the reversed pattern and text, respectively, then the Rabin-Karp algorithm above will identify matches between them: every index i such that $T'[i \ \ldots \ i+m-1] = P'[1 \ \ldots \ m]$. Since T' is the reversal of T, $T'[i] = T[n-i+1]$ and similarly $P'[i] = P[m-i+1]$. Therefore, for every index i, we have found locations where $T[n-i-1 \ \ldots \ n-i-m] = P[m \ \ldots \ 1] \Rightarrow T[n-i-m \ \ldots \ n-i-1] = P[1 \ \ldots \ m]$. If we let $j = n-i-m$, then we have found locations such that $T[j \ \ldots \ j+m-1] = P[1 \ \ldots \ m]$, which is precisely the definition of a match at index j. Thus we have shown that the matches identified by our algorithm using the reversed pattern are exactly the correct matches in the forward order.

10/10

6.046	Dan Ports
Problem 6-2	
No collab	Recitation #8

Assume that we are given a pattern and text, both randomly generated such that each symbol in both is selected independently and unifromly at random from $0, 1$. We show that the naive string-matching algorithm runs in expected time $\Theta(n)$.

We will first show that the inner loop of the naive algorithm requires expected constant time to execute. The inner loop performs comparisons of $T[s+j]$ and $P[j]$ with j from 1 to m. It continues to perform comparisons until it has performed j comparisons, or until it has found a pair that do not match. The sample space contains four possibilities for the pair $(T[s+j], P[j])$: $(0,0)$, $(0,1)$, $(1,0)$, and $(1,1)$. All of these are equally likely, since the pattern and text are randomly generated. Two of these four ($(0,1)$ and $(1,0)$) are not matches, so they will immediately stop the loop. If this is the only condition that can stop the loop, we know that the loop stops with probability $\frac{1}{2}$ at each iteration, so the expected number of iterations executed is $\frac{1}{\frac{1}{2}} = 2$. In fact, there is also the constraint that the loop will stop after j iterations, so the expected number of iterations will be even less. Each iteration requires constant time to execute, so the inner loop requires expected $\Theta(1)$ time to execute.

Next, note that the outer loop runs a total of $n-m = \Theta(n)$ times. Each iteration of the outer loop requires constant time overhead plus whatever time is required to run the inner loop. The expected runtime of each run of the inner loop (for each iteration of the outer loop) is not independent; however, by linearity of expectation, we can add their expectations to find the expected runtime. Thus, running the procedure requires $\Theta(n)$ overhead plus n times the runtime of the inner loop. This is $\Theta(n) + n\Theta(1) = \Theta(n)$, expected.

The same result continues to hold if the pattern is fixed and the text is random. The only part of the analysis that differs is the probability that an element of the pattern and an element of the text will be the same. With both the pattern and text random, this probability is $\frac{1}{2}$. However, even with only the text random, the probability is still $\frac{1}{2}$ because exactly one of the two possible values of the text element will always match a fixed pattern element, and both values of the text element are equally likely. The rest of the analysis proceeds in exactly the same way to give the same result. The same result is possible if only the text is fixed; it only does not hold if both the text and pattern are fixed.

6.046		Dan Ports
	Problem 6-3	
No collab		Recitation #8

9/10

Part a:

Algorithm Let A and B be given as two sets containing elements in the range $\{0 \ldots m\}$. We will compute the set $C = \{x + y : x \in Ay \in B\}$. To do this, we first create two polynomials of degree m, represented as arrays of coefficients $X[0 \ldots m]$ and $Y[0 \ldots m]$. We initialize the array X to contain zeros, and iterate through the set A; for each element we let x be the element's value then set $X[x] = 1$. When we are complete, $X[x] = 1$ if an element with value x is in A, and 0 otherwise. We initialize Y in the same way, using the set B. We next use the FFT-based fast polynomial multiplication algorithm to multiply these two polynomials. The output is a polynomial represented as an array $Z[0 \ldots 2m]$ of coefficients. We iterate through this array; if $Z[x] \neq 0$, we know $x \in C$ and output x. ✓

