Contents

Part I

Fundations

Chapter 1

The Role of Algorithms in Computing

Chapter 2

Getting Started

Exercises

2.1-2

INSERTION-SORT(A)

```
1  for j = 2 to A.length
2      key = A[j]
3      i = j − 1
4      while i > 0 and A[i] < key
5          A[i + 1] = A[i]
6          i = i − 1
7      A[i + 1] = key
```

2.1-3

LINEAR-SEARCH(A, v)

```
1  for i = 1 to A.length
2      if v == A[i]
3          return i
4  return NIL
```

Loop invariant: subarray $A[1 \,..\, i-1]$ does not contain such element $A[i]$ that $A[i] == v$.
Initialization: Piror to the first iteration of the loop, subarray $A[i \,..\, i-1]$ contains no element, so it does not contain element whose value equals to v.
Maintenance: The for loop works by scanning the whole array element by element. At the end of each scan, it either add an element whose value does not equal to v to the subarray or exit the program. So before the next

iteration, the subarray does not contain the element that equals to v.
Termination: The program terminates in two cases: find $A[i]$ that equals to v, or return NIL, hence the algorithm is correct.

2.1-4

Input: Two n-element array A and B whose elements corresponding to two n-bit binary integers bit by bit.
Output: A $(n+1)$-element array C whose elements corresponding to the sum of A and B.

BINARY-ADDITION(A, B)

```
1  carry = 0
2  for i = A.length to 1
3       C[i] = A[i] xor B[i] xor carry
4       if A[i] + B[i] + carry ≥ 2
5            carry = 1
6       else
7            carry = 0
8  if A[i] + B[i] + carry ≥ 2
9       C[i + 1] = 1
10 else
11      C[i + 1] = 0
```

2.2-1

$n^3/1000 - 100n^2 - 100n + 3 = \Theta(n^3)$

2.2-2

SELECTION-SORT(A)

```
1  for i = 1 to A.length − 1
2       smallest = i
3       for j = i + 1 to A.length
4            if A[j] < A[smallest]
5                 smallest = j
6       swap A[i] and A[smallest]
```

Loop invariant: Subarray $A[1\,..\,i-1]$ is sorted.
Since after each iteration of the outer for loop, the elements in the subarray $A[i+1\,..\,A.length]$ are all greater or equal to $A[i]$, which, according to the loop invariant, is the largest element of subarray $A[1\,..\,i]$, hence they also

greater than every other element in that array. After run the algorithm for the first $n-1$ elements, the last element left must be the largest element, so the array is ordered.
Best case is when the input array is already ordered.
The inner for loop affects the running time most significantly, it gives the time complexity $\Theta(\sum_{i=1}^{n-1}) = \Theta(n^2)$ for both best case and worest case running time.

2.2-3

Since the element being searched for is equally likely to be any element in the array, then the average elements being searched will be: $1/n \sum_{i=1}^{n} i = (n+1)/2$, which also gives the time complexity of $\Theta(n)$
Since all the elements will be scanned in the worest case, the time complexity is $\Theta(n)$

2.2-4

Modify the algorithm to test what we known as the best case, and solve it specifically.

2.3-2

Modify the last for loop as:

```
1  for k = p to r
2      if i == n_1 + 1
3          for m = j to n_2
4              A[k] = R[j]
5          break
6      if j == n_2 + 1
7          for m = i to n_1
8              A[k] = R[i]
9          break
```

2.3-3

Base case: When $n = 2$, $T(n) = n \lg n = 2$.
Induction: Assume that when $n = 2^k$, $T(2^k) = 2^k \lg 2^k = k \cdot 2^k$. When $n = 2^{k+1}$, $T(2^{k+1}) = 2T(2^k) + 2^{k+1} = 2^{k+1} \lg 2^k + 2^{k+1} = (k+1) \cdot 2^{k+1}$, which is $2^{k+1} \lg 2^{k+1}$, hence the solution of the recurrence is $n \lg n$.

