CPSC 320 2018W2: Assignment 4

This assignment gives you practice at using recurrences to analyze algorithm runtimes. It also gives you practice at using recurrences as a means to design efficient algorithms, by inductively describing quantities that we'd like to compute.

Please follow the guidelines given in Assignment 1 for submission to Gradescope, and for group collaboration. Remember to provide short justifications for your answers. Submit by the deadline **Monday March 11, 2019 at 10PM**.

Use LaTeXto prepare your answers. Easiest will be to use the .tex file provided. Please enclose each paragraph of your solution in `\soln{Your solution here...}`.

Your solution will then appear in dark blue, making it a lot easier for TAs to find the parts that you wrote.

1 Computer scientist makes $0.38 a day – find out how!!!

A popular pastime of computer scientists (especially academics) is fantasizing about how to use our knowledge to get rich playing the stock market. The first advice one receives about playing with stocks is "buy low, sell high". Easier said than done.

Suppose we have an array $S = [1..n]$ of integer values where $n \geq 1$ and $S[i]$ represents the price of a stock on date i. Given this historic price data, the stock market problem is to (retrospectively) find the best dates to buy and sell the stock so as to maximize profit. If i and j are the buying and selling dates (with $i \leq j$), then $[i, j]$ should be a contiguous interval with the highest jump in price, and $S[j] - S[i]$ is the profit. (You can buy and sell stock on the same date, with a profit of zero.)

Example: The $S[1..13] = [5, 10, 3, 4, 9, 21, 13, 1, 12, 20, 15, 17, 4]$ has a maximum jump of $20 - 1 = 19$ if purchased on date $i = 8$ and sold on date $j = 10$.

1. Describe a brute-force algorithm that identifies the best buy and sell dates, and the corresponding profit.

```
Algorithm Buy-low-Sell-high(S)
    n = |S|
    maxProfit = −∞
    buy = −1
    sell = −1
    for i = 1 to n:
        for j = i to n:
            profit = S[j] − S[i]
            If profit > maxProfit :
                maxProfit = profit
                buy = i
                sell = j
    Return (buy, sell)
```

2. What is the running time of your brute force algorithm?

   The running time for this brute force algorithm would be $O(n^2)$. Because we have two nested for loops, which grow with n.

3. A divide-and-conquer approach for the stock market problem can consider three cases:

   (a) The best buy and sell dates are both at most $\lfloor n/2 \rfloor$.

   (b) The best buy and sell dates are both greater than $\lfloor n/2 \rfloor$.

   (c) The best buy date is at most $\lfloor n/2 \rfloor$ and the best sell date is greater than $\lfloor n/2 \rfloor$.

   In order to determine the solution in case (c), what specific pieces of information do you need to know about the left half of the array, i.e., $S[1..\lfloor n/2 \rfloor]$, and the right half, $S[\lfloor n/2 \rfloor + 1..n]$?

   For getting the solution on case (c) we would need to get the *argmin* value of $S[1..\lfloor n/2 \rfloor]$ for the best buy day and the *argmax* value of the $S[\lfloor n/2 \rfloor + 1..n]$ for the best sell day.

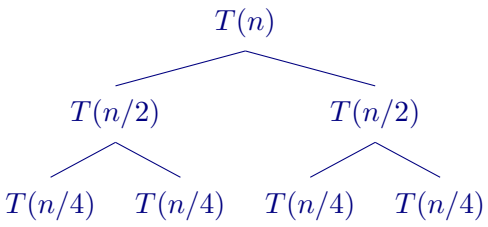
4. Write a recursive divide and conquer algorithm for the stock market problem. (It may be handy for the algorithm to return the information needed about subproblems, as documented in your answer from part 3, in addition to the buy date, sell date and profit.)

```
Algorithm BlSh(S, start, end)
    n = end − start
    If n = 0 :
        return (S[start], start, start, start, start)
    Else
        // Get the profit of the left and right side
        profitL, buyL, sellL, minL, maxL = BlSh(S, start, start + ⌊n/2⌋)
        profitR, buyR, sellR, minR, maxR = BlSh(S, start + ⌊n/2⌋ + 1, end)

        // Get the min and max value of S[start....end]
        amin = argmin(S, minL, minR)
        amax = argmax(S, maxL, maxR)

        // Calculate the profit based on the min, max values
        profit = S[maxR] − S[minL]

        // Return the maximum profit, best buy day, best sell day, minimum price, maximum price
        If profit > profitL && profit > profitR :
            return (profit, minL, maxR, amin, amax)
        Else if profitL > profit && profitL > profitR :
            return (profitL, buyL, sellL, amin, amax)
        Else :
            return (profitR, buyR, sellR, amin, amax)
```

5. Give a recurrence relation for the runtime of your algorithm, and solve it to get a big-O bound on the running time.

$$\text{BlSh}(n) = \begin{cases} c_1, & \text{if start} = \text{end} \\ \text{BlSh}(n/2) + \text{BlSh}(n/2) + c_2, & \text{otherwise} \end{cases}$$

   The recursion tree would be:



We are dividing on each level the size in half, so our algorithm will hit the base case on level i when $1 = \frac{n}{2^i}$, so all our leaf nodes will be on level $i = \log(n)$, thus we have $\log(n)$ levels. We can also notice that on each level i we are spending $2^i c_2$ time.

