COMP 3711 – Design and Analysis of Algorithms
2021 Fall Semester – Written Assignment # 3
Distributed: October 7, 2021
Due: October 18, 2021, 11:59 PM
Solution Key – Revised October 31, 2021

Your solution should contain
(i) your name, (ii) your student ID #, and (iii) your email address
at the top of its first page.

Some Notes:

- Please write clearly and briefly. Your solutions should follow the guidelines given at
  https://canvas.ust.hk/courses/38226/pages/assignment-submission-guidelines

  In particular, your solutions should be written or printed on *clean* white paper with no watermarks, i.e., student society paper is not allowed.

- Please also follow the guidelines on doing your own work and avoiding plagiarism as described on the class home page.
  You must acknowledge individuals who assisted you, or sources where you found solutions. Failure to do so will be considered plagiarism.

- The term *Documented Pseudocode* in Problems 1 and 4 means that your pseudocode must contain documentation, i.e., comments, inside the pseudocode, briefly explaining what each part does.

- Many questions ask you to explain things, e.g., what an algorithm is doing, why it is correct, etc. To receive full points, the explanation must also be *understandable* as well as correct.

- Please make a *copy* of your assignment before submitting it. If we can't find your submission, we will ask you to resubmit the copy.

- Submit a SOFTCOPY of your assignment to Canvas by the deadline. The softcopy should be one PDF file (no word or jpegs permitted, nor multiple files).

- 10/31/2021. Solution 3 of Q1 was slightly modified.

  If your submission is a scan of a handwritten solution, make sure that it is of high enough resolution to be easily read. At least 300dpi and possibly denser.

Problem 1: ***k*-swapped arrays** [30 pts]

An n element array $A[1 \ldots n]$ is **k-swapped** if k is the *minimum* integer in $[0, n]$ satisfying the following:
for all $i, j \in [1, n]$ satisfying $i > j$ and $A[i] < A[j]$, we have $i - j \leq k$.

The problem is to sort a given k-swapped array in $O(n \log k)$ time.
Time in this problem refers to the number of comparisons performed by the algorithm.

In what follows A is a k-swapped array of size n, where k is provided as an extra parameter to the algorithm, and k can be used in the code.

(a) (i) Write documented pseudocode for an $O(n \log k)$ time procedure to sort k-swapped array $A[1 \ldots n]$.
(ii) Below the pseudocode, provide a description in words (as opposed to code) of what your algorithm is doing.

(b) Prove why your algorithm is correct.
This should be a clear FORMAL mathematical style proof.

(c) Prove that your algorithm runs in $O(n \log k)$ time.

(d) In class we proved that any comparison-based algorithm requires $\Omega(n \log n)$ time. Why does this not contradict having an $O(n \log k)$ solution for this problem (when k is much smaller than n).

(e) Prove that any algorithm for solving this problem requires $\Omega(n \log k)$ comparisons (so your solution is the best possible).

Rules, recommendations and hints

- In part (a), your algorithm may use any procedures or algorithms that we learned in class as a black-box subroutine (without having to provide code for that subroutine).
- In parts (b)-(d) you may use theorems, lemmas and facts that we learned in class or tutorial. If you use such theorems, lemmas and/or facts you must (i) explicitly state the text of the theorem/lemma/fact that you are using and (ii) identify exactly where in the class notes you are copying them from. Everything else must be proven from scratch.
- Hint. The solution to part (e) is in one of the tutorials (in a slightly different form).
- Proofs of correctness that are not formal and clear will have points deducted. There are multiple algorithms for this problem, some of which are quite intuitive. A main goal of this problem is to show that you can prove the correctness of an intuitive idea.
- Your proof of correctness may assume that all elements in the array are distinct.

Solution: We provide four different algorithms for (a), (b).

In what follows $A[1,\dots n]$ denotes the original array and $y_1,\dots,y_n$ are the elements in A written in sorted order.

Algorithm 1 (a) Sort k-swapped array $A[1\dots n]$

```
1: Create min priority queue $Q$
2:  Insert $A[1\dots k+1]$ into $Q$                    ▷ $O(k\log k)$ time
3: for $i = 1$ to $n-(k+1)$ do   ▷ Remove $A[i]$ from $Q$ and add next item to $Q$
4:     $A[i] = Extract\_min(Q)$
5:     Insert $A[i+k+1]$ into $Q$
6: for $i = n-k$ to $n$ do                               ▷ Remove $A[i]$ from $Q$
7:     $A[i] = Extract\_min(Q)$
```

The algorithm starts by creating a min-priority queue on $A[1,\dots,k+1]$.
It then iteratively
(i) pulls the smallest item from the queue and makes it the next item in the sorted output
(ii) adds the next item in the array (if any exists) to the priority queue

(b) The proof of correctness uses the following Lemma:

Lemma: If A is a k-swapped array, and $A[j]=y_i$, then $j\le i+k$.

Proof: Suppose not and there exists i,j with $A[j]=y_i$ and $j>i+k$.

Then $j-k>i$. Now consider subarray $B=A[1\dots j-k-1]$.

Since $y=A[j]$, $y_i\notin B$. Since B only has $j-k-1$ slots and $i\le j-k-1$, B must contain some y_v with $v>i$, i.e., there must exist a $t\le j-k-1$ such that $A[t]=y_v>y_i=A[j]$. But, since $j-t\ge k+1$, this contradicts the definition of A being k-swapped. ■

Since $A[1,\dots,\min(i+k,n)]$ are all inserted into Q before implementing line 4 or line 7, the lemma immediately implies

Corollary: y_i was inserted into Q before line 4 or line 7 was called.

