Introduction to Algorithms (CLRS) - Personal Solutions

Morgan Bruce

June 22, 2022

Contents

Chapter 1

The Role of Algorithms in Computing

1.1 Algorithms

1.1-1 Give a real-world example that requires sorting or a real-world example that requires computing a convex hull.

Solution. Any sort of problem involving organizing probably has sorting. For example, ordering a set of people by any quantitative characteristic (height, weight, test scores, age, their one mile run time, etc.).

Convex hulls are used widely in graphical simulations, for example to detect collisions. □

1.1-2 Other than speed, what other measures of efficiency might one use in a real-world setting?

Solution. Computer memory. Programmer time (time to implement algorithm). Energy efficiency. □

1.1-3 Select a data structure that you have seen previously, and discuss its strengths and limitations.

Solution. Linked lists have the advantage of being extendible up to the memory limit of your computer. Disadvantage of needing to traverse entire list to find an element (or determine if one isn't there) □

1.1-4 How are the shortest-path and traveling-salesman problems given above similar? How are they different?

Solution. Both are problems on weighted graphs involving attempting to minimize some subset of the graph (a path or a cycle). However, the shortest-path problem makes no requirement to end at the starting point, whereas the traveling-salesman problem does. This makes the shortest-path problem in P (solved by several algorithms, such as Dijkstra's), and the traveling-salesman in NP-complete. □

1.1-5 Come up with a real-world problem in which only the best solution will do. Then come up with one in which a solution that is "approximately" the best is good enough.

Solution. A problem where only the best solution will do is one that requires a fully correct solution, like needing a sorted list for a directory. One where an approximate solution, as discussed earlier in the chapter, would be solving traveling-salesman. □

1.2 Algorithms as a technology

1.2-1 Give an example of an application that requires algorithmic content at the application level, and discuss the function of the algorithms involved.

Solution. Google Maps when giving directions to users uses algorithms to determine the quickest route, factoring in distance, tolls, predicted traffic at the time the user would be at a certain road, any reported accidents, etc. □

1.2-2 Suppose we are comparing implementations of insertion sort and merge sort on the same machine. For inputs of size n, insertion sort runes in $8n^2$ steps, while merge sort runs in $64n \lg n$ steps. For which values of n does insertion sort beat merge sort?

Solution. This is asking for what values of n we have that $8n^2 \leq 64n \lg n$. We can simplify this to $2^{n/8} \leq n$. Plugging in values yields the largest valid integer value of n being 43. □

1.2-3 What is the smallest value of n such that an algorithm whose running time is $100n^2$ runs faster than an algorithm whose running time is 2^n on the same machine?

Solution. This is asking what the smallest value of n such that $100n^2 \leq 2^n$ is, which is equivalent to when $n \leq \frac{2^{n/2}}{10}$. The value of n is 15. □

1.3 Problems

1-1 For each function $f(n)$ and time t in the following table, determine the largest size n of a problem that can be solved in time t, assuming that the algorithm to solve the problem takes $f(n)$ microseconds.

Solution.

	1 second	1 minute	1 hour	1 day	1 month	1 year	1 century
$\lg n$	2^{1e6}	2^{6e7}	$2^{7.2e9}$	$2^{8.64e10}$	$2^{2.59e12}$	$2^{3.15e13}$	$2^{3.15e15}$
$\sqrt{n}$	$1e12$	$3.6e15$	$1.29e19$	$7.46e21$	$6.71e24$	$9.67e26$	$9.67e30$
n	$1e6$	$6e7$	$3.6e9$	$8.64e10$	$2.59e12$	$3.1104e13$	$3.11e15$
$n \lg n$	62746	$2.80e6$	$1.33e8$	$2.75e9$	$7.18e10$	$7.8709e11$	$6.76e13$
n^2	1000	7745	60000	293938	1609968	5577096	55770960
n^3	100	391	1532	4420	13736	98169	455661
2^n	19	25	31	36	41	44	51
$n!$	9	11	12	13	15	17	18

□

Chapter 2

Getting Started

2.1 Insertion sort

2.1-1 Using Figure 2.2 as a model, illustrate the operation of `INSERTION-SORT` on the array $A = \langle 31, 41, 59, 26, 41, 58 \rangle$.

Solution. First pass: key set to 41. `A[1] = 31 < 41`, so the while loop breaks, and the second element stays put.

Second pass: key set to 59. Same steps as above.

Third pass: Key set to 26. `A[3] = 59 > 26`, so set `A[4] = 59`. `A[2] = 41 > 26`, so set `A[3] = 41`. `A[1] = 31 > 26`, so set `A[2] = 31`. Loop breaks, set `A[1] = 26`.

