# Assignment 2

Due 23:59, Tuesday, February 5, 2019

**CS ID 1**:

**CS ID 2**:

**Instructions:**

1. Do not change the problem statements we are giving you. Simply add your solutions by editing this latex document.
2. Take as much space as you need for each problem. You'll tell us where your solutions are when you submit your paper to gradescope.
3. Export the completed assignment as a PDF file for upload to gradescope.
4. On gradescope, upload only **one** copy per partnership. (Instructions for uploading to gradescope will be posted on the HW2 page of the course website.)

1. **[Three Pile Radix Sort – 42 points].**

### 1.1 Quicksort

In practice, quicksort is one of the fastest sorting algorithms.

1. Pick a **pivot** (e.g. the first element)

| bee | bed | cab | ace | dad | baa | add | ebb |
|---|---|---|---|---|---|---|---|

2. Reorder the array such that all elements $<$ pivot are to its left, and all elements $\geq$ pivot are to its right.

| add | bed | ace | baa | bee | cab | dad | ebb |
|---|---|---|---|---|---|---|---|
| left_partition | | | | | right_partition | | |

3. Recursively sort each partition.

```
void qsort(vector<string> & x, int lo, int hi) {
  if (lo >= hi) return;
  int p = lo;
  for( int i=lo+1; i <= hi; i++ )
    if( x[i] < x[lo] ) { p++; swap(x[p], x[i]); }
  swap(x[lo], x[p]);
  qsort(x, lo, p-1);
  qsort(x, p+1, hi);
}
void quicksort(vector<string> & x) {
  qsort(x, 0, x.size()-1);
}
```

(a) (4 points) Show the contents of the array `x` at the start of each iteration of the for-loop in the call `qsort(x, 0, 3)` on the input shown below:

| | `lo=0` | 1 | 2 | `hi=3` | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| $i = 1$ | add | bed | ace | baa | bee | cab | dad | ebb |
| 2 | | | | | bee | cab | dad | ebb |
| 3 | | | | | bee | cab | dad | ebb |
| 4[1] | | | | | bee | cab | dad | ebb |
| 5[2] | | | | | bee | cab | dad | ebb |

[1] at the end of iteration $i = 3$. [2] after `swap(x[lo],x[p])`.

(b) (4 points) Complete the following loop invariant by providing the range of indices for which the statement is true. Note that `x[` $a$ ... $b$ `]` is empty if $a > b$.

At the start of iteration $i$ of the for-loop in `qsort`:

(A) `x[` _____ ... _____ `]` $<$ `x[lo]` and

(B) `x[` _____ ... _____ `]` $\geq$ `x[lo]`

(c) (4 points) To prove the loop invariant, we use induction on the iteration index $i$. In the base case, when `i=lo+1` and `p=lo`, the ranges for parts (A) and (B) are empty and the invariant is trivially true. For the inductive step, going from iteration $i$ to iteration $i' = i + 1$:

i. If `x[i] < x[lo]`, the value of $p$ in the next iteration is $p' = p+1$ and the order of strings in the vector $x$ changes. If we replaced `swap(x[p],x[i])` with `x[p]=x[i]`, which part of the invariant would be false at iteration $i + 1$, (A) or (B)?

ii. If `x[i] >= x[lo]`, which part of the invariant at iteration $i + 1$ is *the same* as at iteration $i$, (A) or (B)?

(d) (4 points) What is the worst case running time of Quicksort on inputs of size $n$? For full credit, show the recurrence relation representing the running time for a worst case input.

(e) (4 points) Suppose in every recursive call to sort a subarray, we choose a pivot, in time proportional to the size of the subarrary, that is never one of the smallest 1/4 or largest 1/4 elements. What is the worst case running time of Quicksort on inputs of size $n$ in this case?