We know that we will have n leaves reaching the base case with $T(1)$, and on the base case we spend c_1 time, so for all our base cases we will have $c_1 n$ time.

So we have to add the time at each level i which is $2^i c_2$. And we have $\log(n) - 1$ levels without the leaf nodes. So we have $c_2 \sum_{i=0}^{\log(n)-1} 2^i$, using the geometric series formula to solve that summatory, $c_2(\frac{1-2^{\log(n)}}{1-2})$ which yields $c_2 n$.

The running time of the algorithm would be the time of the leaves + the internal nodes. So the algorithm will have $c_1 n + c_2 n$ running time which is $O(n)$.

2 Medians and finding the kth largest element

We saw in the Divide and Conquer worksheet that the QuickSelect algorithm finds the median in $O(n)$ time, **if** we assume that we "get lucky" with the choice of pivot at every call to QuickSelect. By getting lucky on a call of size n, we mean that the rank of the pivot lies in the range $[n/4, 3n/4]$, and so the size of the next recursive call is at most $3n/4$. However in the worst case (when we are not lucky), the pivot might have rank 1 or rank n, in which case the algorithm takes $\Theta(n^2)$ time.

Better algorithms for choosing the pivot don't rely on luck, but guarantee that the pivot partitions a problem of size n into subproblems of size at most $7n/10$. One such algorithm carefully chooses $\lceil n/5 \rceil$ elements in $O(n)$ time, and then sets the pivot to be the median of these elements using a recursive call. Exactly how the $\lceil n/5 \rceil$ elements are chosen is not important for this problem. The rest of the algorithm is identical to our QuickSelect algorithm. Here is pseudocode:

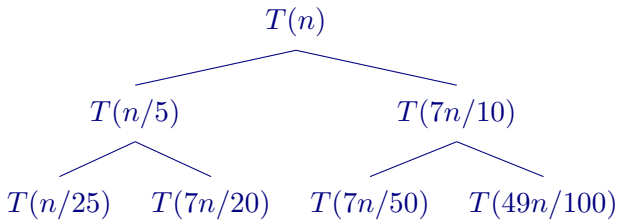
```
Algorithm BetterQuickSelect(A[1..n], k)
    // Return the kth largest element of A
    // Precondition: |A| = n ≥ k, and all elements of A are unique

    // Select the pivot element p
    Create array A' of ⌈n/5⌉ "carefully choosen" elements of A // this takes O(n) time
    Let p = BetterQuickSelect(A', ⌈ |A'|/2 ⌉)

    Let Lesser = all elements from A less than p
    Let Greater = all elements from A greater than p
    If |Greater| = k − 1
        Return p
    Else If |Greater| > k − 1
        Return BetterQuickSelect(Greater, k)
    Else // |Greater| < k − 1
        Return BetterQuickSelect(Lesser, k − |Greater| − 1)
```

1. Let $T'(n)$ be the running time of BetterQuickSelect. Write a recurrence for $T'(n)$. Use the facts that the pivot partitions a problem of size n into subproblems of size at most 7n/10 and that the time to create list A' is $O(n)$.

$$T'(n) = \begin{cases} c, & \text{if } |A| = 1 \\ \\ n + T'(n/5) + T'(7n/10), & \text{otherwise} \end{cases}$$

2. Solve your recurrence of part 1 to show that BetterQuickSelect takes time $O(n)$ in the worst case.

   The recursion tree would be:



   By analizing the recursion tree we can notice that at the 2nd level we are spending $cn\frac{9}{10}$ time because each call is divided into two calls of sizes $\frac{n}{5}$, $\frac{7n}{10}$ This is a pattern that repeats on following i level with a running time of $cn\frac{9}{10}^i$. So the total cost, would be represented by the following sum $cn\sum_{i=0}^{\infty}\frac{9}{10}^i$, that can be solved by the geometric series, $cn\sum_{i=0}^{\infty}\frac{9}{10}^i = cn\frac{1}{1-\frac{9}{10}} = 10cn$, thus the total running time of BetterQuickSelect is $O(n)$

3 More Expressions of 1's (Adapted from Jeff Erickson)

As in the tutorial of week 8, we can represent positive integers in terms of 1's, +, ×, and parentheses "(" and ")". For example, all of the following expressions represent the integer 14:

$$1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1$$
$$((1+1)\times(1+1+1+1+1))+((1+1)\times(1+1))$$
$$(1+1)\times(1+1+1+1+1+1+1)$$
$$(1+1)\times(((1+1+1)\times(1+1))+1)$$

Given a positive integer n, we'd like to find the minimum number of 1's in such an expression. Let $M(n)$ denote this minimum. For example, the fourth expression above is such an expression for 14, and so $M(14) = 8$.