Fourth pass: Key set to 41. `A[4] = 59 > 41`, so set `A[5] = 59`. `A[3] = 41`, so the while loop breaks. Set `[4] = 41`.

Final pass: Key set to 58. `A[5] = 59 > 58`, so set `A[6] = 59`. `A[4] = 41 < 58`, so the while loop breaks. Set `[5] = 58`. The final sorted array is $\langle 26, 31, 41, 41, 58, 59 \rangle$. □

2.1-2 Rewrite the `INSERTION-SORT` procedure to sort into nonincreasing instead of nondecreasing order.

Solution. Rewrite the algorithm as follows.

REVERSE-INSERTION-SORT(A)

```
1 for j = 2 to A.length
2     key = A[j]
3     // Insert A[j] into the sorted sequence A[1..j-1].
4     i = j - 1
5     while i > 0 and A[i] < key // The only change
6         A[i + 1] = A[i]
7         i = i - 1
8     A[i + 1] = key
```

The implementation of this can be found in
`code/<language>/chapter02/reverse_insertion_sort/`. □

2.1-3 Consider the **searching problem**:

Input: A sequence of n numbers $A = \langle a_1, a_2, \ldots, a_n \rangle$ and a value v.

Output: An index i such that $v = A[i]$ or the special value `NIL` if v does not appear in A.

Write pseudocode for **linear search**, which scans through the sequence, looking for v. Using a loop invariant, prove that your algorithm is correct. Make sure that your loop invariant fulfills the three necessary properties.

Solution. The algorithm is as follows.

LINEAR-SEARCH(A,v)

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

The loop invariant can be stated as follows: At the start of each iteration of the for loop, the subarray $A[1..i-1]$ does not contain the value v. We will show this invariant has the three necessary properties to prove the correctness of the algorithm. Prior to the first loop, the subarray $A[1..0]$ is an empty array, which clearly does not contain v. Suppose the invariant is true after loop i, that is, $A[1..i]$ does not contain v. Then on loop $i+1$, if $A[i+1] == v$, the algorithm will exit, returning $i+1$. If the algorithm does not exit, the loop iteration completes and thus $A[1..i+1]$ does not contain v. When the loop terminates without the algorithm prematurely returning, that means that v is not contained in $A[1..n]$, and thus `NIL` is returned. So the loop invariant fulfills the necessary properties to prove correctness of the algorithm. □

2.1-4 Consider the problem of adding two n-bit binary integers, stored in two n-element arrays A and B. The sum of the two integers should be stored in binary form in an $(n+1)$-element array C. State the problem formally and write pseudocode for adding the two integers.

Solution. Define the problems as follows:

Input: Two sequences A, B representing binary numbers, where the sequences are length n and contain only 0 or 1. The endianness of A, B should be defined, as it affects the algorithm to solve it. We will assume big-endian.

Output: A length $n+1$ sequence representing the binary sum $A+B$.

Write the algorithm solving the problem as follows:

ADD(A,B)

```
1  carry = 0
2  for i = A.length to 1
3      sum = A[i] + B[i] + carry // Binary addition
4      if sum % 10 == 1 // Sum either 01 or 11
5          S[i+1] = 1
6      if sum >= 10 // Sum either 10 or 11
7          carry = 1
8      else
9          carry = 0
10 S[0] = carry
11 return S
```

□

2.2 Analyzing algorithms

2.2-1 Express the function $n^3/1000 - 100n^2 - 100n + 3$ in terms of Θ-notation.

Solution. Considering the informal definition of Θ-notation in this chapter, we simply observe that the $n^3/1000$ term is the leading term and dominates for large n, and drop the coefficient, expressing the function as $\Theta(n^3)$. □

2.2-2 Consider sorting n numbers stored in array A by first finding the smallest element of A and exchanging it with the element in $A[1]$. Then find the second smallest element of A, and exchange it with $A[2]$. Continue in this

manner for the first $n-1$ elements of A. Write pseudocode for this algorithm, which is known as **selection sort**. What loop invariant does this algorithm maintain? Why does it need to run for only the first $n-1$ elements, rather than for all n elements? Give the best-case and worst-case running times of selection sort in Θ-notation.

Solution.

SELECTION-SORT(A)

```
1  for j = 1 to A.length - 1
2      current_min = stored_val = A[j]
3      stored_index = j
4      for i = j + 1 to A.length
5          if A[i] < current_min
6              current_min = A[i]
7              stored_index = i
8      A[j] = current_min
9      A[stored_index] = stored_val
```

See `code/<language>/chapter02/selection_sort/` for implementation of this algorithm.

