1. Rank the following by growth rate: *n*, , , , , , 4, , 

*Solution*: Define a partial order  on the functions under consideration by

.

We will show the following.



Let , , , , , , , , and. Then for , it suffices to show that

 (\*)

since this implies .

Recall that in terms of asymptotic analysis, the base used for a logarithmic function may be any constant greater than 1, since if  and , then , for the constant . Therefore, we can ignore the base of a logarithm inside of asymptotic notation. That is,. Since we use L’Hopital’s rule in proving (\*) for several values of *i*, it is convenient to use, the natural logarithm of *n*, in problems requiring asymptotic evaluation of logarithmic functions.

For , (\*) is true since .

For , (\*) is true since by L’Hopital’s rule,

.

For , the proof of (\*) is a straightforward consequence of L’Hopital’s rule.

For , we have  (by L’Hopital’s rule)

 (by L’Hopital’s rule) .

For , we have .

For , (by L’Hopital’s rule)

.

To show (\*) is valid for , observe that for , . Hence, for , . Thus, .

1. Prove or disprove each of the following.
   1. 
   2. 
   3. 
   4. 
   5. 

*Solution*:

a) The statement is **false**. This is easily seen if we choose  and .

b) The statement is **true** for nonnegative functions. For nonnegative functions  and , we have . It follows that . Note that if we omit the assumption of nonnegativity, the statement is false, as illustrated by the example , .

c) The statement is **false** in general. Consider . It is easily seen that . Therefore,  is not .

If for  we have , then  implies

.

Thus, . For such a function, it is true that 

d) The statement is **true**.



for some and , if then 



 if  then 

.

e) The statement is **true**. Let *g(n)* be a nonnegative function such that. Then . Therefore, for some , .

1. Use , , , , and  to describe the relationship between the following pairs of functions.
   1. , , where *k* and  are positive constants
   2. , , where *k* and *c* are constants, , 
   3. , 

*Solution*:

a) Since , we know that . Therefore, it suffices to show .

We use the well-known inequality  for all . Substituting  for *x*, this gives , or . Therefore, for , . It follows that .

b) Note . Thus, to show , we may assume without loss of generality that *k* is an integer. We give two arguments.

* Our first argument may be unfair for some students, in that it uses mathematical induction, which is discussed in the next chapter. It can be shown by mathematical induction and L’Hopital’s rule that  for integer *j* such that . Hence, for , . It follows that .
* An argument not using induction follows. We begin by considering  (by L’Hopital’s rule) . It follows that there is a positive *N* such that . Since the exponential function is monotone increasing,  implies

.

Therefore, .

c) , hence .

1. Prove that .

*Solution*: . Therefore, .

1. Prove that .

*Solution:* This follows from the example in the chapter that .

1. Given a set of *n* **integer** values in the range of , give an efficient sequential algorithm to sort these items. Discuss the time, space, and optimality of your solution.

*Solution*: We could use the BinSort algorithm that appeared in the chapter to solve this problem in optimal  time, using  space, including  extra space. An alternative solution, using  time and only  extra space, is based on Counting Sort, an algorithm that will be more fully discussed later in the book.

Roughly, the idea is that we can make one pass through the data, counting the frequency of each of the possible data values. Then we can place each data value in its appropriate position. The extra space used is dominated by an array that is used to count the frequencies of the at most  distinct data values. The algorithm is described as follows.

**Procedure Sort(*list*)**

{Sort an array of integer values restricted to the range of .}

**Input**: an array, *list*, of integers indexed from 1 to *n* with values in 

**Output**: the *list* in ascending order

**Local variables**: integer *l\_index*, integer *f\_index,* integer *count*, integer array 

**Action**:

For *f\_index* = 1 to 100, do

 {initialization}

End For

{Next, count the frequencies of the data values.}

For *l\_index* = 1 to *n*, do





End For

{Now, place the data values in their proper positions.}





Repeat

While , do



End While

For  to , do





End For



Until 

End Sort

1. **(Total function)** Determine the asymptotic running time of the following algorithm, which is used to sum a set of values. Show that the running time is optimal.