Proof of correctness We assume the correctness of the fast polynomial multiplication algorithm. It simply remains to show that the reduction of the sums problem to a polynomial multiplication is valid. From algebra, it is clear that the polynomial $Z = XY$ has a term of degree n with coefficient $(x_0 y_n + x_1 y_{n-1} + \ldots + x_{n-1} y_1 + x_n y_0)$. Since the coefficients of x and y are only zero or one, Z has a non-zero term of degree n if and only if there exist an i and j such that $i + j = n$ and x_i and y_j are non-zero. We generated the polynomials such that x_i and y_j are only non-zero if A contains i and B contains j, so our algorithm always gives the correct result. ✓

Runtime analysis Initializing the coefficient arrays takes constant time to set them to zero, plus $\Theta(|A| + |B|)$ time to iterate through the two arrays A and B. We apply the FFT fast polynomial multiplication algorithm to two polynomials of degree m, which requires $O(m \lg m)$ time. Finally, we iterate through the array Z, which also takes $\Theta(m)$ time. The algorithm thus overall requires $O(m \lg m)$ time. ✓

Part b:

Algorithm In order to solve this problem, we will generate and make use of a polynomial defined by $B(x) = x^{A[1]} + x^{A[2]} + \cdots + x^{A[n]}$. As before, we will represent this polynomial as an array B of coefficients that has length m. We can generate the array B (in $\Theta(n) = O(m)$ time) by initializing it to zeros and iterating through the array, incrementing $B[A[i]]$ by one for every i. We wil also make use of a polynomial $C(x) = B(x^2) = x^{2A[1]} + x^{2A[2]} + \cdots + x^{2A[n]}$. We represent this as an array C of coefficients with length $2m$. We can generate it in the same way, with the same $O(m)$ time bound; we simply increment $C[2A[i]]$ for every i.

We will show (below) that, given a target value t, there exist three different indices i, j, k such that $A[i] + A[j] + A[k] = t$ if and only if the polynomial $F(x) = B^3(x) - B(x^2)B(x) = B^3(x) - C(x)B(x)$ has a non-zero coefficient of degree t. We can generate this polynomial by using the FFT to multiply B by itself, then multiplying the result by B again; this only requires $O(m \log m)$ time. We put the resulting coefficients in an array $D[0 \ldots 3m]$. We also generate $E = C(x)B(x)$ by multiplying B and C (again in $O(m \log m)$ time) and putting the results in an array $E[0 \ldots 3m]$. We then generate the array F in $\Theta(n)$ time by iterating from $i = 0$ to $i = 3m$ and filling it with $F[i] = D[i] - 3E[i]$.

Finally, we check (in constant time) whether $F[t] > 0$; if it is we indicate that a matching triple of indices exists, and if it is not we indicate that no such triple exists.

Proof of correctness We assume the correctness of the fast polynomial multiplication algorithm and the results of part a. We will justify the correctness of this algorithm by showing that there exist three different indices i, j, k such that $A[i] + A[j] + A[k] = t$ if and only if the polynomial $F(x) = B^3(x) - B(x^2)B(x) = B^3(x) - C(x)B(x)$ has a non-zero coefficient of degree t. We first consider the simpler problem in which the indices i, j, k are not required to be different. In this case, we simply compute B^3 by multiplying B by itself twice and check whether $B^3[t] > 0$. The proof of this statement follows trivially from the results of part a.

This argument does not apply when we require i, j, k to be distinct, because it would find cases in which $i = j = k$ and $A[i] + A[i] + A[i] = t$, or $i = j \neq k$ and $A[i] + A[i] + A[k] = t$, for example. Subtracting three times the polynomial $E(x) = B(x)C(x)$ handles this constraint. This is because $C(x) = B(x^2)$ represents the numbers that can be achieved by summing two indices that are the same, that is $A[i] + A[i]$ for any i. When we multiply by $B(x)$, we obtain a representation for the numbers that can be achieved by summing two identical indices and one other: $A[i] + A[i] + A[j]$ for any i, j including $i = j$. We need to subtract three times this polynomial because there are three different ways to assign the indices (i.e. $(i = j = 1, k = 2)$ is equivalent to $(i = k = 1, j = 2)$ and $(j = k = 1, i = 2)$). The resulting polynomial is $F(x)$, so checking $F[t]$ gives the correct result. (The correctness of the procedures used to generate the polynomials B, D, E, F is easy to see.)

