CS 7B - Spring 2014 - Assignment 4. 4/17/14

Write responses on paper and submit programs by email.

1 Recursion and the divide-and-conquer approach

Many useful algorithms are ***recursive*** in structure: to solve a given problem, they call themselves recursively one or more times to deal with closely related sub-problems. These algorithms typically follow a divide-and-conquer approach: they break the problem into several subproblems that are similar to the original problem but smaller in size, solve the subproblems recursively, and then combine these solutions to create a solution to the original problem.

The divide-and-conquer paradigm involves three steps at each level of the recursion:
Divide the problem into a number of subproblems that are smaller instances of the same problem.
Conquer the subproblems by solving them recursively. If the subproblem sizes are small enough, however, just solve the subproblems in a straightforward manner.
Combine the solutions to the subproblems into the solution for the original problem.

The ***merge sort*** algorithm closely follows the divide-and-conquer paradigm. Intuitively, it operates as follows.
Divide: Divide the n-element sequence to be sorted into two subsequences of $n/2$ elements each.
Conquer: Sort the two subsequences recursively using merge sort.
Combine: Merge the two sorted subsequences to produce the sorted answer.

The recursion "bottoms out" when the sequence to be sorted has length 1, in which case there is no work to be done, since every sequence of length 1 is already in sorted order.

The key operation of the merge sort algorithm is the merging of two sorted sequences in the "combine" step. We merge by calling an auxiliary procedure $\textsc{Merge}(A, p, q, r)$, where A is an array and p, q, and r are indices into the array such that $p \leq q < r$. The procedure assumes that the subarrays $A[p \ldots q]$ and $A[q+1 \ldots r]$ are in sorted order. It merges them to form a single sorted subarray that replaces the current subarray $A[p \ldots r]$.

Our MERGE procedure takes time $\Theta(n)$, where $n = r - p + 1$ is the total number of elements being merged, and it works as follows. Think of two piles of cards face up on a table. Each pile is sorted, with the smallest cards on top. We wish to merge the two piles into a single sorted output pile, which is to be face down on the table. Our basic step consists of choosing the smaller of the two cards on top of the face-up piles, removing it from its pile (which exposes a new top card), and placing this card face down onto the output pile. We repeat this step until one input pile is empty, at which time we just take the remaining input pile and place it face down onto the output pile. Computationally, each basic step takes constant time, since we are comparing just the two top cards. Since we perform at most n basic steps, merging takes $\Theta(n)$ time.

The following pseudocode implements the above idea, but with an additional twist that avoids having to check whether either pile is empty in each basic step. We place on the bottom of each pile a sentinel card, which contains a special value that we use to simplify our code. Here, we use 1

as the sentinel value, so that whenever a card with 1 is exposed, it cannot be the smaller card unless both piles have their sentinel cards exposed. But once that happens, all the nonsentinel cards have already been placed onto the output pile. Since we know in advance that exactly $r - p + 1$ cards will be placed onto the output pile, we can stop once we have performed that many basic steps.

```
MERGE(A, p, q, r)
1   n1 = q - p + 1
2   n2 = r - q
3   let L[1 .. n1 + 1] and R[1 .. n2 + 1] be new arrays
4   for i = 1 to n1
5       L[i] = A[p + i - 1]
6   for j = 1 to n2
7       R[j] = A[q + j]
8   L[n1 + 1] = ∞
9   R[n2 + 1] = ∞
10  i = 1
11  j = 1
12  for k = p to r
13      if L[i] ≤ R[j]
14          A[k] = L[i]
15          i = i + 1
16      else A[k] = R[j]
17          j = j + 1
```

In detail, the MERGE procedure works as follows. Line 1 computes the length n_1 of the subarray $A[p \dots q]$, and line 2 computes the length n_2 of the subarray $A[q+1 \dots r]$. We create arrays L and R ("left" and "right"), of lengths $n_1 + 1$ and $n_2 + 1$, respectively, in line 3; the extra position in each array will hold the sentinel. The for loop of lines 45 copies the subarray $A[p \dots q]$ into $L[1 \dots n_1]$, and the for loop of lines 6-7 copies the subarray $A[q+1 \dots r]$ into $R[1 \dots n_2]$. Lines 8-9 put the sentinels at the ends of the arrays L and R. Lines 10-17 perform the $r - p + 1$ basic steps by maintaining the following loop invariant:

> At the start of each iteration of the **for** loop of lines 12-17, the subarray $A[p \dots k-1]$ contains the $k - p$ smallest elements of $L[1 \dots n_1 + 1]$ and $R[1 \dots n_2 + 1]$, in sorted order. Moreover, $L[i]$ and $R[j]$ are the smallest elements of their arrays that have not been copied back into A.

In computer science, we prove an algorithm works by showing that an invariant, in this case the loop invariant, holds *prior to the first iteration* of the **for** loop of lines 12-17, that each iteration of the loop *maintains the invariant*, and that the invariant provides a useful property to show *correctness when the loop terminates.*

Initialization: Prior to the first iteration of the loop, we have $k = p$, so that the subarray $A[p \dots k-1]$ is empty. This empty subarray contains the $k - p = 0$ smallest elements of L and R,

