Midterm 2 Review

CPSC 320 | 2024W1

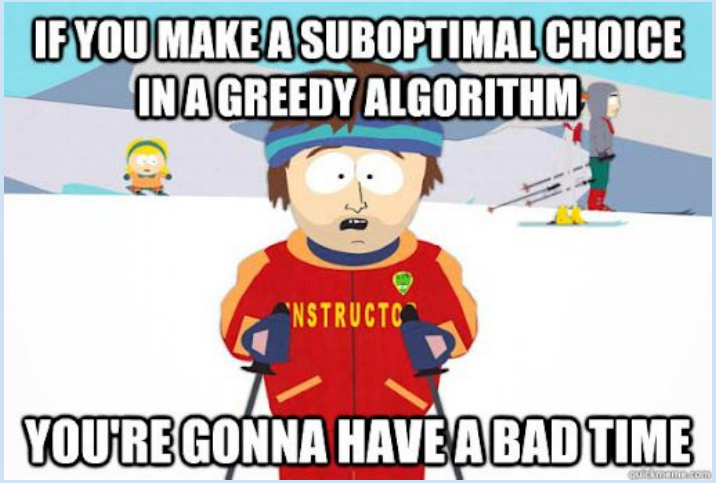
References

Content/Problem/Solutions adapted from:

- Course Material
- Stanford CS161 Handouts: Guide to Greedy Algorithms, Recurrences
- Berkeley Handout: Berkeley Divide & Conquer Algorithms
- CS 482 Greedy Stays Ahead Handout

Part I

Greedy Algorithms

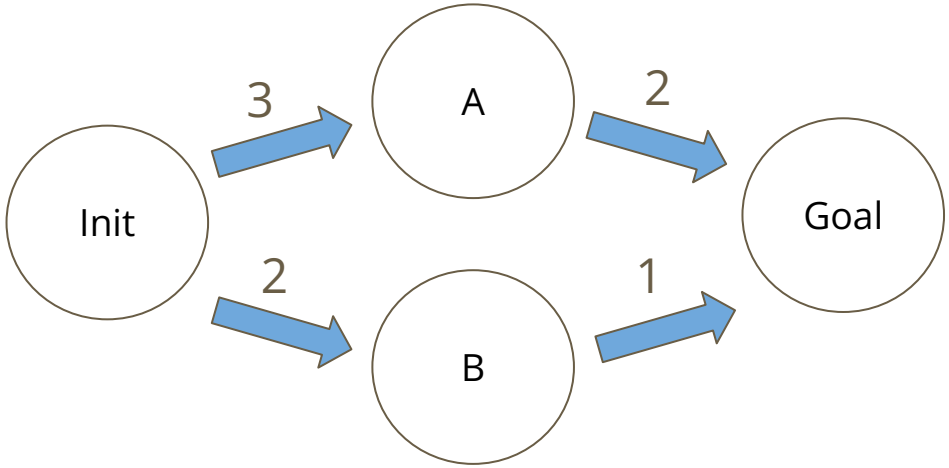


Greedy Algorithms

Key Idea: algorithm that makes locally optimal choices at each step

- best immediate decision that minimizes/maximizes a certain metric.

Optimality Condition: a globally optimal solution can be reached by making a sequence of locally optimal choices.



Approach: always selecting lowest cost edge.

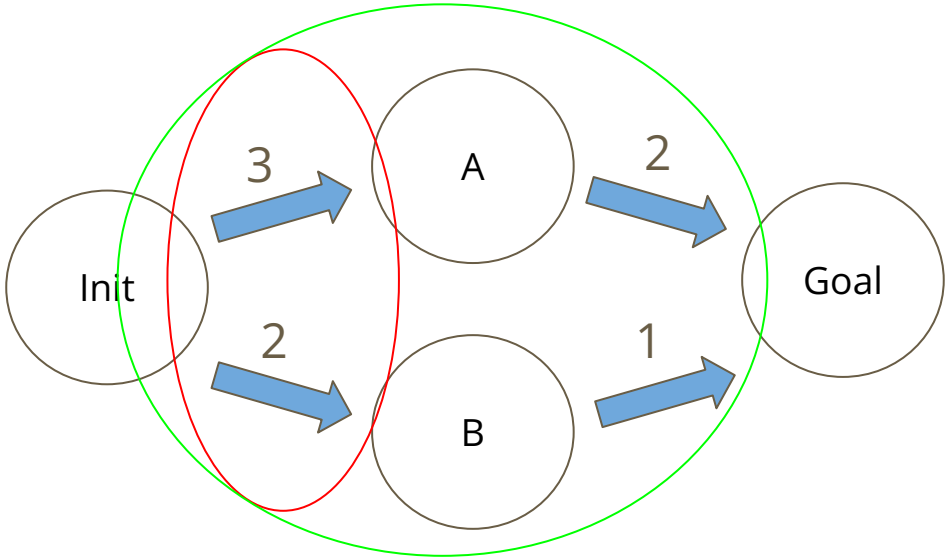
Does the greedy algorithm lead to the globally optimal path in this graph?