The loop invariant that the outer loop maintains is that the subarray $A[1..j]$ is sorted. Informally, this holds because the minimum element is selected to the first index, the second smallest element is selected to the second index, and so forth through the first j elements. The loop only needs to run for the first $n-1$ elements because after $n-1$ iterations, the largest element is necessarily in the final index, as the $n-i$th largest elements are already all fixed to the ith indices. The best case runtime is $\Theta(n^2)$, since even with an input list that is already sorted, there is no way to early stop either for loop, as all elements need to be checked to confirm each candidate element is the current minimum. The worst case is equivalently $\Theta(n^2)$, as the same checks as in the best case are still performed, but the current minimum and stored index are updated on each increment of the i index (in Θ notation, this only adds a linear term which is already accounted for by the nested loop). $\square$

2.2-3 Consider linear search again (see Exercise 2.1-3). How many elements of the input sequence need to be checked on the average, assuming that the element being searched for is equally likely to be any element in the array? How about in the worst case? What are the average-case and worst-case running times of linear search in Θ notation? Justify your answers.

Solution. Assume the element to be checked is present in the array (otherwise no information is given on the ratio of present elements to non-present elements, and an average case is impossible to derive). The average case is equal to $\sum_i^n p(i) \cdot i$, where i is equal to each index. Since every index has equal probability to have the element, $p(i) = 1/n$. Thus, we have $\sum_i^n p(i) \cdot i = n(n+1)/(2n) = (n+1)/2$. In Θ notation, the average case is $\Theta(n)$ (strip away the coefficients / constants on the previous solution), and the worst case is $\Theta(n)$ (check n indices). □

2.2-4 How can we modify almost any algorithm to have a good best-case running time?

Solution. Have an initial check to verify if the input is the solution (e.g., if the input array to a sorting algorithm is sorted). □

2.3 Designing algorithms

2.3-1 Using Figure 2.4 as a model, illustrate the operation of merge sort on the array $A = \langle 3, 41, 52, 26, 38, 57, 9, 49 \rangle$.

Solution. The initial array of $\langle 3, 41, 52, 26, 38, 57, 9, 49 \rangle$ gets split up into arrays of length 1. Then, the arrays are paired off in order and merged together, to form $\langle 3, 41, 26, 52, 38, 57, 9, 49 \rangle$. Then, pairs are combined into 4 to give $\langle 3, 26, 41, 52, 9, 38, 49, 57 \rangle$. Then, the two sets of 4 are merged to get the final sorted array of $\langle 3, 9, 26, 38, 41, 49, 52, 57 \rangle$ □

2.3-2 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 into A and then copying the remainder of the other array back into A.

Solution.

MERGE(A, p, q, r)

```
1  n_1 = q - p + 1
2  n_2 = r - q
3  let L[1..n_1] and R[1..n_2] be new arrays
4  for i = 1 to n_1
5      L[i] = A[p + i - 1]
6  for j = 1 to n_2
7      R[j] = A[q + j]
8  i = 1
9  j = 1
10 k = p
11 while i <= n_1 and j <= n_2
12     if L[i] <= R[j]
13         A[k] = L[i]
14         i = i + 1
15         k = k + 1
16     else
17         A[k] = R[j]
18         j = j + 1
19         k = k + 1
```

□

2.3-3 Use mathematical induction to show that when n is an exact power of 2, the solution of the recurrence

$$T(n) = \begin{cases} 2 & \text{if } n = 2, \\ 2T(n/2) + n & \text{if } n = 2^k, \text{ for } k > 1 \end{cases}$$

is $T(n) = n \lg n$.

Solution. Let $k = 1$. Then by definition of T, $T(2) = 2 = 2 \lg 2$.

Suppose the hypothesis is true for $k = m$. Let $k = m + 1$. We find that $T(2^{m+1}) = 2T(2^{m+1}/2) + 2^{m+1} = 2T(2^m) + 2^{m+1} = 2 \cdot 2^m \lg 2^m + 2^{m+1} = m \cdot 2^{m+1} + 2^{m+1} = (m+1)2^{m+1} = 2^{m+1} \lg 2^{m+1}$. This proves the inductive hypothesis, and thus the result is true for all $k \in \mathbb{Z}^+$, or when n is a power of 2. □

2.3-4 We can express insertion sort as a recursive procedure as follows. In order to sort $A[1..n]$, we recursively sort $A[1..n-1]$ and then insert $A[n]$ into

the sorted array $A[1..n-1]$. Write a recurrence for the worst-case running time of this recursive version of insertion sort.

