Isai Lopez Rodas
1605542 ilopezro@ucsc.edu

CMPS 102 — Spring 2020 – Homework 2

Four problems, 10 points each, due Monday April 27 (11:59 PM) on Gradescope.

Before you begin the assignment, please read the following carefully.

- Read the *Homework Guidelines*.
- The assignment consists of four problems worth ten points each. Unless otherwise stated, algorithms and their proofs of correctness are weighted equally.
- Due to class size, it is likely that only a subset of the problems will be graded. We will announce the problem(s) which will be graded after the due date of the assignment.
- Every part of each question begins on a new page. Do not change this.
- This does not mean that you should write a full page for every question. Your answers should be short and precise. Lengthy and wordy answers will lose points.
- Do not change the format of this document. Simply type your answers as directed.
- You are **not** allowed to work in teams.

I have read and agree to the collaboration policy. – Isai Lopez Rodas, ilopezro@ucsc.edu
Collaborators: None

Recommended exercises (not to be turned in)

1. The solved exercises in 5 (Divide and Conquer).

Continued on next page.

Problems to be turned in

1. You are interested in building a coffeeshop and charging station beside a freeway running between two cities. Let n be the number of interchanges between the cities, and number them consecutively so that interchange 0 is at the first city, 1 is the next interchange, and so on, with $n-1$ being the interchange at the other city. We say each interchange i is adjacent to interchanges $i-1$ and $i+1$ (only).

   Each interchange i has a cost $c(i)$ to construct a coffee shop/charging station. Call an interchange i a *local minimum* if (and only if) the cost of building there is less than the costs of building at the adjacent interchanges, i.e. both $c(i) < c(i+1)$ and $c(i) < c(i-1)$. You want to ensure that you build at a local minimum so that no one can build a coffeeshop/charging station more cheaply at an adjacent interchange.

   You can assume that:

   - $n \geq 3$, so there is at least one interchange between the cities.
   - The costs at the cities $c(0)$ and $c(n-1)$ are both known to be the same large number (say $100n$).
   - The other costs are all different and are all strictly less than $c(0)$ (and thus also less than $c(n-1)$).

   The difficulty is that the $c(i)$ values are not known ahead of time (except for $c(0)$ and $c(n-1)$) and it is expensive to determine the cost of building at any interchange.

   The goal is to create a divide and conquer algorithm that finds any interchange k that is a local minimum while examining relatively few of the $c(i)$ values. Although there may be many local minimums, your algorithm need only find one of them.

(a) (2 points) First, prove that a local minimum exists.

Solution.

I will use proof by contradiction to show that a local minumum exists.

Base Case: When n = 3, we know that $c(0)$ and $c(2)$ are both equal to each other in terms of costs. Additionally, we also know that $c(1)$ is stricly less than $c(0)$ and $c(2)$. This means that there is a local minimum at n = 2 since $c(1) < c(0)$ **AND** $c(1) < c(2)$.

For the sake of contradiction, I will assume that no local minimum exists. By the definition of local maxima and minima, this means that $c(0)$ will have to be the largest cost and every interchange thereafter is either stricly lower or stricly higher than the cost of building that first interchange. So if $c(0)$ is the largest cost, that still satisfies the 3^{rd} requirement of being able to build an interchange. According to the 2^{nd} requirement, $c(0)$ is the largest cost. So I will say that interchange city $c(1)$ will be stricly less than $c(0)$. $c(2)$ will be stricly less than $c(1)$ and every city until $c(n-1)$ will all be stricly less than their previous interchange city. This creates the inequality $c(0) > c(1) > c(2) \cdots > c(n-1)$; however, we've got a contradiction here because this breaks the 2^{nd} requirement in that $c(0) = c(n-1)$.

(b) (3 points) Clearly describe a divide and conquer algorithm for the problem

Algorithm.

I will design a Binary Search algorithm that returns the first local minumum that it finds. It will contain two recursive calls on the first $\frac{n}{2}$ and the last $\frac{n+1}{2}$ elements.