We can then prove the following by induction.
$I(i)$: After line 4 or line 7, $A[1,\dots i]$ contains exactly the items $y_1,\dots,y_i$ in increasing sorted order.

$I(1)$ is true since, the Corollary implies that Q contains y_1 before the start of Line 3. Thus the *Extract_min* at line 4 will set $A[1]=y_1$.

Now assume that $I(t)$ is true. Then at the start of step $t+1$ none of $y_1,\dots,y_t$ are in Q (since they've already been extracted). From the Corollary we know that y_{t+1} is in Q. Thus the *Extract_min* at line 4 or line 7 will set $A[t+1]=y_{t+1}$.

So $I(t)$ is true for all t. In particular, $I(n)$ is true, so the algorithm is correct.

Alternative Algorithm

Here is a totally different algorithm with its correctness proof. In what follows we use k-*array* to mean an array that is k'-swapped for some $k' \le k$ (this just removes the "minimum" from the definition). Note that any algorithm that works on a k-array will also work on a k-swapped array.

Algorithm 2 (a') Sort k-array $A[1 \dots n]$

```
1: $\ell = 1$; $r = \min(2k, n)$
2: while $\ell \le n$ do
3:     $Mergesort(A, \ell, r)$                    ▷ Mergesort a window of $2k$ items
4:     $\ell = \ell + k$; $r = \min(r + k, n)$    ▷ Slide the window $k$ units to the right
```

This algorithm just maintains a window of $2k$ contiguous items (truncating the window so that it does not extend past the end of the array).

It starts with the window containing $A[1, \dots, 2k]$.

At each step it mergesorts the items in that window and then slides the window k units to the right and repeats.

(b') The main fact that we will use is

Lemma 2: If $A[1, \dots, n]$ is a k-array and $A[1, \dots, 2k]$ is then sorted then

(i) $A[1, \dots k]$ are exactly the items $y_1, \dots y_k$ in sorted order.

(ii) $A[k+1, n]$ is a k-array.

Proof: From the Lemma on the previous page we know that, before the sort, if $i \le k$ and $A[j] = i$ then $j \le i + k \le 2k$.

Thus $y_1, \dots, y_k$ are all originally located in $A[1, \dots 2k]$. Since these k items must be the k smallest items in $A[1, \dots 2k]$, after sorting $A[1, \dots, 2k]$, (i) is true.

To prove (ii) we first assume that $n > 2k$ since otherwise $A[k+1, n]$ contains at most k items and is trivially a k-array.

Recall that an *inversion* is a pair i, j with $i > j$ but $A[i] < A[j]$.

Now consider the inversions in $A[k+1, \dots, n]$ *after the sort is performed.* To prove (ii) we need to show that, after the sort, if $A[i] < A[j]$ then $i - j \le k$.

There are 3 possible cases of such inversions with $i > j$ and $A[i] < A[j]$.

- If $j > 2k$: then $A[i]$ and $A[j]$ are unchanged from before the sort so $i - j \le k$;
- If $i \le 2k$: Because of the sort, there are no inversions in $A[k+1, \dots, 2k]$;
- If $k + 1 \le j \le 2k < i$: Suppose $i - j > k$

  After the sort, if $j < t \le 2k$ then $A[i] < A[j] < A[t]$. All of these $A[t]$ were located in $A[1, \dots 2k]$ before the sort, so, before the sort $A[1 \dots 2k]$

contains at least $2k-(j-1)$ elements larger than $A[i]$. In particular, this means that before the sort there a $t' \le j < i$ such that $A[t'] > A[i]$. But $i - t' \ge i - j > k$. But this would contradict the fact that A was a k-array before the sort. So $i - j \le k$.

These 3 cover all possible cases, so (ii) is correct. ■

We now see by induction on n that the algorithm correctly sorts a size n k-array.

If $n \le 2k$ the algorithm is obviously correct.

If $n > 2k$, the first step correctly places the first k items in the array. The algorithm then recurses on $A[k+1, n]$. But, from Lemma 2, this is also a k-array. So, by induction, it correctly sorts the remainder of the items.

A 3rd Algorithm

A 3rd approach, which we will only sketch, and not write in detail, is to partition A into $t = \lceil \frac{n}{k} \rceil$ subarrays, B_s, of size k.

Let $B_s = A[sk, \ldots, sk+k-1]$ for $0 \le s \le t$ (B_t might have fewer than k items).

- Sort each B_s separately. This uses $\Theta(n \log k)$ time in total.. Call the outputted arrays B'_s.
- Because the original A was k-swapped, all items in B_s are less than all the items in B_{s+2}.

  Concatenate $B'_0, B'_2, B'_4, \ldots$ into one list. From the observation above, this list is sorted.

  Concatenate $B'_1, B'_3, B'_5, \ldots$ into another list. Again, from the observation above, this list is sorted.

  Creating thse two sorted lists took $O(n)$ time
- Now merge the two sorted lists using another the $O(n)$ time merge procedure from class. This yields the final sorted list using only $O(n \log k)$ time in total.

A 4th Algorithm

A 4th approach, which we will also only sketch, and not write in detail, is to use $O(n)$ time to partition A into $k+1$ subarrays, B_s, $s = 1, \ldots k+1$ where

$$B_s = s, s + (k+1,)s + 2(k+1), s + 3(k+1), \ldots.$$

From the earlier lemma we know that for all i, $A[i] < A[i+(k+1)]$ so each B_s is sorted in increasing value.

