4-5

a. Assume there are 3 chips A, B and C. Since at least 2 chips are bad, there are four possibilities,
   – All chips are bad.
   – A is good.
   – B is good.
   – C is good.
If the good chip exists, then it must say “the other is bad” in any pairwise test, since there is at most one good chip. When the bad chips always say “the other is bad,” all chips must say “the other is bad” in all pairwise tests. In this scenario, no strategy based on pairwise tests can distinguish the four possibilities, because there is only one possible outcome.

b. The following algorithm finds one good chip if there are more than n/2 good chips.
1. If there is only one chip, then it must be good.
2. Split the chips into 2-chip pairs. If the number of chips is odd, then let c denote the lonely chip.
3. Test each pair. If the result is good-good, then remove arbitrary one, otherwise remove both chips.
4. Go to 1.
The first three steps of the algorithm above uses floor(n/2) pairwise tests, since we can only split n chips into floor(n/2) pairs. Moreover, there will be at most floor(n/2)+1 chips remaining after these steps, since we remove at least one chip per pair. The rest is to show that at least half of the remaining chips are good.
If the result of pairwise test is good-good, then the chips are either both good or both bad, otherwise at least one of the chips is bad. Assume
   – x pairs consist of two good chips.
   – y pairs consist of a good chip and a bad chip.
   – z pairs consist of two bad chips.
   – There are good chips and b bad chips.
There are three possibilities,
   – If n is even, then g = 2x + y ≥ b = y + 2z. This implies x ≥ y. In this case, x good chips and y bad chips remain.
   – If n is odd and c is bad, then g = 2x + y ≥ b = y + 2z + 1. This implies x ≥ z + 1, since x and z are integers. In this case, x good chips and z + 1 bad chips remain.
   – If n is odd and c is good, then g = 2x + y + 1 ≥ b = y + 2z. This implies x + 1 ≥ z, since x and z are integers. In this case, x + 1 good chips and z bad chips remain.
We conclude that at least half of remaining chips are still good after the first three step.

c. Use the result of b. Find a good chip G in T1(n) = T1(⌈n/2⌉) + O(n) = O(n). Then test G with all the other chips in T2(n) = n − 1. G says “the other is good” only when tested with good chips.
a. 

<table>
<thead>
<tr>
<th></th>
<th>2</th>
<th>4</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>16</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

b. It follows from the heap property that the element that is leftmost and topmost is the minimum value in the tableau. That is, any element to the right of this element, or below it, is by definition greater than it. Hence, if \( Y[1,1] = \text{EMBED Equation.3} \), it is a nonexistent element, that is, no elements exist to the right of it or below it. The tableau is thus empty.

If \( Y[m, n] < \text{EMBED Equation.3} \), and if one of its parent slots is not , then \( Y[m, n] \) would have to be exchanged with that parent to maintain the heap invariant, and the comparison with its (new) parents would have to be repeated, until the heap invariant is satisfied. Hence, if the tableau satisfies the heap invariant with \( Y[m,n] < \text{EMBED Equation.3} \), it follows that all elements to the left and above \( Y[m, n] \) are less than it, that is, there are no empty spaces in the tableau.

c. 4 for the algorithm, 2 for the timebound] EXTRACT-MIN in a tableau is similar to the operation in a standard heap. The minimum value is retrieved from the cell \( Y[1,1] \), which is then set to \( \text{EMBED Equation.3} \). The value of \( \text{EMBED Equation.3} \) is “percolated” down the tableau, so that at each step the lesser of the children replaces the value. This is repeated until no elements that are less than \( \text{EMBED Equation.3} \) are to the right or below it. The pseudocode is given below:

```plaintext
ExtractMin(Y)
    minValue = Y[1, 1]
    Y[1, 1] = \text{EMBED Equation.3}
    percolateDown (Y, 1, 1)
    return minValue

percolateDown (Y, x, y)
    if Y[x, y+1] = Y[x+1, y] = \text{EMBED Equation.3}
        return
    if Y[x, y+1] < Y[x+1, y]          // compare two children
        Y[x, y] = Y[x, y+1]
        Y[x, y+1] = \text{EMBED Equation.3}
        return percolateDown (Y, x, y+1)
    else
        Y[x, y] = Y[x+1, y]
        Y[x+1, y] = \text{EMBED Equation.3}
        return percolateDown (Y, x+1, y)
```