1: **procedure** BINARY SEARCH($\mathcal{A}$, $\mathcal{L}$, $\mathcal{R}$) ▷ $\mathcal{A}$ is array, $\mathcal{L}$ is left index, $\mathcal{R}$ is right index
2: $len \leftarrow$ length of $\mathcal{A}$
3: **if** $len = 3$ **then**
4: **return** 1
5: **else** ▷ $len > 3$
6: $mid \leftarrow \frac{\mathcal{L}+\mathcal{R}}{2}$
7: $midElement \leftarrow \mathcal{A}[mid]$
8: $leftElement \leftarrow \mathcal{A}[mid - 1]$
9: $rightElement \leftarrow \mathcal{A}[mid + 1]$
10: **if** $midElement < leftElement$ and $midElement < rightElement$ **then**
11: **return** mid
12: **else if** $leftElement > rightElement$ **then**
13: BINARY SEARCH($\mathcal{A}$, $mid + 1$, $\mathcal{R}$)
14: **else**
15: BINARY SEARCH($\mathcal{A}$, $\mathcal{L}$, mid)
16: **end if**
17: **end if**
18: **end procedure**

I have a base case on line 3 when the length of the array = 3. Since we proved that a local minimum must exists, I start off by saying that if our array is of size 3, then return the middle value. I then compare the middle element of the array to its adjacent elements and if they middle element is less than its adjacent elements, we have found a local minimum. I return the index of the minima. If the local minimum does not exists within the initial $\mathcal{L}$ and $\mathcal{R}$ indeces, I recursively call on $\mathcal{L}$ to mid and $mid + 1$ to $\mathcal{R}$ until a minumum is found.

(c) (3 points) Prove that your algorithm is correct

Proof of correctness.

Base Case: $n = 3$.
My algorithm will correctly return the local minima at index 1 because we are given the rule that the first and last elements/cities are the largest and all other interchanging stops are stricly lower than the first and last element.

Inductive Step: I will assume that my algorithm correctly sorts elements up until n. I will prove that my algorithm correctly sorts $n + 1$ elements.

Since we are only looking for first lowest element we can find, will initially compare our median value to its neighboring elements. If the median value is not a local minima, we will recurse on the side where the smallest of the three elements was, which reduces the number of elements we are looking at by half. Because we recurse on the side that has the smallest element, we are guaranteed to find the smallest value because we are following the smallest element all the way through until we find a local minima.

(d) (2 points) Analyze the (worst case) number of interchange costs (i.e. $c(i)$ values) examined by your algorithm as a function of n.

Solution.

If we have n number of cities, I can deduce that in the worst case, the algorithm will compare $\frac{n}{2}$ every time it is called. Within each recursive call, we can have up to three comparisons made. We compare median value to its adjacent element values and if no minima is found, we go on to compare if the left and right elements are less than or greater than each other. These comaprisons are made in constant time so our recurrance relation is $T(n) = T(\frac{n}{2}) + \mathcal{O}(1)$. Using Master's Theorem, we can say that:

$$a = 1,\ b = 2,\ d = 0$$

Using case #2, I get $f(n) = \mathcal{O}(n^{\log_b a})$. We can then say that $T(n) \leq \mathcal{O}(n^{\log_b a} \log n)$. Plugging in the above values, I get:

$$\begin{aligned} T(n) &\leq \mathcal{O}(n^{\log_2 1} \log n) \\ &\leq \mathcal{O}(n^0 \log n) \\ &\leq \mathcal{O}(\log n) \end{aligned}$$

Therefore, we can conclude, through the Master's Theorem, that the worst case runtime of my algorithm is $\mathcal{O}(\log n)$.

I have read and agree to the collaboration policy. – Isai Lopez Rodas, ilopezro@ucsc.edu
Collaborators: None

---

2. We all know how to search a sorted 1-dimensional array. This problem involves searching two dimensional arrays that are partially sorted.

   Call a two-dimensional square ($n \times n$) array A *line-sorted* if

   - the entries of A are all distinct numbers,
   - the entries in each row are sorted in ascending order (left to right), and
   - the entries in each column are sorted in ascending order (top to bottom).

   If A is line-sorted then both $A[i-1,j] < A[i,j] < A[i+1,j]$ and $A[i,j-1] < A[i,j] < A[i,j+1]$ (assuming the indices are inside the array).

   Devise a *two-dimensional divide and conquer* algorithm which takes a line-sorted array A and number k to search for, and returns either indices i and j such that $A[i,j] = k$ or an indicator that k is not in the array. Describe your algorithm (in pseudo-code), argue that your algorithm is correct, give and and analyze its (worst-case) running time through a recurrence. You may assume that n is a power of two.

   The goal is for your algorithm to take asymptotically less time than the number of elements in the array, but it will probably not be as efficient as simply performing binary search on each row.

Continued on the next page.

Algorithm.

```
1:  procedure TwoDSearch(A, k)                    ▷ A - array, k - element to find
2:      row ← 0
3:      column ← len(A) − 1
4:      while row ≤ len(A) or column ≥ 0 do
5:          if A[row][column] = k then
6:              return True
7:          else if A[row][column] < k then
8:              row ++
9:          else
10:             column − −
11:         end if
12:     end while
13:     return False
14: end procedure
```

Proof of correctness.