Greedy Algorithms

Key Idea: at each step, choose the best option available without considering previous or future choices.

- best immediate decision that minimizes/maximizes a certain metric.

Optimality Condition:



Locally Optimum:
2 < 3 (so go to B)

(leads to)

Globally Optimum:
2+1<3+2
found shortest path!

Proving Optimality

Greedy Stays Ahead:

- Define the partial solutions.
- Define the stay-ahead measure (greedy-stays-ahead lemma).
- Prove that greedy stays ahead.
- Prove optimality.

Exchange Argument:

- Define the solutions.
- Compare solutions.
- Exchange moves.
- Iterate.

*bulk of work

Greedy-Stays-Ahead

According to some measure, the greedy algorithm always is at least as far ahead as the optimal solution during each iteration of the algorithm.

- Define the partial solutions. The greedy will produce some partial solution G(i) that you will probably compare against some optimal solution O(i).
- Define the stay-ahead measure (greedy-stays-ahead lemma). Find the measure you will use to compare G(i) and O(i). This is often the metric the algorithm is being greedy about.
- Prove that greedy stays ahead. Use your measure and show that using it, greedy's partial solution never falls behind of the optimal solution (**induction**)
- Prove optimality. Using the fact that greedy stays ahead, prove that the greedy algorithm must produce an optimal solution (*often done by contradiction by assuming the greedy solution isn't optimal and using the fact that greedy stays ahead to derive a contradiction*)

Exchange Argument

Work by showing that you can iteratively transform any optimal solution into the solution produced by the greedy algorithm without increasing the cost of the solution.

- Define the solutions. You are comparing a greedy solution G to an optimal solution O, so it's best to define these variables explicitly.
- Compare solutions. Next, show that if G ≠ O, then they must differ in some way. Define this point of departure to use in the rest of the proof.
- Exchange moves. Show how to transform O by exchanging some move by O for some move by G. Then, prove that in by doing so, you did not increase the cost of G.
- Iterate. Argue that you have decreased the number of differences between G and O by performing the exchange, and that by iterating this exchange you can turn O into G without impacting the quality of the solution. Therefore, X must be optimal. *(To be very rigorous here, we should prove this by induction, but for the purposes of this class it is okay to just informally explain that iteratively proceeding works.)*

Motivating Example

Interval Scheduling: suppose you have set of *n* requests *{1,2, ..., n}* and each request *i* has a desired start and finish time (s_i, f_i). Want to find schedule with maximum number of non-overlapping requests.

Interval Scheduling

Give a (brief) description of the optimal greedy algorithm

Interval Scheduling (Solutions)

Give a (brief) description of the optimal greedy algorithm

Repeatedly select remaining requests with earlier finish time and removing all conflicting requests from the set.

Interval Scheduling

Prove that algo is optimal using **greedy-stays-ahead**

Interval Scheduling (Solutions)

Prove that algo is optimal using **greedy-stays-ahead**

Let $A = \{i_1, \ldots, i_k\}$ be the requests selected by our greedy algorithm, in the order in which they were added. Let $O = \{j_1, \ldots, j_m\}$ be an optimal solution, ordered by finish times.

Let $f(S)$ be the last finish time of the jobs in a set S. Note that by our orderings of A and O, $f(i_1, \ldots, i_r) = f(i_r) = f_{i_r}$ and $f(j_1, \ldots, j_r) = f(j_r) = f_{j_r}$, so we will use these interchangeably in this example. Now our goal is to show that for all $r \leq k$, $f(i_r) \leq f(j_r)$.

Solutions from Alexa Sharp, Tom Wexler (CS 482)

Define partial solutions + Define the stay-ahead measure

Interval Scheduling (Solutions)

Prove that algo is optimal using **greedy-stays-ahead**

This can be shown by induction. As the base case, we take $r = 1$. Since we selected the job with the earliest finish time, it certainly must be the case that $f(i_1) \leq f(j_1)$.

