
Selected Solutions for Chapter 2: Getting Started

Solution to Exercise 2.2-2

```
SELECTION-SORT(A)
  n = A.length
  for j = 1 to n - 1
    smallest = j
    for i = j + 1 to n
      if A[i] < A[smallest]
        smallest = i
    exchange A[j] with A[smallest]
```

The algorithm maintains the loop invariant that at the start of each iteration of the outer **for** loop, the subarray $A[1..j-1]$ consists of the $j-1$ smallest elements in the array $A[1..n]$, and this subarray is in sorted order. After the first $n-1$ elements, the subarray $A[1..n-1]$ contains the smallest $n-1$ elements, sorted, and therefore element $A[n]$ must be the largest element.

The running time of the algorithm is $\Theta(n^2)$ for all cases.

Solution to Exercise 2.2-4

Modify the algorithm so it tests whether the input satisfies some special-case condition and, if it does, output a pre-computed answer. The best-case running time is generally not a good measure of an algorithm.

Solution to Exercise 2.3-5

Procedure **BINARY-SEARCH** takes a sorted array A , a value v , and a range $[low..high]$ of the array, in which we search for the value v . The procedure compares v to the array entry at the midpoint of the range and decides to eliminate half the range from further consideration. We give both iterative and recursive versions, each of which returns either an index i such that $A[i] = v$, or **NIL** if no entry of

$A[\text{low}.. \text{high}]$ contains the value v . The initial call to either version should have the parameters $A, v, 1, n$.

ITERATIVE-BINARY-SEARCH($A, v, \text{low}, \text{high}$)

```

while  $\text{low} \leq \text{high}$ 
     $\text{mid} = \lfloor (\text{low} + \text{high}) / 2 \rfloor$ 
    if  $v == A[\text{mid}]$ 
        return  $\text{mid}$ 
    elseif  $v > A[\text{mid}]$ 
         $\text{low} = \text{mid} + 1$ 
    else  $\text{high} = \text{mid} - 1$ 
return NIL

```

RECURSIVE-BINARY-SEARCH($A, v, \text{low}, \text{high}$)

```

if  $\text{low} > \text{high}$ 
    return NIL
 $\text{mid} = \lfloor (\text{low} + \text{high}) / 2 \rfloor$ 
if  $v == A[\text{mid}]$ 
    return  $\text{mid}$ 
elseif  $v > A[\text{mid}]$ 
    return RECURSIVE-BINARY-SEARCH( $A, v, \text{mid} + 1, \text{high}$ )
else return RECURSIVE-BINARY-SEARCH( $A, v, \text{low}, \text{mid} - 1$ )

```

Both procedures terminate the search unsuccessfully when the range is empty (i.e., $\text{low} > \text{high}$) and terminate it successfully if the value v has been found. Based on the comparison of v to the middle element in the searched range, the search continues with the range halved. The recurrence for these procedures is therefore $T(n) = T(n/2) + \Theta(1)$, whose solution is $T(n) = \Theta(\lg n)$.

Solution to Problem 2-4

- a. The inversions are $(1, 5), (2, 5), (3, 4), (3, 5), (4, 5)$. (Remember that inversions are specified by indices rather than by the values in the array.)
- b. The array with elements from $\{1, 2, \dots, n\}$ with the most inversions is $\langle n, n-1, n-2, \dots, 2, 1 \rangle$. For all $1 \leq i < j \leq n$, there is an inversion (i, j) . The number of such inversions is $\binom{n}{2} = n(n-1)/2$.
- c. Suppose that the array A starts out with an inversion (k, j) . Then $k < j$ and $A[k] > A[j]$. At the time that the outer **for** loop of lines 1–8 sets $\text{key} = A[j]$, the value that started in $A[k]$ is still somewhere to the left of $A[j]$. That is, it's in $A[i]$, where $1 \leq i < j$, and so the inversion has become (i, j) . Some iteration of the **while** loop of lines 5–7 moves $A[i]$ one position to the right. Line 8 will eventually drop key to the left of this element, thus eliminating the inversion. Because line 5 moves only elements that are less than key , it moves only elements that correspond to inversions. In other words, each iteration of the **while** loop of lines 5–7 corresponds to the elimination of one inversion.

the front of the list and taking halls from the front of the list—so that a new hall doesn't come to the front and get chosen if there are previously-used halls.)

