Solve all 7 (SEVEN) problems in the space provided

Read the Problems CAREFULLY!

There are 6 (SIX) pages this page included; the last page is left blank

STATEMENT

On my honor, I pledge that I have not violated the provision of the NJIT Student Honor Code.

Sign below at the end of the exam

Signature

Any algorithm you present must be given in concise and complete form. Argue about its correctness. Analyze its (worst-case) running time and express it in asymptotic notation. Random ramblings or sketches will not be given any points. You may use algorithms presented in class as black-boxes without further description. For example, instead of repeating the code of MergeSort you can just write MergeSort(B,m) to indicate that you sort an array B of m keys.

List of useful formulae

\[
\sum_{i=1}^{n} i = \frac{n(n+1)}{2}, \quad \sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}, \quad \sum_{i=1}^{n} i^3 = \frac{n^2(n+1)^2}{4}
\]

\[n! \approx \left(\frac{n}{e}\right)^n, \quad a^{\log_a n} = n^{\log_a a},\]

For \(x \neq 1\), we have that

\[
\sum_{i=0}^{n} x^i = \frac{x^{n+1} - 1}{x - 1}, \quad \sum_{i=0}^{n-1} i x^i = \frac{(n - 1)x^{n+1} - nx^n + x}{(1 - x)^2}.
\]

B1. \(f(n) = \Theta(g(n))\) iff \(\exists\) positive constants \(c_1, c_2, n_0: 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \ \forall \ n \geq n_0.\)

B2. \(f(n) = \Omega(g(n))\) iff \(\exists\) positive constants \(c_1, n_0: 0 \leq c_1 g(n) \leq f(n) \ \forall \ n \geq n_0.\)

B3. \(f(n) = O(g(n))\) iff \(\exists\) positive constants \(c_2, n_0: 0 \leq f(n) \leq c_2 g(n) \ \forall \ n \geq n_0.\)

Master Method. \(T(n) = aT(n/b) + f(n)\), such that \(a > 0, b > 1, f(n) > 0).\)

M1 If \(f(n) = O(n^{\log_b a - \epsilon})\) for some constant \(\epsilon > 0\), then \(T(n) = \Theta(n^{\log_b a}).\)

M2 If \(f(n) = \Theta(n^{\log_b a} \log^k n)\), then \(T(n) = \Theta(n^{\log_b a} \log^{k+1} n)\), where \(k \geq 0\) is a non-negative constant.

M3 If \(f(n) = \Omega(n^{\log_b a + \epsilon})\) for some constant \(\epsilon > 0\), and if \(af(n/b) \leq cf(n)\) for some constant \(c < 1\) and for large \(n\), then \(T(n) = \Theta(f(n)).\)
Problem 1. (50 points)

**TYPE A and B**

We show in general how to find the fake coin among \( n = 3^k \) coins in \( k \) weighings. For **TYPE A** use \( k = 3 \) i.e. \( n = 3^3 = 27 \) and for **TYPE B** use \( k = 4 \), i.e. \( n = 3^4 = 81 \) to obtain the corresponding answer.

Let us have \( n = 3^3 \) coins. Split the coins into three piles of \( n/3 \) coins each. Call the piles \( A, B, C \). Weigh \( A \) and \( B \). If they weigh the same the fake coin is in \( C \). If \( A \) weighs less the fake is in \( A \), otherwise it is in \( B \).

So after one weighing we have located the fake coin in one of \( A, B, C \) i.e. we need to find the fake coin among \( n/3 = 3^{k-1} \) coins. Therefore \( T(n) = T(n/3) + 1 \) and the solution gives \( T(n) = \log_3 n \). Since \( n = 3^k \), we get \( T(n) = T(3^k) = k \). The result follows.

Problem 2. (50 points)

**TYPE A.** This is similar to an Exam 1 problem.

For \( T(n) = 8T(n/2) + n^4 \), master method case (3) gives \( T(n) = \Theta(n^4) \).