2.3-4

Since it takes $\Theta(n)$ to insert an element into the right place, so the recurrence is:

$$T(n) = \begin{cases} 1 & \text{if } n = 1, \\ T(n-1) + n & \text{otherwise} \end{cases}$$

2.3-5

BINARY-SEARCH(A, m, n, t)

```
1  if m − n == 0
2       if t == A[n]
3            return n
4       else
5            return NIL
6  p = (m + n)/2
7  if t ≤ A[p]
8       return BINARY-SEARCH(A, m, p, t)
9  else
10      return BINARY-SEARCH(A, p + 1, n, t)
```

On each recursion, the problem is divided into two subproblems of $1/2$ the size of the original problem, and terminates when only one element left. Assume the total number of division is k, then $2^k = n$, so the time complexity is $\Theta(\lg n)$

2.3-6

The while loop searches backward for the proper place for the key element, meanwhile moves the elements that are greater than the key element one position to the right. Even though binary search can improve the search time to $\Theta(\lg n)$, it does not affect the time required to move the elements, so use binary search can not improve the overall worest-case running time to $\Theta(n \lg n)$.

2.3-7

We can incorporate MERGE-SORT and BINARY-SEARCH in exercise 2.3-5 into out ADDITION-SEARCH. The basic idea is using merge sort sorting S, scan S, then use binary search to find out if there exists an element $S[j]$ in subarray $S[i+1\,..\,S.\mathit{length}]$ such that $S[i] + S[j] = x$. Since merge sort take

$\Theta(n \lg n)$ and binary search take $\Theta(n \lg n)$ for iterating the entire array, so the time complexity of ADDITION-SEARCH is $\Theta(n \lg n)$.

```
ADDITION-SEARCH(S)
1  MERGE-SORT(S, 1, S.length)
2  for i = 1 to S.length
3       m = S[i]
4       n = x − m
5       if BINARY-SEARCH(S, i + 1, S.length, n)
6            return m, n
7       else
8            return NIL
```

Problems

2-1

a. Selection sort take $\Theta(k^2)$ to sort all k elements in each sublist, hence the cost of n/k sublists will be nk.

b. Since we start merging when we split up to n/k sublists, the depth of the recursion tree is $\lg(n/k)$. Moreover, we merge n elements at each level of the tree, hence the worst case running time is $\Theta(n \lg(n/k))$

c. Notice that the largest asymptotic value of k is $k = \lg n$, because otherwise there would be higher order term in $\Theta(nk + n \lg(n/k))$ than $\Theta(n \lg n)$. Replace k with $\lg n$ in the worst case running time of our modified merge sort algorithm, yields $\Theta(n \lg n + n \lg \lg(n/k))$, which, just consider the higher order term, equals to the worst case running time of merge sort in terms of Θ notation. Therefore, the largest value of k is $\lg n$.

d. k should be the largest sublist length so that selection sort outperforms merge sort.

2-2

a. We need to show that the elements in A' forms a permutation of the elements in A.

b. **Loop invariant:** $A[j] = minA[j \,..\, A.length]$ and the subarray $A[j \,..\, A.length]$ is a permutation of the elements in $A[j \,..\, A.length]$ at the time when the loop started.

Initialization: On initialization, subarray $A[j\,..\,A.\,length]$ only contains one element, so its the smallest element in the subarray.
Maintenance: The only possible operation to the subarray is exchange $A[j]$ and $A[j-1]$ when $A[j] < A[j-1]$. Since $A[j]$ is the smallest element in $A[j\,..\,A.\,length]$, if the exchange happens, $A[j-1]$ would become the smallest element in $A[j-1\,..\,A.\,length]$. Meanwhile, since subarray $A[j\,..\,A.\,length]$ is a permutation of the elements in $A[j\,..\,A.\,length]$ at the time the loop started, after exchanging $A[j]$ and $A[j-1]$, $A[j-1\,..\,A.\,length]$ is still a permutation of the elements that were in $A[j-1\,..]$ at the time the loop started.
Termination: Since j decreases by 1 after each iteration, the condition that causes the loop terminate is $j = i$. By the state of the loop invariant, $A[i] = minA[i\,..\,A.\,length]$, and subarray $A[i\,..\,A.\,length]$ is a permutation of the elements that were in $A[i\,..\,A.\,length]$ at the time the loop started.