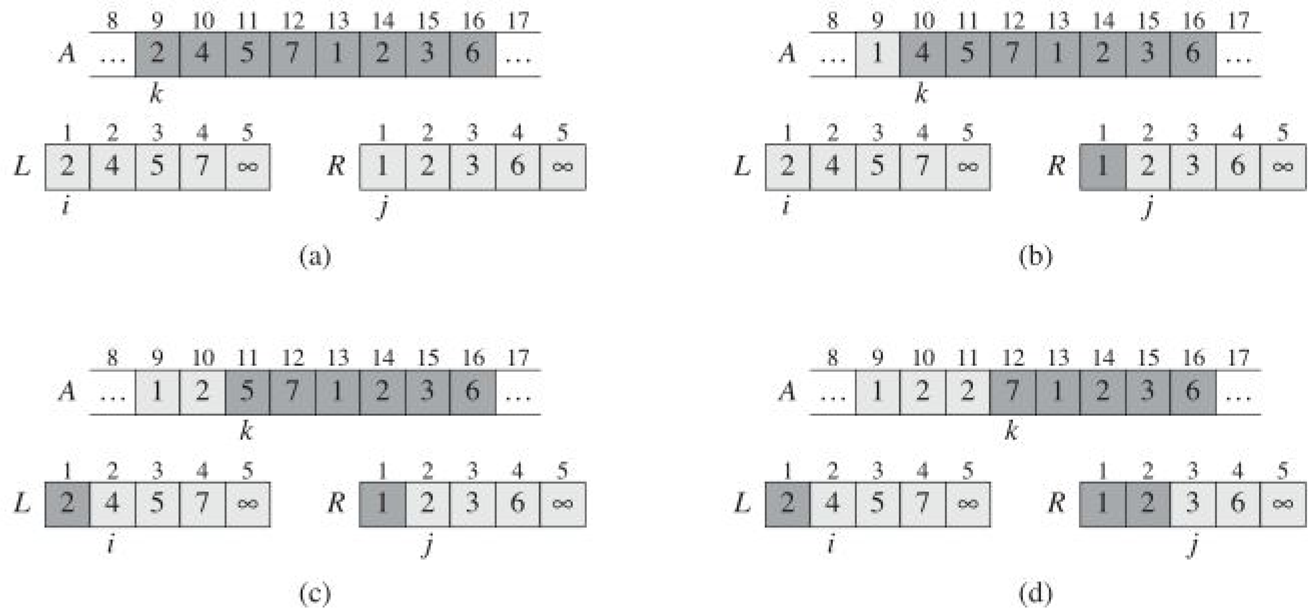


Figure 1: The operation of lines 10-17 in the call MERGE$(A, 9, 12, 16)$, when the subarray $A[9 \dots 16]$ contains the sequence $\langle 2, 4, 5, 7, 1, 2, 3, 6 \rangle$. After copying and inserting sentinels, the array L contains $\langle 2, 4, 5, 7, 1 \rangle$, and the array R contains $\langle 1, 2, 3, 6, 1 \rangle$. Lightly shaded positions in A contain their final values, and lightly shaded positions in L and R contain values that have yet to be copied back into A. Taken together, the lightly shaded positions always comprise the values originally in $A[9 \dots 16]$, along with the two sentinels. Heavily shaded positions in A contain values that will be copied over, and heavily shaded positions in L and R contain values that have already been copied back into A. **(a)-(h)** The arrays A, L, and R, and their respective indices k, i, and j prior to each iteration of the loop of lines 12-17.

and since $i = j = 1$, both $L[i]$ and $R[j]$ are the smallest elements of their arrays that have not been copied back into A.
Maintenance: To see that each iteration maintains the loop invariant, let us first suppose that $L[i] \leq R[j]$. Then $L[i]$ is the smallest element not yet copied back into A. Because $A[p \dots k-1]$ contains the $k - p$ smallest elements, after line 14 copies $L[i]$ into $A[k]$, the subarray $A[p \dots k]$ will contain the $k - p + 1$ smallest elements. Incrementing k (in the for loop update) and i (in line 15) reestablishes the loop invariant for the next iteration. If instead $L[i] > R[j]$, then lines 16-17 perform the appropriate action to maintain the loop invariant.

Termination: At termination, $k = r + 1$. By the loop invariant, the subarray $A[p \dots k-1]$, which is $A[p \dots r]$, contains the $k - p = r - p + 1$ smallest elements of $L[1 \dots n_1 + 1]$ and $R[1 \dots n_2 + 1]$, in sorted order. The arrays L and R together contain $n_1 + n_2 + 2 = r - p + 3$ elements. All but the two largest have been copied back into A, and these two largest elements are the sentinels.

To see that the MERGE procedure runs in $\Theta(n)$ time, where $n = r - p + 1$, observe that each of lines 1-3 and 8-11 takes constant time, the for loops of lines 4-7 take $\Theta(n_1 + n_2) = \Theta(n)$ time, and there are n iterations of the **for** loop of lines 12-17, each of which takes constant time.

We can now use the MERGE procedure as a subroutine in the merge sort algorithm. The procedure MERGE-SORT(A, p, r) sorts the elements in the subarray $A[p \dots r]$. If $p \geq r$, the subarray has at most one element and is therefore already sorted. Otherwise, the divide step simply computes