For $t > 1$, assume the statement is true for $t - 1$ and we will prove it for t. The induction hypothesis states that $f(i_{t-1}) \leq f(j_{t-1})$, and so any jobs that are valid to add to the optimal solution are certainly valid to add to our greedy solution. Therefore, it must be the case that $f(i_t) \leq f(j_t)$.

Prove that greedy stays ahead.

So we have that for all $r \leq k$, $f(i_r) \leq f(j_r)$. In particular, $f(i_k) \leq f(j_k)$. If A is not optimal, then it must be the case that $m > k$, and so there is a job j_{k+1} in O that is not in A. This job must start after O's k^{th} job finishes at $f(j_k)$ (i.e. $s_{j_{k+1}} > f(j_k)$), and hence after $f(i_k)$, by our previous result. But then this job would have been compatible with all the jobs in A, and so our greedy algorithm would have added j_{k+1} to A. This is a contradiction, and thus A is optimal.

Prove optimality

Solutions from Alexa Sharp, Tom Wexler (CS 482)

Interval Scheduling

Prove that algo is optimal using **exchange argument**

(more challenging)

Interval Scheduling (Solutions)

Prove that algo is optimal using **exchange argument**

Here, this exchange argument follows a bit more of an inductive style (but non-inductive argument is very similar)

Proof. Every optimal solution contains the empty set and thus the claim holds for the base case $i = 0, S_0 = \emptyset$. Now suppose S_i can be extended to an optimal solution $\mathcal{O}$. It remains to show that S_{i+1} can also be extended to some optimal solution $\mathcal{O}^*$, not necessarily the same as $\mathcal{O}$. To prove this, we use an **exchange argument**.

Defining the solutions

Interval Scheduling (Solutions)

Prove that algo is optimal using **exchange argument**

Compare solutions

There are two cases to consider. At step $i+1$, either no interval was added to S_i, in which case S_{i+1} can be extended to $\mathcal{O}$, since $S_i = S_{i+1}$. Otherwise, let I be the interval added to S_i and let J be the first interval in $\mathcal{O}$ but not in S_i. If $I = J$ then we're done. Suppose then $I \neq J$. Since the algorithm selected I at step i, I must have the earliest finish time for all the remaining intervals in $\mathcal{I}$. In particular

$$f_I < f_J \quad [1] \tag{1}$$

Clearly I and J are compatible with every interval in S_i. Since $J \in \mathcal{O}$, J is compatible with every interval J' in $\mathcal{O}$ that comes after J. In particular

$$f_J < f_{J'} \tag{2}$$

Using (1) and (2), it follows that I is compatible with every interval J' in $\mathcal{O}$ that comes after J. So we can simply **exchange** J with I in $\mathcal{O}$, and obtain $\mathcal{O}^* = \mathcal{O} \cup \{I\} \backslash \{J\}$ where $\mathcal{O}^*$ is an optimal solution that extends S_{i+1}. □

Exchange

Part II

Divide and Conquer

Divide and Conquer Strategies

The ***divide-and-conquer*** strategy solves a problem by:

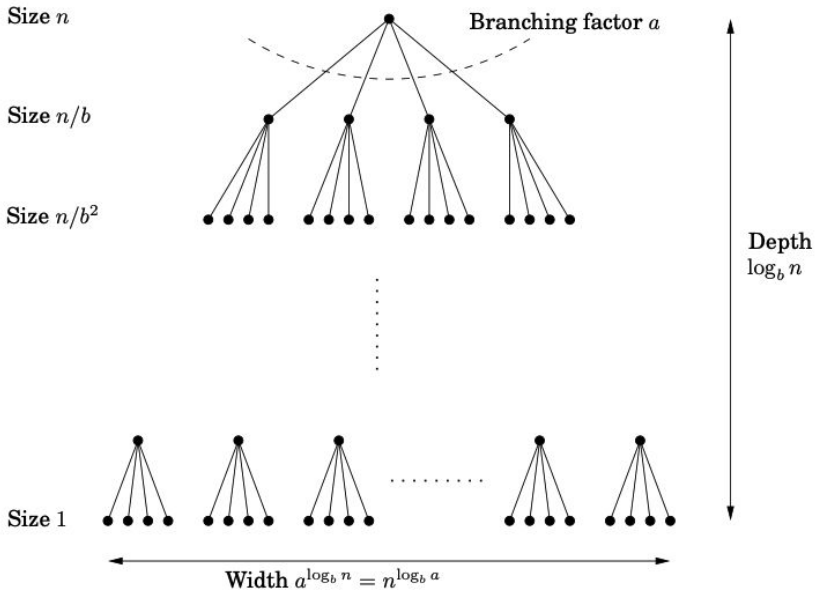
1. Breaking it into *subproblems* that are themselves smaller instances of the same type of problem
2. Recursively solving these subproblems
3. Appropriately combining their answers