We can then use the solution of SS5 (merging m sorted lists of total size n in $O(n \log m)$ time) to merge those $k+1$ lists in $O(n \log(k+1)) = O(n \log k)$ time to get the fully sorted array.

(c) For Algorithm (1) note that it performs n Inserts and n Extract Mins, each of which requires $O(\log k)$ time (since the priority queue has size k). this uses $O(n \log k)$ in total.

For Algorithm 2, note that each sort requires $O(2k \log 2k) = O(k \log k)$ time. The total number of sorts is $\lceil \frac{n}{k} \rceil = O(n/k)$. The total time required is then $O(\frac{n}{k} k \log k) = O(n \log k)$.

For the sketched, 3rd, and 4th algorithms, we already showed that they are $O(n \log k)$ time.

(d) The lower bound was actually $\Omega(\log N(n))$ where $N(n) = n!$ was the number of possible different outcomes, i.e., permutations, that could occur.
But, the number of possible k-arrays is not $n!$. It's actually much smaller.

(e) Let $N_k(n)$ denote the number of k-swapped arrays of size n.

We need to show that $\log(N_k(n)) = \Omega(n \log k)$. The lower-bound theorem we learned in class then immediately gives the lower bound.

We actually provide two solutions. The first is for the simpler version that we said in Piazza that you could solve instead. The second is for the full problem.

Simplified proof:
As mentioned in Piazza, in solving this problem you *may* assume that $k+1$ (or k) divides n and that you are deriving a lower bound for sorting k-arrays and not k-swapped arrays, i.e for an array that is k'-swapped for some $k' \le k$
Let $N'_k(n)$ be the number of k-arrays of size n.
Assume that n is divisible by $k+1$, i.e., $n = t(k+1)$ for some positive integer t.
We use a slight modification of the technique from Tutorial SS10.
Consider any permutation of $[1 \dots n]$ written in A satisfying the following properties:

(i) Let $S_m = \{m(k+1), m(k+1+2), \dots, m(k+1)+k\}$.
Note that the $|S_m| = k+1$ for all m and the items in S_m are all smaller than the items in S_{m+1}.

(ii) $A[m(k+1), \dots, m(k+1)+k]$ is a permutation of S_m.

Note that such an A is a k-array because if $i > j$ and $A[i] < A[j]$ then i and j must be in the same S_m so $i - j \le k$.
The number of permutations of any S_m is $(m+1)!$.
Multiplying, we find that total number of arrays that satisfy (i) and (ii) is $((k+1)!)^t$. Since any array that satisfies (a) and (ii) is a k-array we have that

$N'_k(n) \geq ((k+1)!)^t$, so

$$\begin{aligned}
\log_2 N'_k(n) &\geq \log_2\left(((k+1)!)^t\right) \\
&= t\log_2(k+1)!) \\
&\geq \Theta\left(t(k+1)\log(k+1)\right) \qquad \text{(Stirling's formula)} \\
&= \Theta\left(\frac{n}{k}(k+1)\log(k+1)\right) \\
&= \Theta(n\log k) \qquad \left(\text{Since } \frac{k+1}{k} = \Theta(1) \text{ and } \log_2(k+1) = \Theta(\log k)\right),
\end{aligned}$$

which is what we needed.

Proof for full case:

As in the simple proof, we first assume that n is divisible by $k+1$, i.e., $n = t(k+1)$ for some positive integer t.

Consider any permutation of $[1 \dots n]$ written in A satisfying the following properties:

(i) Let $S_m = \{m(k+1), m(k+1+2), \dots, m(k+1)+k\}$.
Note that the $|S_m| = k+1$ for all m and the items in S_m are all smaller than the items in S_{m+1}.

(ii) $A[m(k+1), \dots, m(k+1)+k]$ is a permutation of S_m.

(iii) $A[k+1] = 1$ and $A[1] = k+1$.

Note that such an A is k-swapped. As seen in the simple proof this is a k-arry. But, from (iii) $1, k+1$ are also an inversion with $k+1 > 1$ but $A[k+1] < A[1]$. Since $k+1-1 = k$, A is k-swapped.

The number of permutations of any S_m is $(k+1)!$. The number of permutations of S_1 with $A[1] = k+1$ and $A[k+1] = 1$ is $(k-1)!$.

Multiplying, we find that total number of arrays that satisfy (i), (ii) and (iii) is

$$(k-1)!((k+1)!)^{t-1} = \frac{((k+1)!)^t}{k(k+1)}.$$

Thus

$$N_k(n) \geq (k-1)!((k+1)!)^{t-1} = \frac{((k+1)!)^t}{k(k+1)}.$$

Form Stirling's formula we have

$$\Theta((k+1)! = \Theta((k+1)\log(k+1)) = \Theta(k\log k).$$

Thus

$$\begin{aligned}
\log_2 N_k(n) &\geq \log_2\left(\frac{((k+1)!)^t}{k(k+1)}\right) \\
&= t\log_2(k+1)!) - \log_2 k - \log_2(k+1) \\
&\geq t\log_2(k+1)!) - 2\log(k+1) \\
&= \Theta(tk\log) - O(\log(k)) \\
&= \Theta(tk\log k) \\
&= \Theta\left(\frac{n}{k}k\log(k)\right) \\
&= \Theta\left(n\log k\right).
\end{aligned}$$

Now suppose that n is not a multiple of $k+1$, i.e., $n = t(k+1)+r$ where $t \geq 1$ and $0 < r < k+1$. In particular, $n-r$ is a multiple of $(k+1)$ so we can apply the result to it.