Solution.

$$T(n) = T(n-1) + (n-1)$$

This follows from considering the worst case of inserting $A[n]$ into sorted $A[1..n-1]$: needing to swap $A[n]$ with every index until it is at the first index. This yields $n-1$ swaps, in addition to all swaps to sort $A[1..n-1]$. Solving the recurrence is done by summing all integers from 1 to $n-1$, yielding $\Theta(n^2)$. □

2.3-5 Referring back to the searching problem (see Exercise 2.1-3), observe that if the sequence A is sorted, we can check the midpoint of the sequence against v and eliminate half of the sequence from further consideration. The **binary search** algorithm repeats this procedure, halving the size of the remaining portion of the sequence each time. Write pseudocode, either iterative or recursive, for binary search. Argue that the worst-case running time of binary search is $\Theta(\lg n)$.

Solution.

BINARY-SEARCH(A, v, p, q)

```
1 if p == q and A[p] != v
2     return NIL
3 midpoint = floor((p + q) / 2)
4 if A[midpoint] == v
5     return midpoint
6 if A[midpoint] > v
7     return BINARY-SEARCH(A, v, p, midpoint)
8 return BINARY-SEARCH(a, v, midpoint, q)
```

The worst case is when the element is not in the list, in which case the array size must halve until it is one element large and the element is not the value to search for, returning `NIL`. Halving A this way to a size of one element takes $\lg n$ steps, so the procedure takes $\Theta(\lg n)$ time. □

2.3-6 Observe that the **while** loop of lines 5-7 of the `INSERTION SORT` procedure in Section 2.1 uses a linear search to scan (backward) through the sorted subarray $A[1..j-1]$. Can we use a binary search (see Exercise 2.3-5)

instead to improve the overall worst-case running time of insertion sort to $\Theta(n \lg n)$?

Solution. It could be implemented, but it would not improve the running time to $\Theta(n \lg n)$. Once the index that $A[j]$ is to be inserted in is found, say index i, the elements of the subarray $A[i..j-1]$ still need to be shifted into $A[i+1..j]$. This will involve a nested loop that, in worst case, will still iterate over the entire subarray $A[1..j-1]$, so the running time will still be $\Theta(n \lg n)$. □

2.3-7* Describe a $\Theta(n \lg n)$-time algorithm that, given a set S of n integers and another integer x, determines whether or not there exist two elements in S whose sum is exactly x.

Solution. Name this procedure `PAIR-TO-SUM`. The pseudocode for this procedure is given by the following.

PAIR-TO-SUM(S, x)

```
1  S = MERGE-SORT(S)
2  n = S.size
3  for i=1 to n
4      value = x - S[i]
5      subarray = Union(S[1..i-1], S[i+1..n])
6      j = BINARY-SEARCH(subarray, value, 1, n)
7      if j != NIL
8          return i, j
9  return NIL
```

First, this procedure calls `MERGE-SORT`, which is $\Theta(n \lg n)$. Then, this procedure iterates over the n elements of S, binary searching for the unique element that would sum to x. Binary search is $\Theta(\lg n)$, and it is called n times in worst case, so this procedure is $\Theta(n \lg n)$. □

2.4 Problems

2-1 **Insertion sort on small arrays in merge sort**

Although merge sort runs in $\Theta(n \lg n)$ worst-case time and insertion sort runs in $\Theta(n^2)$ worst-case time, the constant factors in insertion sort can

make it faster in practice for small problem sizes on many machines. Thus, it makes sense to **coarsen** the leaves of the recursion by using insertion sort within the merge sort when subproblems become sufficiently small. Consider a modification to merge sort in which n/k sublists of length k are sorted using insertion sort and then merged using the standard merging mechanism, where k is a value to be determined.

(a) Show that insertion sort can sort the n/k sublists, each of length k, in $\Theta(nk)$ worst-case time.

(b) Show how to merge sublists in $\Theta(n \lg(n/k))$ worst-case time.

(c) Given that the modified algorithm runs in $\Theta(nk + n \lg(n/k))$ worst-case time, what is the largest value of k as a function of n for which the modified algorithm has the same running time as standard merge sort, in terms of Θ-notation?

(d) How should we choose k in practice?

Solution.

(a) There exist n/k sublists of length k. With insertion sort, each sublist takes $\Theta(k^2)$ time to sort. Since we do this $\Theta(k^2)$ operation n/k times, we multiply to find the worst time for insertion sort to sort all sublists is $\Theta(nk)$.

