**Chapter 10**

**Algorithm Efficiency and Sorting**

**1** **a** *O(n)*

**b** *O(n)*

**c** *O(n)*

**d** *O(n)*

**e** *O(1)*

**f** *O(n)* We assume that the list is singly linked. Worst case: the entire list must be traversed first.

**g** *O(log n)*

**h** *O(n log n)*

**2** If we perform a pass on the elements of the array, and no swaps were needed, then we know we are done sorting.

**3a** Adding an item to a stack.

Array implementation (pp. 365-367): *O(1).* We check to see that the array is not full. If it is we throw an exception. If not full, we initialize one of the elements of the array and increment the array index. All of these operations take a constant amount of time. However, if the implementation allowed for the possibility of resizing the array when the stack is full, then we would need to move all the values into the new larger array, and the complexity would be *O(n)* due to the loop.

Reference implementation (pp. 368-369): *O(1).* We create a new Node object and make it the top of the stack. These operations take a constant amount of time.

List implementation (pp. 369-371): *O(1).* Adding an item to the 0 index of a list is a special case (for example, see page 267). Again, adding to the front of a list requires only constant time.

**3b** Adding an item to a queue.

Array implementation (pp. 422-424): *O(1).* This is analogous to the array implementation of the stack. If the

array is full, we throw an exception. Otherwise, we perform a constant amount of work to initialize one of the

array elements and update the array index and the count of elements. On the other hand, if we wanted to allow

the queue to grow beyond its current size, *O(n)* time would be required to populate the values of the new array.

Reference implementation (pp. 418-419): O(1). All operations required for enqueue( ) require constant time.

First, we determine if the queue is empty. Then we update references.

List implermentation (pp. 425-426): *O(n).* The enqueue( ) operation calls the list add( ) operation and passes it

an index value. We have to traverse the list until we find the place to insert the element in the list.

**4** 1000 5 10 20 30 40 50 60

**5** The algorithm is of order *O(n2).* The outer and middle loops each have n iterations, but the inner loop has a constant (exactly 9) iterations. As n increases, the number of iterations of the inner loop does not change.

**6** The outer loop makes *n+*1 comparisons. The first *n* of them precede another pass through the course of the for loop and the last one fails the test and forces the loop to stop.

For each of the *n* passes through the loop, *j*+1 comparisons are made by the test for the while loop, *J* being the value of the index for the for loop, i.e.: *j* = [0, ... , *n*-1]. Within this while loop, an additional comparison is made by the if statement so 2\**j* + 1 total comparisons are made by the inner while loop.

Since *J* increases from 0 to *n*-1, the overall total of comparisons is:

[2\*(0) + 1] + [2\*(1) + 1] + ... + [2\*(*n*-1) + 1] + 1

the last being the comparison that bumps us out of the outer for loop. Thus the total is:

**7** Selection sort is moderately faster than Insertion sort. This is due to the constant multiple of the largest term in the count of comparisons and data moves each algorithm performs. As determined in the text, selection sort takes *n*2/2 + 5\**n*/2 - 3 steps for the worst case while insertion sort takes *n*2 + *n* - 2 steps. For large *n*, this confers a 2:1 advantage for selection sort.

**8** Let *c* = *c*[*n*] + *c*[*n*-1] + ... + *c*[0]. Then, *f(x)* <= *c*\**xn* and is thus *O*(*xn*).

**9** Let *a* > 1 and *b* > 1. To show: *f*(*n*) is *O*(loga *n*) => *f*(*n*) is *O*(logb *n*).

Define *c* = 1/(logb *a*). Then, *f*(*N*) is *O*(loga *n*) = *c* \* *O*(logb *n*) = *O*(logb *n*).

The proof is symmetrical for *f*(*n*) is *O*(logb *n*) => *f*(*n*) is *O*(loga *n*).

**10** Any operations outside of loops contribute a constant amount of time and would not affect the overall behavior of the selection sort for a large *n*.

**11** Apply insertion sort to: 80 40 25 20 30 60 in ascending order.

|  |  |
| --- | --- |
| array | Action |
| 80 40 25 20 30 60 | Initial array |
| 80 80 25 20 30 60 | Save 40 and shift 80 |
| 40 80 25 20 30 60 | Insert 40 |
| 40 40 80 20 30 60 | Save 25 and shift 40, 80 |
| 25 40 80 20 30 60 | Insert 25 |
| 25 25 40 80 30 60 | Save 20 and shift 25, 40, 80 |
| 20 25 40 80 30 60 | Insert 20 |
| 20 25 40 40 80 60 | Save 30 and shift 40, 80 |
| 20 25 30 40 80 60 | Insert 30 |
| 20 25 30 40 80 80 | Save 60 and shift 80 |
| 20 25 30 40 60 80 | Insert 60; final array |

**12** Selection sort for the array 8 11 23 1 20 33 into ascending order.

8 11 23 1 20 33 The largest element is already in correct position.

8 11 23 1 20 33 The largest among the first 5 needs to swap with 20.

8 11 20 1 23 33 The largest among the first 4 needs to swap with 1.

8 11 1 20 23 33 The largest among the first 3 needs to swap with 1.

8 1 11 20 23 33 The largest among the first 2 needs to swap with 1.

1 8 11 20 23 33 Done.

**13** Bubblesort on the array 10 12 23 34 5 to put in descending order.

Pass 1: 4 comparisons required

10 12 23 34 5

12 10 23 34 5

12 23 10 34 5

12 23 34 10 5

12 23 34 10 5 Now the 5 is in the correct position.

Pass 2: 3 comparisons required

12 23 34 10 5

23 12 34 10 5

23 34 12 10 5

23 34 12 10 5 Now the 10 is also in the correct position.

