CSCI-GA.1170-001/002 Fundamental Algorithms October 3, 2015

Solutions to Problem 1 of Homework 3 (10 (+5) Points)

Name: GOWTHAM GOLI (N17656180) *Due: Tuesday, September 29*

Sometimes, computing "extra" information can lead to more efficient divide-and-conquer algorithms. As an example, we will improve on the solution to the problem of maximizing the profit from investing in a stock (page 68-74).

Suppose you are given an array A of n integers such that entry $A[i]$ is the value of a particular stock at time interval i. The goal is to find the time interval (i, j) such that your profit is maximized by buying at time i and selling at time j. For example, if the stock prices were monotone increasing, then $(1, n)$ would be the interval with the maximal profit $(A[n] - A[1])$. More formally, the current formulation of the problem has the following input/output specification:

Input: Array A of length n.

Output: Indices $i \leq j$ maximizing $(A[j] - A[i])$.

(a) (6 Points) Suppose your change the input/output specification of the stock problem to also compute the largest and the smallest stock prices:

Input: Array A of length n.

Output: Indices $i \leq j$ maximizing $(A[j] - A[i])$, and indices α, β such that $A[\alpha]$ is a minimum of A and $A[\beta]$ is a maximum of A.

Design a divide-and-conquer algorithm for this modified problem. Make sure you try to design the most efficient "conquer" step, and argue why it works. How long in your conquer step? (**Hint**: When computing optimal i and j, think whether the "midpoint" $n/2$ is less than i, greater than j or in between i and j.)

Solution:

Approach

The algorithm given in the book has a running time of $O(n \log n)$ time. This is due to the $O(n)$ time taken in the conquer step. So instead of traversing the array from the middle element in both the directions, a better and efficient step will be to return the minimum and maximum elements from each half of the array using the recursive calls and then using these minimum and maximum elements to find the ideal buy-sell dates across the both halves in the conquer step.

Thus the conquer step will now take only $O(1)$ time instead of $O(n)$ time. Therefore, the recursion now returns

- Buy ans Sell dates for maximum profit
- Maximum price of the stock in the given range
- Mimumum price of the stock in the given range

Psuedocode

```
1  Algorithm: Stock-Profit(A, First, Last)
2  if First is Last then
3  |  return (First, Last, First, Last)
4  end
5  else
6  |  Mid ← (First+Last)/2
7  |  (buy_1, sell_1, min_1, max_1) ← Stock-Profit(A, First, Mid)
8  |  (buy_2, sell_2, min_2, max_2) ← Stock-Profit(A, Mid+1, Last)
9  |  min ← MINIMUM(min_1, min_2)
10 |  max ← MAXIMUM(max_1, max_2)
11 |  maxProfit_1 ← A[sell_1] − A[buy_1]
12 |  maxProfit_2 ← A[sell_2] − A[buy_2]
13 |  maxProfit_3 ← A[max_2] − A[min_1]
14 |  maxProfit ← MAXIMUM(maxProfit_1, maxProfit_2, maxProfit_3)
15 |  if maxProfit is maxProfit_1 then
16 |  |  return (buy_1, sell_1, min, max)
17 |  else if maxProfit is maxProfit_3 then
18 |  |  return (buy_2, sell_2, min, max)
19 |  else if maxProfit is maxProfit_3 then
20 |  |  return (max_2, min_1, min, max)
21 end
```

Algorithm 1: Algorithm of Stock-Profit problem in $O(n)$ time using divide and conquer

□

(b) (4 Points) Formally analyze the runtime of your algorithm and compare it with the runtime of the solution for the Stock Profit problem in the book.

Solution: As seen in the above Pseudocode, we are decreasing the $O(n)$ taken in conquer step to $O(1)$ time by maintaining "extra" information from the recursive calls. Thus $T(n)$ now reduces to

$$\begin{aligned} & T(n) = 2T(n/2) + O(1) \\ \implies & f(n) = O(n^{\log_2(2-1)}) \\ \implies & f(n) = O(n^{\log_b(a-\epsilon)}) \ \ with \ \epsilon > 0 \\ \implies & T(n) = \Theta(n^{\log_b a}) \\ \implies & T(n) = \Theta(n^{\log_2 2}) \\ & \quad = O(n) \end{aligned}$$

$\therefore$ The above discussed algorithm is $\log n$ times faster as compared to the book □

(c) (**Extra Credit**; 5 Points) Design a direct, non-recursive algorithm for the Stock Profit problem which runs in time $O(n)$. Write its pseudocode. Ideally, you should have a single "**For** $i = 1$ **to** n" loop, and inside the loop you should maintain a few "useful counters". Formally argue the correctness of your algorithm.
(**Hint**: Scan the array left to right and maintain its running minimum and the best solution found so far. Under which conditions would the best current solution be improved when scanning the next array element?)