d. Insert can be performed in a fashion analogous to the operation in the standard heap, by inserting the new value in an existing \( \text{EMBED Equation.3} \) -cell, and percolating it “up” until
it is greater in value than both its parents. Percolating involves exchanging it with the greater of its two parents. The pseudocode is given below:

Insert \((Y, \text{value})\)
- \(\text{if } Y[m, n] = \text{ EMBED Equation.3}\)
  - return “tableau is full”
- \(Y[m, n] = \text{value}\)
- \(\text{percolateUp (Y, m, n)}\)
- return “insert done”

percolateUp \((Y, x, y)\)
- \(\text{if } Y[x, y] \text{ greater than both } Y[x-1, y] \text{ and } Y[x, y-1]\)
  - return
- \(\text{if } Y[x, y-1] > Y[x-1, y] \quad \text{// compare two parents}\)
  - \(\text{swap } Y[x, y] \text{ and } Y[x, y-1]\)
  - return \(\text{percolateUp (Y, x, y-1)}\)
- else
  - \(\text{swap } Y[x, y] \text{ and } Y[x-1, y]\)
  - return \(\text{percolateUp (Y, x-1, y)}\)

\(e\). An Insert can be done into an \(n \times n\) tableau in \(O(n+n) = O(n)\) time. An Extract-Min can also be performed in the same time bound. To sort \(n^2\) numbers, we simply perform \(n^2\) Insert operations, followed by \(n^2\) Extract-Mins. The Inserts populate the tableau, ordered by values in the rows & columns. The Extract-Min operations return the values in ascending order. Since there are \(n^2\) Insert & Extract-Min operations, the total time = \(\text{ EMBED Equation.3} \), as required.

\(f\). To search the tableau for a given value \(k\), we can modify the Insert procedure given above slightly. The idea is to start off as if we are inserting \(k\). While percolating it up, however, we check whether either of the parent cells is equal to \(k\). If so, we’ve found the entry we’re looking for. If not, we percolate up as before, and repeat. Note that we do not actually modify the tableau as we go along; the insertion and percolateUp are conceptual. If in this process we get to two parent cells which are both lesser in value than \(k\), we conclude that \(k\) does not exist in the tableau. It is easily seen that this algorithm runs in the same time-bound as Insert.

8-5

\(a\). Simply in sorted order
\(b\). \(\{2, 1, 3, 4, 5, 6, 7, 8, 9, 10\} \quad \{2, 1, 4, 3, 6, 5, 8, 7, 10, 9\} \quad \{6, 1, 7, 2, 8, 3, 9, 4, 10, 5\}\).
\(c\). It can simply noted that :

\[
A_i \leq A_{i+k} \iff \sum_{j=i}^{i+k-1} A_j = \sum_{j=i+1}^{i+k} A_j = A_i + A_{i+1} + \ldots + A_{i+k-2} +
\]

\[
A_{i+k-1} - A_{i+1} - A_{i+2} - \ldots - A_{i+k-1} - A_{i+k} = A_i - A_{i+k} \leq 0 \iff \frac{1}{k} \sum_{j=i}^{i+k-1} A_j \leq
tfrac{1}{k} \sum_{j=i+1}^{i+k} A_j.
\]