c. **Loop invariant:** Subarray $A'[1\,..\,i-1]$ contains $i-1$ smallest elements come from array $A[1\,..\,A.\,length]$ and in ordered manner.
Initialization: Piror to the first iteration of the outer **for** loop, $i = 1$, so there is no element in $A'[1\,..\,i-1]$, the loop invariant holds.
Maintenance: Since $A'[1\,..\,i-1]$ is the $i-1$ smallest elements from $A[1\,..\,A.\,length]$, after the execution of the inner **for** loop, which makes $A'[i]$ the smallest element of subarray $A[i\,..\,[.\,A][length]]$, and the values in the subarray all come from $A[1\,..\,A.\,length]$, hence subarray $A'[1\,..\,i]$ contains the i smallest elements in $A[1\,..\,A.\,length]$.
Termination: The loop terminates when i reaches $A.\,length$, by the state of the loop invariant, $A'[1\,..\,A.\,length\text{-}1]$ contains $A.\,length\text{-}1$ smallest elements of $A[1\,..\,A.\,length]$, moreover, since $A[A.\,length]$ is the only element in the remaining subarray, therefore it is the largest element of $A[1\,..\,A.\,length]$. In conclusion, array $A'[1\,..\,A.\,length]$ is a permutation of $A[1\,..\,A.\,length]$ but in sorted manner.

d. The worest case running time of the BUBBLESORT is:

$$\sum_{i=1}^{A.\,length-1} (n-i+1) = \frac{n^2}{2} = \Theta(n^2)$$

It is the same as the worest case running time of insertion sort.

2-3

a. The **for** loop executes $n+1$ times, so the running time of the code is $\Theta(n)$.

b. The naive polynomial evaluation algorithm is:

POLY-EVAL(A, x)

```
1  y = 0
2  for i = 1 to A.length
3        y = x^{i-1} + A[i]
```

Since it has to perform $i - 1$ mutiplication and an addition at each iteration, the running time of POLY-EVAL is:

$$\sum_{n=1}^{A.length} i = \Theta(n^2)$$

It is n times slower than Horner's rule.

c. **Initialization:** Piror to the first iteration, $y = 0$, and since a summation with no terms equals to 0, therefore the loop invariant holds. **Maintenance:** At the start of the iteration when i reaches t, $y = \sum_{k=0}^{n-(t+1)} a_{k+t+1}x^k$. Prior to i reaches $t - 1$,

$$\begin{aligned} y &= a_t x^0 + x \sum_{k=0}^{n-(t+1)} a_{k+t+1}x^k \\ &= a_t x^0 + \sum_{k=1}^{n-t} a_{k+t}x^k \\ &= \sum_{k=0}^{n-t} a_{k+t}x^k \end{aligned}$$

which is the same as $\sum_{k=0}^{n-(i+1)} a_{k+i+1}x^k$, hence the loop invariant holds. **Termination:** i reaches -1 when the loop terminates. Plug $i = -1$ into the equation, yield $y = \sum_{k=0}^{n} a_k x^k$.

d.