Solution:

As seen in the book, Maximum Stock-Profit problem can be reduced to the change in stock prices as a maximum-subarray problem. Now we need to compute the Max-sum subarray in $O(n)$ time using a non-recursive algorithm i.e just using a single "**For** $i = 1$ **to** n"loop

Let B be the new array obtained by calculating the change in stock prices on two consecutive days i.e $B[i] = A[i+1] - A[i]$

Let $S(i)$ be the sum of the largest sub-array ending at i in the array B. Therefore, $B[i]$ surely contributes to $S(i)$. In order to solve the problem, it is sufficient if we compute $S(i)$ for $i = 0, \ldots, n-1$. Therefore, we need to formulate some method of computing the $S(i)$ for $i = 0, \ldots, n-1$ in $O(n)$ time

If $S(i-1) > 0$ then clearly, $S(i) = S(i-1) + B(i)$

If $S(i-1) \leq 0$ then $S(i-1) + B(i) \leq B(i) \implies S(i) = B(i)$

$$\therefore S(i) = \begin{cases} B(i), & \text{if } S(i-1) < 0 \\ S(i-1) + B(i), & \text{if } S(i-1) > 0 \end{cases}$$

Psuedocode

```
1  Algorithm: Stock-Profit(A)
2  B ← newArray[0 . . . n − 1]
3  S ← newArray[0 . . . n − 1]
4  buyIndex ← newArray[0 . . . n − 1]
5  for i = 0 to n-1 do
6  |  B[i] = A[i+1] - A[i]
7  end
8  S[0] ← B[0]
9  buyIndex[0] ← 0
10 for i = 1 to n-1 do
11 |  if S[i-1] > 0 then
12 |  |  S[i] ← S[i-1] + B[i]
13 |  |  buyIndex[i] ← buyIndex[i-1]
14 |  else
15 |  |  S[i] = B[i]
16 |  |  buyIndex[i] ← i
17 end
18 max ← −∞
19 for i = 0 to n-1 do
20 |  if S[i] > max then
21 |  |  max ← S[i]
22 |  end
23 end
24 return (buyIndex[i], i+1)
```

Algorithm 2: Non-recursive algorithm of Stock-Profit problem in $O(n)$ time

□

CSCI-GA.1170-001/002 Fundamental Algorithms October 3, 2015

Solutions to Problem 2 of Homework 3 (10 (+6) points)

Name: GOWTHAM GOLI (N17656180) *Due: Tuesday, September 29*

The Tower of Hanoi is a well known mathematical puzzle. It consists of three rods, and a number n of disks of different sizes which can slide onto any rod. The puzzle starts with all disks stacked up on the 1st rod in order of increasing size with the smallest on top. The objective of the puzzle is to move all the disks to the 3rd rod, while obeying the following rules.

- Only one disk is moved at a time
- Each move consists of taking one disk from top of a rod, and moving it on top of the stack on another rod
- No disk may be placed on top of a smaller disk.

A recursive algorithm that solves this problem is as follows: We first move the top $n-1$ disks from rod 1 to rod 2. Then we move the largest disk from rod 1 to rod 3 and then move the $n-1$ smaller disks from rod 2 to rod 3. Using the symmetry between the rods, the number of steps that this algorithm takes is given by the recurrence

$$T(n) = 2T(n-1) + 1 \ ,$$

which can be solved to get $T(n) = 2^n - 1$.

(a) (5 points) Show that the above algorithm is optimal, i.e., there does not exist a strategy that solves the Tower of Hanoi puzzle in less than $2^n - 1$ steps.