\(d\). Split the array (arbitrarily) into \(k\) subarrays of size \(\text{floor}(n/k)\) or \(\text{floor}(n/k)+1\) each. Sort
each subarray at a cost of $O(n/k \log(n/k))$ each to form sorted subarrays $A_1, \ldots, A_k$. So far this is a total of $O(k(n/k \log n/k)/k) = O(n \log n/k)$. Finally, interleave the $k$ sorted subarrays to form one $k$-sorted array $A$: $A[i] = A[i \mod k[i \div k]]$. This involves $O(n)$ operations but no comparisons. Thus the total time is $O(n \log n/k)$. This works because for $i$ which is in the range 1 to $n-k$, we know that $A[i] = A[i \mod k[i \div k]] < A[i \mod k[(i \div k) + 1] = A[(i+k) \mod k[(i + k) \div k]] = A[i + k]$ since $A[i \mod k]$ is a sorted array, and therefore $A$ is $k$-sorted, by part (c) above.

e. We build $k$ subarrays by extracting $i, i+k, i+2k, \ldots$ from the $k$-sorted array. Now we get $k$ sorted arrays with length $n/k$. We can use the method in 6.5-8 to sort all elements in $O(n \log k)$ time.

f. Use the fact that comparison based sort needs $\Omega(n \log n)$.

9-1

a. Sort the numbers using mergesort or heapsort, which takes $\Theta(n \log n)$ worst-case time. Put the $i$ largest elements (directly accessible in the sorted array) into the output array, taking $\Theta(i)$ time.

Total worst-case running time: $\Theta(n \log n + i) = \Theta(n \log n)$ (because $i \leq n$).

b. Implement the priority queue as a heap. Build the heap using BUILD-HEAP, which takes $\Theta(n)$ time, then call HEAP-EXTRACT-MAX $i$ times to get the $i$ largest elements, in $\Theta(i \log n)$ worst-case time, and store them in reverse order of extraction in the output array. The worst-case extraction time is $\Theta(i \log n)$ because $i$ extractions from a heap with $O(n)$ elements takes $i \cdot O(\log n) = O(i \log n)$ time, and half of the $i$ extractions are from a heap with $\geq n/2$ elements, so those $i/2$ extractions take $(i/2) \Omega(\log(n/2)) = \Omega(i \log n)$ time in the worst case.

Total worst-case running time: $\Theta(n + i \log n)$.

c. Use the SELECT algorithm of Section 9.3 to find the $i$th largest number in $\Theta(n)$ time.

Partition around that number in $\Theta(n)$ time. Sort the $i$ largest numbers in $\Theta(i \log i)$ worst-case time (with merge sort or heapsort).

Total worst-case running time: $\Theta(n + i \log i)$.

10-1

<table>
<thead>
<tr>
<th></th>
<th>Unsorted Singly linked</th>
<th>Sorted Singly linked</th>
<th>Unsorted Doubly linked</th>
<th>Sorted Doubly linked</th>
</tr>
</thead>
<tbody>
<tr>
<td>Search(L,k)</td>
<td>O(n)</td>
<td>O(n)</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
<tr>
<td>Insert(L,x)</td>
<td>O(1)</td>
<td>O(n)</td>
<td>O(1)</td>
<td>O(n)</td>
</tr>
<tr>
<td>Delete(L,x)</td>
<td>O(n)</td>
<td>O(n)</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
<tr>
<td>Successor(L,X)</td>
<td>O(n)</td>
<td>O(1)</td>
<td>O(n)</td>
<td>O(1)</td>
</tr>
<tr>
<td>Operation</td>
<td>Predec(L, x)</td>
<td>Min(L)</td>
<td>Max(L)</td>
<td></td>
</tr>
<tr>
<td>---------------</td>
<td>-------------</td>
<td>--------</td>
<td>--------</td>
<td></td>
</tr>
<tr>
<td>Time Complexity</td>
<td>O(n)</td>
<td>O(n)</td>
<td>O(n)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>O(n)</td>
<td>O(1)</td>
<td>O(n)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>O(n)</td>
<td>O(n)</td>
<td>O(n)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>O(n)</td>
<td>O(n)</td>
<td>O(n)</td>
<td></td>
</tr>
</tbody>
</table>