**Function Total(*list*)**

**Input:** an array, *list*, of numeric entries indexed from 1 to *n*.

**Output:** the total of the entries in the array

**Local variables:** integer *index*, numeric *subtotal*

**Action:**

*subtotal* = 0

For *index* = 1 to *n*, do



return *subtotal*

*Solution*: The initialization of the subtotal runs in  time, as does every update of the subtotal in the body of the For-loop. Therefore, the entire For-loop runs in  time. Hence, the algorithm runs in  time. Since each data item must be considered, the time is optimal.

1. **(Selection Sort)** Determine the asymptotic running time of the following algorithm, which is used to sort a set of data. See Figure 1-13. Determine the total asymptotic space and the additional asymptotic space required.

**Subprogram SelectionSort(*List*)**

**Input**: array , to be sorted in ascending order according to the *key* field of the records

**Output**: the ordered *List*

**Algorithm**: SelectionSort

For each position in the *List*, do the following.

1. Determine the index corresponding to the entry from the unsorted portion of the *List* that is a minimum.

2. Swap the item at the position just determined with the current item.

**Local variables:** indices *ListPosition, SwapPlace*

**Action:**

{ListPosition is only considered for values up to n-1, because once the first n-1 entries have been swapped into their correct positions, the last item must also be correct.}

For *ListPosition =* 1 to 

{Determine the index of correct entry for ListPosition and swap the entries.}

**

**

End For

End *Sort*

**Subprogram Swap(*A, B*)**

**Input**: Data entities *A, B*

**Output**: The input variables with their values interchanged, *e.g.*, if on entry we have *A=*3 and *B=*5, then at exit we have *A=*5 and *B=*3*.*

**Local variable**: *temp*, of the same type as *A* and *B*

Action:

*temp = A* {Backup the entry value of *A*}

*A = B* {*A* gets entry value of *B*}

*B = temp* {*B* gets entry value of *A*}

end *Swap*

**Function MinimumIndex(*List, startIndex*)**

**Input**: , an array of records to be ordered by a *key* field; *startIndex*, the first index considered.

**Output**: index of the smallest *key* entry among those indexed *startIndex … n* (the range of indices of the portion of the *List* presumed unordered).

**Local variables**: indices *bestIndexSoFar, at*

**Action**:

*bestIndexSoFar = startIndex*

{at is used to traverse the rest of the index subrange}

For  to *n*,do

If 

then *bestIndexSoFar = at*

End For

Return *bestIndexSoFar*

End *MinimumIndex*

*Solution*: In the *MinimumIndex* function, the body of the For-loop runs in  time per iteration. So, if , the running time of *MinimumIndex* is . Since *MinimumIndex* is called from the For-Loop of *SelectionSort* for , the time utilized by *MinimumIndex* in the For-loop of *SelectionSort* is  (by substituting  and, hence, ) . It is easy to see that the *Swap* subprogram runs in  time. Therefore, the total time utilized by all the *Swap* operations in the For-loop of *SelectionSort* is . Thus, the running time of *SelectionSort* is .

The array *List* utilizes  space. If we assume that it is passed by reference, then the space occupied by *List* is not duplicated from the calling program. The additional space required is for a constant number of simple local variables, so the extra space is .

1. Earlier in this chapter, we gave an array-based implementation of Insertion Sort. In this problem, we consider a linked list based version of the algorithm.

**Subprogram InsertionSort(*X*)**

For every *current* entry of the list after the first entry do

Search the sublist of all entries from the first entry to the *current* entry for the proper placement, indexed *insertPlace*, of the *current* entry in the sublist.

Insert the *current* entry into the same sublist at the position *insertPlace*.

end for

Suppose we implement the Insertion Sort algorithm as just described for a linked list data structure.

1. What is the worst-case running time for a generic iteration of the Search step?

*Solution*: In the worst case, all items up to the *current* item must be examined, as the *current* item could belong at the end of the portion of the list that has been sorted. Thus, for the  initial item, the search step runs in  time in the worst case.