Solution:

Proof by Induction

Base Case - For $n = 1$, Directly move the disc from the first rod to third rod. Number of steps is less than $2^1 - 1$

Induction - Assume that the optimal solution takes less than $2^n - 1$ steps to solve the puzzle $\forall n = 1$ to n discs

Now in the case of $n + 1$ discs

- We move the first n discs from the first rod to second rod. Number of steps required is $2^n - 1$
- Now we move the last $(n+1)^{th}$ disc from the first rod to third rod. Number of steps required is 1
- Now move the n discs from the second rod to the third rod. Number of steps is $2^n - 1$

Therefore, total number of steps in moving $n + 1$ discs from first rod to third rod is $(2^n - 1) + (1) + (2^n - 1) = 2^{n+1} - 1$

$\therefore$ By Induction, This is the optimal solution and there doesn't exist a strategy that solves the puzzle in less than $2^n - 1$ steps $\square$

(b) (5 points) Suppose the moves are restricted further such that you are only allowed to move disks to and from rod 2. Give an algorithm that solves the puzzle in $O(3^n)$ steps.

Solution:

With the given restrictions, the puzzle can be solved by following the steps given below

- Move the top $n-1$ disks from rod 1 to rod 2
- Move the $n-1$ disks from rod 2 to rod 3
- Move the n^{th} disk from rod 1 to rod 2
- Move the $n-1$ disks from rod 3 to rod 2
- Move the $n-1$ disks from rod 2 to rod 1
- Move the n^{th} disk from rod 2 to rod 3
- Move the $n-1$ disks from rod 1 to rod 2
- Move the $n-1$ disks from rod 2 to rod 3

Let $T_{x2}(n)$ be the time taken to move n discs from rod x to rod 2 and $T_{2x}(n)$ be the time taken to move n discs from rod 2 to rod x

$$T(n) = 3T_{x2}(n-1) + 3T_{2x}(n-1) + 2$$

Note that, $T_{x2}(n) = T_{12}(n) = T_{32}(n)$ and $T_{2x}(n) = T_{21}(n) = T_{23}(n)$

To move n discs from rod 1 to rod 2, we proceed as follows

- Move the top $n-1$ discs from rod 1 to rod 2
- Move the $n-1$ discs from rod 2 to rod 3
- Move the n^{th} disc from rod 1 to rod 2
- Move the $n-1$ discs from rod 3 to rod 2

$$\therefore T_{x2}(n) = 2T_{x2}(n-1) + T_{2x}(n-1) + 1$$

To move n discs from rod 2 to rod 1, we proceed as follows

- Move the top $n-1$ discs from rod 2 to rod 1
- Move the n^{th} disc from rod 2 to rod 3
- Move the $n-1$ discs from rod 3 to rod 2
- Move the $n-1$ discs from rod 2 to rod 1

$$\therefore T_{2x}(n) = 2T_{2x}(n-1) + T_{x2}(n-1) + 1$$

We can see that $(T_{x2} + T_{2x})(n) = 3[(T_{x2} + T_{2x})(n-1)] + 2 \implies (T_{x2} + T_{2x})(n) = O(3^n)$

Substituting $(T_{x2} + T_{2x})(n)$ in $T(n)$ we get

$$\begin{aligned} T(n) &= 3[(T_{x2} + T_{2x})(n-1)] + 2 \\ &= 3.O(3^{n-1}) + 2 \\ \therefore T(n) &= O(3^n) \end{aligned}$$

□

(c) (6 points(**Extra credit**)) Suppose the moves are restricted such that you are only allowed to move from rod 1 to rod 2, rod 2 to rod 3, and from rod 3 to rod 1. Give an algorithm that solves the puzzle in $O((1+\sqrt{3})^n)$ steps.

Solution:

We can imagine that the rods are arranged in a cyclic order. Therefore time taken to move discs by one step (i.e from one rod to the immediate next rod) will be same for 1 to 2, 2 to 3 and 3 to 1. Let it be $T_1(n)$ i.e time taken to move n discs by 1 step.
Similarly, define $T_2(n)$ which is time taken to move n discs by 2 steps.

To solve the puzzle with the given restrictions, we proceed as follows

- Move the top $n-1$ discs from rod 1 to rod 3
- Move n^{th} disc from rod 1 to rod 2
- Move the $n-1$ discs from rod 3 to rod 1
- Move n^{th} disc from rod 2 to rod 2
- Move the $n-1$ discs from rod 1 to rod 3

$$\therefore T_2(n) = 2T_2(n-1) + T_1(n-1) + 2$$

To move the discs by 1 step proceed as follows (Consider that we are moving discs from 1 to 2)