2.4

a. $(1, 5)$, $(2, 5)$, $(3, 4)$, $(3, 5)$, $(4, 5)$.

b. Array $[n \,..\, 1]$ has the most inversions, since for each $i < j$, $A[i] > A[j]$. It has $\sum_{k=1}^{n-1}(n - k) = n^2/2$ inversions.

c. The running time is in proportion to the amount of inversions in the array. Since in the **for** loop of lines 1-8, we set the key to $A[j]$ and $i = j - 1$, and the **while** loop of lines 5-7 iterates the subarray $A[1 \,..\, j]$ and eliminates the inversions. In other words, the amount of inversions

need to be eliminated contributes to the running time of insertion sort, the more the inversions, the longer the running time, vice versa.

d. (In reference to clrs instructor's manual)
To start off, we define a subroutine called MERGE-INVERSIONS that has value x in L and y in R such that $x > y$, and, like the MERGE procedure in merge sort, but count the amount of inversions at the same time. Consider an inversion (i, j), and $x = A[i]$, $y = A[j]$, such that $x > y$, we claim that there is exactly one MERGE-INVERSIONS that involves x and y. To see why, notice that since L and R are sorted subarrays, the only way that two elements can change their relative position to each other is for the greater one in L and the smaller one in R within the MERGE-INVERSIONS, hence at least one $Merge - Inversions$ involving x and y. Moreover, observe that at the end of MERGE-INVERSIONS, the elements in L and R are stored into another subarray either L or R in an ordered manner, therefore x and y both appear in the same subarray in the right order, thus there is only one MERGE-INVERSIONS involving x and y, our claim has been proven.
We have proven that one inversion implies one MERGE-INVERSIONS, now we prove that the claim still holds reversely. Observe that since we have a MERGE-INVERSIONS, we have x and y such that $x > y$. Notice that either L or R are ordered, and x in L and y in R, therefore $i < j$, thus (i, j) is an inversion.
We have proven one-to-one relationship between MERGE-INVERSIONS and inversions, now we count the inversions. Notice that if there exists an inversion (i, j), there must be an x that is greater than y at some point of time. Moreover, since the subarray is ordered, all the elements with index greater than i are also greater than y, hence they are all values that compose inversions with y. Let us denote i' the index of the smallest element z in L that is greater than y, thus the number of inversions involving y would be $n_1 - i' + 1$. We need to detect the first time such an z and y are exposed, and compute the number of inversions thereafter.
The following pseudocode represents the idea above:

```
COUNT-INVERSIONS(A, p, r)
1  inversions = 0
2  if p < r
3       q = ⌊(p + r)/2⌋
4       inversions = inversions + COUNT-INVERSIONS(A, p, q)
5       inversions = inversions + COUNT-INVERSIONS(A, q + 1, r)
6       inversions = inversions + MERGE-INVERSIONS(A, p, q, r)
7  return inversions
```

```
MERGE-INVERSIONS(A, p, q, r)
 1  n_1 = q - p + 1
 2  n_2 = r - q
 3  for i = 1 to n_1
 4       L[i] = A[p + i - 1]
 5  for j = 1 to n_2
 6       R[j] = A[q + j]
 7  L[n_1 + 1] = ∞
 8  R[n_2 + 1] = ∞
 9  i = 1
10  j = 1
11  inversions = 0
12  counted = FALSE
13  for k = p to r
14       if counted = FALSE
15            inversions = inversions + n_1 - i + 1
16            counted = TRUE
17       if L[i] ≤ R[j]
18            A[k] = L[i]
19            i = i + 1
20       else
21            A[k] = R[j]
22            j = j + 1
23            counted = FALSE
24  return inversions
```

On lines 14-16, we count the number of inversions upon each time z and y are exposed, this is a linear time operation, thus the running time of the algorithm above is the same as merge sort, which is $\Theta(n \lg n)$.

Chapter 3

Growth of Functions

Exercises

3.1-1

To prove $\max(af(n), g(n)) = \Theta(f(n) + g(n))$, that means we need to prove that for some c_1, c_2, and n_0 such that $0 \leq c_1(f(n)+g(n)) \leq \max(f(n), g(n)) \leq c_2(f(n) + g(n))$ for all $n \geq n_0$. By choosing $c_2 = 1$, we can easily see that the right condition is satisfied. The left condition can be satisfied when we choose c_1 to be $1/2$.