For \( T(n) = 4T(n/2) + n^2 \), master method case (2) gives \( T(n) = \Theta(n^2 \log n) \).

For \( T(n) = T(n-1) + n \), the iteration method gives \( T(n) = T(n-1) + n = T(n-2) + (n-1) + n = \ldots = T(1) + 2 + \ldots + (n-1) + n \). Choose \( T(1) = 1 \) (our choice), and \( T(n) = n(n+1)/2 = \Theta(n^2) \).

**TYPE B.** This is similar to an Exam 1 problem.

For \( T(n) = 8T(n/2) + n^3 \), master method case (2) gives \( T(n) = \Theta(n^3 \log n) \).

For \( T(n) = 16T(n/2) + n^2 \), master method case (3) gives \( T(n) = \Theta(n^5) \).

For \( T(n) = T(n-2) + n \), the iteration method gives \( T(n) = T(n-2) + n = T(n-4) + (n-2) + n = \ldots = T(1) + T(2) + \ldots + (n-4) + (n-2) + n \). Choose \( T(1) = 1, T(2) = 2 \) (our choice), and \( T(n) = \Theta(n^2) \).

Alternatively, use the substitution method to guess \( T(n) = n^2/4 + n/2 \).

Problem 3. (50 points)

**TYPE A.** This is similar to an Exam 1 problem; details are omitted.

\[ T(n) = 2T(n/2) + 8n, \quad T(4) = 9. \]

Use the iteration method as in Exam 1. We obtain \( T(n) = (n/4)T(4) + (\log n - 2)8n = 8n \log n - 13.75n \).

**TYPE B.** This is similar to an Exam 1 problem; details are omitted.

\[ T(n) = 2T(n/2) + 4n, \quad T(4) = 18. \]

Use the iteration method as in Exam 1. We obtain \( T(n) = (n/4)T(4) + (\log n - 2)4n = 4n \log n - 3.5n \).

Problem 4. (50 points)

**TYPE A.**

1. QuickSort is stable. **FALSE**
2. CountSort is stable. **TRUE**
3. RadixSort sorts in-place. **FALSE**
4. On the increasing input sequence of integers \( \langle 1, 2, \ldots, n \rangle \), InsertionSort is asymptotically faster than CountSort. **FALSE** they have the same asymptotic performance.
5. On the increasing input sequence of integers \( \langle 1, 2, \ldots, n \rangle \), MergeSort is asymptotically faster than CountSort. **FALSE** the latter is linear the former is \( \Omega(n \log n) \).
6. On the decreasing input sequence of integers \( \langle n, n-1, \ldots, 1 \rangle \), RadixSort is asymptotically faster than HeapSort. **TRUE** (justification follows that of previous question).
7. The solution of \( T(n) = T(n/5) + T(7n/10) + n \), is \( T(n) = \Theta(n \log n) \). **FALSE** it is \( T(n) = \Theta(n) \).
8. \( \lg (n!) = O(n \log n) \). **TRUE** see first page though a tighter bound is \( \Theta \).
9. For sorting \( n \) keys in the range \( 0, \ldots, n^2 - 1 \), RadixSort is asymptotically faster than HeapSort. **TRUE** two rounds of CountSort in RadixSort give \( \Theta(n) \) performance.
10. For sorting \( n \) keys in the range \( 0, \ldots, 2^n - 1 \), RadixSort is asymptotically faster than MergeSort. **FALSE** RadixSort would need \( \Theta(n^2 / \log n) \) time much worse than MergeSort’s \( O(n \log n) \).
TYPE B.