1. Use the recurrence for $M(n)$ from the solutions to the tutorial of week 8, plus memoization, to develop an efficient recursive algorithm to compute $M(n)$.

Algorithm $M(n, Soln[-1... - 1_n])$
// Soln is an array with solutions, at the begining it is filled with n -1s
$Min = \infty$
If $n = 1$:
    return 1
If $Soln[n]! = -1$:

return $Soln[n]$
For $i = 1$ to $n - 1$:
$Min = \texttt{min}(Min,\ M(n - i, Soln) + M(i, Soln))$
If $(i + 1) < n/2$ && $(i + 1)|n$:
$Min = \texttt{min}(Min,\ M(n/(i + 1), Soln) + M(i + 1, Soln))$
$Soln[n] = Min$
return Min

2. What is the running time of your memoized algorithm?

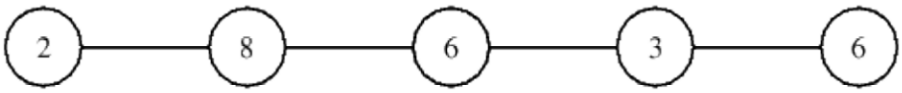
   Since we are saving the results of all subproblems we know that we won't repeat work, so there will be at most n recursive cases, each of these cases will have a running time of n because so the total time on the recursive cases would be n^2.

   For the rest of the calls to the algorithm they will be leaves or base cases. This base cases take constant time so for computing all of the leaves we will have n time, thus the total running time of the algorithm would be $n^2 + n$ which is $O(n^2)$

4 Max-Weight Independent Sets

Let $G = (V, E, w())$ be an undirected graph, with a weight $w(v)$ associated with each node $v \in V$. An *independent set* of G is a subset of nodes, none of which are connected by an edge of the graph. A *max-weight* independent set is an independent set such that the sum of the weights of the nodes in the set is as large as possible. Let MIS(G) be the weight of a max-weight independent set for graph G. Finding max-weight independent sets and their corresponding weights can be done efficiently for simple types of graphs, although no efficient algorithm is known for general graphs. Here, you'll consider graphs that are paths.

For $0 \leq i \leq n$, let $P = ([1..n], E, w)$ be a weighted path, that is $E = \{(1, 2), (2, 3), \ldots, (i, i+1), \ldots (n - 1, n)\}$. For example, in the following path, the weights are the numbers drawn inside the nodes.



Let $\text{MIS}^+(P, i)$ be the max weight of an independent set of P involving only nodes from $[1..i]$ and including i. Similarly, let $\text{MIS}^-(P, i)$ be the max weight of an independent set involving only nodes from $[1..i - 1]$ (node i is excluded from the independent set). We can write $\text{MIS}^+(P, i)$ as:

$$\text{MIS}^+(P, i) = \begin{cases} 0, & \text{if } i = 0 \\ w(i) + \text{MIS}^-(P, (i - 1)), & \text{if } i \geq 1. \end{cases}$$

1. Write a recurrence for $\text{MIS}^-(P, i)$, $i \geq 0$:

$$\text{MIS}^-(P, i) = \begin{cases} 0, & \text{if i<= 1} \\ \max\{\text{MIS}^+(P, i - 1), \text{MIS}^+(P, i - 2)\}, & \text{if i > 1} \end{cases}$$

2. We can express $\text{MIS}(P)$ as $\max\{\text{MIS}^+(P, n), \text{MIS}^-(P, n)\}$. Use this to write an algorithm that computes $\text{MIS}(P)$ in $O(n)$ time. Your algorithm should use memoization and recursion to compute both

$\text{MIS}^+(P, n)$ and $\text{MIS}^-(P, n)$, and use these quantities to compute $\text{MIS}(P)$. (No justification is needed, just the algorithm.)