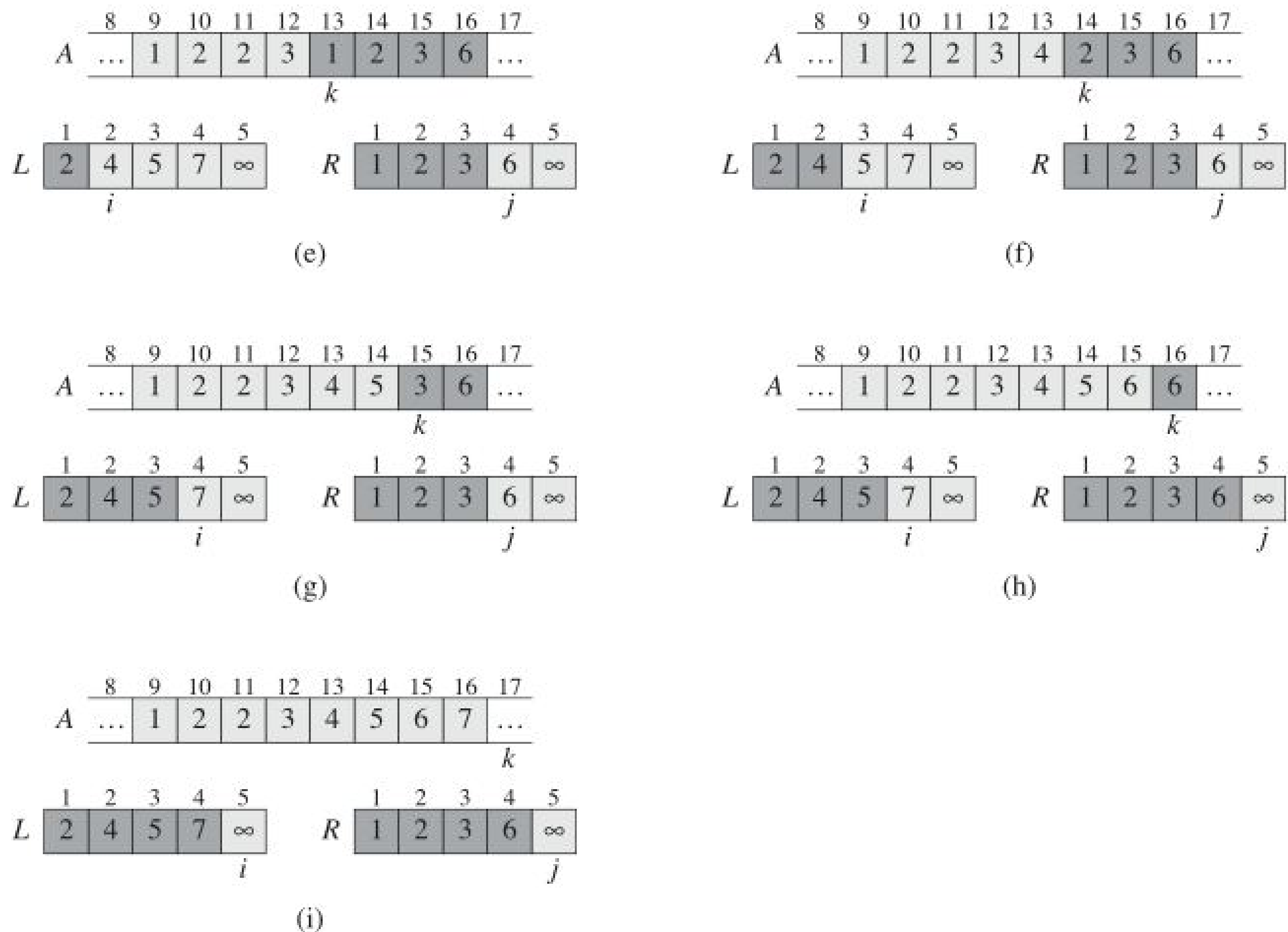


Figure 2: continued (i) The arrays and indices at termination. At this point, the subarray in $A[9\ldots16]$ is sorted, and the two sentinels in L and R are the only two elements in these arrays that have not been copied into A.

an index q that partitions $A[p\ldots r]$ into two subarrays: $A[p\ldots q]$, containing $\lceil n/2\rceil$ elements, and $A[q+1\ldots r]$, containing $\lfloor n/2\rfloor$ elements.

```
MERGE-SORT(A, p, r)
1  if p < r
2      q = ⌊(p + r)/2⌋
3      MERGE-SORT(A, p, q)
4      MERGE-SORT(A, q + 1, r)
5      MERGE(A, p, q, r)
```

To sort the entire sequence $A = \langle A[1], A[2], \ldots, A[n]\rangle$, we make the initial call MERGE-SORT(A, 1, A.`length`), where once again A.`length` $= n$. The figure illustrates the operation of the procedure bottom-up when n is a power of 2. The algorithm consists of merging pairs of 1-item sequences to form sorted sequences of length 2, merging pairs of sequences of length 2 to form sorted sequences of length 4, and so on, until two sequences of length $n/2$ are merged to form the final sorted sequence of length

n.