It is easy to see that $N_k(n)$ is a nondecreasing function in n since any k-swapped array of size n can be transformed into a unique k-swapped array of size $n+1$ by setting $A[n+1] = n+1$. Thus $N_k(n) \geq N_k(n-r)$.

Since $n \geq k+1$ and $0 < r < k+1$ this implies $\frac{n}{2} \leq n-r$ so

$$\begin{aligned}
\log_2(N_k(n)) &\geq \log_2(N_k(n-r)) \\
&= \Theta\left((n-r)\log k\right) \qquad \text{(from result above)} \\
&= \Theta\left(\frac{n}{2}\log k\right) \\
&= \Theta\left(n\log k\right).
\end{aligned}$$

Marking Note 1a1.

Many student got confused by the distinction between k and $k+1$. As an example, they thought that item i could only go at most to location $A[k]$ and didn't realize that it could go to location $A[k+1]$. That caused issues with the various algorithms. We deducted a small number of points for this error on the code (because the algorithms would not work!) but otherwise marked them as essentially correct.

But, if this error was continued in (b) we deducted more points, because this meant that the proof of correctness was also wrong.

Marking Note 1a2.

Some students thought that A could be split up into $n/(k+1)$ (or n/k) continuous disjoint intervals of size $k+1$ (or k) in which all of the items in each interval are smaller than all of the items in the next intervals. They then sorted the intervals separately. This does NOT work and was marked as wrong (points were deducted for parts (a), (b), (c)). See Piazza note 212 for a worked counterexample.

This idea could be modified to show that all the items in one interval were smaller than all of the items in the interval AFTER the next interval. This was the idea behind the 3rd solution given and required a lot more work, e.g., an extra merge to get correct)

Marking Note 1e.

The only correct way to prove this was by finding a lower bound on $N_k(n)$ (or $N'_k(n)$) and then showing that $\log_2 N_k(n) = \Omega(k \log n)$.

You could NOT prove the lower bound by saying saying that there are n/k pieces of size k and each piece requires $k \log k$ comparisons to sort. This would only be a proof that an algorithm that works by splitting the array up into those pieces would require $\Omega(n \log k)$ comparisons. It is not a proof that all possible algorithms would require $\Omega(n \log k)$ comparisons

Marking Note 1eii. Some students used the observation from the 4th algorithm to argue that a k-swapped array can be partitioned into $k+1$ sorted arrays, that is, for every $j \in [1, k+1]$ the items $A[j], A[j+(k+1)], A[j+2(k+1)], A[j+3(k+1)]$, etc. are sorted. Call such an array a k-sorted array. You can show that (if $(k+1)|n$) then the number of k-sorted arrays is

$$g(n,k) = \frac{n!}{((n/(k+1))!)^k}.$$

They then showed that $\log_2 g(n,k) = \Theta(n \log k)$. they ended by saying that since every k-sorted array is a k-swapped array, this proves the statement.

The flaw here is that while a k-swapped array is a k-sorted array, not every k-sorted array is a k swapped one. To make this proof correct,

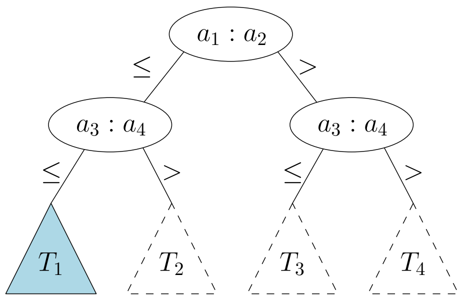
you needed to explicitly note that this is an undercount. If you stated that they are the same, it would be wrong.

Problem 2 Decision Trees [15 pts]

Recall that a comparison-based sorting algorithm can be represented in the (binary) decision tree model.

Following the worked example of Tutorial SS13, the figure below shows part of the decision tree for mergesort operating on a list of 4 numbers, a_1, a_2, a_3, a_4. Please expand subtree T_1, i.e., draw and label all of the edges, internal (comparison) nodes and leaves in subtree T_1.

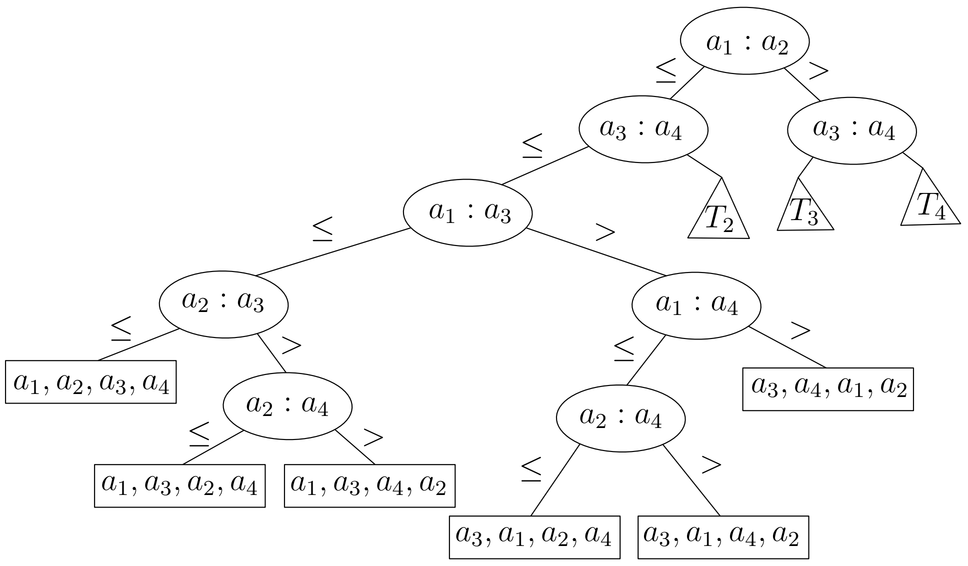
Edges in the tree must be labelled with $\leq$ or $>$ and leaves must show the final sorted order.