(b) Perform the same steps as in normal merge sort, but since the branching tree of sublists is only $\lg(n/k)$ iterations deep instead of $\lg n$, merging the sublists takes $\Theta(n \lg(n/k))$ time.

(c) The threshold is $k = \lg n$, at which point the first part of the sum will begin to dominate as k gets larger.

(d) We can choose some constant value which isn't too large, for instance $k = 64$.

□

***2-2* Correctness of bubblesort**

Bubblesort is a popular, but inefficient, sorting algorithm. It works by repeatedly swapping adjacent elements that are out of order.

```
BUBBLESORT(A)
1 for i = 1 to A.length - 1
2     for j = A.length to i + 1
3         if A[j] < A[j - 1]
4             exchange A[j] with A[j - 1]
```

(a) Let A' denote the output of BUBBLESORT(A). To prove that BUBBLESORT is correct, we need to prove that it terminates and that

$$A'[1] \leq A'[2] \leq \cdots \leq A'[n], \tag{2.1}$$

where $n = A.length$. In order to show that BUBBLESORT actually sorts, what else do we need to prove?

The next two parts will prove inequality (2.1).

(b) State precisely a loop invariant for the **for** loop in lines 2-4, and prove that this loop invariant holds. Your proof should use the structure of the loop invariant proof presented in this chapter.

(c) Using the termination condition of the loop invariant proved in part (b), state a loop invariant for the **for** loop in lines 1-4 that will allow you to prove inequality (2.1). Your proof should use the structure of the loop invariant proof presented in this chapter.

(d) What is the worst-case running time of bubblesort? How does it compare to the running time of insertion sort?

Solution. □

2-3 Correctness of Horner's rule

The following code fragment implements Horner's rule for evaluating a polynomial

$$\begin{aligned} P(x) &= \sum_{k=0}^{n} a_k x^k \\ &= a_0 + x(a_1 + x(a_2 + \cdots + x(a_{n-1} + x a_n) \cdots)), \end{aligned}$$

Given the coefficients $a_0, a_1, \ldots, a_n$ and a value for x:

```
1  y = 0
2  for i = n downto 0
3      y = a_i + x * y
```

(a) In terms of Θ-notation, what is the running time of this code fragment for Horner's rule?

(b) Write pseudocode to implement the naive polynomial-evaluation algorithm that computes each term of the polynomial from scratch. What is the running time of this algorithm? How does it compare to Horner's rule?

(c) Consider the following loop invariant: At the start of each iteration of the **for** loop of lines 2-3,

$$y = \sum_{k=0}^{n-(i+1)} a_{k+i+1} x^k .$$

Interpret a summation with no terms as equaling 0. Following the structure of the loop invariant proof presented in this chapter, use this loop invariant to show that, at termination, $y = \sum_{k=0}^{n} a_k x^k$.

(d) Conclude by arguing that the given code fragment correctly evaluates a polynomial characterized by the coefficients $a_0, a_1, \ldots, a_n$.

Solution. □

***2-4* Inversions**

Let $A[1..n]$ be an array of n distinct numbers. If $i < j$ and $A[i] > A[j]$, then the pair (i, j) is called an **inversion** of A.

(a) List the five inversions of the array $\langle 2, 3, 8, 6, 1 \rangle$.

(b) What array with elements from the set $\{1, 2, \ldots, n\}$ has the most inversions? How many does it have?

(c) What is the relationship between the running time of insertion sort and the number of inversions in the input array? Justify your answer.

(d) Give an algorithm that determines the number of inversions in any permutation on n elements in $\Theta(n \lg n)$ worst-case time. (*Hint:* Modify merge sort.)

Solution.

(a) The inversions are $(1, 5), (2, 5), (3, 5), (4, 5), (3, 4)$.

(b) The array with the most inversions is the reverse sorted array: $[n, n - 1, \ldots, 2, 1]$. It has $n - 1$ inversions for the first element, $n - 2$ for the second, and so forth, for a total of $1 + 2 + \cdots + n - 1 = n(n - 1)/2$.

(c) Insertion sort works by iterating through each element in the array, undoing every inversion an element has with elements appearing prior to it in the array. Thus, if $I(A)$ is a formula for the number of inversions in an array, the running time of insertion sort is equal to $I(A) + \Theta(n)$, where the code to maintain the state of the procedure is $\Theta(n)$.

(d) Take merge sort, and add to a counter for every element added to the array from the right array before all elements from the left are depleted. Return this count for every subprocess, and add these counts together to get the total number of inversions.

□