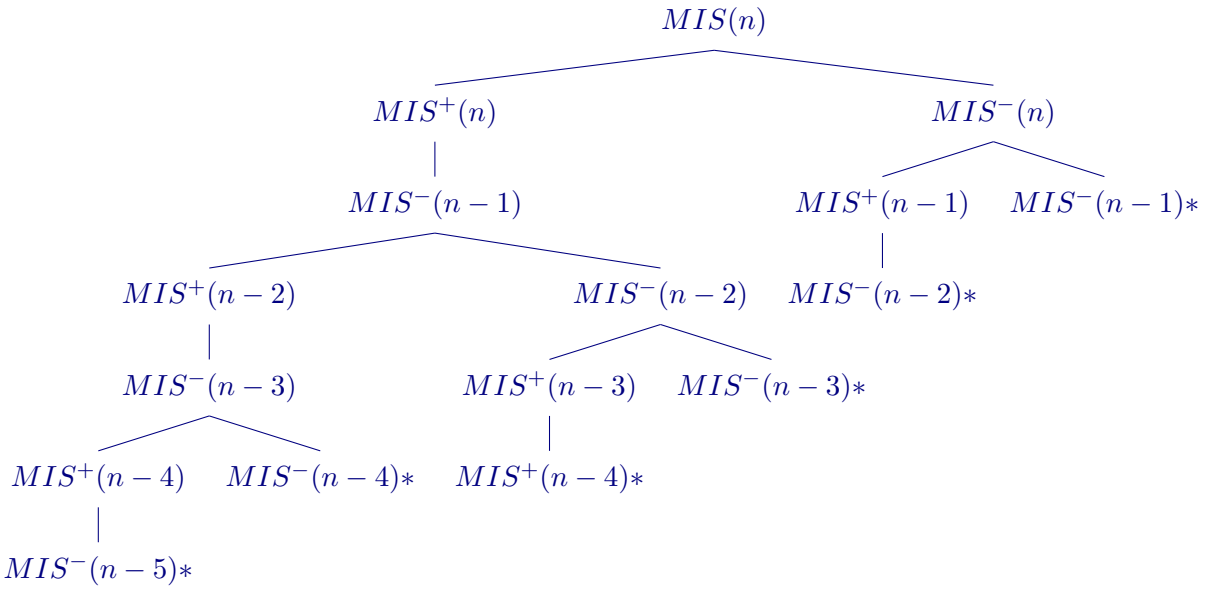
Algorithm $MIS(P)$
// Soln is an array with solutions, at the begining it is filled with n -1s
$Soln = [-1... - 1_n]$
$n = |V|$
return $\texttt{max}\{MIS^+(P, n, Soln), MIS^-(P, n, Soln)\}$

Algorithm $MIS^+(P, i, Soln)$
If $i = 0$
return 0
return $w(i) + MIS^-(P, i - 1, Soln)$

Algorithm $MIS^-(P, i, Soln)$
If $i \leq 1$
return 0
If $Soln[i] = -1$
$Soln[i] = \texttt{max}\{MIS^+(P, i - 1, Soln), MIS^-(P, i - 1, Soln)\}$
return $Soln[i]$

3. Using a recursion tree, explain why the running time of your algorithm of part 3 is $\Theta(n)$.

Below is the recursion tree for the example graph path $(2 - 8 - 6 - 3 - 6)$, the leaf nodes are marked with an *:



Using this recursion tree, we can notice that we due to the memoization on the function $MIS^-(P, i)$ we wont have more than n recursive calls to $MIS^-(P, i)$, and we will also have no more than n calls to $MIS^-(P, i)$ that are leaves, either reaching the base case or stopped because the solution is already computed. So we would have $2n$ calls to $MIS^-(P, i)$ doing constant c_1 operations, thus $2c_1n$

for calling all the $MIS^-(P,i)$ on $MIS(P)$.

We also need the time of calling all the $MIS^+(P,i)$ which are upperbounded to n because one call either is a base case or call $MIS^-(P,i)$, since $MIS^-(P,i)$ is upperbounded by n, the calls to $MIS^+(P,i)$ will be also upperbounded by n, hence the running time of all $MIS^+(P,i)$ calls is $c_2 n$.

By adding the running time of both all the calls we get, $2c_1 n + c_2 n$ which is $O(n)$.

5 More Max-Weight Independent Sets

This problem concerns independent sets on "rail line" graphs. These undirected graphs have $2n$ nodes, labeled $[1,i]$, $[2,i]$, $1 \leq i \leq n$. A function $w()$ assigns a weight to each node. There are edges between $[1,i]$ and $[2,i]$ for each i, as well as edges between $[1,i]$ and $[1,i+1]$ for $1 \leq i \leq n-1$ and between $[2,i]$ and $[2,i+1]$ for $1 \leq i \leq n-1$, for a total of $2(n-1)+n = 3n-2$ edges.

Develop recurrences that could be used to efficiently compute MIS(G) when the input G is a rail line graph. You can express MIS(G) in terms of other quantities of your choosing that you can define, and then provide recurrences for these quantities. You do *not* need to describe an algorithm to compute MIS(G).