Exercise Set #5

Fall 2019, CMPUT 204
Department of Computing Science
University of Alberta

In this exercise, we focus on divide & conquer and have more problems on the greedy approach.

Some students ask for more practice problems. We include in this exercise addtional problems. Only problems from 1-8 are considered for quizzes. Problems 9 and up are for practice.

Problem 1. For the exponentiation problem discussed in class, show the sequence of recursive calls to compute `Power`$(b, 61)$. What are the values of n such that recursive calls are alternating between odd values of n and even valuses of n? For example, this is the case when we compute `Power`$(b, 15)$.

Problem 2. Exercise 4.2-3, 4.2-6 (CLRS p.82-83)

Exercise 4.2-3: How would you modify Strassen's algorithm to multiply $n \times n$ matrices in which n is not an exact power of 2? Show that the resulting algorithm runs in time $\Theta(n^{\log 7})$.

Exercise 4.2-6: How quickly can you multiply a $kn \times n$ matrix by an $n \times kn$ matrix, using Strassen's algorithm as a subroutine? Answer the same question with the order of the input matrices reversed.

Problem 3. Given a set P of n teams in some sport, a Round-Robin tournament is a collection of games in which each team plays each other team exactly once. Design an efficient algorithm for constructing a Round-Robin tournament assuming n is a power of 2. A restriction is that every team can play at most one game per day and the goal is to schedule all the games in $n - 1$ days.

Problem 4. The Maximum Subarray Sum Problem: You are given a one dimensional array that may contain both positive and negative integers, find the sum of contiguous subarray of numbers which has the largest sum. For example, if the given array is $A = [-2, -5, \mathbf{6}, \mathbf{-2}, \mathbf{-3}, \mathbf{1}, \mathbf{5}, -6]$, then the maximum subarray sum is 7 (see highlighted elements).

A naive solution is to check each contiguous subarray. A divide-and-conquer algorithm can reduce the running time to $O(n \log n)$. Describe such an algorithm and analyze its running time.

Problem 5. Consider the following matrix:

$$F = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}$$

a. Recall the definition of Fibonacci numbers where $f_0 = 0$, $f_1 = 1$, and for $n \geq 2$: $f_n = f_{n-1} + f_{n-2}$. Prove by induction that for each $n \geq 1$:

$$F^n = \begin{bmatrix} f_{n-1} & f_n \\ f_n & f_{n+1} \end{bmatrix}$$

b. Use part (a) to design an $O(\log n)$ time algorithm to compute f_n. Justify the running time of your algorithm.

Problem 6. A server has n customers waiting to be served. The service time required by each customer is known in advance: it is t_i minutes for customer i. So if, for example, the customers are served in order of increasing i, then the ith customer has to wait $\sum_{j=1}^{i} t_j$ minutes.
We wish to minimize the total waiting time

$$T = \sum_{i=1}^{n} (\text{time spent waiting by customer } i)$$

Give an efficient algorithm for computing the optimal order in which to process the customers. Prove that your algorithm gives an optimal solution.

Problem 7. Suppose you have started a security company and need to buy licenses for n different cryptography softwares. Due to regulations you can only buy one license per month. Right now all these softwares are \$100 each but they are all becoming more expensive at different rates. In particular the cost of software i is multiplied by a factor of $r_i > 1$ each month (so after t months it will cost $\$100 * r_i^t$. We assume all r_i's are distinct (i.e. $r_i \neq r_j$). The question is: given that you can buy only one software each month in which order should you buy the licenses to that the total amount of money spent is as small as possible? Give a an algorithm and analyze the running time and correctness.

Problem 8. You have a manufacturing company who produces some form of glass that is more resistant than a typical glass. In your quality control section you do testing of samples and the setup for a particular type of cup produced is as follows. You have a ladder with n steps and you want to find the highest step from which you can drop a sample cup and not have it break. We call this the highest safe step. A natural approach to find that highest safe step is binary search: drop a cup from step $n/2$ and depending on whether it breaks or not try step $n/4$ or $3n/4$ (and of course if it broke you take another sample), and so on. Of course you can find the answer quickly (in $O(\log n)$ drops) but the drawback is you may break many cups.

If your main goal is to break as few cups as possible, you can try dropping from the 1st, then 2nd, then 3rd, and so on until you find the first step from which dropping the cup will break it. This way you only break one cup but the drawback is you may need up to n expriments.

So the trade-off is: To perfrom the expriment more quickly you might break many more cups. Suppose you are given a fixed budget $k \geq 1$ and you are allowed to break at most k cups in the entire experiment (for quality control) and yet want to do the expriment as quickly as possible. So you want to find the highest safe step using at most k cups and as quickly as possible.

a. Suppose that $k = 2$. Describe a strategy that uses at most 2 cups to find the highest safe step among the n steps. If $T(n)$ is the function that upper bounds the number of expriements performed by your algorithm (i.e. how many times you drop a cup) it must be the case that $T(n) \in o(n)$. Try to make $T(n)$ as small as you can and also explain why your algorithm uses at most this many cups.

b. Now suppose your budget is $k > 2$ for some given k that is at most $O(\log n)$. Describe an algorithm for finding the highest safe step among n steps that uses at most k cups. If $T_k(n)$ denotes the number of times your algorithm drops a cup, it must be the case that $T_1(n), T_2(n), T_3(n), \ldots$ have the following property that each is asymptotically smaller than the previous one: i.e. $\lim_{n\to\infty} T_k(n)/T_{k-1}(n) = 0$ for each k. Explain why your algorithmuses at most this many cups.

