CSC2710 Analysis of Algorithms — Fall 2022
Unit 9 - Transform and Conquer - Problem Reductions

Book: §8.2; §32.2

9.1 Problem Reduction

- In problem reduction you take a problem you don't know how to solve and convert it to an instance of a problem you know how to solve.
  - This is an *extremely* important idea in theoretical Computer Science.
    - Used in the study of complexity theory (part of CSC3555).
    - Used in problem solving, our main focus in CSC2710.
- The process of mapping one problem onto another is called a *reduction*.
  - The reduction must take a polynomial amount of time (practical).
  - The solution to the new problem must be efficient.
- A reduction is useful if the amount of time required to reduce the problem and solve it is less than the time it takes to solve the original problem.
- We will present several examples of this strategy.

9.1.1 Example: Computing the Least Common Multiple (LCM)

- Recall: the least common multiple LCM of two numbers a and b is the smallest integer that is divisible by both a and b.
  - Recall: this can be found by looking at the prime factorization of a and b and computing the following product:
    - The product of all of the common factors, times
    - the product of all prime factors in a that are not in b, times
    - the product of all prime factors in b that are not in a.
  - Example to compute the LCM of 24 and 60 we first compute the respective prime factorizations:

    $$24 = 2 \cdot 2 \cdot 2 \cdot 3$$

    and

    $$60 = 2 \cdot 2 \cdot 3 \cdot 5$$

    The common factors are 2, 2, and 3 which give us a product of 12. There is only one factor in 24 not in 60 and that is a 2. Likewise there is one factor in 60 not in 24 which 5. We compute the product of the three sub products to obtain 120.
- It turns out you can relate $\operatorname{lcm}(a, b)$ to $\gcd(a, b)$.
  - Observation the GCD of two numbers is just the product of all the common prime factors.
  - $\operatorname{lcm}(a, b) \gcd(a, b) = ab$
    - as every prime factor of a and b is included exactly once in the product.
    - The LCM provides the uncommon factors times one copy of the all the common factors.
    - The GCM provides the product of all the common factors.

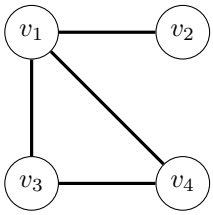
– Therefore we can reduce the problem of computing the LCM of a and b to the problem of computing:

$$\operatorname{lcm}(a,b) = \frac{ab}{\gcd(a,b)}$$

- Note we know that the GCD can be efficiently computing using Euclid's algorithm. We also no that the transformation is efficient so we have a good reduction.

9.1.2 Example: Counting Paths in a Graph

- Sometimes we want to know the number of possible paths between two vertices in a graph.
- It turns out that the number of paths of length less than or equal to k ($k \geq 1$) between vertices i and j is located in location (i,j) in A^k which is the k-th power of the adjacency matrix A.
- We have reduced the problem of determining the number of paths of length k or less between vertex i and j to the problem of exponentiating a matrix.
  - We can use algorithms similar to binary exponentiation to exponetiate matrices as well.
- Example: Given the graph



with adjacency matrix :

$$A = \begin{array}{c} \\ v_1 \\ v_2 \\ v_3 \\ v_4 \end{array} \begin{array}{c} \begin{array}{cccc} v_1 & v_2 & v_3 & v_4 \end{array} \\ \begin{pmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{pmatrix} \end{array}$$

We want to compute all the paths of length at most 2.

– we simply compute A^2.

$$A^2 = \begin{array}{c} \\ v_1 \\ v_2 \\ v_3 \\ v_4 \end{array} \begin{array}{c} \begin{array}{cccc} v_1 & v_2 & v_3 & v_4 \end{array} \\ \begin{pmatrix} 3 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 1 & 2 & 1 \\ 1 & 1 & 1 & 2 \end{pmatrix} \end{array}$$

– This means there are 3 paths in the graph that start at v_1 and end at v_1

∗ **Practice**: what are they?

9.1.3 Example: Reduction to Graph Problems

- Graph based reductions are important in many fields including AI which uses it for state space exploration.
- General Idea
  1. Express each state in the search space as its own vertex in the graph.
  2. An edge (i, j) in the graph denotes that state j can be reached directly from state i.
     - Sometimes the edges are labeled with a transition.
- The original problem has now been reduced to finding a path from some initial vertex to some goal vertex in the resulting graph.
  - How might you solve this?
- **Practice**: Missionaries and Cannibals problem.