Prune-and-search: *recursion covers only a constant fraction of the input set (eliminating branches that cannot lead to a valid solution)*

Master Theorem

Master theorem[2] *If* $T(n) = aT(\lceil n/b \rceil) + O(n^k)$ *for some constants* $a > 0$, $b > 1$, *and* $d \geq 0$,

Figure 2.3 Each problem of size n is divided into a subproblems of size n/b.



Note: you are also expected to be able to work through recursions that don't follow Master Theorem format (using recursion tree, intuition, etc.)

Master Theorem

Theorem: Let $T : \mathbb{N} \to \mathbb{R}_{\geq 0}$ be defined by

$$T(n) = \begin{cases} aT(n/b) + cn^k, & \text{for } n \geq n_0, \\ c, & \text{for } n < n_0, \end{cases}$$

where $a > 0$, $b > 1$, $c > 0$, and $k \geq 0$ are constants.

- If $a > b^k$, then $T(n) = \Theta(n^{\log_b a})$.
- If $a = b^k$, then $T(n) = \Theta(n^k \log n)$.
- If $a < b^k$, then $T(n) = \Theta(n^k)$.

This version will be sufficient for assignment problems and exams in this course.

*notice d is used instead of k

Master Theorem (Optional Proof)

Master theorem[2] *If* $T(n) = aT(\lceil n/b \rceil) + O(n^d)$ *for some constants* $a > 0$, $b > 1$, *and* $d \geq 0$, *then*

$$T(n) = \begin{cases} O(n^d) & \text{if } d > \log_b a \\ O(n^d \log n) & \text{if } d = \log_b a \\ O(n^{\log_b a}) & \text{if } d < \log_b a \,. \end{cases}$$

*notice d is used instead of k

Master Theorem (Optional Proof)

Proof. To prove the claim, let's start by assuming for the sake of convenience that n is a power of b. This will not influence the final bound in any important way—after all, n is at most a multiplicative factor of b away from some power of b [redacted]—and it will allow us to ignore the rounding effect in $\lceil n/b \rceil$.

Next, notice that the size of the subproblems decreases by a factor of b with each level of recursion, and therefore reaches the base case after $\log_b n$ levels. This is the height of the recursion tree. Its branching factor is a, so the kth level of the tree is made up of a^k subproblems, each of size n/b^k (Figure 2.3). The total work done at this level is

$$a^k \times O\left(\frac{n}{b^k}\right)^d = O(n^d) \times \left(\frac{a}{b^d}\right)^k.$$

*notice d is used instead of k

Master Theorem (Optional Proof)

As k goes from 0 (the root) to $\log_b n$ (the leaves), these numbers form a geometric series with ratio a/b^d. Finding the sum of such a series in big-O notation is easy [redacted], and comes down to three cases.

1. *The ratio is less than* 1.

   Then the series is decreasing, and its sum is just given by its first term, $O(n^d)$.

2. *The ratio is greater than* 1.

   The series is increasing and its sum is given by its last term, $O(n^{\log_b a})$:

$$n^d \left(\frac{a}{b^d}\right)^{\log_b n} = n^d \left(\frac{a^{\log_b n}}{(b^{\log_b n})^d}\right) = a^{\log_b n} = a^{(\log_a n)(\log_b a)} = n^{\log_b a}.$$

3. *The ratio is exactly* 1.

   In this case all $O(\log n)$ terms of the series are equal to $O(n^d)$.

These cases translate directly into the three contingencies in the theorem statement. ∎

Recursion Tree

Consider the recurrence $T(n) = T(n/3) + T(2n/3) + O(n)$. Let c represent the constant factor in $O(n)$ term.

1. Draw the recursion tree for this recurrence.
2. Find the length of the longest path from the root to a leaf node.
3. Determine an upper bound on the cost at each level of the tree.
4. What is an upper bound on the total runtime for the entire recursion tree?