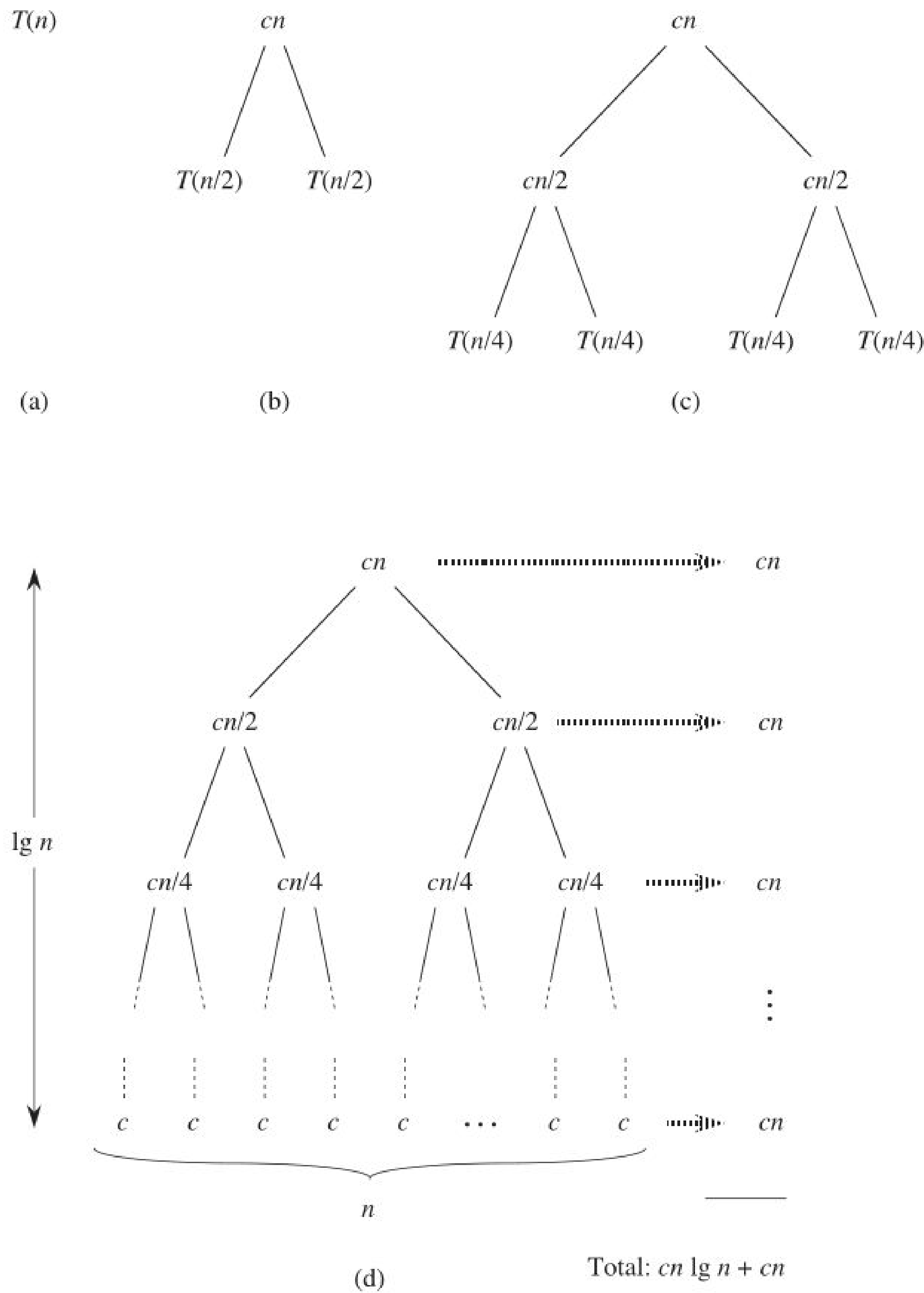
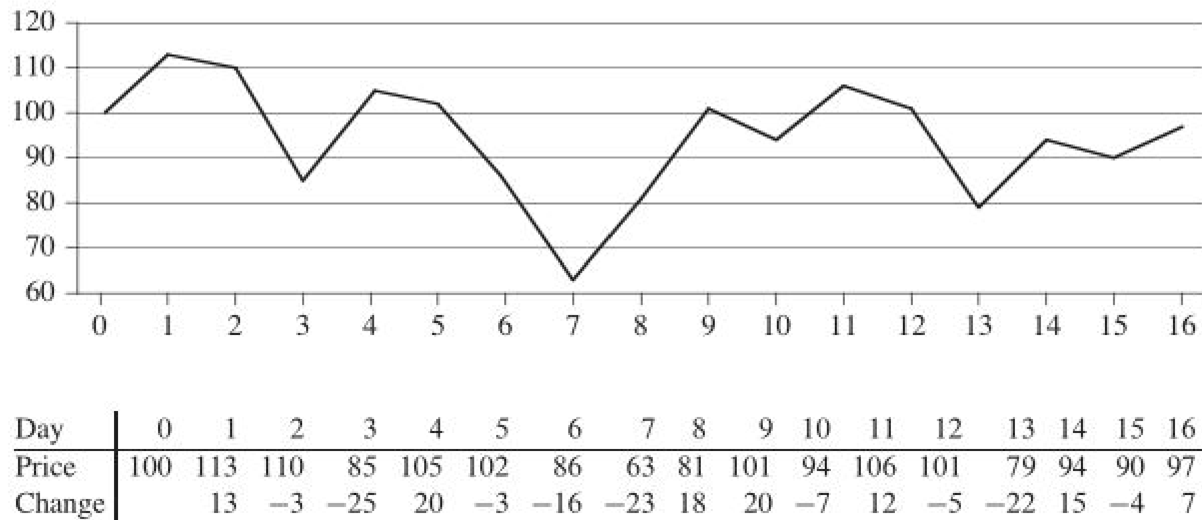


Figure 3: How to construct a recursion tree for the recurrence $T(n) = 2T(n/2) + cn$. Part **(a)** shows $T(n)$, which progressively expands in **(b)(d)** to form the recursion tree. The fully expanded tree in part (d) has $\lg n + 1$ levels (i.e., it has height $\lg n$, as indicated), and each level contributes a total cost of cn. The total cost, therefore, is $cn \lg n + cn$, which is $\Theta(n \lg n)$.

1.1 The maximum-subarray problem

Suppose that you been offered the opportunity to invest in the Eccentric Artist Cooperative. Like the art the company produces, the stock price of the Eccentric Artist Cooperative is rather erratic. You are allowed to buy one unit of stock only one time and then sell it at a later date, buying and selling after the close of trading for the day. To compensate for this restriction, you are allowed to learn what the price of the stock will be in the future. Your goal is to maximize your profit. Figure 4 shows the price of the stock over a 17-day period.