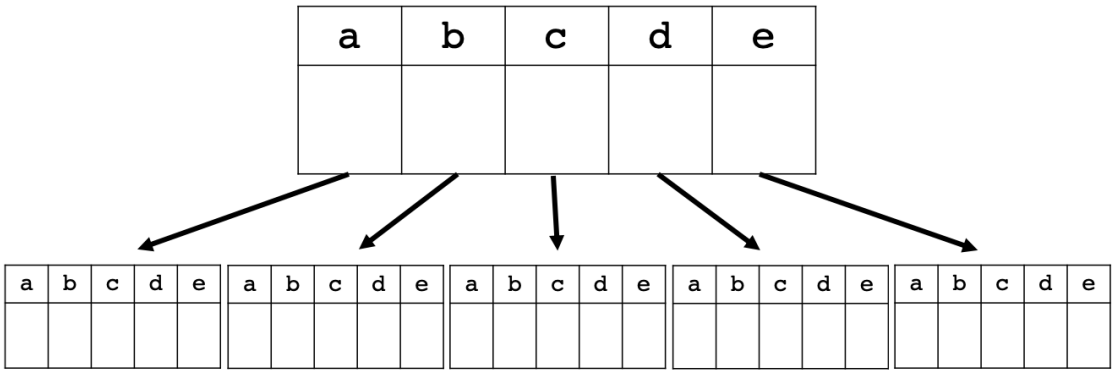
### 1.2 Radix sort

Left-to-Right Radix sort does not sort strings by comparing them. It looks at the first character in each string and divides the strings into groups (called piles) based on this character. So all the strings beginning with "a" go into the same pile. In other words, it uses the first character as the index for where to store the string in an array of piles. It then recursively sorts the strings in each pile, which all have the same first character, starting with the second character. The integer `j` in the code below indicates which character to start at: the first call is L2RRadixSort( `S, 0` ). We assume that the strings are composed of the characters from the alphabet `{a,b,c,d,e}`.

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| S[0] | a | c | c | e | d | e | d | ∅ |
| S[1] | a | d | d | ∅ | | | | |
| S[2] | b | a | c | c | a | e | ∅ | |
| S[3] | b | e | a | d | ∅ | | | |
| S[4] | b | e | d | d | e | d | ∅ | |
| S[5] | c | e | d | e | d | ∅ | | |
| S[6] | d | a | b | b | e | d | ∅ | |
| S[7] | d | e | c | a | d | e | ∅ | |

```
L2RRadixSort( vector<string> & S, int j )
1. If S.size() <= 1 return
2. For every string str in S
3.     Put str in Pile[c] where c=str[j]
4. vector<string> T
5. For every character c in alphabetic order
6.     L2RRadixSort( Pile[c], j+1 )
7.     Append Pile[c] to T
8. S=T
```

(a) (4 points) Show us that you understand the algorithm by executing two levels of recursion on the given set of 8 strings, placing the piles of strings in the appropriate array cells. Please identify the strings using only their index numbers from the table above. At the end of each call to the function do not reassemble the data into a vector–we just want to see the piles! After you have finished the two levels, circle all of the piles that will require further classification.




(b) (4 points) In the original table of strings, mark the cells that are examined by a complete execution of Left-to-Right Radix sort. We have marked (shaded) the cells corresponding to the examination of the first character for you.

(c) (1 point) If we measure the size of the data by the total number of characters in the data set, what fraction of this data set was explored in the complete execution of the algorithm? (Ignore the terminating nulls.)

(d) (4 points) Finally, fill in the table below with a set of 8 unique strings on which Left-to-Right Radix sort examines all the characters. Assume `abc < abca`

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| `S[0]` | | | | | | | | ∅ |
| `S[1]` | | | | ∅ | | | | |
| `S[2]` | | | | | | | ∅ | |
| `S[3]` | | | | | ∅ | | | |
| `S[4]` | | | | | | | ∅ | |
| `S[5]` | | | | | | ∅ | | |
| `S[6]` | | | | | | | ∅ | |
| `S[7]` | | | | | | | ∅ | |

