1. **(Exercise 3.1-3)**

Explain why the statement, “The running time of algorithm *A* is at least *O*(*n*).” is meaningless.

**Solution:**

By definition, *O*(*g*(*n*)) = {*f*(*n*): there exist positive constants *c* and 0 such that 0 ≤ *f*(*n*) ≤ *cg*(*n*) for all *n* ≥ 0}. Let *T*(*n*) be the running time of algorithm *A* then the statement above is equivalent to saying *T*(*n*) ≥ *O*(*n*) which means *T*(*n*) ≥ *f*(*n*) for some *f*(*n*) in *O*(*n*) but it doesn't tell us which *f*(*n*), in fact, it doesn't really tell us anything.

1. **(Exercise 3.1-2)**

Show that for any real constants *a* and *b*, where *b* > 0, = .

**Solution:**

By definition of , we need to find three constants *c*1, *c*2 and *n*0 such that 0 *c*1­ *c*2­, for all *n* 0 .

Since *n* , we have

*n* + *a* *n* + |*a*| 2*n*.

Since *n* (i.e. ) , we have

*n* + *a* *n* -

Thus, when *n* we have

0 2*n*.

Since *b* is a positive constant,

0 *b* *b* (2*n*)*b*

0 b *b* (2*b*)*nb*

So, *c*1 = , *c*2 = 2*b* , *n*0 = 2, satisfy the definition.

**3.**

Can the master method be applied to the recurrence *T* (*n*) = 4*T* (*n*/2) + *n*2? Why or why not? Give an asymptotic upper bound for this recurrence.

**Solution:**

Let *T* (*n*) = *a T* (*n*/*b*) + *f* (*n*)

a = 4, b = 2 =>

, satisfy Master Theorem case 2.

**4.(Exercise 4.3-8)**

Using the master method, you can show that the solution to the recurrence *T*(*n*) = 4T(*n*/2)+*n* is *T*(*n*)=Θ(*n*2). Show that a substitution proof with the assumption *T*(*n*) ≤ c*n*2 fails. Then show how to subtract off a lower-order term to make the substitution proof work.

**Solution:**

If we guess *T*(*n*) ≤ c*n*2

*T*(*n*) ≤ 4c(*n*/2)2 +*n*

≤ c*n*2+n=> This is not equal to *cn*2

Now we guess*T*(*n*)≤*cn*2−*n*

*T*(*n*) ≤ 4(c(*n*/2)2−*n*/2)+*n*

≤ cn2−2*n*+*n*

≤ *cn*2−*n* =>exact equal to our assumption.

**5. (Exercise 6.5-9)**

Give an *O*(*n* lg *k*)-time algorithm to merge *k* sorted lists into one sorted list where *n* is the total number of elements in all input lists. (Hint: Use a min-heap for *k*-way merging)

**Solution:**

Take out the first element from the sorted lists respectively and build a min-heap with them. For each element in the heap, we shall record the list-id of the list where it is taken out. The heap size is k, so this step takes *O*(*k*) time.

Put the root into the new sorted list and replace the root with the next element from the same list. If there is an empty list, pick the next element from the next non-empty list. Run MIN-HEAPIFY on the root. Repeat this step until all k sorted lists are empty. Each time takes *O*(lg *k*).

Now, the k elements in the heap are sorted. We can append them to the new sorted list and get the sorted list of length n.

The time complexity of this algorithm is *O*(*k*)+(*n*-*k*) *O*(lg *k*) = *O*(*n* lg *k*).

**6. (Exercise 7.4-3)**

Show that the expression achieves a maximum over when or .

**Solution:**

We can give an upper bound for the expression by the following equation:

The “=” of the inequality holds only when , which implies or . So has maximum when or .

**7.**

Give an efficient in-place algorithm to arrange an array of *n* elements so that all the negative keys precede all the non-negative keys. How fast is your algorithm.

**Solution:**

Input: Array *A*[0,*n*-1] of real numbers

Output: Array *A*[0,*n*-1] in which all the negative keys precede all the non-negative keys(including zeros).

ARRANGE(*A*, *n*)

1. *i* = 0
2. *j* = *n* – 1
3. while *i* *j*
4. if *A*[*j*] < 0
5. *i* = *i* + 1
6. else
7. swap( *A*[*i*], *A*[*j*])
8. *j* = *j* – 1

Time complexity is *O*(*n*).