Recursion Tree (Solutions)

1. Draw the recursion tree for this recurrence.

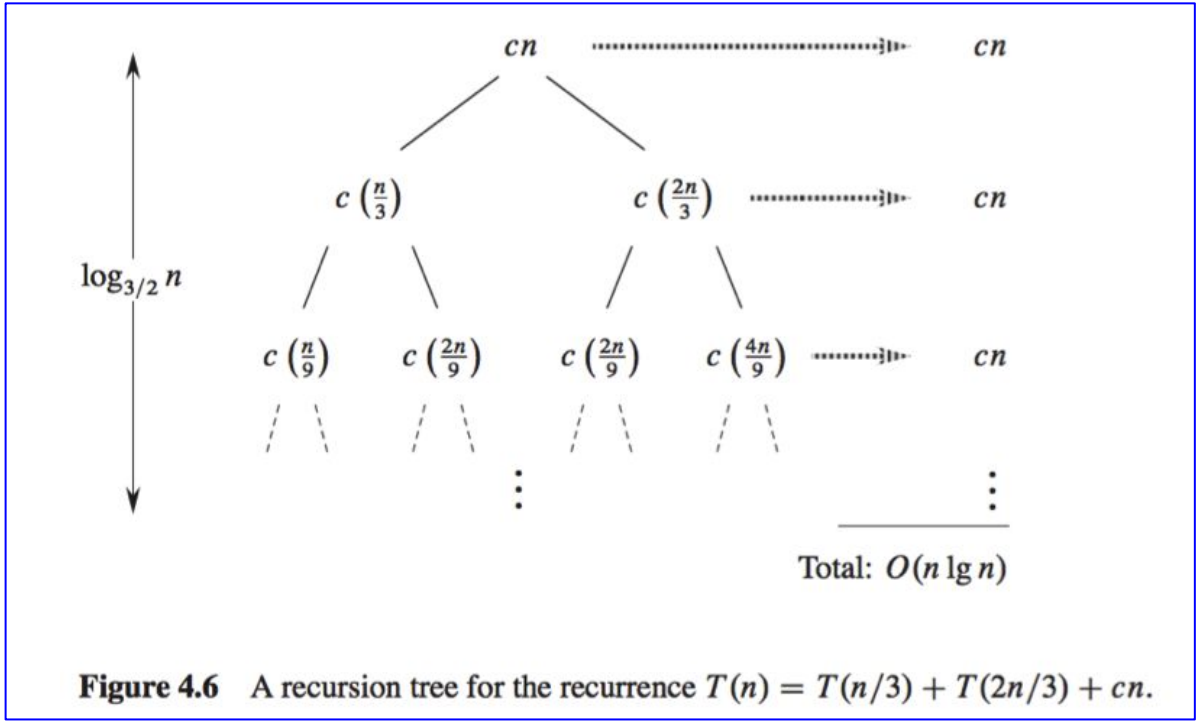


Figure 4.6 A recursion tree for the recurrence $T(n) = T(n/3) + T(2n/3) + cn$.

Recursion Tree (Solutions)

2. Find the length of the longest path from the root to a leaf node.

The longest simple path from the root to a leaf is $n \to (2/3)n \to (2/3)^2 n \to \cdots \to 1$. Since $(2/3)^k n = 1$ when $k = \log_{3/2} n$, the height of the tree is $\log_{3/2} n$.

Recursion Tree (Solutions)

3. Determine an upper bound on the cost at each level of the tree.

We get that each level costs at most cn, but as we go down from the root, more and more internal nodes are absent, so the costs become smaller. Fortunately we only care about an

Recursion Tree (Solutions)

4. What is an upper bound on the total runtime for the entire recursion tree?

O(nlogn)

Binary Tree Problem

Consider the **Binary Tree Search** algorithm.

1. Write the recurrence relation for the runtime of Binary Search.
2. Using the Master Theorem, determine the asymptotic runtime of Binary Search.

Binary Tree Problem (Solutions)

Consider the **Binary Tree Search** algorithm.

1. Write the recurrence relation for the runtime of Binary Search.

$$T(n) = 1T(n/2) + O(1)$$

Binary Tree Problem (Solutions)

Consider the **Binary Tree Search** algorithm.

2. Using the Master Theorem, determine the asymptotic runtime of Binary Search.

Which case of the master theorem are we using?

Final answer is O(logn).

Maximum Subarray Problem *(more challenging)*

Now we will present another divide and conquer algorithm.

Suppose you are given the price of a stock on each day, and you have to decide when to buy and when to sell to maximize your profit. Note that you cannot sell before you buy (so you can't just sell on the highest day and buy on the lowest day).