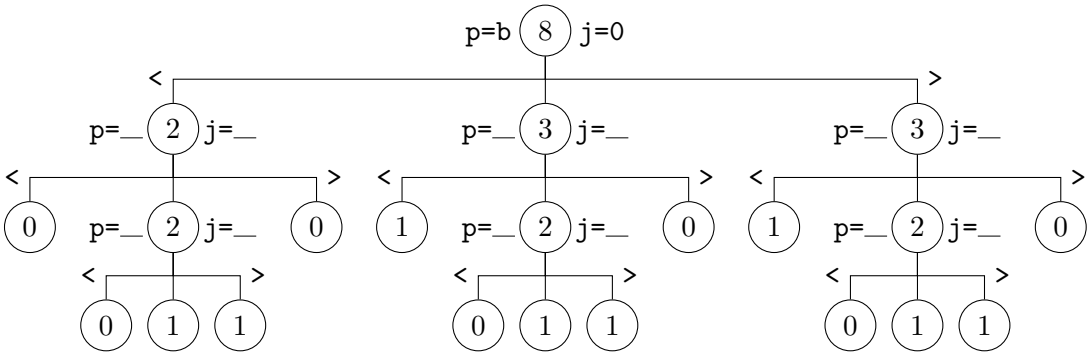
### 1.3 Three Pile Radix sort

Radix sort works for strings over larger alphabets than five characters – just use more piles – but at some point it seems wasteful to have so many piles. Many might be empty. Rather than using more piles, let's consider using less. Following Quicksort's approach, we could pick a character $p$ from the alphabet and partition the strings into those whose first character is less than $p$, those whose first character equals $p$, and those whose first character is greater than $p$. Three piles! The middle pile is exactly the same as the pile for character $p$ in radix sort. The first and last pile are like the two partitions formed using $p$ as the pivot in quicksort.

(a) (3 points) Fill in the blanks in the following pseudo-code to supply the appropriate value of $j$ to the recursive calls:

P3RadixSort(`vector<string>` $S$, `int` $j$)
1. If $S$ contains 0 or 1 string, return.
2. Pick a pivot character $p$
3. Divide $S$ into three piles:
   $P_<$: strings whose $j$th character is less than $p$
   $P_=$: strings whose $j$th character equals $p$
   $P_>$: strings whose $j$th character is greater than $p$
4. P3RadixSort($P_<$, ☐)
5. P3RadixSort($P_=$, ☐)
6. P3RadixSort($P_>$, ☐)
7. Copy (the now sorted) $P_<$, $P_=$, $P_>$ back into $S$ in that order.

(b) (6 points) The following recursion tree shows the execution of P3RadixSort on the set of strings in problem 1.2. The circles contain the number of strings passed to the subproblem (the root represents the initial problem of sorting 8 strings). The "`j=0`" next to the circle is the value of `j` passed to the subproblem (0 for the initial problem). The character `b` below the circle is the pivot character $p$ chosen by the subproblem to divide the strings into three piles, which are the three children of the root. Your job is to complete the recursion tree by supplying the $j$ values and pivot values for all the nodes except the root (which is given) and the leaves (which have no children).



2. **[Soccer Analytics for the Win – 30 points].**

   Christine Sinclair (https://en.wikipedia.org/wiki/Christine_Sinclair) is a Canadian National Team soccer player who spends her professional life playing for the Portland Thorns FC. Suppose you would like to develop a query tool that would allow you to assemble statistics quickly for Christine and other NWSL players. As proof of concept, you will store Christine's career scoring data in an array whose indices represent games, ordered by date and whose entries are the number of goals she scored. That is, `G[j]` contains the number of goals she scored in game j, for games in the range 1 to $n$. (`G[0]` is always 0.)

   (a) (4 points) Assuming that $1 \leq i \leq j \leq n$, prove the correctness of the following algorithm by arguing that it returns the total number of goals between games $i$ and $j$, inclusive, for any $1 \leq i \leq j \leq n$.

```
1  int getTotal(const vector<int> & G, int i, int j){
2     if (i == j)
3        return G[i];
4     else
5        return G[j] + getTotal(G, i, j-1);
```

   i. Base case: argue briefly that the base case is correct.

   ii. State an appropriate inductive hypothesis:

   iii. Prove the inductive case:

   (b) (2 points) What is the worst case running time of `getTotal(G, 1, n)` as an asymptotic function of $n$?