Runtime analysis The arrays B and C are each generated by iterating through the input array A, which requires time $\Theta(n) = O(m)$. Generating D and E requires polynomial multiplication in time $O(m \log m)$ via the FFT. F is easily generated in $\Theta(m)$ time. Checking $F[t]$ requires constant time. Thus, the algorithm requires $O(m \log m)$ time overall.

-1 C(x)B(x) can be permuted 3 ways

10/10

6.046	Dan Ports
Problem 6-4	
No collab	Recitation #8

Algorithm We are given a text T of length n, a pattern P of size m, in an alphabet Σ. We will transform this into a set of texts and patterns in the alphabet $\{A, G, T, C\}$ that can be handled by Tidor's algorithm. We assume that the elements of Σ are the integers $\{0 \ldots |\Sigma| - 1\}$, and we give a mapping between each element of Σ and a sequence of $\lceil \log_4(|\Sigma|) \rceil$ elements of the $\{A, G, T, C\}$ alphabet. We write each integer from 0 to $|\Sigma|$ in base-4, then map the base-4 digits $\{0, 1, 2, 3\} \mapsto \{A, G, T, C\}$ in that arbitrary order. Let $d = \lceil \log_4(|\Sigma|) \rceil$. Since d digits are required to write these integers in base-4, we have found a mapping between each element of Σ and a sequence of d elements of the $\{A, G, T, C\}$ alphabet.

We use this result to generate a set $\{T_1, T_2, \ldots, T_d\}$ of texts each of length n and a set $\{P_1, P_2, \ldots, P_d\}$ of patterns each of lengths m. We define these such that $T_i[j]$ is the ith element of the $\{A, G, T, C\}$-representation of $T[j]$, and likewise for $P_i[j]$. Thus the texts T_i and patterns P_i are in the $\{A, G, T, C\}$ alphabet, so we can use Tidor's algorithm to perform matches between T_i and P_i. We do this (independently) for every i from 1 to d. Assume that the results are placed in a set of arrays $R_i[j]$ that equals 1 if T_i matches P_i at position j. For every j from 0 to n, we output a match at position j if $R_i[j] = 1$ for every i from 0 to d.

Proof of correctness Since the elements of Σ can be represented by integers from 0 to $|\Sigma|$, we can certainly represent them uniquely by their base-4 representation, which has d digits. Using any arbitrary mapping between $\{0, 1, 2, 3\}$ and $\{A, G, T, C\}$, we see that we can uniquely represent every element of Σ as a sequence of d elements of the $\{A, G, T, C\}$ alphabet. We can therefore represent the text and pattern this way. Splitting the text and pattern by base-4 digit, we can certainly apply Tidor's algorithm to the resulting d sets of texts and patterns. We assume that Tidor's algorithm gives the correct result. Two numbers are certainly equal if and only if every digit in their base-4 representation are equal. Therefore, if we have found a match at position j in each text/pattern pair, it matches for each base-4 digit, and so it matches in the original alphabet Σ. Thus the algorithm gives the correct result.

Runtime analysis We need to transform the pattern and text into their $\{A, G, T, C\}$-representations. This requires $\Theta((n + m)d) = O(nd)$ time (since $m \leq n$). Next, we apply Tidor's algorithm to d individual texts and patterns, all of length n and m respectively. This requires time $dT(n, m)$. Finally, for every position from 1 to n, we need to check whether all d strings indicate a match; this requires $\Theta(nd)$ time. Thus the algorithm overall has the desired runtime of $\Theta((n + T(n, m))d) = \Theta((n + T(n, m)) \log |\Sigma|)$.