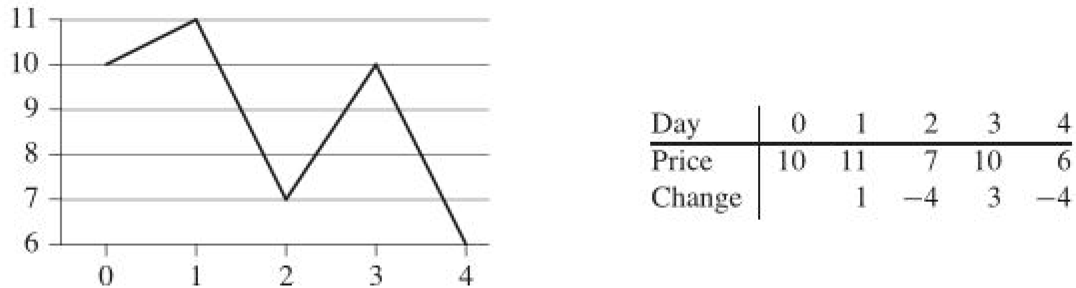
Day	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Price	100	113	110	85	105	102	86	63	81	101	94	106	101	79	94	90	97
Change		13	−3	−25	20	−3	−16	−23	18	20	−7	12	−5	−22	15	−4	7

Figure 4: Information about the price of stock in the Eccentric Artist Cooperative after the close of trading over a period of 17 days. The horizontal axis of the chart indicates the day, and the vertical axis shows the price. The bottom row of the table gives the change in price from the previous day.

You may buy the stock at any one time, starting after day 0, when the price is \$100 per share. Of course, you would want to "buy low, sell high"-buy at the lowest possible price and later on sell at the highest possible price-to maximize your profit. Unfortunately, you might not be able to buy at the lowest price and then sell at the highest price within a given period. In Figure 1, the lowest price occurs after day 7, which occurs after the highest price, after day 1. You might think that you can always maximize profit by either buying at the lowest price or selling at the highest price. For example, in Figure 4, we would maximize profit by buying at the lowest price, after day 7. If this strategy always worked, then it would be easy to determine how to maximize profit: find the highest and lowest prices, and then work left from the highest price to find the lowest prior price, work right from the lowest price to find the highest later price, and take the pair with the greater difference. Figure 5 shows a simple counterexample, demonstrating that the maximum profit sometimes comes neither by buying at the lowest price nor by selling at the highest price.

1.2 A brute-force solution

We can easily devise a brute-force solution to this problem: just try every possible pair of buy and sell dates in which the buy date precedes the sell date. A period of n days has $\binom{n}{2}$ such pairs of dates. Since $\binom{n}{2}$ is $\Theta(n^2)$, and the best we can hope for is to evaluate each pair of dates in constant time, this approach would take $\Omega(n^2)$ time. Can we do better?



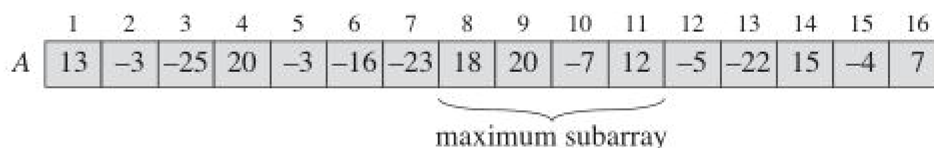
Day	0	1	2	3	4
Price	10	11	7	10	6
Change		1	−4	3	−4

Figure 5: An example showing that the maximum profit does not always start at the lowest price or end at the highest price. Again, the horizontal axis indicates the day, and the vertical axis shows the price. Here, the maximum profit of \$3 per share would be earned by buying after day 2 and selling after day 3. The price of \$7 after day 2 is not the lowest price overall, and the price of \$10 after day 3 is not the highest price overall.

1.3 A transformation

In order to design an algorithm with an $O(n^2)$ running time, we will look at the input in a slightly different way. We want to find a sequence of days over which the net change from the first day to the last is maximum. Instead of looking at the daily prices, let us instead consider the daily change in price, where the change on day i is the difference between the prices after day $i-1$ and after day i. The table in Figure 4 shows these daily changes in the bottom row. If we treat this row as an array A, shown in Figure 6, we now want to find the nonempty, contiguous subarray of A whose values have the largest sum. We call this contiguous subarray the ***maximum subarray***. For example, in the array of Figure 6, the maximum subarray of $A[1..16]$ is $A[8..11]$, with the sum 43. Thus, you would want to buy the stock just before day 8 (that is, after day 7) and sell it after day 11, earning a profit of \$43 per share.

At first glance, this transformation does not help. We still need to check $\binom{n-1}{2} = \Theta(n^2)$ subarrays for a period of n days. In the exercises you will show that although computing the cost of one subarray might take time proportional to the length of the subarray, when computing all $\Theta(n^2)$ subarray sums, we can organize the computation so that each subarray sum takes $O(1)$ time, given the values of previously computed subarray sums, so that the brute-force solution takes $\Theta(n^2)$ time.



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
A	13	−3	−25	20	−3	−16	−23	18	20	−7	12	−5	−22	15	−4	7

Figure 6: The change in stock prices as a maximum-subarray problem. Here, the subarray $A[8..11]$, with sum 43, has the greatest sum of any contiguous subarray of array A.

So let us seek a more efficient solution to the maximum-subarray problem. When doing so, we will usually speak of “a” maximum subarray rather than “the” maximum subarray, since there could be more than one subarray that achieves the maximum sum.

The maximum-subarray problem is interesting only when the array contains some negative numbers.

If all the array entries were nonnegative, then the maximum-subarray problem would present no challenge, since the entire array would give the greatest sum.