Pass 3: 2 comparisons required

23 34 12 10 5

34 23 12 10 5

34 23 12 10 5 Now the 12 is also in the correct position.

Pass 4: 1 comparison required (We have to execute this pass because pass 3 encountered a swap.)

34 23 12 10 5

34 23 12 10 5 The 23 and 34 are in the correct positions, and we are done.

**14a** Apply (ascending versions of) selection sort, bubble sort and insertion sort to sorted and inverse sorted arrays:

Inverse sorted array: 9 7 5 3 1

i selection sort

|  |  |
| --- | --- |
| Action | array |
| initial array | 9 7 5 3 1 |
| first swap | 1 7 5 3 9 |
| second swap | 1 3 5 7 9 |
| third swap not needed | 1 3 5 7 9 |
| fourth swap not needed | 1 3 5 7 9 |

ii. bubble sort

|  |  |
| --- | --- |
| Action | array |
| initial array | 9 7 5 3 1 |
| pass 1 | 9 7 5 3 1 |
|  | 7 9 5 3 1 |
|  | 7 5 9 3 1 |
|  | 7 5 3 9 1 |
|  | 7 5 3 1 9 |
| pass 2 | 7 5 3 1 9 |
|  | 5 7 3 1 9 |
|  | 5 3 7 1 9 |
|  | 5 3 1 7 9 |
| pass 3 | 5 3 1 7 9 |
|  | 3 5 1 7 9 |
|  | 3 1 5 7 9 |
| pass 4 | 3 1 5 7 9 |
|  | 1 3 5 7 9 |

iii. insertion sort

|  |  |
| --- | --- |
| Action | array |
| initial array | 9 7 5 3 1 |
| save 7 and shift 9 | 9 9 5 3 1 |
| insert 7 | 7 9 5 3 1 |
| save 5 and shift 7, 9 | 7 7 9 3 1 |
| insert 5 | 5 7 9 3 1 |
| save 3 and shift 5, 7, 9 | 5 5 7 9 1 |
| insert 3 | 3 5 7 9 1 |
| save 1 and shift 3, 5, 7, 9 | 3 3 5 7 9 |
| insert 1 | 1 3 5 7 9 |

**14b** Sorted array: 1 3 5 7 9

i selection sort

|  |  |
| --- | --- |
| Action | array |
| initial array | 1 3 5 7 9 |
| first swap not needed | 1 3 5 7 9 |
| second swap not needed | 1 3 5 7 9 |
| third swap not needed | 1 3 5 7 9 |
| fourth swap not needed | 1 3 5 7 9 |

ii. bubble sort

|  |  |
| --- | --- |
| Action | array |
| initial array | 1 3 5 7 9 |
| pass 1 | 1 3 5 7 9 |
|  | 1 3 5 7 9 |
|  | 1 3 5 7 9 |
|  | 1 3 5 7 9 |

1. insertion sort

|  |  |
| --- | --- |
| Action | Array |
| initial array | 1 3 5 7 9 |
| copy 1 on itself | 1 3 5 7 9 |
| copy 3 on itself | 1 3 5 7 9 |
| copy 5 on itself | 1 3 5 7 9 |
| copy 7 on itself | 1 3 5 7 9 |

**15a** If there are n elements in the array, then in the worst case of bubble sort, we must do n – 1 comparisons on the first pass, n – 2 comparisons on the second pass, etc. until the (n – 1) pass when we do just 1 comparison.

The total number of comparisons is



And in this case, n = 100, so the number of comparisions is 100(99)/2 = 4950.

**15b** In the best case of bubble sort, the array is already sorted, so we only make the first pass, which requires n – 1 comparisons. If n = 100, the number of comparisons is 99.

**16** void swap(Object x, Object y) // standard swap method

{

Object temp = x;

x = y;

y = temp;

} // end swap

int indexOfLargest(Object a, int size)

{

int max = 0;

for (int cur=1; cur<size; cur++)

{

if(a[cur].key > a[max].key)

max = cur;

} // end for loop

return max;

} // end indexOfLargest

void selectionSort(Object inA[], int last)

{

int current;

if (last > 0)

{ current = indexOfLargest(inA, last);

swap (inA[current], inA[last]);

selectionSort (inA, last-1);

} // end if

} // end selectionSort

A recursive version of bubbleSort:

void bubble(Object a[], int n, int index, boolean sorted)

// helping method

{ if(n > index+1)

{ int nextIndex = index + 1;

if(a[index] > a[nextIndex])

{ swap(a[index], a[nextIndex]);

sorted = false;

} // end if

bubble(a, n, index+1, sorted);

} // end if

} // end bubble

void bubbleSort(Object a[], int n)

{ if(n > 1)

{ boolean sorted = true;

bubble(a, n, 0, sorted);

if(!sorted)

bubbleSort(a, n-1);

} // end if

} // end bubbleSort

A recursive version of insertionSort. Here the sorted region is at the end rather than at the beginning of the array.

void shift(Object a[], Object nextItem, int loc)

{ if((loc > 0) && (a[loc-1] > nextItem))

{ a[loc] = a[loc-1];

loc--;

shift(a, nextItem, loc);

} // end if

} // end shift

void insertionSort(Object a[], int unsorted, int n)

// Preconditions: unsorted = 1, n = size of a.

// Postcondition: a is sorted in ascending order.

{ if(unsorted < n)

{ Object nextItem = a[unsorted];

int loc = unsorted;

shift(a, nextItem, loc);

a[loc] = nextItem;

insertionSort(a, unsorted+1, n);

} // end if

} // end insertionSort