(1) MergeSort is stable. **TRUE**
(2) CountSort sorts in-place. **FALSE**
(3) RadixSort is stable. **TRUE**
(4) On the decreasing input sequence of integers \(\langle n, n-1, \ldots, 1 \rangle\), MergeSort is asymptotically faster than CountSort. **FALSE** CountSort is linear.
(5) On the decreasing input sequence of integers \(\langle n, n-1, \ldots, 1 \rangle\), HeapSort is asymptotically faster than QuickSort. **TRUE** HeapSort is \(O(n \log n)\) but QuickSort quadratic.
(6) On the decreasing input sequence of integers \(\langle n, n-1, \ldots, 1 \rangle\), QuickSort is asymptotically faster than InsertionSort. **FALSE** both have quadratic performance.
(7) The solution of \(T(n) = T(7n/10) + T(n/5) + n\), is \(T(n) = \Theta(n)\). **TRUE**
(8) \(n! = o(n \log n)\). **FALSE** \(n! = \omega(n \log n)!\) (and this last ! is an exclamation mark).
(9) For sorting \(n\) keys in the range 0, \(\ldots\), \(n^7 - 1\), RadixSort is asymptotically faster than MergeSort. **TRUE** 7 rounds of CountSort give RadixSort a linear time performance over MergeSort’s \(\Omega(n \log n)\).
(10) For sorting \(n\) keys in the range 0, \(\ldots\), \(n^n - 1\), RadixSort is asymptotically faster than HeapSort. **FALSE** RadixSort would be \(\Theta(n^2)\) but HeapSort is \(O(n \log n)\).

**Problem 5.** (50 POINTS)

<table>
<thead>
<tr>
<th>Type A</th>
<th>Type B</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>1</td>
<td>3 6 12</td>
</tr>
</tbody>
</table>

**Problem 6.** (50 POINTS)

**TYPE A.**

Case 1. \(n = 2k + 1\).

\(n=2k+1\)

<table>
<thead>
<tr>
<th>A4</th>
<th>A7</th>
<th>A6</th>
<th>A3</th>
<th>A1</th>
<th>A2</th>
<th>A5</th>
</tr>
</thead>
<tbody>
<tr>
<td>o</td>
<td>o</td>
<td>o</td>
<td>m</td>
<td>o</td>
<td>o</td>
<td>x</td>
</tr>
</tbody>
</table>

\(<----m-A4------------x<-----A1-m----->\)
\(<----m-A7------> <--------A2-m----->\)
\(<m-A6> <--------A5-m--------->\)

\(m\) \(m+d\)

\| A4 | A7 | A6 | A3 | A1 | A2 | A5 |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>o</td>
<td>o</td>
<td>o</td>
<td>o</td>
<td>m</td>
<td>d</td>
<td>o</td>
</tr>
</tbody>
</table>

\(<------m-A4+d----------x<A1-m-d>\)
\(<m-A7+d----------> <----A2-m-d---->\)
\(<---m-A6+d--> <-----A5-m-d----->\)

Suppose \(x\) is not the median but \(x = m + d\)
4 distances are increased by \(d\)
3 distances are decreased by \(d\)
Net gain is \(d\).

“Order” the numbers. Let \(x\) be the median \(m\) of the \(n\) numbers. We claim that \(D(x) = \sum_{i=0}^{n-1} |A[i] - x|\) is then minimized for \(x = m\), i.e. \(D(m)\) is minimal. Why?

Suppose that \(x\) is not the median but a value \(x = m + d\) to the right of the median by a displacement \(d\). Then the distance \(|A[i] - x|\) of (at least) \(k + 1\) points (including the median) has been increased by \(d\) and the distance of (at most) \(k\) points has been decreased by \(d\). The end result is that the sum \(\sum_{i=0}^{n-1} |A[i] - x|\) has been
increased by at least \(d\), i.e. \(D(m + d) \geq D(m) + d\). If \(x\) is on the left of the median \(m\) we have a similar situation and an increase by at least \(d\) of the sum (i.e. \(D(m - d) \geq D(m) + d\)). Therefore the median \(m\) is the optimal \(x\) because it gives the lowest value for the sum.

We do not need to sort the numbers to find the median, i.e. \(x\). We just use the linear-time Select algorithm. This takes \(\Theta(n)\).