- Move the top $n-1$ discs from rod 1 to rod 3
- Move the n^{th} disc from rod 1 to rod 2
- Move the $n-1$ discs from rod 3 to rod 2

$$\begin{aligned} \therefore T_1(n) &= 2T_2(n-1) + 1 \\ \therefore T_2(n) &= 2T_2(n-1) + (2T_2(n-2)+1) + 2 \\ \implies T_2(n) &= 2T_2(n-1) + 2T_2(n-2) + 3 \end{aligned}$$

□

CSCI-GA.1170-001/002 Fundamental Algorithms October 3, 2015

Solutions to Problem 3 of Homework 3 (10 Points)

Name: GOWTHAM GOLI (N17656180) *Due: Tuesday, September 29*

A *local minimum* of an array $A[1], \ldots, A[n]$ is an index $i \in \{1, \ldots, n\}$ such that either (a) $i = 1$ and $A[1] \leq A[2]$; or (b) $i = n$ and $A[n] \leq A[n-1]$; or (c) $1 < i < n$ and $A[i] \leq A[i-1]$ and $A[i] \leq A[i+1]$. Note that every array has at least (and possibly more than) one local minimum, since the "global" minimum of the entire array is also a local minimum. Design an $O(\log n)$ divide-and-conquer algorithm to find some local minimum of a given (unsorted) array A of size n.
(**Hint**: Think of binary search for inspiration.)

Solution: We need to note that if the last two elements of the array are increasing then there exists a local minima in the array. We can prove this by contradiction. Assume that there exists no local minima, then the $(n-1)^{th}$ element is smaller than the n^{th} element, $(n-2)^{th}$ element is smaller than than $(n-1)^{th}$ element and continuing so on, we get that the 1st element is smaller than the 2nd element. Hence our assumption is wrong. Therefore, there exists a local minima in the array.
Similarly, we can prove that if the first two elements of the array are decreasing then there exists a local minima in the array.
Using this logic we proceed as follows to calculate the local minima in any array A which is similar to the binary search

- Compute the middle element $A[mid]$. See if it is the local minima. If so, we are done
- Else, Check for the neighbor of the $A[mid]$ that is smaller than it. If it is $A[mid-1]$ then there exists a local minima in $A[0 \ldots mid-1]$. If it is $A[mid+1]$ then there exists a local minima in $A[mid+1 \ldots n-1]$

Psuedocode

```
1 Algorithm: Local-Minima(A, low, high)
2 mid ← (low+high)/2
3 if A[mid] is a local minima then
4 |   return A[mid]
5 else if A[mid-1] < A[mid] then
6 |   Local-Minima(A, low, mid-1)
7 else if A[mid+1] < A[mid] then
8 |   Local-Minima(A, mid+1, high)
```

Algorithm 3: Local minima in an array in $O(n)$ time

It is clear that the algorithm follows the same lines as that of binary search. Therefore, the running time of this algorithm is $O(\log n)$ □

CSCI-GA.1170-001/002 Fundamental Algorithms October 3, 2015

Solutions to Problem 4 of Homework 3 (18 points)

Name: GOWTHAM GOLI (N17656180) *Due: Tuesday, September 29*

A *local minimum* of a two-dimensional array $\{A[i,j] \mid 1 \leq i,j \leq n\}$ is an index $(i,j) \in \{1,\ldots,n\} \times \{1,\ldots,n\}$ which is less or equal than all of its neighbors, where we say that two nodes are neighboring if they are either vertically or horizontally (but not diagonally) adjacent in the array. Note that every array has at least (and possibly more than) one local minimum, since the "global" minimum of the entire array is also a local minimum. The eventual goal of this problem is to design an efficient divide-and-conquer algorithm to find some local minimum of a given (unsorted) array A of size $n \times n$.

(a) (2 points) Consider the following "greedy" algorithm. Start with any node $v = (i,j)$. If v is a local minimum, then output v. Else take the smallest neighbor of v (break ties arbitrarily) and repeat the above process with the neighbor until you find a local minimum. Prove that this algorithm always terminates in time $O(n^2)$.