(c) (2 points) Unfortunately, the result in the previous part of the problem just isn't fast enough. We would like to be able to report Christine's total scoring within any range of games, in constant time. To do this, we pre-compute her total scores in *all possible* consecutive sets of games. Exactly how many totals do we need to compute assuming $n$ games?

(d) (4 points) Complete the code below to populate a 2D vector that we could use to compute the total number of goals in any range of games. You may use the `getTotal` function from part (a).

```
1  int main() {
2      vector <int> G;
3      ... // initialize G to contain Christine's stats
4      vector <vector <int>> Q;
5
6      for (int i = 1; i <= ________; ________) {
7          vector <int> temp;
8
9          for (int j = ________; j <= ________; ________)
10
11             temp.pushBack(_________________);
12
13         ___.pushBack(temp);
14     }
15     ... // use Q to answer queries
16     return 0;
17 }
```

What is the worst case running time of this algorithm to build the query structure, and how much space does it require, asymptotically in terms of $n$?

Time: Space:

(e) (2 points) What is the running time to use `Q` to answer queries like "how many goals did Christine score between games $i$ and $j$? (inclusive)" for any $1 \leq i \leq j \leq n$?

(f) (4 points) Let's be clever about this... suppose we build an array `A` whose size is the same as `G`, but whose entries are cumulative total goals since the start of the season (assume `A[0]=0`). What is the running time to build such a structure, and how much space does it use as an asymptotic function of $n$? (We are not asking you to show us the pseudocode solution to this problem, but you should sketch it out in order to answer the question!)

Time: Space:

(g) (4 points) Give an expression in terms of the new array `A` that can be used to answer queries like "how many goals did Christine score between games $i$ and $j$, inclusive? Also tell us the asymptotic running time for answering the query.

Expression: [ ] Time: [ ]

(h) (4 points) Finally, we are going to design an algorithm to build *another* structure that will allow us to answer queries of the form "at game $j$ of her career, what was her most productive run of $k$ games?" for any $1 \leq j \leq n$, and $1 \leq k \leq j$.

Our goal, in the end, will be to have an array `V`, whose entry `V[j][k]` gives the number of the last game of the most productive $k$ game stretch, in the range of games from 1 through $j$, and for $1 \leq k \leq j$. For example, if `V[28][3] == 14`, then the most productive 3 game stretch in the first 28 games of her career is the set of games numbered 12, 13, and 14.

Suppose you have accurately built array `A`. Assume, in addition, you have (perhaps magically) completed iteration $j$ of the algorithm to compute `V`, so that `V[j-1][k]` indicates the last game of the most productive interval of length `k` within games 1 to $j-1$, for all $k = 1, 2, .., j-1$. This is an invariant on the state of `V`.

Complete the code below to maintain the invariant on array `V` in the next iteration of a loop. To do this, think about how the entries of `V` might change, given `A[j]`.

```
1  void updateMostProductive(const vector<int> & A, vector<int> & V, int j) {
2      //assume A[0..n] contains correct cumulative sums
3      //assume V[1..j-1][1..j-1] is correct.
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18     //end condition: V[1..j][1..j] is correct.
19     return;
20 }
```

Note that articulating this algorithm gets you most of the way toward implementing an algorithm to compute `V` for all of the data, since you just have to run it once for each $j$.

(i) (2 points) Suppose you have used this algorithm to compute `V`. It can be used to ask interesting questions about the trajectory of the athlete's career. Suppose, for example, that you observe that `V[j][4]` $>$ `V[j/2][4]`, for all $j$. What does the observation mean? Should Christine retire?

(j) (2 points) What is the asymptotic running time of an algorithm for answering the question from the previous part, assuming that `V` has already been computed?

Blank sheet for extra work.