**8. (Problem8-2)**

Suppose that we have an array of *n* data records to sort and that the key of each record has the value 0 or 1. An algorithm for sorting such a set of records might possess some subset of the following three desirable characteristics:

(1) The algorithm runs in *O*(*n*) time.

(2) The algorithm is stable.

(3) The algorithm sorts in place, using no more than a constant amount of storage space in addition to the original array.

(a)(5%) Give an algorithm that satisfies criteria 1 and 2 above.

(b)(5%) Give an algorithm that satisfies criteria 2 and 3 above.

**Solution:**

1. Counting sort runs in *O*(*n*) time and is a stable sorting algorithm.
2. Insertion sort is an in place sorting algorithm. It is also stable. To see this consider *A*[*i*] = *A*[*j*] and *i* < *j*. Since *i* < *j*, *A*[*i*] will be considered first. *A*[*i*] will be added at the correct position (by shifting) into the sorted array *A*[1..*i*−1]. This will result in a sorted array *A*[1..*i*] containing the original *A*[*i*] at some position *k* ≤ *i*. So *A*[*i*] is now *A*[*k*]. When *A*[*j*] is considered, *A*[*j*] has to be shifted down into *A*[1..*j* − 1] of which *A*[1..*i*] is a subarray containing *A*[*k*] (originally *A*[*i*]). *A*[*j*] cannot bypass *A*[*k*] in the shifting process because *A*[*k*] = *A*[*j*]. Therefore, the original *A*[*i*] and the original *A*[*j*] will preserve their relative order.

**9. (Exercise9.3-7)**

Describe an *O*(*n*) algorithm that, given a set S of *n* distinct numbers and a positive integer *k* ≤ *n*, determines the *k* numbers in *S* that are closest to the median of *S*.

**Solution**:

Assume for simplicity that *n* is odd and *k* is even. If the set *S* was in sorted order, the median is in position *n*/2 and the *k* numbers in *S* that closest to the median are in positions (*n* − *k*)/2 through (*n* + *k*)/2. We first use linear time selection to find the (*n* − *k*)/2th, *n*/2th, and (*n* + *k*)/2th elements and then pass through the set *S* to find the numbers less than (*n* + *k*)/2th element, greater than the (*n*−*k*)/2th elements, and not equal to the *n*/2th elements. The algorithm takes *O* (*n*) time as we use linear time selection exactly three times and traverse the *n* numbers in *S* once.

**10. (Modified from Exercise 15.2-3)**

Use the substitution method to show that the solution to the following recurrence is

**Solution:**

We need to show that there exists a constant and an integer such that for all . Assume that there exist and such that such that Show that

We need to find a and an so that .

by definition.

then .

then

then

…

then

then

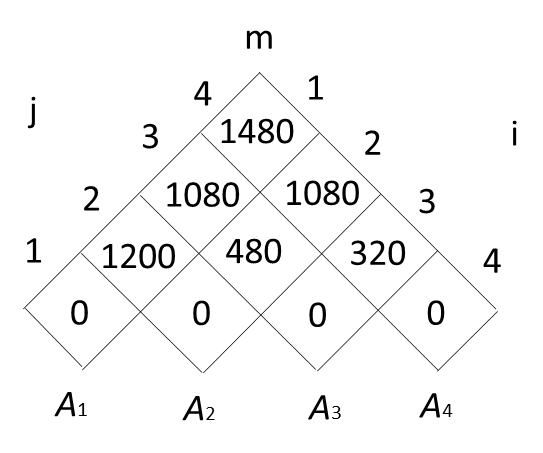
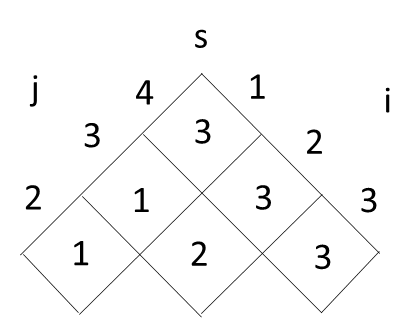
then

After *P*(11) the bound of *c* is increasing, then we can choose and , so that .

**11. (P.372)**

Please use the bottom-up approach of Dynamic Programming to find the optimal order, and its cost, for computing the product *A*1*A*2*A*3*A*4, where *A*1=(10x15), *A*2=(15x8), *A*3=(8x4), and *A*4=(4x10). (You need to show your answer with two-dimensional tables.)

**Solution:**

🡺 ((*A*1(*A*2*A*3))*A*4)

**12.**

What are the two key properties that an optimization problem must have in order for greedy algorithm to apply?