3.1-2

To prove the equation, we have to show that there exist positive constants c_1, c_2 and n_0 such that $0 \leq c_1 n^b \leq (n + a)^b \leq c_2 n^b$ for any real constant a and $b > 0$, and $n \geq n_0$. This can be written as $0 \leq \sqrt[b]{c_1}n \leq n + a \leq \sqrt[b]{c_2}n$.
Let us examine 3 cases below:
$a > 0$: We can easily verify that the left condition satisfies when taking $c_1 \leq 1$ for all n. For the right condition, by examining the graph of the two function $f(n) = n + a$ and $g(n) = \sqrt[b]{c_2}n$, we know that when the gradient of $g(n)$ is greater than 1, there must be some n_0 that makes $g(n) \leq f(n)$ for $n \geq n_0$. Therefore $\sqrt[b]{c_2}$ should be greater than or equal to 1, which makes $c_2 > 1$.
$a < 0$: Similar to $a > 0$, we can conclude that $c_1 < 1$ and $c_2 \geq 1$.
$a = 0$: The inequation satisfies when taking $c_1 \leq 1$ and $c_2 \geq 1$.
In conclusion, when taking $c_1 < 1$ and $c_2 > 1$, there must be a n_0 such that the inequation satisfies when $n \geq n_0$ for any real constant a and b,

3.1-3

The big O notation set a upper bound of the running time of an algorithm, it means the running time of an algorithm is no worse than n^2. "At least" in the statement indicates that there exists any worse case, therefore it is meaningless.

3.1-4

$2^{n+1} = O(2^n)$. To see why, we have to prove that there eixsts a positive constant nubmer c and n_0 such that $0 \leq 2^{n+1} \leq c2^n$ for all $n \geq n_0$. Dividing both sides of the inequation by 2^n, yields $0 \leq 2 \leq c$, therefore any $c \geq 2$ satisfies the equation.
$2^{2n} \neq O(2^n)$. To see why, we have to prove that the inequation $0 \leq 2^{2n} \leq 2^n$ does not hold. Dividing both sides of the inequation yeilds $0 \leq 2^n \leq c$, 2^n is a monotonically increasing function, therefore we cannot find a constant that satisfies $2^n \leq c$.

3.1-5

By definition, $f(n) = \Theta(g(n))$ implies there exists positive contants c_1, c_2 and n_0 such that $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n \geq n_0$. Notice that the left part and right part of the inequation are the definitions of Ω and O respectively, hence the necessity has been proven.
If $f(n) = \Omega(g(n))$ and $f(n) = O(g(n))$, that means there exists positive contants c_1, c_2 n_1 and n_2 such that $0 \leq c_1 g(n) \leq f(n)$ for all $n \geq n_1$ and $0 \leq f(n) \leq c_2 g(n)$ for all $n \geq n_2$. Combining these two inequations, we have there exists positive constants c_1, c_2, n_1 and n_2 such that $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n \geq \max(n_1, n_2)$. Hence we have proven the sufficiency.

3.1-6

Assume the running time of the algorithm is $f(n)$, so $f(n) = \Theta(g(n))$, according to Theorem 3.1, the assumption can be proved.

3.1-7

By definition of o, for any positive constant c, there exists $n_0 > 0$ such that $0 \leq f(n) < cg(n)$ for all $n \geq n_0$; and by definition of ω, for any positive constant c, there exists $n_0 > 0$ such that $0 \leq cg(n) < f(n)$ for all $n \geq n_0$. However, $f(n)$ cannot be greater than $cg(n)$ and less than $cg(n)$ at

the meantime, hence this is a contradiction. Therefore $o(g(n)) \cap \omega(g(n))$ is empty.