This guarantees that the algorithm uses as few lecture halls as possible: The algorithm will terminate with a schedule requiring $m \leq n$ lecture halls. Let activity i be the first activity scheduled in lecture hall m . The reason that i was put in the m th lecture hall is that the first $m - 1$ lecture halls were busy at time s_i . So at this time there are m activities occurring simultaneously. Therefore any schedule must use at least m lecture halls, so the schedule returned by the algorithm is optimal.

Run time:

- Sort the $2n$ activity-starts/activity-ends events. (In the sorted order, an activity-ending event should precede an activity-starting event that is at the same time.) $O(n \lg n)$ time for arbitrary times, possibly $O(n)$ if the times are restricted (e.g., to small integers).
- Process the events in $O(n)$ time: Scan the $2n$ events, doing $O(1)$ work for each (moving a hall from one list to the other and possibly associating an activity with it).

Total: $O(n + \text{time to sort})$

Solution to Exercise 16.2-2

The solution is based on the optimal-substructure observation in the text: Let i be the highest-numbered item in an optimal solution S for W pounds and items $1, \dots, n$. Then $S' = S - \{i\}$ must be an optimal solution for $W - w_i$ pounds and items $1, \dots, i - 1$, and the value of the solution S is v_i plus the value of the subproblem solution S' .

We can express this relationship in the following formula: Define $c[i, w]$ to be the value of the solution for items $1, \dots, i$ and maximum weight w . Then

$$c[i, w] = \begin{cases} 0 & \text{if } i = 0 \text{ or } w = 0, \\ c[i - 1, w] & \text{if } w_i > w, \\ \max(v_i + c[i - 1, w - w_i], c[i - 1, w]) & \text{if } i > 0 \text{ and } w \geq w_i. \end{cases}$$

The last case says that the value of a solution for i items either includes item i , in which case it is v_i plus a subproblem solution for $i - 1$ items and the weight excluding w_i , or doesn't include item i , in which case it is a subproblem solution for $i - 1$ items and the same weight. That is, if the thief picks item i , he takes v_i value, and he can choose from items $1, \dots, i - 1$ up to the weight limit $w - w_i$, and get $c[i - 1, w - w_i]$ additional value. On the other hand, if he decides not to take item i , he can choose from items $1, \dots, i - 1$ up to the weight limit w , and get $c[i - 1, w]$ value. The better of these two choices should be made.

The algorithm takes as inputs the maximum weight W , the number of items n , and the two sequences $v = \langle v_1, v_2, \dots, v_n \rangle$ and $w = \langle w_1, w_2, \dots, w_n \rangle$. It stores the $c[i, j]$ values in a table $c[0..n, 0..W]$ whose entries are computed in row-major order. (That is, the first row of c is filled in from left to right, then the second row,

b. Let f be the maximum flow before reducing $c(u, v)$.

If $f(u, v) = 0$, we don't need to do anything.

If $f(u, v) > 0$, we will need to update the maximum flow. Assume from now on that $f(u, v) > 0$, which in turn implies that $f(u, v) \geq 1$.

Define $f'(x, y) = f(x, y)$ for all $x, y \in V$, except that $f'(u, v) = f(u, v) - 1$. Although f' obeys all capacity constraints, even after $c(u, v)$ has been reduced, it is not a legal flow, as it violates flow conservation at u (unless $u = s$) and v (unless $v = t$). f' has one more unit of flow entering u than leaving u , and it has one more unit of flow leaving v than entering v .

The idea is to try to reroute this unit of flow so that it goes out of u and into v via some other path. If that is not possible, we must reduce the flow from s to u and from v to t by one unit.

Look for an augmenting path from u to v (note: *not* from s to t).

- If there is such a path, augment the flow along that path.
- If there is no such path, reduce the flow from s to u by augmenting the flow from u to s . That is, find an augmenting path $u \rightsquigarrow s$ and augment the flow along that path. (There definitely is such a path, because there is flow from s to u .) Similarly, reduce the flow from v to t by finding an augmenting path $t \rightsquigarrow v$ and augmenting the flow along that path.

Time

$O(V + E) = O(E)$ if we find the paths with either DFS or BFS.