Solution:

T_1 merges the two sorted lists a_1 a_2 and a_3 a_4 of size 2.

The decision tree below lists the questions that the algorithm asks in the order in which it asks them.



Problem 3 Radix sort [16 pts]

You are given a set of 10 decimal integers in the range of 1 to 65535:

$$A = [29681, 53846, 43521, 39427, 32433, 35700, 30764, 16892, 52608, 19583].$$

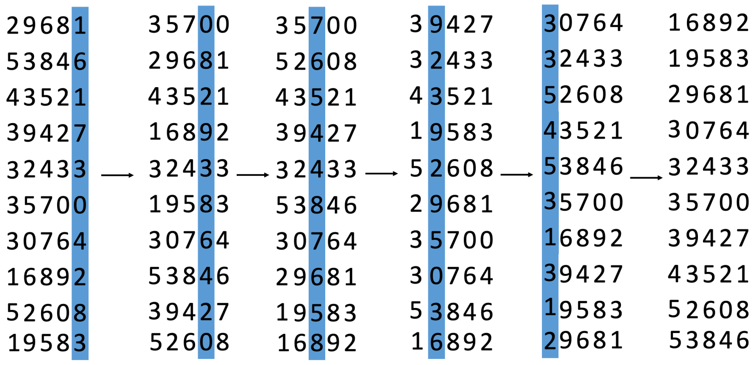
(a) Please conduct Radix sort on A using Base 10. Illustrate your result after each step following the worked example on Page 36 in the 08_Linearsort lecture slides.

(b) Now convert these decimal integers to hexadecimal and conduct Radix sort again, this time using Base 16 (hexadecimal). Illustrate your result after each step following the worked example on Page 36 in the 08_Linearsort lecture slides.

Note: a digit in hexadecimal is in the range of $[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F]$.
To start you off, note that 29681 *is* $73F1$ *in hexadecimal.*
There are many decimal to hexadecimal converters available on the intern.
We recommend you use one of them to covert the data rather than doing it by hand.

Solution

(a) Radix sort using base 10 (digits to be sorted are highlighted in blue):

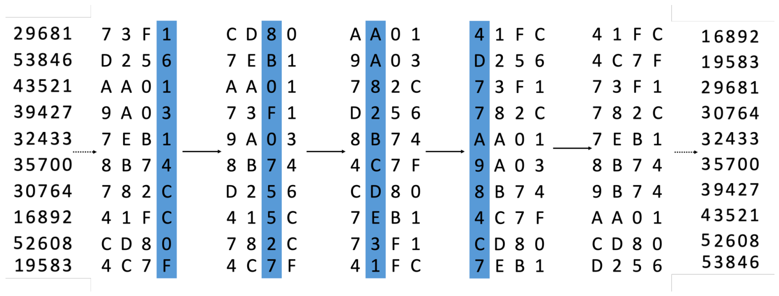


29681	35700	35700	39427	30764	16892
53846	29681	52608	32433	32433	19583
43521	43521	43521	43521	52608	29681
39427	16892	39427	19583	43521	30764
32433	32433	32433	52608	53846	32433
35700	19583	53846	29681	35700	35700
30764	30764	30764	35700	16892	39427
16892	53846	29681	30764	39427	43521
52608	39427	19583	53846	19583	52608
19583	52608	16892	16892	29681	53846

(b) Convert A to hexadecimal:

$$A = [4C7F, CD80, 41FC, 782C, 8B74, 7CB1, 9A03, AA01, D256, 73F1]$$

Radix sort using base 16 (digits to be sorted are highlighted in blue):



29681	7 3 F 1	C D 8 0	A A 0 1	4 1 F C	4 1 F C	16892
53846	D 2 5 6	7 E B 1	9 A 0 3	D 2 5 6	4 C 7 F	19583
43521	A A 0 1	A A 0 1	7 8 2 C	7 3 F 1	7 3 F 1	29681
39427	9 A 0 3	7 3 F 1	D 2 5 6	7 8 2 C	7 8 2 C	30764
32433	7 E B 1	9 A 0 3	8 B 7 4	A A 0 1	7 E B 1	32433
35700	8 B 7 4	8 B 7 4	4 C 7 F	9 A 0 3	8 B 7 4	35700
30764	7 8 2 C	D 2 5 6	C D 8 0	8 B 7 4	9 B 7 4	39427
16892	4 1 F C	4 1 5 C	7 E B 1	4 C 7 F	A A 0 1	43521
52608	C D 8 0	7 8 2 C	7 3 F 1	C D 8 0	C D 8 0	52608
19583	4 C 7 F	4 C 7 F	4 1 F C	7 E B 1	D 2 5 6	53846

Problem 4 Greedy Covering [29 pts]

Consider the following (unrealistic) scenario. You are running a political campaign and want to collect petition signatures in the park. Your research team tells you that everyone who goes to the park visits for one of n known time intervals during the day (if they come for a particular interval, they stay for the entire time interval; note that different time intervals might overlap).

If you go to the park at any time during one of the intervals you will get the signatures of everyone there for that interval. You want to figure out a minimum sized set of times that you have to go to the park to get everyone's signature.

In computer science such a situation is called a *covering problem* and is modelled formally as described below.