Solution:

Note that the given greedy algorithm always moves to an element smaller than the current entry. So it will not revisit any entry of the matrix more than one time. As there are total n^2 number of entries, the algorithm will indeed terminate in $O(n^2)$ time □

(b) (3 points) What is the exact length (number of nodes, not "edges") of the "local minimum path" of the greedy algorithm on the following 7×7 grid, starting with the initial point $v = (1,1)$ (equal to 30). By generalizing this picture from $n = 7$ to general n, show that the worst case running time of the greedy algorithm is $\Omega(n^2)$.

$$\begin{pmatrix} 30 & 100 & 16 & 15 & 14 & 100 & 0 \\ 29 & 100 & 17 & 100 & 13 & 100 & 1 \\ 28 & 100 & 18 & 100 & 12 & 100 & 2 \\ 27 & 100 & 19 & 100 & 11 & 100 & 3 \\ 26 & 100 & 20 & 100 & 10 & 100 & 4 \\ 25 & 100 & 21 & 100 & 9 & 100 & 5 \\ 24 & 23 & 22 & 100 & 8 & 7 & 6 \end{pmatrix}$$

Solution:

Given below is the sequence of path the greedy strategy follows

$$\begin{pmatrix} (1)30 & 100 & (15)16 & (16)15 & (17)14 & 100 & (31)0 \\ (2)29 & 100 & (14)17 & 100 & (18)13 & 100 & (30)1 \\ (3)28 & 100 & (13)18 & 100 & (19)12 & 100 & (29)2 \\ (4)27 & 100 & (12)19 & 100 & (20)11 & 100 & (28)3 \\ (5)26 & 100 & (11)20 & 100 & (21)10 & 100 & (27)4 \\ (6)25 & 100 & (10)21 & 100 & (22)9 & 100 & (26)5 \\ (7)24 & (8)23 & (9)22 & 100 & (23)8 & (24)7 & (25)6 \end{pmatrix}$$

Total number of nodes visited is 31. Generalizing this picture from $n = 7$ to general n we can conclude that, In the worst case, the greedy algorithm visits $[n(n+1)/2 + (n-1)/2]$. nodes

Therefore, Time complexity is $\Omega(n^2)$ □

(c) (3 points) For simplicity of calculation, assume that $n = 2^k - 1$ for some integer $k \geq 1$. Consider the following divide-and-conquer algorithm: at every step, find the minimum element v of the middle row 2^{k-1} and the middle column 2^{k-1}. If v is a local minimum, output v. Else take the smallest neighbor of v (call it w), and recurse in the quadrant (north-east, north-west, south-east or south-west) of A of size $(n-1)/2 = (2^{k-1} - 1)$ where w lies (not counting the middle row/column in the quadrant, so one row/column is eliminated before dividing by 2).

While this algorithm looks appealing, you will prove that it is *not* correct in general. For this, consider the following grid containing all the numbers from 1 to 49 exactly once, except three weights $10, 31, 39$ are marked with the ?. Fill the missing numbers marked with ? with $10, 31, 39$ in a way such that the algorithm given above gives the wrong answer, and state what this wrong answer is.

$$\begin{pmatrix} 45 & 48 & ? & 20 & ? & ? & 32 \\ 42 & 41 & 47 & 30 & 36 & 34 & 37 \\ 46 & 43 & 44 & 40 & 38 & 33 & 35 \\ 21 & 22 & 23 & 24 & 25 & 26 & 49 \\ 2 & 3 & 9 & 27 & 13 & 12 & 16 \\ 4 & 1 & 8 & 28 & 11 & 15 & 19 \\ 5 & 6 & 7 & 29 & 14 & 17 & 18 \end{pmatrix}$$

Solution:

Consider the matrix given below

$$\begin{pmatrix} 45 & 48 & 31 & 20 & 10 & 39 & 32 \\ 42 & 41 & 47 & 30 & 36 & 34 & 37 \\ 46 & 43 & 44 & 40 & 38 & 33 & 35 \\ 21 & 22 & 23 & 24 & 25 & 26 & 49 \\ 2 & 3 & 9 & 27 & 13 & 12 & 16 \\ 4 & 1 & 8 & 28 & 11 & 15 & 19 \\ 5 & 6 & 7 & 29 & 14 & 17 & 18 \end{pmatrix}$$

The minimum element of the middle row 4 and the middle column 4 is 20. But this is not the local minima. The smallest neighbor of 20 is 10. Therefore, we start to recurse in the top left quadrant of A of size (7-1)/2 i.e

$$\begin{pmatrix} 10 & 39 & 32 \\ 36 & 34 & 37 \\ 38 & 33 & 35 \end{pmatrix}$$

The minimum element of the middle row 2 and the middle column 2 is 33. Also it is a local minimum, so by the algorithm we can conclude that the local minimum of the given matrix is 33. However, we can see that 33 is not a local minimum as the neighbor element 26 is smaller than 33. Therefore the given algorithm is not correct in general □

