CISC 4080 —Computer Algorithms

Fall 2021

Final Review Quide

1. Understand the basic concepts: algorithm, problem instance, input size, running time, space requirement (memory required), best case, worst case and average case performance (running time or space requirement)

2. Running time analysis: given pseudocode, write $T(n)$ (i.e., number of computer steps executed by the algorithm on an input of size n).

   (a) non-recursive: analyze the loop (including nested loop) to count the total number of steps executed

   (b) recursive algorithm: write a recursive formula for $T(n)$.

   Expectation: Given an algorithm with reasonable complexity, analyze its running time. Solving recursive formula using method of iteration, or applying master theorem.

3. Master theorem: know how/when to apply it to solve recursive formula for closed one For example, it cannot be used to solve
$$T(n) = 2T(n-2) + n^2,$$
   It can be used to solve
$$T(n) = 3T(3n/4) + n^2,$$
   with $a = 3, b = 4/3, d = 2$, because $n/b = 3n/4$.

4. Understand the meaning of O, Ω, Θ notations, and can use the simple rules of thumbs (book DPV and slides) to decide for given pairs of functions of n, what is the relation between them.

5. Understand the algorithms studied for the problems listed below

   (a) Search problem: linear search and binary search

   (b) Sorting: bubble sort, selection sort, insertion sort, mergesort, quick sort, counting sort, radix sort and so on (counting sort and radix sort algorithms were not covered this semester)

   (c) Selection (find the k-th largest element): adapted bubble sort

   Expectation: you should be able to answer questions about these algorithms, adapte the algorithms, and trace the codes.

6. Sorting algorithms concepts

   (a) comparison based vs non-comparison based (not covered)

   (b) lower bound for comparison based sorting: $n \log n$.

   (c) in-place vs out-of-place sorting (briefly explained)

   (d) stable vs unstable sorting algorithms (not covered)

   (e) n^2 vs $n \log n$ vs n running time sorting algorithms

7. Problem solving paradigm: how to come up an algorithm to solve a new problem

   (a) brute force: enumerate (try) all possibilities to find the best one

   (b) Divide and conquer: divide the problem into smaller problem instances, solve each of the smaller problem instances (recursively), and then do some processing to combine the solutions to the smaller problems to find the solution to the original problem.

(c) Decision Tree, recursive solution: if the problem is a combinarotics problem, we can model the multiple decisions involved as a decision tree. With the help of decision tree thinking, we can come up recursive algorithm (see hw3).

(d) Dynamic Programming: If you have a recursive solution to a problem, and observe "overlapping subproblems", then you can try to use dynamic programming to improve its running time.

Expectation: Apply them to solve simple problems with given hints. An example was discussed during the review session:

Given a vector of int, price, where $price[i]$ is the price of the stock at i-th day. Find the best pair of buy and sell dates, so that one makes the maximum profit, i.e., $price[sell_date] - price[buy_date]$ is maximized. Note that we assume you can only sell either after you buy the stock, or on the same date.

Write a function that finds the maximum profit (and optionally: the buy and sell dates).

e.g., for a vector storing 2, 12, 10, 1, the maximum profit is 10 (buy on day 0, and sell on day 1); for a vector storing 12, 8, 3, 1, the maximum profit is 0.

The function should just find these max profit and buy/sell dates, and printing out these information found should be done in the main().

1. Brute force approach:

```
/* Check all possible buy date, paired with all possible sell date, and evaulate
    the resulting profit, while keeping track of the max. profit achieved so far
*/
int BestBuySell (vector<int> price, int & buy, int & sell)
{
    int maxProf=0;

    for (int i=0; i<price.size(); i++) {
       //iterates through all possible buy date

       for (int j=i; j<price.size(); j++){
           //iterates through possible sell date
           //for each buy date i

           profit = price[j]-price[i];

           if (profit>maxProf){
                     maxProf = profit;
                     buy = i; sell=j;
                }
        }
    }
    return maxProf;
}
```

The running time of this algorithm will be $\Theta(n^2)$.