Loop Invariant: Element k is contained within the submatrix whose top right corner is (row, col) only if k exists in the matrix.

Base Case: $n = 1$
If we have a 1×1 matrix that contains the element 1 and we try and find element 1 in that matrix, our loop invariant holds true because the top right corner of our 1×1 matrix holds the element we are looking for.

Inductive Hypothesis: Assume that our loop invariant holds true from $1 \rightarrow n$.

Inductive Step: We want to prove that it also holds true for $n + 1$.
For an $n \times n$ matrix:
If element $k > \mathcal{A}[row][col]$ we know that all the elements within that row are also less than k, so we have to move down a row to try and search for that item. Because we move down a row, our loop invariant still holds true because the element is still within the submatrix whose topright corner is now $(row + 1, col)$.

Second case is if $k < \mathcal{A}[row][col]$. With this case, we know that all the elements in that specific col are way too big for element k. Therefore, we'd have to move a column down to try and find element k. Our loop invariant still holds true for this case because if our element is within our matrix, it is for certain that submatrix $\mathcal{A}[row][col - 1]$ contains that element.

Since we know that our loop invariant holds true up until n for an $n \times n$ matrix, if we add another column and row, our loop invariant would still hold since element k would still be within submatrix whose top right corner is (row, col).

Runtime Analysis: The runtime of the algorithm is $\mathcal{O}(n)$. In its worst case, when the element you're looking for is at the bottom left corner, it has to traverse across n times and down another $n - 1$ times. The total runtime would be $\mathcal{O}(n + n - 1) = \mathcal{O}(2n - 1) = \mathcal{O}(n)$.

I have read and agree to the collaboration policy. – Isai Lopez Rodas, ilopezro@ucsc.edu
Collaborators: None

---

3. Problem 3 in Chapter 5 (Divide and Conquer). This is the equivalent back card problem: determine if *more than* half the n possible fraudulent bank cards are from the same account using a machine that detects if two cards are from the same account or not. If this doesn't sound like Problem 3 in Chapter 5 of your version of the text, then let us know.

   Clearly describe your algorithm, prove that it is correct, and analyze its running time. For full credit, the running time should be $O(n \lg n)$.

   Algorithm.

```
1:  procedure FRAUDULENTCARDS(A)                          ▷ A is the array of cards
2:      if len(A) = 1 then
3:          return A[0]
4:      end if
5:      firstHalf ← A[0 ... len(A)/2]
6:      secondHalf ← A[(len(A)+1)/2 ... len(A)]
7:      if firstHalf ≠ NULL then
8:          counter ← 0
9:          for card in A do
10:             if card = firstHalf then
11:                 counter++
12:             end if
13:         end for
14:         if counter > len(A)/2 then
15:             return card
16:         end if
17:     end if
18:     if secondHalf ≠ NULL then
19:         counter ← 0
20:         for card in A do
21:             if card = secondHalf then
22:                 counter++
23:             end if
24:         end for
25:         if counter > len(A)/2 then
26:             return card
27:         end if
28:     end if
29:     return None
30: end procedure
```

   I will recursively call on the first and second half of the card set. If there is anything returned from the function call, we will iterate through the set of cards and if there are matches greater than $\frac{n}{2}$ then we will return that Card as fraudulent.

Proof of correctness.

Base Case: $n = 1$
When the set of cards is equal to one, it will return that card and since it is greater than $\frac{n}{2}$ it is a card from the single account.

Inductive Hypothesis: Assume algorithm works from $1 \rightarrow k$

Inductive Step: Prove it works for set of cards $k + 1$
The algorithm would run and find all cards that are either fraudulent or from the same account through the inductive hypothesis. They will continue to split into $\frac{k}{2}$ subsets until it eventually return a single card and iterates through the set of cards to find a match. If we add another card to the set $\mathcal{A}$, the algorithm would return that card and iterate through $\mathcal{A}$ to count how many other cards are the same. Therefore, our algorithm works for $1 \rightarrow k + 1$ elements.