9.2 Space Time Trade-Offs

- Sometimes you can gain some speed if you allow the algorithm to use extra space.
- This is a fundamental issue in computer Science base on the following observation:
  - Observation: One can precompute some of the computation and store it in a table for later use.
  - This practice speeds up the computation at the sacrifice of additional space usage.
- This act of preprocessing is sometimes called input enhancement.
- Another way of using extra space is through the use of *prestructuring*
  - Uses the extra space to facilitate fast or more flexible access to the data.
  - This means we must do some work before the problem is actually solved.
    - it only deals with access structure however.
- There is a third technique called dynamic programming but, we defer our discussion to unit 11.
- Sometimes we can realize a decrease in both time and space through space efficient data structures.
  - methods for represent graphs for graph traversals (e.g., adjacency list vs adjacency matrix).
- Other examples of space-time tradeoffs:
  - Compression
  - Procedural generation (vector graphics) vs. stored data (jpg or png graphics)
  - Loop unrolling (optimization at the compiler level)

9.2.1 Sorting by Counting

- We can apply input enhancement to the sorting problem.
- Idea: Use a table as follows:
  - For each element of a list to be sorted record the total number of elements smaller than the current element in the table.
  - The numbers in the table indicate the location of the element in the final sorted array.
  - An index i in the table corresponds to index i in the sequence being sorted.
- The process we just discussed is called *comparison counting sort*.
- The formal algorithm implementing this solution is as follows

```
# Input: An array A of integers
# Output: An arry S of A's elements in sorted order
def ComparisonCountingSort( A ):
    # Initialize an array to hold the counts
    C = [ 0 for x in range( len(A) ) ]                    # Line 1
    S = [ 0 for x in range( len(A) ) ]                    # Line 2
    # Count the frequency of each element
    for i in range( len(A)-1 ):                           # Line 3
        for j in range( i+1, len(A) ):                    # Line 4
            if( A[i] < A[j] ):                            # Line 5
                C[j] = C[j] + 1                           # Line 6
            else:                                         # Line 7
                C[i] = C[i] + 1                           # Line 8
    # Place the elements in their final location
    for i in range( len(A) ):                             # Line 9
        S[ C[i] ] = A[i]                                  # Line 10
    return S                                              # Line 11
```

- Notice that this algorithm has running time $\Theta\left(n^2\right)$
  - **Practice**: Why is that correct?
- In addition it also uses $\Theta(n)$ extra space.
- This amounts to ComparisonCountingSort being a pig!
  - SelectionSort and InsertionSort is better in terms of space with the same time.
- Not all is lost! We can gain some use of this technique if the values in A are known to be in the range $[l, u]$.
  - The frequency of each element in A can then be stored in an array $F[0..u-l]$.
  - Element $F[0]$ stores the frequency of the number l
  - In general $F[i]$ stores the frequency of the number $l+i$.
- To use the table, the first $F[0]$ cells of the sorted array with contain l, followed by $F[1]$ cells of $l+1$.
  - The $l+1$ values will start at location $F[0]+F[1]-1$.
- Due to the fact that we use the accumlated sum of frequencies are often called distributions in statisitics, this method of sorting is sometimes called *distribution counting*.
  - It more commonly goes by the counting sort.
- The formal algorithm is as follows:

```
# Input: An array A of comparable objects, upper and lower
#   are the upper and lower bounds of the elements of A
# Output: An array S of A's elements sorted
def CountingSort( A, upper, lower ):
    # Initialize the elements frequencies to 0
    F = [ 0 for x in range( upper-lower+1 ) ]                    # Line 1
    S = [ 0 for x in range( len(A) ) ]                           # Line 2
    # Compute the frequencies of the elements of A
    for j in range( len(A) ):                                    # Line 3
        F[ A[j] - lower ] = F[ A[j] - lower ] + 1                # Line 4

    # Compute the distribution
    for j in range( 1, upper-lower+1  ):                         # Line 5
        F[j] = F[ j-1 ] + F[j]                                   # Line 6

    # Place the elements of A into S in sorted order
    for i in range( len(A)-1, -1, -1 ):                          # Line 7
        j = A[i] - lower                                         # Line 8
        S[ F[j] - 1 ] = A[i]                                     # Line 9
        F[j] = F[j] - 1                                          # Line 10
    return S                                                     # Line 11
```