2. Divide-and-Conquer
We can use divide-and-conquer approach to get an algorithm with running time of $n \log n$.

```
/*
   return max profit achievable from sell/buy within the date range: first...last
   first<=last
```

```
*/
int BestBuySell_DAC (vector<int> price, int first, int last, int & buy, int & sell)
{
    /* base case */
    if (first==last)
    {
        buy=first; sell=first;
        return 0;
    }

    /* general case: divide and conquer */
    int mid = (first+last)/2;

    int p1 = BestBuySell_DAC (price, first, mid, buy, sell);
    int p2 = BestBuySell_DAC (price, mid+1, last, buy2, sell2);

    if (p2>p1){
        buy=buy2;
        sell=sell2;
        bestProf=p2;
    }
    else
        bestProf=p1;

    //Third option: buy in the first half, sell in the second half

    //find smallest in price[first...mid]
    int low=price[first];
    for (int i=first+1;i<=mid;i++) {
         if (price[i]<low){
            low=price[i];
            lowD = i;
        }
    }

    //find largest in price[mid+1...last]
    int high = price[mid+1];
    for (int i=mid+2;i<=last;i++)
       if (price[i]>high){
         high = price[i];
         highD = i;
       }

    if (high-low > bestProf){
        bestProf = high-low;
        buy = lowD;
        sell = highD;
    }
    return maxProf;
}
```

3. Some called it Dynamic Programming approach

The last approach has running time of $\Theta(n)$, see below:

```
/*
   Consider if to sell on day j, then the best day to buy is the lowest
   price point prior to j. We can just keep track the lowest price so far...
*/
int BestBuySell (vector<int> price, int & buy, int & sell)
{
    int lowest = price[0];
    int lowestDay = 0;

    for (int i=0; i<price.size(); i++) {
       //update lowest price so far
       // this part is kind of like "overlapping subproblem"
       if (price[i]<lowest){
            lowest = price[i];
            lowestDay = i;
       }

        //if we sell on day i, the max profit is:
        int maxProf_SellDatei = price[i] - lowest;

        //update maxProf seen so far, and buy/sell date
        if (maxProf_SellDatei > maxProf){
             maxProf = maxProf_SellDatei;
             buy = lowestDay; sell=i;
        }

    }
    return maxProf;
}
```

8. Summary and comparison of two different methods to implement dynamic programming:

   If during a recursive algorithm's execution, same subproblems are encountered mutliple times, then we can use a table to store subproblems' solutions, and look up the table if the subproblem is encountered again.

   For both memoization and tabulation approach, similar tables are used. When designing the table, you need to consider what input parameters are required to specify a subproblem.

   - If a single integer parameter is required, such as the case of unlimited Knapsack problem (where we have unlimited supply of all types of items), then use an one dimensional table/array.
   - If each subproblem is specified with two integer inputs (such as n and W as in 0/1 Knapsack problem), then we need to use a 2D array/table (e.g., a vector of vectors), using n as row index, W as column index (or reverse).
   - Higher dimension tables or hashtables might be used for subproblems with more integer parameters, or other type of parameters.

   The difference of memoization and tabulation lies in the structure of the algorithm:

   - Memoizatin approach: implement the algorithm recursively (i.e., it keeps the recursive structure). The typical steps to implement memoization approach include 1) Write a wrapper function, in which a table is allocated and initialized, 2) the table is passed to the recursve function by reference, 3) in the recursive function, add logics to check if the subproblem has been solved before to avoid recomputing same problem, and to save solution found before the end of computation.

- Tabulation approach: works by setting up the table, and then use loop to solve all subproblems that are smaller than the current problem (of interest). The logic of recursive algorithm is used to calculate all subproblems: when subproblems solutions are needed, they are just looked up from the table.

Pros and Cons of the two approaches* (This is not discussed in class):

Memoization: only solve subproblems as needed by the final problem; but it incurs recursive function call overhead.

Tabulation: solve all subproblems (even those not needed); but it does not have recursive function call overhead.

9. Familiar with problems and algorithms studied in the class, able to trace their execution (e.g., drawing recursion tree), and able to solve similar problems.
   (a) Sorting Problem: bubblesort, selection sort, insertion sort, mergesort, quicksort
   (b) Search problem:
   (c) Merge problem: two-way merge, and K-way merge
   (d) Add up to K: pairs, triples, subsets (i.e., coin change problem)
   (e) Combinatorics problems: Rod Cutting, 0/1 Knapsack, general Knapsack, and unlimited Knapsack

10. Basic understanding of the usage of dictionary, different ways of implementing dictionary: direct access tablem, binary search tree, and hashtable. (And be able to use the C++ STL classes).
    - additional data structures learned: set and multiset.