The following problems are for practice only. They will not be considered for the quizzes.

Problem 9. Count number of occurrences in a sorted array. Given a sorted array A and a number x, write an algorithm that counts the occurrences of x in A. Expected time complexity is $O(\log n)$. For example, if $A = [2, 3, 5, 5, 5, 7, 7, 8]$ amd $x = 5$. Your algorithm should return 3.

Problem 10. You have n coins of the same weight, except for a "special" one which is slightly heavier. You want to find out this special coin. The only thing you have is a scale — on which you can weigh any set of coins against another set of coins.

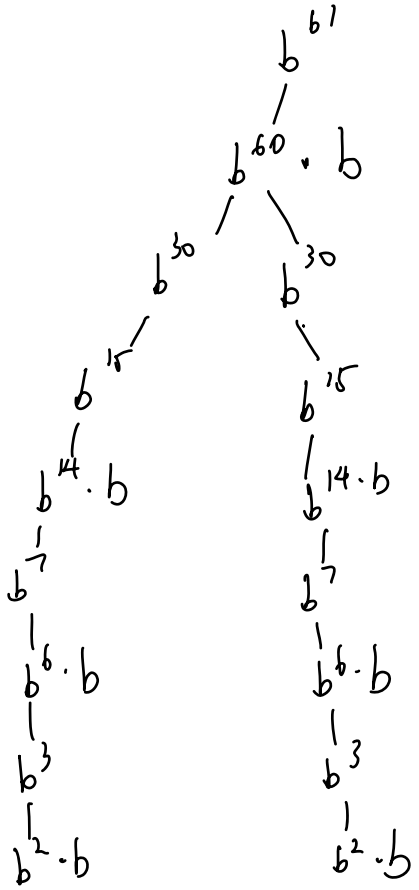
a. Design an algorithm to find out which one is the special coin, your algorithm should minimize the number of weighings required until the special coin is found.

b. Consider the case that the special coin could either be heavier or lighter than the others. That is, you know its weight is different from the others, but whether it is heavier or lighter is unknown. In the worst-case, what is the minimum number of weighings you will need to find the special coin among 12 coins? Describe your algorithm.

Problem 11. Exercise 16.1-2 from CLRS.

Exercise 16.1-2: Suppose that instead of always selecting the first activity to finish, we instead select the last activity to start that is compatible with all previously selected activities. Describe how this approach is a greedy algorithm, and prove that it yields an optimal solution.

Problem 1. For the exponentiation problem discussed in class, show the sequence of recursive calls to compute `Power`$(b, 61)$. What are the values of n such that recursive calls are alternating between odd values of n and even valuses of n? For example, this is the case when we compute `Power`$(b, 15)$.



when $n = 2^k - 1$

Problem 2. Exercise 4.2-3, 4.2-6 (CLRS p.82-83)

Exercise 4.2-3: How would you modify Strassen's algorithm to multiply $n \times n$ matrices in which n is not an exact power of 2? Show that the resulting algorithm runs in time $\Theta(n^{\log 7})$.

Exercise 4.2-6: How quickly can you multiply a $kn \times n$ matrix by an $n \times kn$ matrix, using Strassen's algorithm as a subroutine? Answer the same question with the order of the input matrices reversed.

4.2-3

pad the matrix to be power of two and run the algorithm. find the next largest power of two (m) will at most double the value of n since each power of two is off from eachother, so we have

$n^{\lg 7} \geq (2n)^{\lg 7} = 7n^{\log 7} \in O(n^{\log 7})$

and $m^{\log 7} \geq n^{\lg 7} \in \Omega(n^{\log 7})$

$\therefore$ run in $\Theta(n^{\log 7})$

4.2-6

time. since $(kn \times n)(n \times kn)$ produce $kn \times kn$ matrix. which produce k^2 multiplication of $n \times n$ matrix. runs in $\Theta(k^2 n^{\log 7})$

Problem 3. Given a set P of n teams in some sport, a Round-Robin tournament is a collection of games in which each team plays each other team exactly once. Design an efficient algorithm for constructing a Round-Robin tournament assuming n is a power of 2. A restriction is that every team can play at most one game per day and the goal is to schedule all the games in $n-1$ days.

```
RoundRobin(n)
  let L be an empty list
  if n=1
      return {}

  if n=2 then
    return {1, 1, 2}

  list 1 ← RoundRobin(n/2)   // schedule for the first half
  list 2 ← RoundRobin(n/2)   // schedule for the second half

  for each team t1 in list 1
      for each team t2 in list 2
        if (t1, t2) ∉ L
              L = L ∪ (list1, list2)

  L = L ∪ list1 ∪ list2

  return L
```

Problem 4. The Maximum Subarray Sum Problem: You are given a one dimensional array that may contain both positive and negative integers, find the sum of contiguous subarray of numbers which has the largest sum. For example, if the given array is $A = [-2, -5, \mathbf{6}, \mathbf{-2}, \mathbf{-3}, \mathbf{1}, \mathbf{5}, -6]$, then the maximum subarray sum is 7 (see highlighted elements).