(d) (8 points) For simplicity of calculation, assume that $n = 2^k + 1$ for some integer $k \geq 0$ and all the numbers $A[i, j]$ are distinct. Consider a slightly more complex divide-and-conquer algorithm. In addition to the middle row $(2^{k-1} - 1)$ and column $(2^{k-1} - 1)$, also look at the boundary nodes $\{(1, i), (n, i), (i, 1), (i, n) \mid i = 1 \ldots n\}$, and find the global minimum a (located at node v of all these $6n - 9$ numbers (3 rows and 3 columns of size n, except 9 "intersection points" are counted twice) in $O(n)$ time. If v is a local minimum, output v. Else take the smallest neighbor of v (call it w), and recurse in the quadrant (north-east, north-west, south-east or south-west) of A of size $(n + 1)/2 = 2^{k-1} + 1$ (where the quadrant *includes the boundary*, so you effectively duplicate the middle row/colum before dividing by 2) where w lies.

Prove the correctness of this modified algorithm. To do that, prove the following stronger inductive statement: the algorithm computes an answer which is (a) correct local minimum (i, j) *and* (b) $A[i, j]$ is less or equal than every number on the boundary of the square. Make sure you stress where (i) you use the fact that a *smaller* neighbor w is selected; (ii) that all the numbers are *distinct*.

Solution:

-	-	-	-	-	-	-	-	-
-	*	*	*	*	*	*	*	-
-	*			p	<p		*	-
-	*			-			*	-
-	-	-	-	-	-	-	-	-
-	*			-			*	-
-	*			-			*	-
-	*	*	*	-	*	*	*	-
-	-	-	-	-	-	-	-	-

Notations Used

- *Frame* refers to the - - - - part
- *Quadrant* refers to the matrix formed by the the enclosing frame
- *Buundary* refers to the * * * * part
- *Sub-matirx* refers to the matrix formed by the enclosing boundary

Theorem 1. *If we start recursion on any quadrant then there is a local minimum in that quadrant*

Proof. Let p be the smallest element in the frame. Therefore, the quadrant selected now becomes

>p	>p	>p	>p	>p
>p	*	>p	*	>p
p	<p	>p	-	>p
>p	*	>p	*	>p
>p	>p	>p	>p	>p

Assuming that all the elements in the matrix are distinct from each other, the smallest element in this quadrant must be less than p, let it be q. Also, we know that all the elements in the frame are greater than p. Therefore, if the smallest element q of this sub-matrix falls on the boundary, it is a local minima as it is guaranteed that q is lesser than all it's neighbors which lie on the frame (as $q < p$) and obviously it is lesser than all it's neighbors which are in the sub-matrix. If q falls inside of the sub-matrix i.e not on the boundary, then clearly q is lesser than all of it's neighbors. Therefore a local minima exists in the quadrant in which the recursion is called □

Theorem 2. *We will always find the local minima in the quadrant*

Proof. In case, we don't find a local minima in the frame, then we recurse down to the next smaller quadrant. If we don't succeed to find a local minima on the frame on this smaller quadrant then we recurse down to the next smaller quadrant. This way we keep recursing down to the next smaller quadrant until we find a local minima. In the worst case, we get down to the smallest quadrant such that the frame and boundary spans the entire matrix (i.e a 3x3 matrix). Now by *Theorem 1*, there is a local minima in this quadrant since we recursed down to it and as we are observing the entire matrix we will definitely find a local minima in this quadrant. Thus we will always find a local minima in the quadrant using the above algorithm. □

Therefore by *Theorem 1* and *Theorem 2*, There always exists a local minima in the quadrant we recursed down to and we are guaranteed to find this local minima using the given algorithm. This proves the correctness of our algorithm □

(e) (2 points) Write the recurrence equation and solve it to compute the running time of the algorithm in part (d).

Solution: The time taken to find the minimum element among the $6n-9$ elements is $O(n)$. Then we recurse down to smaller matrix or quadrant of size $(n/2Xn/2)$. Therefore time takes is

$$\begin{aligned} T(n) &= T(n/2) + O(n) \\ By\ Master\ Theorem,\ a = 1, b = 2, d = 1 &\implies d > \log_b a \\ \implies T(n) &= O(n^d) \\ \therefore T(n) &= O(n) \end{aligned}$$

□