- Let's look at an example. Consider the sequence $A = \langle 13, 11, 12, 13, 12 \rangle$ with $l = 11$ and $u = 13$ which gives us a three element table F.
  - Initialize table F to all zeros $F = [0, 0, 0]$
  - Walk the sequence A and update the frequencies appropriately. $F = [1, 2, 2]$.
  - Compute the distribution. $F = [1, 3, 5]$
  - We then apply the process in the last loop of CountingSort to arrive at the sequence $S = \langle 11, 12, 12, 13, 13 \rangle$.
- If the size of the interval $[l, u]$ remains fixed, CountingSort runs in $\Theta(n)$ time using $\Theta(u - l)$ extra space.
  - Note: this does not violate our optimal sorting time belief as CountingSort requires very special types of inputs.

9.2.2 Input Enhancement in String Matching

- We can also apply the input enhancement technique to the string matching problem.
  - Recall:

    Definition 1 (String Matching Problem)**.**
    Input*: An n symbol string*

    $$\sigma = \sigma_1\sigma_2 \cdots \sigma_n$$

    and an $m \leq n$ symbol search string

    $$s = s_1 s_2 \ldots s_m.$$

    Output*: If there exists an index i such that*

    $$\sigma_i = s_1, \sigma_{i+1} = s_2, \ldots, \sigma_{i+m} = s_m,$$

    output i; otherwise, output $\perp$.

- We have already seen a brute-force method of solving this problem.
- Input enhanced string matching algorithms, which is most of them, work as follows:
  - Preprocess the search pattern to gain information about it
  - Store the preprocessed information in a table
  - Use the table to aid in speeding up the search time.
- We will investigate two such methods for solving the string matching problem.
  1. Horspool's Algorithm
  2. Boyer-Moore's Algorithm (in the lecture notes, but we will not go over in class)

9.2.3 Example: Horspool's Algorithm

- Horspool's Algorithm has the following high level structure:
  1. Line up a the pattern s under a consecutive substring of σ.
  2. Work right to left through the pattern comparing each letter with the letter above it in σ.
     - If all letter match, a match has been found and return the location of the start of the match in σ.
     - If we encounter a letter that does *not* match, the pattern must be shifted right.
       * We want to maximize this shift without missing any potential matches.
- We can handle the shift in the above algorithm as follows, assuming that σ_c is the letter in σ that is aligned with the rightmost character of s
  1. The character σ_c does not occur in s, then shift right by the entire length of the pattern s.
  2. The character σ_c occurs in the pattern s and is not the last letter, then shift s right to align the right most character of s (that is not the last character of s) with σ_c.
  3. If character σ_c occurs *only* in the last character of s then shift the pattern right by its entire length.
  4. If character σ_c happens to be the last character in the pattern and there are other σ_c's among the first $|s| - 1$ characters, align the rightmost occurrence of c, in s, with σ_c.

- We have gained some benefits from the fact that we move right to left through the pattern string but, it can still devolve into searching all the characters every time.
- Thankfully, we can precompute the shifts and reduces what we must check.
- We compute the shift S, for character c we using the following piecewise defined function:

$$S(c) = \begin{cases} |s|, & \text{if } c \text{ is not among the first } |s|-1 \text{ characters.} \\ \text{distance from right most} & \\ c \text{ among first } |s|-1 \text{ characters,} & \text{otherwise} \\ \text{of the pattern to its last character} & \end{cases}$$

  For all $c \in \Sigma$.

  - Recall: Σ denotes the alphabet.