Runtime Analysis: Since the algorithm recursively calls on $\frac{n}{2}$ subsets of the array for a depth k, we get an initial recurrence relation of $T(n) = T(\frac{n}{2})$. Since it does it for the two halves, we multiply it by 2. We also iterate through the set twice every recursive call. Our final recurrence relation is the following:

$$T(n) \leq 2T(g)\tfrac{n}{2} + 2n$$

Again, we use coefficients $a = 2$, $b = 2$, and $d = 0$. Our $f(n) = \mathcal{O}(n^{\log_b a})$. Our recurrence relation would become:

$$T(n) \leq \mathcal{O}(n^{\log_b a} \log n)$$

After plugging in our equation, we get:

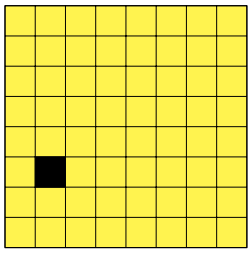
$$\begin{aligned} T(n) &\leq \mathcal{O}(n^{\log_2 2} \log n) \\ &\leq \mathcal{O}(n^1 \log n) \\ &\leq \mathcal{O}(n \log n) \end{aligned}$$

Therefore, in the worst case, our algoritm would take $\mathcal{O}(n \log n)$

I have read and agree to the collaboration policy. – Isai Lopez Rodas, ilopezro@ucsc.edu
Collaborators:

---

4. Tiling a Room. You have a square room that is 2^k feet on a side (for some $k \geq 1$), so you can think of the room as a grid of $2^k \times 2^k$ cells. One of these cells is used for a floor vent. You are to find a way to fill in the $(2^k \times 2^k) - 1$ other cells perfectly with L-shaped tiles, like the one shown. The L-shaped tiles can be rotated in four different ways, but cannot be cut or extend outside the room. Give a divide and conquer algorithm that finds a tiling and argue that it is correct.



For full credit, your algorithms should take $O(n)$ time where $n = 2^{2k} = (2^k)^2$ is the number of cells in the room. The intent of the problem is to show that rooms of this kind can be tiled, do not worry about the book-keeping needed to record the particular solution found by your divide and conquer algorithm.

Continued on next page.

Algorithm.

1: **procedure** TILING(n, p) ▷ n number of tiles, p is missing cell
2: **if** $n = 2$ **then** ▷ Base Case
3: Place tile around air vent
4: **end if**
5: Place Tile in the center, but in such a way that it is not inside the $\frac{n}{2} \times \frac{n}{2}$ square that contains air vent
6: TILING($\frac{n}{2}, p$)
7: TILING($\frac{n}{2}, p$)
8: TILING($\frac{n}{2}, p$)
9: TILING($\frac{n}{2}, p$)
10: **end procedure**

Proof of correctness.

Base Case: $n = 2$
When we have a 2×2 square, we can simply place the L-shaped tile around the air vent covering the entire 2×2 square.

Inductive Hypothesis: We will assume that this algorithm works for $2^k \times 2^k$ squares given that $k \geq 1$.

Inductive Step: I will try to prove that this algorithm works for a square of size $2^{k+1} \times 2^{k+1}$. When our algorithm begins, we place an L-shaped tile in the middle of the $2^{k+1} \times 2^{k+1}$ square. When we begin our recursive calls, our first call we attempt to tile a $\frac{2^{k+1}}{2} \times \frac{2^{k+1}}{2}$ square. Mathematically, we know that:

$$\begin{aligned}\frac{2^{k+1}}{2} \times \frac{2^{k+1}}{2} &= \frac{2^k \times 2}{2} \times \frac{2^k \times 2}{2} \\ &= 2^k \times 2^k\end{aligned}$$

Through our *Inductive Hypothesis*, we can tile a $2^k \times 2^k$ square. Therfore, our induction is complete and we can conclude that our algorithm is correct.

Runtime Analysis: The algorithm recursively calls itself 4 times on $\frac{n}{2}$ sized squares each time, where n is the dimension of the entire square, 2^k. Our recurrence relation is then:

$$T(n) = 4T(\frac{n}{2}) + \mathcal{O}(1)$$

We add a constant because of the initial two comparisons that is done at the beginning of the algorithm that takes constant time. I will use Master's Theorem once again to show the runtime of this algorithm. I will use coefficients $a = 4$, $b = 2$. I will also use case #1:

$$\begin{aligned}f(n) &= \mathcal{O}(n^{\log_b a}) \\ T(n) &\leq \mathcal{O}(n^{\log_b a}) \\ &\leq \mathcal{O}(n^{\log_2 4}) \\ &\leq \mathcal{O}(n^2)\end{aligned}$$

We can conclude from Master's Theorem that this algorithm runs in $\mathcal{O}(n^2)$; however, I said that n was equivalent to 2^k, which is the dimension of the square. This means that if we plug in my value for n, we get that the runtime is $\mathcal{O}((2^k)^2)$ or $\mathcal{O}(2^{2k})$. I can set $n = 2^2k$ since the total number of tiles that we look at is $2^k \times 2^k = (2^k)^2$. Therefore, our algorithm runs in $\mathcal{O}(n)$.