1. What is the worst-case running time for a generic instance of the Insert step?

*Solution*: Once the search step has located the position in the sorted portion of the list at which the *current* item should be inserted, the insertion runs in  time by using at most 4 pointer assignments. Specifically, there are at most 2 pointer assignments to excise the item from its input position while maintaining the other  links in a singly linked list, and at most 2 more pointer assignments to insert the link at its proper position.

1. Show that the algorithm has a worst-case running time of .

*Solution*: It follows from the previous discussion is that the worst-case running time is .

1. Although both the array-based and linked list-based implementations of Insertion Sort have worst-case running times of , in practice, we usually find that the linked list based implementation, assuming the same data, in the same input order, is faster. Why should this be? Think in terms of entries consisting of large data records.

*Solution*:The array-based implementation requires, in the worst case, asymptotically as many data copying operations as the worst-case linked-list based implementation requires pointer operations. Copying large records is likely to be much more expensive than pointer assignments, as has been discussed in the text.

1. Array implementations of both Insertion Sort and Selection Sort have  worst-case running times. Which is likely to be faster if we time both in the same hardware/software environment for the same input data? Why?

*Solution*: Both sorting routines make  comparisons in the worst case. However, each pass through the main loop of SelectionSort uses only one swap, for a total of  data movement operations. By contrast, it is easily seen that an array-based InsertionSort might make as many as  data movement operations. Note that if the data is made up of large records, data movement operations may be quite expensive. Therefore, SelectionSort, with significantly fewer of these expensive operations, is likely to be faster.

1. The **Stable Marriage Problem** (**SMP**) requires establishing a stable matching between two sets of elements, given a set of preferences for each element. Suppose there are arrays  and  of identically structured *person* records, where one of the fields in the record is *partner*. Suppose the entries of these arrays represent members of couples, with the value of the *partner* field indexing the member of the opposite array that is the entry’s partner. For example, if  and  are a couple, then  and . Suppose there is an *evaluation* function of two *person* records that executes in  time, returning a numerical evaluation of how the *person* represented by the first parameter evaluates the *person* represented by the second parameter. There is an unstable situation if an uncoupled pair, one from the *him* array and one from the *her* array, each evaluates the other higher than his/her own *partner*.
2. Give an efficient algorithm that determines whether or not the *him* and *her* arrays represent an unstable situation, and analyze its worst-case running time.
3. Analyze the best-case running time of this algorithm.
4. If your algorithm is efficient, argue that it has optimal worst-case running time.

*Solutions*:

1. It will be convenient to use *Boolean* variables, that is, variables that only store values *True* and *False*. Consider the following algorithm.

* In time, initialize,.



* Do while *stable* and 
  +  (this takes time)



* +  ( time)



* + Do while *stable* and 
    -  ( time)



* + -  

or



*I.e.*, *stable* is *True* so far if and only if the complex condition to the right of the equal sign is *True*. This step runs in time.



* + - If *stable* then increase *herIndex* by 1 ( time)



* + End inner Do-While loop. Clearly, one iteration of the loop body requires  time, and in the worst case, there are *n* iterations of the loop body, so the worst-case running time of the loop is .
  + If *stable* then increase *himIndex* by 1 ( time)



* End outer Do-While loop. For the worst case, one iteration of the loop body runs in  time and there are *n* iterations of the loop body, so the worst-case running time of the loop is .
* Return *stable* and, if desired, *himIndex* and *herIndex* so that if *stable* is *False*, an unstable couple is identified to the subprogram’s caller. This can be done in time.



Clearly, the worst-case running time of this algorithm is .



1. In the best case, *stable* becomes *False* when  and. In this case, both the inner and outer loops are exited after only 1 iteration apiece of their loop bodies. It follows easily that in the best case, the running time is .
2. The *n* pairs  such that  represent couples, need not be considered, but all other of the  pairs must be considered in the worst case, since any of them, including the last one considered, might be the first such pair that yields an example of an unstable couple. Since , any algorithm for this problem has a worst-case running time of. Therefore, our algorithm has an optimal worst-case running time of.