- An algorithm to construct this table is as follows:

```
# Input: A search string S and an alphabet of possible characters E
# Output: A table T indexed by alphabet characters and filled with
#   appropriately computed shift sizes
def BuildShiftTable(S, E):
    # Initialize T as a Dictionary
    T = {}                                                    # Line 1
    # Initialize every character a complete shift
    for x in range(len(E)):                                   # Line 2
        T[ E[x] ] = 0                                         # line 3

    # Set the shift distance of each character in S to the distance
    #   from the right most occurrence to the end of the pattern
    for i in range( len(S) - 1 ):                             # Line 4
        T[ S[i] ] = len(S) - 1 - i                            # Line 5
    return T                                                  # line 6
```

- **Practice**: What is the running time of BuildShiftTable?
- **Practice**: How much extra space does BuildShiftTable use?
- Horspool's Algorithm can no be phrased as a simple three step algorithm.

  1. Construct the shift table using BuildShiftTable and search string s.
  2. Align the pattern against the beginning of the text.
  3. Repeat until either a matching substring is found *or* the pattern extends past the end of the string σ.
     (a) Starting with the last character in the pattern compare it with the corresponding character in the text σ
     - If all $|s|$ characters are matched we are done.
     - If a mismatch is encountered, look up the character above $s_{|s|}$ in σ. Call that character σ_c. Shift the pattern right by $T[\sigma_c]$.

- The Python for Horspool's algorithm becomes:

```
# Input: A patter P and a text T
# Output: The index of the left end of the first substring of T
#    that matches P, or -1
def Horspool( P, T ):
    # Build the shift table for the pattern
    S = BuildShiftTable(P, E)                                    # Line 1

    # Start at the end of the pattern
    i = len(P) - 1                                               # Line 2
    while( i <= len(T)-1 ):                                      # Line 3
        k = 0                                                    # Line 4
        # Match all of the characters possible
        while( k <= len(T) - 1 and P[len(P)-1-k] == T[i-k] ):    # Line 5
            k = k + 1                                            # Line 6
        # Now check to see if we matched the whole pattern
        if( k == len(P) ):                                       # Line 7
            return i - len(P) + 1                                # Line 8
        # If nothing matched, shift appropriately
        else:                                                    # Line 9
            i = i + S[ T[i] ]                                    # Line 10
    return -1                                                    # Line 11
```

- Let's look at an example quotation from the 1977 RSA-129 challenge [1].

  THE MAGIC WORDS ARE SQUEAMISH OSSIFRAGE

  The pattern string will will search for is $s =$ SQUEAMISH

  – We first generate the shift table which will have every letter in the alphabet, Σ set to a shift of 9 except for the following table of characters

Symbol	**Shift**
S	1
Q	7
U	6
E	5
A	4
M	3
I	2
H	9

[1] It was a $100 prize. It is named RSA-129 as to break the ciphertext you must factor a 129 bit number into two primes. Additionally, an Ossifrage is an outdated word for a bearded vulture.

- We then proceed to search for the string as follows:
  1. Start at the beginning.
     ```
     THE MAGIC WORDS ARE SQUEAMISH OSSIFRAGE
     SQUEAMISH
     ```
  2. Seeing a `C` in the text means we shift right 9.
     ```
     THE MAGIC WORDS ARE SQUEAMISH OSSIFRAGE
              SQUEAMISH
     ```
  3. Seeing a `R` in the text means we shift right 9.
     ```
     THE MAGIC WORDS ARE SQUEAMISH OSSIFRAGE
                       SQUEAMISH
     ```
  4. Seeing a `I` in the text means we shift right 2.
     ```
     THE MAGIC WORDS ARE SQUEAMISH OSSIFRAGE
                         SQUEAMISH
     ```
     We have now matched the substring and we are done.

- The running time of the algorithm has running time $O(mn)$
  - Under random texts, the algorithm has running time $\Theta(n)$

Challenge Problem

1. Design (not necessarily in Python) faster algorithms for solving the following problems, assuming you have infinite space in which to work. Then analyze the time and space complexities of each algorithm.
   - Finding the n^{th} Fibonacci number
   - The nearest-neighbor query
   - The mode of a list of numbers
2. Design a representation-change algorithm that uses a problem reduction (using an algorithm to solve a different problem on your new representation) to solve the "Knight's Walk" problem. That is, given a source square and a destination square on an n-square chess board (so for the standard chess board, $n = 64$), what is the shortest path a knight chess piece can take to reach the destination from the source?
   - What is the space requirement?
   - What is the time efficiency?