- A real *interval* is $I = [s, f]$ where $s \le f$
- A real *point* x *covers* interval $I = [s, f]$ if $x \in I$, i.e., $s \le x \le f$.
- Let $\mathcal{I} = \{I_1, \ldots, I_n\}$ be a set of n intervals and $A = \{a_1, a_2, \ldots a_k\}$ a set of points.

  Then "A covers $\mathcal{I}$" if every interval in $\mathcal{I}$ is covered by at least one point in A. More formally, if, for every $I_j \in \mathcal{I}$ there exists $a_i \in A$ such that $a_i \in I_j$.
- $|A|$, the size of A, is just the number of points in A.

  A is a *minimal cover* of $\mathcal{I}$ if A is a smallest cover of $\mathcal{I}$.

  That is, for every cover A' of $\mathcal{I}$, $|A| \le |A'|$.

The input to this problem is the $2n$ numbers $s_1, f_1, s_2, f_2, \ldots, s_n, f_n$ with $s_j \le f_j$, that define $\mathcal{I}$; $I_j = [s_j, f_j]$.

The problem is to construct a minimal cover for $\mathcal{I}$ in $O(n \log n)$ worst-case time. Your output should be a set A which is a minimal cover.

When solving this problem you may use any algorithm we taught in class (but not the tutorial) as a black box as long as you quote which algorithm you are using. You can assume that any properties of that algorithm taught in class are correct as long as you quote what the properties are. Any other subroutines and their properties must be derived from scratch.

(a) Prove that there exists a minimal cover A such that every point in A is one of the f_j. That is, $A \subseteq \{f_1, \ldots, f_n\}$.

(b) Give documented pseudocode for your algorithm.

In addition, below your algorithm's pseudocode, explain in words/symbols what your algorithm is doing

(c) Prove correctness of your algorithm. Your proof must be mathematically formal and understandable.

Note: Break your proof up into clear logical pieces and skip space between the pieces. Explicitly state what each part is assuming and proving.

For examples of such proofs please see the lecture notes and tutorials.

As seen in class, greedy algorithms tend to be simple. Their proofs of correctness are more complicated.

(d) Explain why your algorithm runs in $O(n \log n)$ worst case time.

Solution:

Note. this problem was taken and modified from the book "Learning Algorithms Through Programming and Puzzle Solving", Alexander S. Kulikov and Pavel Pevzner, Active Learning Technologies, 2019.

(a) *Note. this part was meant as a first hint to solving the problem. In reality a proof of part (c) (written properly) could also prove (a).*

Let A be any cover of $\mathcal{I}$, Suppose $x \in A$ is not one of the f_j.
Let $\mathcal{I}' = I_{i_i}, I_{i_2}, \ldots I_{i_k}$ be the set of intervals covered by x.
This means that, for all $t = 1, 2, \ldots, k$, $x \in [s_{i_t}, f_{i_t}]$. Set

$$\bar{s} = \max_{1 \le t \le k} s_{i_t} \quad \text{and} \quad \bar{f} = \min_{1 \le t \le k} f_{i_t}.$$

Then $x \in \left[\bar{s}, \bar{f}\right]$ and for all $t = 1, 2, \ldots, k$,

$$s_{i_t} \le \bar{s} \le \bar{f} \le f_{i_t}$$

so $\bar{f}$ covers all of the intervals in $\mathcal{I}'$.
This means that if interval $I \in \mathcal{I}$ is covered by A then I is also covered by $A' = A \cup \{\bar{f}\} - \{x\}$. Since A covers $\mathcal{I}$, A' also covers $\mathcal{I}$.
Since $|A'| = |A|$, A' is also a minimal cover of $\mathcal{I}$ so is A'.
Thus starting with any minimal cover of A we can successively replace any point in A that is not a f_j with a point that is a f_j, maintaining that A is a minimal cover.

Marking Note 4a: One incorrect solution was to start with A being the set of all the n finishing times. This A is a cover. Then, for every point in A, check if removing it keeps A as a cover. If yes, remove that point. The resulting set was then claimed to be a minimal cover of A.

This was wrong. The resulting set is certainly a minimal cover of A among all subsets of the finishing points but this does not necessarily make it a universally minimum cover.

Another incorrect solution is to start with a optimal cover OPT and, if $x \in OPT$ that covers interval I_i, replace x in OPT with f_i. This fails because it assumes that x covers only one interval I_i. If x covers I_i and I_j then replacing x with f_i might leave I_j uncovered.

(b)

```
1. Sort the $I_j = (s_j, f_j)$ by increasing $f_j$.    % Afterwards $f_1 \le f_2 \le \cdots \le f_n$

2. last = $f_1$; $A = \{\text{last}\}$                  % Initialize

3. For $j = 2$ to $n$ do                               % greedily check if $f_j \in A$.
4.     If $s_j >$ last then
5.         last = $f_j$; $A = A \cup \{\text{last}\}$
```

The algorithm greedily check the intervals by ascending finishing time. If current interval is not covered by A so far, its finishing time is added to A.

Marking Note 4b: The instructions explicitly required in-code documentation and a later explanation. If either of those were missing, points were deducted.

(c)

Fact 1: G is a cover (this needs to be proven).

Proof: This follows directly from the algorithm. Consider the time that I_j is processed. If f_j is added to A, then I_j is trivially covered. If f_j is not added to A then $s_j \le \texttt{last}$. But $\texttt{last} = f_t$ for some $t < j$ so $s_j \le \texttt{last} = f_t < f_j$ so last covers I_j. Thus, every I_j is covered by some point in A. ■

Now let

$$O = \{o_1, o_2, \ldots, o_k\} \quad \text{and} \quad G = \{g_1, g_2, \ldots, g_{k'}\}$$

where $O \subseteq \{f_1, f_2, \ldots, f_n\}$ is some minimal cover (we know such a cover exists from part (a)) and G is a greedy cover.