Naive strategy: Try all pairs of (buy, sell) dates, where the buy date must be before the sell date. This takes $\Theta(n^2)$ time.

```
bestProfit = -MAX_INT
bestBuyDate = None
bestSellDate = None
for i = 1 to n:
    for j = i + 1 to n:
        if price[j] - price[i] > bestProfit:
            bestBuyDate = i
            bestSellDate = j
            bestProfit = price[j] - price[i]
return (bestBuyDate, bestSellDate)
```

Maximum Subarray Problem *(more challenging)*

Instead of the daily price, consider the daily change in price, which (on each day) can be either a positive or negative number. Let array A store these changes. Now we have to find the subarray of A that maximizes the sum of the numbers in that subarray.

Now divide the array into two. Any maximum subarray must either be entirely in the first half, entirely in the second half, or it must span the border between the first and the second half. If the maximum subarray is entirely in the first half (or the second half), we can find it using a recursive call on a subproblem half as large.

If the maximum subarray spans the border, then the sum of that array is the sum of two parts: the part between the buy date and the border, and the part between the border and the sell date. To maximize the sum of the array, we must maximize the sum of each part.

We can do this by simply (1) iterating over all possible buy dates to maximize the first part (2) iterating over all possible sell dates to maximize the second part. Note that this takes linear time instead of quadratic time, because we no longer have to iterate over buy and sell dates simultaneously.

Maximum Subarray Problem *(more challenging)*

We have provided you with the following helper:

```
FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
 1  left-sum = −∞
 2  sum = 0
 3  for i = mid downto low
 4       sum = sum + A[i]
 5       if sum > left-sum
 6            left-sum = sum
 7            max-left = i
 8  right-sum = −∞
 9  sum = 0
10  for j = mid + 1 to high
11       sum = sum + A[j]
12       if sum > right-sum
13            right-sum = sum
14            max-right = j
15  return (max-left, max-right, left-sum + right-sum)
```

Maximum Subarray Problem *(more challenging)*

1. Write out the pseudocode for

   Find-Maximum-Subarray(A, low, high)
2. Provide an upper bound on its runtime.

(more challenging)

Maximum Subarray Problem (Solutions)

1. Write out the pseudocode for

   Find-Maximum-Subarray(A, low, high)

```
FIND-MAXIMUM-SUBARRAY(A, low, high)
 1  if high == low
 2      return (low, high, A[low])          // base case: only one element
 3  else mid = ⌊(low + high)/2⌋
 4      (left-low, left-high, left-sum) =
            FIND-MAXIMUM-SUBARRAY(A, low, mid)
 5      (right-low, right-high, right-sum) =
            FIND-MAXIMUM-SUBARRAY(A, mid + 1, high)
 6      (cross-low, cross-high, cross-sum) =
            FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
 7      if left-sum ≥ right-sum and left-sum ≥ cross-sum
 8          return (left-low, left-high, left-sum)
 9      elseif right-sum ≥ left-sum and right-sum ≥ cross-sum
10          return (right-low, right-high, right-sum)
11      else return (cross-low, cross-high, cross-sum)
```

(more challenging)

Maximum Subarray Problem (Solutions)

2. Provide an upper bound on its runtime.

Note that we are omitting the correctness proof, because the main point is to give an example of the divide and conquer strategy. In the homework, you would normally need to provide a correctness proof, unless we say otherwise.

First we analyze the runtime of FindMaxCrossingSubarray. Since each iteration of each of the two for loops takes $\Theta(1)$ time, we just need to count up how many iterations there are altogether. The for loop of lines 3-7 makes $mid - low + 1$ iterations, and the for loop of lines 10-14 makes $high - mid$ iterations, so the total number of iterations is $high - low + 1 = n$. Therefore, the helper function takes $\Theta(n)$ time.

Now we proceed to analyze the runtime of the main function.

For the base case, $T(1) = \Theta(1)$, since line 2 takes constant time.

For the recursive case, lines 1 and 3 take constant time. Lines 4 and 5 take $T(\lfloor n/2 \rfloor)$ and $T(\lceil n/2 \rceil)$ time, since each of the subproblems has that many elements. The FindMaxCrossingSubarray procedure takes $\Theta(n)$ time, and the rest of the code takes $\Theta(1)$ time. So $T(n) = \Theta(1) + T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + \Theta(n) + \Theta(1) = 2T(n/2) + \Theta(n)$ (ignoring the floors and ceilings).

By case 2 of the master theorem, this recurrence has the solution $T(n) = \Theta(n \log n)$.