A naive solution is to check each contiguous subarray. A divide-and-conquer algorithm can reduce the running time to $O(n \log n)$. Describe such an algorithm and analyze its running time.

```
Find_maximum_Subarray(A, low, high)
  if high == low
      return (low, high, A[low])
  else
      mid = ⌊(low + high) / 2⌋
      (left-low, left-high, left-sum) =
          Find-maxim-SubArray(A, low, mid)
      (right-low, right-high, right-sum) =
          F... M- S--- (A, mid+1, high)
      (cross-low, cross-high, cross-sum) =
          find-max-cross-subarray(A, low, mid, high)
  if left-sum > right-sum and left-sum ≥ cross-sum
      return (left-low, left-high, left-sum)
  if right-sum ≥ left-sum and right-sum ≥ cross-sum
      return (right-low, right-high, right-sum).
  else
      return (cross-low, cross-high, cross-sum).
```

$T = \Theta(1 + 2T(\frac{n}{2}) + \Theta(n) + \Theta(1))$

$= 2T(\frac{n}{2}) + \Theta(n)$

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2T(\frac{n}{2}) + \Theta(n) & \text{if } n > 1 \end{cases}$$

$a = 2 \quad b = 2 \quad d = 1$ since $b^d = 2^1 = 2 = a$

$\Theta(n^d \log n) = \Theta(n^1 \log n)$

Problem 5. Consider the following matrix:

$$F = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}$$

a. Recall the definition of Fibonacci numbers where $f_0 = 0$, $f_1 = 1$, and for $n \geq 2$: $f_n = f_{n-1} + f_{n-2}$. Prove by induction that for each $n \geq 1$:

$$F^n = \begin{bmatrix} f_{n-1} & f_n \\ f_n & f_{n+1} \end{bmatrix}$$

b. Use part (a) to design an $O(\log n)$ time algorithm to compute f_n. Justify the running time of your algorithm.

a. $F^1 = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} = \begin{bmatrix} f_0 & f_1 \\ f_1 & f_2 \end{bmatrix}$

Suppose it is true for all $n \geq 1$, for $n+1$ we have

$F^{n+1} = F^n \cdot F = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} f_{n-1} & f_n \\ f_n & f_{n+1} \end{bmatrix} = \begin{bmatrix} f_n & f_{n+1} \\ f_{n+1} & f_{n+2} \end{bmatrix}$

true.

b. fibmatrix (A, n)

if n = 0 then
  return A
else
  if n is odd then
    B ← FibMatrix (A, n−1)
    return A * B
  else
    B ← FibMatrix (A, $\frac{n}{2}$)
    return B * B

O(log n)

Problem 6. A server has n customers waiting to be served. The service time required by each customer is known in advance: it is t_i minutes for customer i. So if, for example, the customers are served in order of increasing i, then the ith customer has to wait $\sum_{j=1}^{i} t_j$ minutes.
We wish to minimize the total waiting time

$$T = \sum_{i=1}^{n} (\text{time spent waiting by customer } i)$$

Give an efficient algorithm for computing the optimal order in which to process the customers. Prove that your algorithm gives an optimal solution.

1. sort the customer lists by their service time

2. select the first customer the from the sorted list

3. select the customers from the list one by one till the last one.

$T = \sum_{i=1}^{n}$ (time spent by customer i)

$= \sum_{i=1}^{n} (n-i)\, t_i$

Suppose the solution is not optimal. Then there must exist an optimal sol. where i is served before customer j.

$T(\text{new}) - T(\text{old}) = (n-j)t_i + (n-i)t_j - (n-i)t_i - (n-j)t_j$

$= nt_i - jt_i + nt_j - it_j - nt_i - it_i - nt_j - jt_j$

$= t_i(i-j) - t_j(j-i)$

$= (t_i - t_j)(i-j) \leq 0$

Problem 7. Suppose you have started a security company and need to buy licenses for n different cryptography softwares. Due to regulations you can only buy one license per month. Right now all these softwares are \$100 each but they are all becoming more expensive at different rates. In particular the cost of software i is multiplied by a factor of $r_i > 1$ each month (so after t months it will cost $\$100 * r_i^t$. We assume all r_i's are distinct (i.e. $r_i \neq r_j$). The question is: given that you can buy only one software each month in which order should you buy the licenses to that the total amount of money spent is as small as possible? Give a an algorithm and analyze the running time and correctness.

ordering the softwares in decreasing order buying them in that order.