**Solution**:

1. Greedy-choice property
2. Optimal substructure

**13. (P.432)**

Construct a Huffman code for the following 6 characters: (Please show your algorithm time complexity and the computation steps in detail.)

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Character | a | b | c | d | e | f |
| Probability | 0.30 | 0.06 | 0.12 | 0.14 | 0.16 | 0.22 |

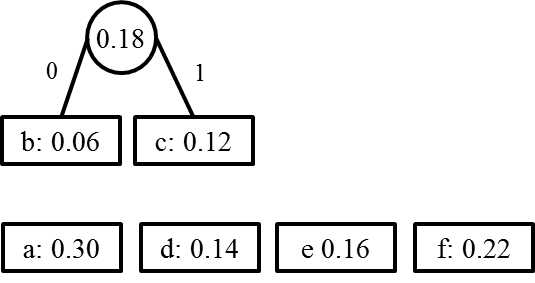
**Solution:**

The following graphs show the steps to construct Huffman code.

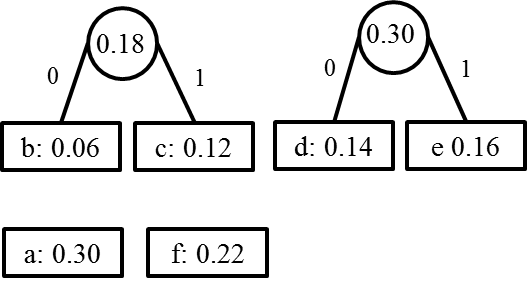
(1)



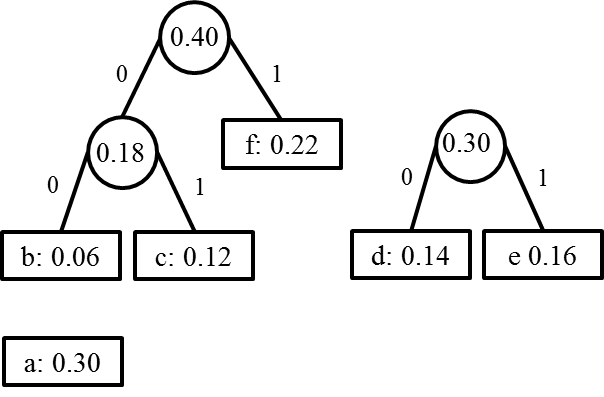
(2)



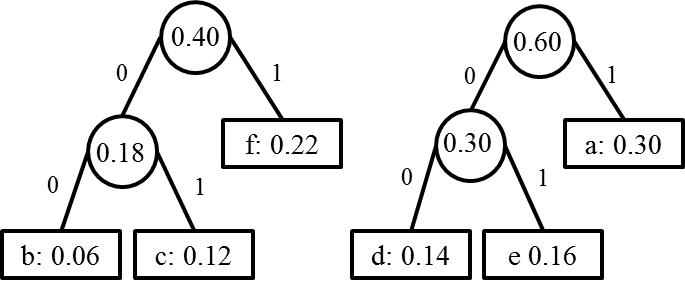
(3)



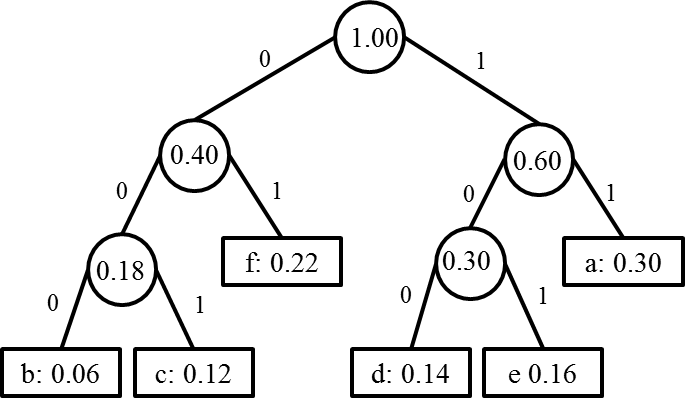
(4)



(5)



(6)



|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Character | a | b | c | d | e | f |
| Probability | 0.30 | 0.06 | 0.12 | 0.14 | 0.16 | 0.22 |
| Huffman  Code | 11 | 000 | 001 | 100 | 101 | 01 |

The time complexity of the Huffman algorithm is O(*n*log*n*). Using a heap to store the weight of each tree, each iteration requires O(log*n*) time to determine the cheapest weight and insert the new weight. There are O(*n*) iterations, one for each item.