Assume they are labelled so that $o_1 < o_2 < \cdots < o_k$ and $g_1 < g_2 < \cdots < g_{k'}$. Let r be the smallest value such that, for all $t < r$, $g_t = o_t$ but $g_r \ne o_r$.

Fact 2: (a) $o_1 \le g_1$ and (b) if $r > 1$, $g_{r-1} < o_r \le g_r$.

Proof: For (a) simply note that if x covers I_1, then $x \le f_1$. O must therefore contain some point $\le f_1$ but if $o_1 > f_1$ this would not be possible.

For (b), let $I_j = (s_j, f_j)$ be the interval that added g_r to G.

This implies $g_{r-1} < s_j \le f_j = g_r$. Now suppose that $o_r > g_r$.

Then, for all $t < r$, $o_t \le o_{r-1} = g_{r-1} < s_j$, so none of those o_t can cover I_j.
But, if $t \ge r$, then $o_t \ge o_r > g_r = f_j$, so none of those t can cover I_j either. But this contradicts that SOME some o_t must cover I_j.

Thus $o_r \le g_r$.
The lower bound follows from the fact that $g_{r-1} = o_{r-1} < o_r$. ■

Fact 3: If $\{o_1, \ldots, o_r\}$ covers interval I_j then $\{g_1, \ldots, g_r\}$ covers I_j.

Proof: Suppose not. Since $\{o_1, \ldots, o_{r-1}\} = \{g_1, \ldots, g_{r-1}\}$ the only way that this can occur is if I_j is covered by o_r and not by g_r (or g_{r-1}).
Because o_r covers I_j, $s_j \le o_r \le f_j$.

If $s_j \le g_{r-1}$, then, from Fact 2, $s_j \le g_{r-1} < o_r \le f_j$ so I_j is covered by g_{r-1}, contradicting that g_{r-1} does not cover I_j. Thus $s_j > g_{r-1}$.

If $f_j \ge g_r$, then, again from Fact 2, $s_j \le o_r \le g_r \le f_j$ so I_j is covered by g_r, contradicting that g_r does not cover I_j. Thus $f_j > g_r$.

But then, $g_{r-1} < s_j < f_j < g_r$ which means that I_j is not covered by ANY point in G, contradicting that G is a cover (from Fact 1). ■

Fact 4: $O' = O \cup \{g_r\} - \{o_r\}$ is a minimal cover of $\mathcal{I}$.

Proof: Since $|O'| = |O|$ and O is minimal we only need to prove that A' is a cover.

Let $I \in \mathcal{I}$ be any interval. I must be covered by some $o_t \in O$. If $t \neq r$ then $o_t \in O'$ so I is also covered by O'. If $t = r$ then I is covered by $\{o_1, \ldots, o_r\}$ which, from Fact 3, implies that I is covered by $\{g_1, \ldots, g_r\} \subseteq O'$ so I is covered by A'. Thus O' is a cover of $\mathcal{I}$. ■

We have just seen that if

$$G = \{g_1, g_2, \ldots, g_{r-1}, g_r, g_{r+1}, \ldots g_{k'}\} \neq \{g_1, g_2, \ldots, g_{r-1}, o_r, o_{r+1}, \ldots o_k\}$$

where O is a minimal cover then

$$O' = \{g_1, g_2, \ldots, g_{r-1}, g_r, o_{r+1}, \ldots o_k\}$$

is a minimal cover as well. By repeating this replacement process, each time increasing the size of the prefix of O that matches with G we can construct an optimal cover which is equal to G, so G is optimal.

Alternative correctness proof via induction.

Let $G(\mathcal{I})$ and $O(\mathcal{I})$, respectively, represent the greedy and some optimal solution for input $\mathcal{I}$. The induction hypothesis will be that,

$$I(n) : \textbf{for all inputs } \mathcal{I} \textbf{ with } |\mathcal{I}| = n\textbf{,}\ |G(\mathcal{I})| = |O(\mathcal{I})|\textbf{.}$$

This is obviously true for $n = 1$.

Assume that the hypothesis is correct for all values $< n$.

Let $\mathcal{I} = \{I_1, \ldots, I_n\}$ be a set of n intervals and set $O = O(\mathcal{I})$ and $G = G(\mathcal{I})$. Assume $f_1 \leq f_2 \leq \cdots \leq f_n$. Set

$$\mathcal{I}' = \{I_j \in \mathcal{I} \,:\, f_1 \notin [s_j, f_j]\}, \quad \mathcal{I}'' = \{I_j \in \mathcal{I} \,:\, o_1 \notin [s_j, f_j]\};$$

$\mathcal{I}'$ is the set of intervals not covered by f_1 while $\mathcal{I}''$ is the set of intervals not covered by o_1.

By Fact 2(a), $o_1 \leq f_1$. Thus, because the f_i are sorted in ascending order,

$$\mathcal{I}' \subseteq \mathcal{I}''.$$

Next observe that, by the workings of the algorithm, $G(\mathcal{I}) = \{f_1\} \cup G(\mathcal{I}')$. Thus $|G(|\mathcal{I})| = 1 + |G(\mathcal{I}')|$.