1.4 A solution using divide-and-conquer

Let's think about how we might solve the maximum-subarray problem using the divide-and-conquer technique. Suppose we want to find a maximum subarray of the subarray $A[low \dots high]$. Divide-and-conquer suggests that we divide the subarray into two subarrays of as equal size as possible. That is, we find the midpoint, say *mid*, of the subarray, and consider the subarrays $A[low \dots mid]$ and $A[mid+1 \dots high]$. As Figure 7(a) shows, any contiguous subarray $A[i \dots j]$ of $A[low \dots high]$ must lie in exactly one of the following places:

- entirely in the subarray $A[low \dots mid]$, so that $low \le i \le j \le mid$,
- entirely in the subarray $A[mid+1 \dots high]$, so that $mid < i \le j \le high$, or
- crossing the midpoint, so that $low \le i \le mid < j \le high$.

Therefore, a maximum subarray of $A[low \dots high]$ must lie in exactly one of these places. In fact, a maximum subarray of $A[low \dots high]$ must have the greatest sum over all subarrays entirely in $A[low..mid]$, entirely in $A[mid+1 \dots high]$, or crossing the midpoint. We can find maximum subarrays of $A[low \dots mid]$ and $A[mid+1 \dots high]$ recursively, because these two subproblems are smaller instances of the problem of finding a maximum subarray. Thus, all that is left to do is find a

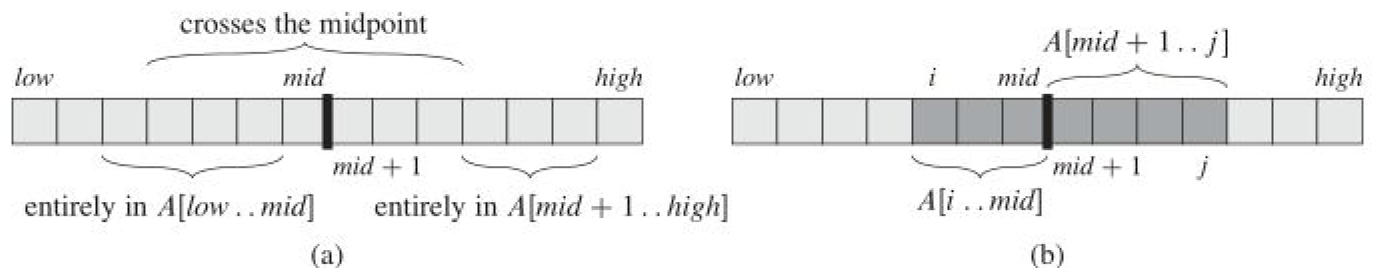


Figure 7: (a) Possible locations of subarrays of $A[low \dots high]$: entirely in $A[low \dots mid]$, entirely in $A[mid+1 \dots high]$, or crossing the midpoint *mid*. (b) Any subarray of $A[low \dots high]$ crossing the midpoint comprises two subarrays $A[i \dots mid]$ and $A[mid+1 \dots j]$, where $low \le i \le mid$ and $mid < j \le high$.

maximum subarray that crosses the midpoint, and take a subarray with the largest sum of the three.

We can easily find a maximum subarray crossing the midpoint in time linear in the size of the subarray $A[low \dots high]$. This problem is not a smaller instance of our original problem, because it has the added restriction that the subarray it chooses must cross the midpoint. As Figure 7(b) shows, any subarray crossing the midpoint is itself made of two subarrays $A[i \dots mid]$ and $A[mid+1 \dots j]$, where $low \le i \le mid$ and $mid < j \le high$. Therefore, we just need to find maximum subarrays of the form $A[i \dots mid]$ and $A[mid+1 \dots j]$ and then combine them. The procedure FIND-MAX-CROSSING-SUBARRAY takes as input the array A and the indices *low*, *mid*, and *high*, and it returns a tuple containing the indices demarcating a maximum subarray that crosses the midpoint, along with the sum of the values in a maximum subarray.

```
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)
```

This procedure works as follows. Lines 1-7 find a maximum subarray of the left half, $A[low \dots i]$. Since this subarray must contain $A[mid]$, the **for** loop of lines 3-7 starts the index i at *mid* and works down to *low*, so that every subarray it considers is of the form $A[i \dots mid]$. Lines 1-2 initialize the variables *left-sum*, which holds the greatest sum found so far, and *sum*, holding the sum of the entries in $A[i \dots mid]$. Whenever we find, in line 5, a subarray $A[i \dots mid]$ with a sum of values greater than *left-sum*, we update *left-sum* to this subarrays sum in line 6, and in line 7 we update the variable *max-left* to record this index i. Lines 8-14 work analogously for the right half, $A[mid + 1 \dots high]$. Here, the **for** loop of lines 10-14 starts the index j at $mid + 1$ and works up to *high*, so that every subarray it considers is of the form $A[mid + 1 \dots j]$. Finally, line 15 returns the indices *max-left* and *max-right* that demarcate a maximum subarray crossing the midpoint, along with the sum $left\text{-}sum + right\text{-}sum$ of the values in the subarray $A[max\text{-}left \dots max\text{-}right]$.

If the subarray $A[low \dots high]$ contains n entries (so that $n = high - low + 1$), we claim that the call FIND-MAX-CROSSING-SUBARRAY$(A, low, mid, high)$ takes $\Theta(n)$ time. 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, and so the total number of iterations is $(mid - low + 1) + (high - mid) = high - low + 1 = n$.