3.2-1

$f(n)$ and $g(n)$ are monotonically increasing means that if $n_1 \geq n_2$, then $f(n_1) \geq f(n_2)$ and $g(n_1) \geq g(n_2)$. Observe that $f(n_1) + g(n_1)$ also greater than or equal to $f(n_2) + g(n_2)$, therefore $f(n) + g(n)$ is monotonically increasing.
Also notice that since $g(n_1) \geq g(n_2)$, and $f(n)$ is monotonically increasing, so $f(g(n_1)) \geq f(g(n_2))$, consequently, $f(g(n))$ is also monotonically increasing.
Since $f(n_1) \geq f(n_2)$, also because $f(n)$ and $g(n)$ are nonnegative, then $f(n_1) \cdot g(n_1) \geq f(n_2) \cdot g(n_1)$. Moreover, $g(n_1) \geq g(n_2)$, therefore $f(n_1) \cdot g(n_1) \geq f(n_2) \cdot g(n_2)$, hence $f(n) \cdot g(n)$ is monotonically increasing.

3.2-2

$$\begin{aligned} a^{\log_b c} &= a^{\frac{\log_a c}{\log_a b}} \\ &= a^{\log_a c \log_b a} \\ &= c^{\log_b a} \end{aligned}$$

3.2-3

Since $n! = \sqrt{2\pi} n^{n+1/2} e^{-n}$,

$$\begin{aligned} \lg(n!) &\approx n \lg n - n + \frac{1}{2} \lg(2\pi n) \\ &= (n + \frac{1}{2}) \lg n - n + \frac{1}{2} \lg(2\pi) \\ &\approx n \lg n - n \\ &= \Theta(n \lg n) \end{aligned}$$

Since $n!/n^n = (n-1)!/n^{n-1} \ldots (n-(n-1))!/n^{n-(n-1)} = 1/n$, so $\lim_{n\to\infty} n!/n^n = 0$, which means $n! = o(n^n)$.
We can easily verify that when $n \geq 4$, $0 \leq c2^n \leq n!$ for any positive constant c, hence $n! = \omega(2^n)$.

3.2-4

3.2-5

3.2-6

Substitute the golden ratio and its conjugate in the equation, which shows they both satisfies the equation.

3.2-7

When $i = 0$,

$$F_i = \frac{\phi^0 - \hat{\phi}^0}{\sqrt{5}}$$
$$= 0$$

When $i = 1$,

$$F_i = \frac{\phi^1 - \hat{\phi}^1}{\sqrt{5}}$$
$$= 1$$

Assume that when $i = k - 1$,

$$F_i = \frac{\phi^{k-1} - \hat{\phi}^{k-1}}{\sqrt{5}}$$

and when $i = k$,

$$F_i = \frac{\phi^k - \hat{\phi}^k}{\sqrt{5}}$$

therefore, when $i = k + 1$,

$$
\begin{aligned}
F_i &= F_k + F_{k-1} \\
&= \frac{\phi^i - \hat{\phi}^i}{\sqrt{5}} + \frac{\phi^{i-1} - \hat{\phi}^{i-1}}{\sqrt{5}} \\
&= \frac{\phi^{i-1}(\phi + 1) - \hat{\phi}^{i-1}(\hat{\phi} + 1)}{\sqrt{5}} \\
&= \frac{\phi^{i-1}(\phi - \phi\hat{\phi}) - \hat{\phi}^{i-1}(\hat{\phi} - \phi\hat{\phi})}{\sqrt{5}} && \text{Since } \phi\hat{\phi} = -1 \\
&= \frac{\phi^i(1 - \hat{\phi}) - \hat{\phi}^i(1 - \phi)}{\sqrt{5}} \\
&= \frac{\phi^{i+1} - \hat{\phi}^{i+1}}{\sqrt{5}} && \text{Since } \phi + \hat{\phi} = 1
\end{aligned}
$$

In conclusion,

$$F_i = \frac{\phi^i - \hat{\phi}^i}{\sqrt{5}}$$

3.2-8

3.1

3.2

A	B	O	o	Ω	ω	Θ
$\lg^k n$	n^ϵ	Yes	Yes	No	No	No
n^k	c^n	Yes	Yes	No	No	No
$\sqrt{n}$	$n^{\sin n}$	No	No	No	No	No
2^n	$2^{n/2}$	No	No	Yes	Yes	No
$n^{\lg c}$	$c^{\lg n}$	Yes	No	Yes	No	Yes
$\lg(n!)$	$\lg(n^n)$	Yes	No	Yes	No	Yes