Combining the pieces then gives

$$\begin{array}{rcll} |G(|\mathcal{I})| & = & 1 + |G(\mathcal{I}')| & \\ & = & 1 + |O(\mathcal{I}')| & \text{(By the induction hypothesis)} \\ & \leq & 1 + |O(\mathcal{I}'')| & \text{(Because } \mathcal{I}' \subseteq \mathcal{I}'' \text{)} \\ & = & |O(\mathcal{I})|. & \text{(Because } OPT(\mathcal{I}) \text{ is optimal for } \mathcal{I}\text{)} \end{array}$$

But $O(\mathcal{I})$ is a minimal cover so $|G(\mathcal{I})| \geq |O(\mathcal{I})|$. Thus $|G(\mathcal{I})| = |O(\mathcal{I})|$, which is what we wanted to prove.

(d) Line (1) of the algorithm uses $O(\log n)$ worst case time, e.g., by using Mergesort.

Line (2) uses $O(1)$ time and lines 3-5 implement a $O(n)$ step loop that does $O(1)$ work each time the loop is incremented.

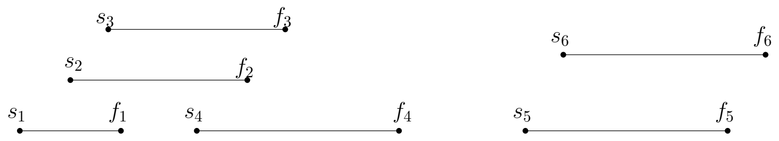
Total time is then $O(n \log n) + O(n) = O(n \log n)$.

Marking note 4c1: In order for the algorithm to be correct you needed to prove that the resulting A is a minimal cover. This first required proving that A is a cover since, unlike in the Interval Scheduling problem, this isn't "obvious" and needed to be explicitly noted.

Solutions missing an explicit statement/proof as to why the algorithm in (b) produced a cover had a few points deducted.

Marking Note 4c2. In the proof structure given above it would be tempting to base a proof on the following statement:
NotAFact: (*) If o_r covers interval I_j then g_r covers I_j.

This is actually not correct as the following example illustrates



$G = \{f_1, f_4, f_5\}$. Note that $O = \{f_1, f_2, f_5\}$ is an optimal solution. Then $r = 2$. Note that (*) is NOT correct since $o_2 = f_2$ covers I_3 but $g_2 = f_4$ does NOT cover I_3.

A proof based on statement NotAFact: would be incorrect. You would instead explicitly need something a little more specific, such as Fact 3 above.

Marking Note 4c3: Some students used the structure above, i.e., defining the equivalent of G, O and r.

They then *stated without proof* that $O \cup \{g_r\} - \{o_r\}$ is also a cover.

They then preceded correctly from there.

This was marked wrong. The MAIN part of the proof of correctness is PROVING that $O \cup \{g_r\} - \{o_r\}$ is a cover. Any "proof" missing a proof of that crucial step was marked partially wrong.

Marking Note 4c4. A solution using Quicksort had a point deducted. The problem explicitly requested an $O(n)$ worst-case time solution and Quicksort is not $O(n \log n)$ worst-case time.

P5: [10 pts] **Huffman Coding**

The table below lists 10 letters (a to j) and their frequencies in a document. Apply Huffman coding (following the algorithm on Page 11 of the 10_Huffman.pptx lecture notes) to construct an optimal codebook. Your solution should contain three parts:

(a) A full Huffman tree.

(b) The final codebook that contains the binary codewords representing a to j (sorted in the alphabetical order on a to j).

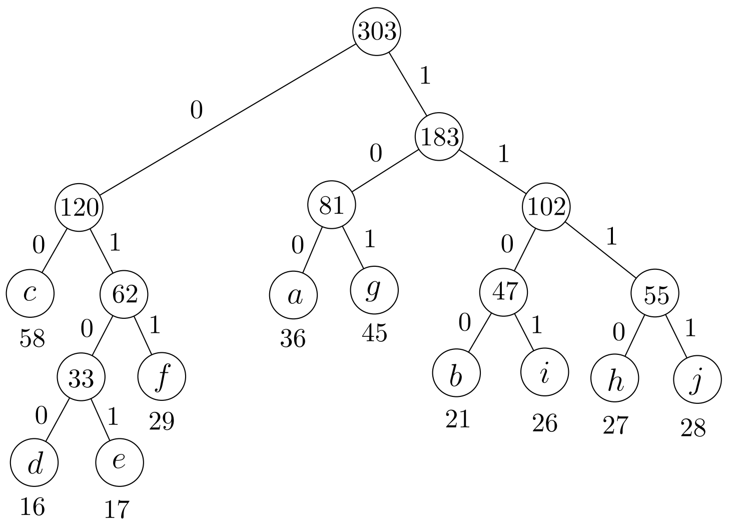
(c) The codebook from part (b) but now sorted by the lengths of the codewords (in increasing order).

i	1	2	3	4	5	6	7	8	9	10
a_i	a	b	c	d	e	f	g	h	i	j
$f(a_i)$	36	21	58	16	17	29	45	27	26	28

Rules, Recommendations and Hints:

- In part (a) you should explicitly label each edge in the tree with a '0' or '1'. You should also label each leaf with its corresponding character a_i and $f(a_i)$ value.
- Part (b) should be a table with 2 columns. The first column should be the letters a to j. The second column should be the codeword associated with each character.
- In part (c) you should break ties using the alphabetical order of the characters. For example, if a, d and e all have codewords of length 5, then write the codeword for a on top of the codeword for d on top of the codeword for e.

Solution 5(a):



Solution 5(b) and 5(c):

a	100
b	1100
c	00
d	0100
e	0101
f	011
g	101
h	1110
i	1101
j	1111

c	00
a	100
f	011
g	101
b	1100
d	0100
e	0101
h	1110
i	1101
j	1111