**Case 2.** \(n = 2k\).

In this case we have two medians, the lower \(m\) and the upper \(M\) median. Let us choose \(x = z\) to be between \(m\) and \(M\) i.e. \(m \leq z \leq M\). Let the sum for this \(x = z\) be \(D = D(z)\). If we move \(x\) by \(d\) to the left (i.e. \(x = z - d\)) but still in that interval or by \(d\) to the right (i.e. \(x = z + d\)) but still in that interval \(k\) points' displacement is increased by \(d\) and \(k\) points' displacement is reduced by \(d\). So the net-gain/loss is 0 and the sum remains equal to \(D\), i.e. \(D(z + d) = D(z - d) = D = D(z)\) as long as \(m \leq z, z + d, z - d \leq M\). The \(d\) is an arbitrarily chosen value.

However if \(x\) is moved to the left of \(m\) or to the right of \(M\) then the sum gets increased. For example if \(x\) moves to the left by \(d\) and falls on the left of \(m\), for \(k - 1\) points or less the displacement gets smaller by \(d\) but for \(k\) points or more the displacement gets bigger by \(d\) and for \(m\) the displacement gets decreased by less than \(d\) because \(x\) moves from the right to the left of \(x\). This is shown in the figure.

Therefore for Case 2, the optimal \(x\) is any value \(z\) such that \(m \leq z \leq M\). We could report \(x = m\) just as we did in Case 1, or use the upper median or any other value in-between.

---

**n=2k median m=a3**

<table>
<thead>
<tr>
<th>A4</th>
<th>A7</th>
<th>A6</th>
<th>A3</th>
<th>x=z</th>
<th>A1</th>
<th>A2</th>
<th>A5</th>
<th>A8</th>
</tr>
</thead>
<tbody>
<tr>
<td>o</td>
<td>o</td>
<td>o</td>
<td>o</td>
<td>o</td>
<td>o</td>
<td>o</td>
<td>o</td>
<td>o</td>
</tr>
<tr>
<td>(&lt;z&gt;A3&gt;)</td>
<td>(&lt;z&gt;A7&gt;A3&gt;)</td>
<td>(&lt;z&gt;A6&gt;A3&gt;)</td>
<td>(&lt;z&gt;A5&gt;A3&gt;)</td>
<td>(&lt;z&gt;A8&gt;A3&gt;)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

If \(z\) is between \(m\) and \(M\) anywhere in \([m,M]\) the sum remains the same

---

\| \= \| \= \| \= \| \= \| \= \|

\(z\) moves to the left by \(d\) i.e. it becomes \(z-d\)

---

3 points' distances get decreased by \(d\)

---

4 points' distances get increased by \(d\)

---

a3's distance get decreased by \(<?\) which is less than \(d\)
**Problem 7.** (33 points)

Determine the smallest time is $O$ a duplicate is detected this signifies a member of $A$ sequence left to right in linear time $O$. Given that $A$ has distinct keys and so does $B$, if during the scanning a duplicate is detected this signifies a member of $A \cap B$. Print it by adding it to set $C$, the output. Running time is $O(n \lg n)$.

**TYPE B**

We use binary search on the sorted array $A$. We search for $-x$. If there is a $k$ such that $A[k] = -x$, this means that $A[k] + x = 0$ i.e. we have found our $i$ to be the $k$ returned by BinarySearch.

If however a $-x$ is not in the array we have one of three cases at the completion of BinarySearch:

(a) Case 1: $-x < A[0]$. Then the closest $A$ to $-x$ is $A[0]$. We return $A[0] + x$. If $-x < A[0]$ it also means that $-x < A[0] < A[1] < \ldots$. Therefore $x + A[0] < x + A[j]$, $j > 0$, i.e. $x + A[0]$ is the smallest possible. Because the sum is positive, it is also equal to its absolute value.