With a linear-time FIND-MAX-CROSSING-SUBARRAY procedure in hand, we can write pseudocode for a divide-and-conquer algorithm to solve the maximum-subarray problem:
The initial call FIND-MAXIMUM-SUBARRAY$(A, 1, A.length)$ will find a maximum subarray of $A[1 \dots n]$. Similar to Find-Max-Crossing-Subarray, the recursive procedure FIND-MAXIMUM-SUBARRAY returns a tuple containing the indices that demarcate a maximum subarray, along with the sum of the values in a maximum subarray. Line 1 tests for the base case, where the subarray has just one element. A subarray with just one element has only one subarray-itself-and so line 2 returns a

```
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)
```

tuple with the starting and ending indices of just the one element, along with its value. Lines 3-11 handle the recursive case. Line 3 does the divide part, computing the index *mid* of the midpoint. Let's refer to the subarray $A[low \ldots mid]$ as the left subarray and to $A[mid+1 \ldots high]$ as the right subarray. Because we know that the subarray $A[low \ldots high]$ contains at least two elements, each of the left and right subarrays must have at least one element. Lines 4 and 5 conquer by recursively finding maximum subarrays within the left and right subarrays, respectively. Lines 6-11 form the combine part. Line 6 finds a maximum subarray that crosses the midpoint. (Recall that because line 6 solves a subproblem that is not a smaller instance of the original problem, we consider it to be in the combine part.) Line 7 tests whether the left subarray contains a subarray with the maximum sum, and line 8 returns that maximum subarray. Otherwise, line 9 tests whether the right subarray contains a subarray with the maximum sum, and line 10 returns that maximum subarray. If neither the left nor right subarrays contain a subarray achieving the maximum sum, then a maximum subarray must cross the midpoint, and line 11 returns it.

2 Analyzing the divide-and-conquer algorithm

Next we set up a recurrence that describes the running time of the recursive FIND-MAXIMUM-SUBARRAY procedure. We make the simplifying assumption that the original problem size is a power of 2, so that all subproblem sizes are integers. We denote by $T(n)$ the running time of FIND-MAXIMUM-SUBARRAY on a subarray of n elements. For starters, line 1 takes constant time. The base case, when $n = 1$, is easy: line 2 takes constant time, and so

$$T(1) = \Theta(1)$$

The recursive case occurs when $n > 1$. Lines 1 and 3 take constant time. Each of the subproblems solved in lines 4 and 5 is on a subarray of $n/2$ elements (our assumption that the original problem size is a power of 2 ensures that $n/2$ is an integer), and so we spend $T(n/2)$ time solving each of them. Because we have to solve two subproblemsfor the left subarray and for the right subarraythe

contribution to the running time from lines 4 and 5 comes to $2T(n/2)$. As we have already seen, the call to FIND-MAX-CROSSING-SUBARRAY in line 6 takes $\Theta(n)$ time. Lines 711 take only $\Theta(1)$ time. For the recursive case, therefore, we have

$$T(n) = \Theta(1) + 2T(n/2) + \Theta(n) + \Theta(1) = 2T(n/2) + \Theta(n).$$

Combining equations gives us a recurrence for the running time $T(n)$ of FIND-MAXIMUM-SUBARRAY:

$$T(n) = \begin{cases} \Theta(1) & \texttt{if} n - 1, \\ 2T(n/2) + \Theta(n) & \texttt{if} n > 1 \end{cases}$$

This recurrence is the same as recurrence that for merge sort. This recurrence has the solution $T(n) = \Theta(n \lg n)$.

Thus, we see that the divide-and-conquer method yields an algorithm that is asymptotically faster than the brute-force method. With merge sort and now the maximum-subarray problem, we begin to get an idea of how powerful the divide-and-conquer method can be. Sometimes it will yield the asymptotically fastest algorithm for a problem, and other times we can do even better. As Exercise 5 shows, there is in fact a linear-time algorithm for the maximum-subarray problem, and it does not use divide-and-conquer.

3 Problems

1. Write a recursive method that for a positive integer n prints odd numbers
   (a) between 1 and n.
   (b) between n and 1.

2. Write a recursive method to print a *Syracuse sequence* that begins with the number n_0 and each element n_i of the sequence is $n_{i-1}/2$ if n_{i-1} is even and $3 \cdot n_{i-1} + 1$ otherwise. It is conjectured that this sequence ends with a 1.

3. Write a recursive function to compute the binomial coefficient according to the definition $\binom{n}{k} = \begin{cases} 1 & \text{if } k = 0 \text{ or } k = n \\ \binom{n-1}{k-1} + \binom{n-1}{k} & \text{otherwise} \end{cases}$

4. Illustrate the operation of merge sort on the array $A = \langle 3, 41, 52, 26, 38, 57, 9, 49 \rangle$.

5. Rewrite the MERGE procedure so that it does not use sentinels, instead stopping once either array L or R has had all its elements copied back to A and then copying the remainder of the other array back into A.

6. (a) Run the program below and use the results to fill in the table.

Milliseconds per algorithm.				
	Size of random array to be sorted.			
$n =$	1024	4096	16384	65536
bubbleSort()				
insertionSort()				
mergeSort()				
n^2				
$n \lg n$				
$\lg n$				

(b) What sort of operating system/architecture will you use to do this experiment? For instance, I have a Intel(R) Core i5-3570K CPU @ 3.40GHz using 8 GB of RAM.