Chapter 4

ch4

Exercises

Chapter 5

ch5

Exercises

Part II

Sorting and Order Statistics

Chapter 6

ch6

Exercises

Chapter 7

ch7

Exercises

Chapter 8

ch8

Exercises

Chapter 9

ch9

Exercises

Part III

Data Structures

Chapter 10

ch10

Exercises

Chapter 11

Hash Tables

Exercises

11.1-1

Assume that the dynamic set is represented by direct-address table T, we can iterate through T, and store the maxmium value in a variable. Since searching operation of direct-address table is $O(1)$, it takes $O(m)$ to find the maxmium element.

11.1-2

Set the i^{th} bit of the vector to 1 if the i^{th} element is inserted and 0 if it is deleted.

11.1-3

Use the basic structure of the traditional direct-address table, but each element should have 2 fields: satellite data and a pointer to the next element, so that elements with identical key value could be stored in a linked list.

11.1-4

11.2-1

Chapter 12

ch12

Chapter 13

ch13

Chapter 14

ch14

Part IV

Advanced Design and Analysis Techniques

Chapter 15

Dynamic Programming

Exercises

15.1-1

For the initial condition, $T(0) = 1 = 2^0 = 2^n$.
Assume that $T(n) = 2^n$. Since $T(n) = 1 + \sum_{j=0}^{n-1} T(j)$, then $T(n+1) = 1 + \sum_{j=0}^{n} T(j) = 1 + \sum_{j=0}^{n-1} T(j) + T(n) = 2T(n) = 2^{n+1}$, therefore equation (15.4) follows equation (15.3) and the initial condition $T(0) = 1$.

15.2-2

Consider the rod with length 3 inches, the price for a cut of length 1 inch is 1, and the price for a cut of length 2 inch is 5, and the price of a cut of length 3 is 7. Since the cut of length 2 has the largest density, the resulting cutting then would be 2 pieces, 1 inch and 2 inches, which has a total revenue of 6, but the cut of length 3 has a total revenue of 7, therefore the greedy approach does not work in this case.

15.2-3

We can use the same algorithm of original rod- cutting problem, but each time when $p[i]$ is called, substract it with the extra cost c.

15.2-4

Modify Memorized-Cut-Rod-Aux as below:

```
MEMORIZED-CUT-ROD-AUX(p, n, r)
 1  if r[n] ≥ 0
 2      return r[n]
 3  if n == 0
 4      q = 0
 5  else q = −∞
 6      for i = 1 to n
 7          t = p[i] + MEMORIZED-CUT-ROD-AUX(p, n − 1, r)
 8          if q < t
 9              s = i
10              q = t
11          l[i] = s
12  r[n] = q
13  return q, l
```

Then use similar procedure as PRINT-CUT-ROD-SOLUTION to compute the actual solution.

15.2-5

These are the top-down version and bottom-up version of the algorithm.

```
TOP-DOWN-COMPUTE-FIBO(n)
1  let r[0..n + 1] be a new array
2  TOP-DOWN-COMPUTE-FIBO-AUX(n, r)
```

```
TOP-DOWN-COMPUTE-FIBO-AUX(n, r)
 1  if r[n] ≥ 0
 2      return r[n]
 3  if n == 0
 4      q = 0
 5  elseif n == 1
 6      q = 1
 7  else
 8      q = TOP-DOWN-COMPUTE-FIBO-AUX(n − 1, r)
        +TOP-DOWN-COMPUTE-FIBO-AUX(n − 2, r)
 9  r[n] = q
10  return q
```

```
BOTTOM-UP-COMPUTE-FIBO-AUX(n)
1  x = 0
2  y = 1
3  for i = 0 to n
4        temp = x
5        x = y
6        y = temp + y
7  return temp
```

The bottom-up version is faster than the top-down version for large data set in that it does not have to traverse the table.