(c) Case 3: $A[j] < -x < A[j+1]$. Then $-x$ is closer to $A[j]$ or $A[j+1]$ than anything else since everything else is smaller that $A[j]$ or greater than $A[j+1]$. We determine in constant time whether $-x - A[j]$ or $A[j+1] + x$ (both distances are positive) is the smallest one. The smallest is the one returned and is the smallest $|A[k] + x|$.

The running time in any of the three cases is dominated by the BinarySearch time. Therefore we can determine the smallest $|A[k] + x|$ in $O(\lg n)$ time.

**Problem 7.** (33 points)

**TYPE A: Intersection**

Sort all the $2n$ keys of both sequences using HeapSort/Mergesort in $O(n \lg n)$ time. Then scan the sorted sequence left to right in linear time $O(n)$. Given that $A$ has distinct keys and so does $B$, if during the scanning a duplicate is detected this signifies a member of $A \cap B$. Print it by adding it to set $C$, the output. Running time is $O(n \lg n)$.

**TYPE B: Union**

Sort all the $2n$ keys of both sequences using HeapSort/Mergesort in $O(n \lg n)$ time. Then scan the sorted sequence left to right in linear time $O(n)$. Given that $A$ has distinct keys and so does $B$, if during the scanning a duplicate is detected this signifies a member of $A \cap B$. Put all the keys of the sorted sequence, into $C$ except for the detected duplicates (i.e. as soon as you detect the two consecutive duplicates in the sorted output sequence, you insert into $C$ the first but not the second one). Running time is $O(n \lg n)$.

Alternatively, we could solve both cases at the same time as follows.

**Common Solution.**

We can solve the two problems at the same time without much effort. Here is how. You can sort the $2n$ keys in place using HeapSort even if they reside in two different arrays. Given that there is no requirement to do this in-place we could use a separate array to copy $A,B$ in place using HeapSort even if they reside in two different arrays. Let’s call it $C$.

Then you scan the output sorted sequence left to right to move the union to one side (left) and the intersection to the other (right). Index $k$ points to the last key of the Union. Everything to its right will at the end be part of the intersection.

```plaintext
Both (A,B,n)
0.1 for(i=0;i<n;i++)
0.2 C[i]=A[i];
0.3 for(i=0;i<n;i++)
0.4 C[i+n]=B[i];
1. HeapSort(C,2n);
2. k=0; // C[0] is always in the union;
3. for(i=1;i< 2n;i++) // Start with C[1] ... C[2n-1] which are sorted
4. if C[i] != C[k] {
5. k++;
6. swap(C[i],C[k]);
7. }
8. else { // duplicate detected
9. ; // Do nothing. Skip C[i]. Lines 8-10 only serve this comment; they can easily go.
10. }
11. return(<C,k>: C[0...k] as the union, C[k+1..2n-1] as the intersection );
```

We can solve the two problems at the same time without much effort. Here is how. You can sort the 2n keys in place using HeapSort even if they reside in two different arrays. Given that there is no requirement to do this in-place we could use a separate array to copy A, B and sort that array. Let’s call it C.

Then you scan the output sorted sequence left to right to move the union to one side (left) and the intersection to the other (right). Index k points to the last key of the Union. Everything to its right will at the end be part of the intersection.

```plaintext
Both (A,B,n)
0.1 for(i=0;i<n;i++)
0.2 C[i]=A[i];
0.3 for(i=0;i<n;i++)
0.4 C[i+n]=B[i];
1. HeapSort(C,2n);
2. k=0; // C[0] is always in the union;
3. for(i=1;i< 2n;i++) // Start with C[1] ... C[2n-1] which are sorted
4. if C[i] != C[k] {
5. k++;
6. swap(C[i],C[k]);
7. }
8. else { // duplicate detected
9. ; // Do nothing. Skip C[i]. Lines 8-10 only serve this comment; they can easily go.
10. }
11. return(<C,k>: C[0...k] as the union, C[k+1..2n-1] as the intersection );
```