(c) By forming various ratios, determine the best match for $\Theta(n)$ of each algorithm.

```
#include <iostream>
#include <vector>
#include <ctime>
#include <cstdlib>
#include <chrono>

#define SIZE 65537  //262145
#define INFINITY 2147483647
#define RND_MAX 32768
using namespace std;

void RANDOM_ARRAY(vector<int>& v, int N) {
    srand(unsigned(time(0)));
    for(int i = 0; i < N; ++i)
        v.push_back(-RND_MAX/2+rand()%RND_MAX);
}



void bubbleSort(vector<int>&);
void insertionSort(vector<int>&);
void merge(vector<int>&,int,int,int);
void mergeSort(vector<int>&,int,int);
void print(vector<int>&);
int main()
{
    vector<int> array
    {-235, 234, -34, 235,  -3,
     -235,  34, 346,  23,2346,
      -34,-239, 983, 982, 348,
      845, 234};
    RANDOM_ARRAY(array, 10);
    //cout << "\nYou have " << array.size() << " elements in the array." << endl;
    //print(array);
    print(array);
    //insertionSort(array);
    //print(array);
    //print(array);
    for(int i = 1024; i <= SIZE; i = 4*i) {
        array.clear();
        RANDOM_ARRAY(array,i);
        auto t1 = std::chrono::high_resolution_clock::now();
```

```
        bubbleSort(array);     // a function of your choice
        auto t2 = std::chrono::high_resolution_clock::now();
        cout << "With i = " << i << " bubbleSort() takes "
             << std::chrono::duration_cast<std::chrono::milliseconds>(t2-t1).
    count()
             << " milliseconds\n";
            // of course, you can replace both "nanoseconds" with "milliseconds"
    or "microseconds"
    }
    for(int i = 1024; i <= SIZE; i = 4*i) {
        array.clear();
        RANDOM_ARRAY(array,i);
        auto t1 = std::chrono::high_resolution_clock::now();
        insertionSort(array);     // a function of your choice
        auto t2 = std::chrono::high_resolution_clock::now();
        cout << "With i = " << i << " insertionSort() takes "
             << std::chrono::duration_cast<std::chrono::milliseconds>(t2-t1).
    count()
             << " milliseconds\n";
    }
    for(int i = 1024; i <= SIZE; i = 4*i) {
        array.clear();
        RANDOM_ARRAY(array,i);
        //cout << "\nBefore sorting, array = " << endl; print(array);
        auto t1 = std::chrono::high_resolution_clock::now();
        mergeSort(array,0,i-1);     // a function of your choice
        auto t2 = std::chrono::high_resolution_clock::now();
        //cout << "\nAfter sorting, array = " << endl; print(array);
        cout << "With i = " << i << " mergeSort() takes "
             << std::chrono::duration_cast<std::chrono::milliseconds>(t2-t1).
    count()
             << " milliseconds\n";
            // of course, you can replace both "nanoseconds" with "milliseconds"
    or "microseconds"
    }

    return 0;
}

void bubbleSort(vector<int>& aRay) {
    int size = aRay.size();
    for(int i = 0; i < size-1; ++i) {
        for(int j = 0;j < size-i; ++j)
            if(aRay[j]>aRay[j+1]) {
                int temp = aRay[j];
                aRay[j]= aRay[j+1];
                aRay[j+1] = temp;
            }
    }
}

void insertionSort(vector<int>& aRay) {
    int length = aRay.size();
    int i, j ,tmp;
    for (i = 1; i < length; i++) {
        j = i; //the first i-1 elements are in order.
        while (j > 0 && aRay[j-1] > aRay[j]) {
            tmp = aRay[j]; //if out of order, swap
            aRay[j] = aRay[j-1];
```

```
97            aRay[j - 1] = tmp;
              j--; // swap with largers until in place
99          }//end of while loop
            //print(aRay);
101       }//end of for loop
   }//end of insertion_sort.
103

105 void merge(vector<int>& v, int p, int q, int r) {
        int n1 = q-p+1;
107     int n2 = r-q;
        vector<int> L;
109     for(int i= 0; i < n1; ++i)
            L.push_back(v.at(p+i));
111     L.push_back(INFINITY);
        vector<int> R;
113     for(int j= 0; j < n2; ++j)
            R.push_back(v.at(q+1+j));
115     R.push_back(INFINITY);
        int i=0, j=0;
117     for(int k = p; k <= r; ++k)
            if(L[i]<=R[j]) v[k]=L[i++];
119         else v[k] = R[j++];
    }
121

123 void mergeSort(vector<int>& v,int low,int high) {
        if(low<high) {
125         int mid = (low+high)/2;
            mergeSort(v,low,mid);
127         mergeSort(v,mid+1,high);
            merge(v,low,mid,high);
129     }
    }
131
    void print(vector<int>& v) {
133     for(int i = 0; i < v.size(); ++i) cout << v.at(i) << ", ";
        cout << endl;
135 }
```

7. What does FIND-MAXIMUM-SUBARRAY return when all elements of A are negative?

8. Write code for the brute-force method of solving the maximum-subarray problem. Your procedure should run in $\Theta(n^2)$ time.

9. Implement both the brute-force and recursive algorithms for the maximum-subarray problem on your own computer. What problem size n_0 gives the crossover point at which the recursive algorithm beats the brute-force algorithm? Then, change the base case of the recursive algorithm to use the brute-force algorithm whenever the problem size is less than n_0. Does that change the crossover point?

10. Use the following ideas to develop a nonrecursive, linear-time algorithm for the maximum-subarray problem. Start at the left end of the array, and progress toward the right, keeping track of the maximum subarray seen so far. Knowing a maximum subarray of $A[1 \ldots j]$, extend

the answer to find a maximum subarray ending at index $j+1$ by using the following observation: a maximum subarray of $A[1 \ldots j+1]$ is either a maximum subarray of $A[1 \ldots j]$ or a subarray $A[i \ldots j+1]$, for some $1 \leq i \leq j+1$. Determine a maximum subarray of the form $A[i \ldots j+1]$ in constant time based on knowing a maximum subarray ending at index j.