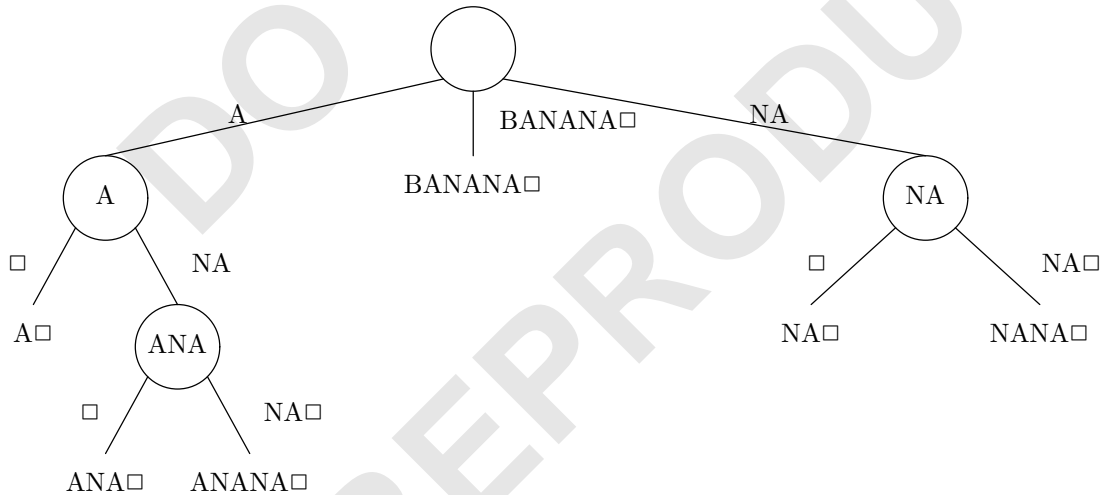


Unless a question says otherwise, time estimates refer to asymptotic bounds ($O(\dots)$, $\Omega(\dots)$, $\Theta(\dots)$). Always give the simplest bounds possible ($O(f(x))$ is “simpler” than $O(f(x) + g(x))$ and $O(Kf(x))$).

1. [3 points] A *suffix tree* for a string S is a trie that contains all suffixes of S . For example, if S is “banana□”, its suffix tree is a trie that contains the seven strings “banana□”, “anana□”, “nana□”, ..., “□” (where ‘□’ is a unique terminating character).

a. [2 points] Draw the suffix tree for “banana□”.

In the following, long chains of states with a single child are condensed into an edge showing all the letters along the way.



b. [1 point] Given a string P and a suffix tree for a string S , describe succinctly how to determine whether P is a substring of S (i.e., not necessarily a suffix) in time proportional only to the length of P .

Simply traverse the tree from the root, following edges labeled by the characters in sequence (as for ordinary search of tries). If one never encounters a non-existent edge, then the substring must be in the trie.

2. [3 points] In a directed graph with N nodes, let the edges be represented by an $N \times N$ array E , so that $E[i][j]$ is the weight (or length) of the edge from node i to node j (assume nodes are identified by numbers $0..N-1$). $E[i][i] = 0$ for all i . If there is no edge from i to j , then $E[i][j] = \infty$.

- a. [1 point] The following algorithm (known as the Floyd-Warshall algorithm) computes $d[i][j]$, the length of the shortest path from node i to node j , for *all* pairs i and j :

```
Initialize d to the contents of E;
for (int k = 0; k < N; k += 1)
  for (int i = 0; i < N; i += 1)
    for (int j = 0; j < N; j += 1)
      d[i][j] = Math.min (d[i][j], d[i][k] + d[k][j]);
```

Assuming that `Math.min` (the minimum function) takes constant time, what is the *best-case* time cost of this algorithm, as a function of N , the number of vertices? What is the worst-case time cost?

This is three nested loops, each executing N iterations once started, independent of the data. Thus, we get $\Theta(N^3)$ time.

- b. [2 points] Under the same assumptions as part (a), what are the best and worst case costs of computing $d[i][j]$ for all i and j using Dijkstra's algorithm, assuming that the graph is connected? Give your reasoning. Under what circumstances, if any, is it better to use Dijkstra's algorithm for this purpose than the Floyd-Warshall algorithm?

Dijkstra's algorithm takes time $\Theta((N + E) \lg N)$, where E is the number of edges to compute all $d[i][j]$ for a single i . Thus, we get $\Theta(N(N + E) \lg N)$ for all $d[i][j]$. E can vary from $N - 1$ up to $\Theta(N^2)$, so the time is $\Omega(N^2 \lg N)$ and $O(N^3 \lg N)$. So when edges are sparse (or you only care about one node), Dijkstra's algorithm can be better.

3. [2 points] Consider 2-4 trees containing the (integer) keys 1–15.

a. Illustrate the 2-4 tree with these keys having the *smallest* possible height.

Root contains (4, 8, 12). Four children are (1, 2, 3), (5, 6, 7), (8, 9, 10), (13, 14, 15).

b. Illustrate a 2-4 tree containing keys 1–15 and having the *largest* possible height.

All nodes have one key. Root has 8; level 1 children have 4 and 12; level 2 children have 2, 6, 10, 14; and level 3 children have 1, 3, 5, 7, 8, 11, 13, and 15.

4. [4 points] Consider the problem of sorting N distinct strings each of length $K = 4 \lg N$ using a radix sort, assuming that the strings contain only the characters 'A' and 'B'.

- a. Using LSD radix sort, how long will this take; that is, what are the best- and worst-case times? Give your reasoning. Assume that (as in Java), moving a string is a constant-time operation (that is, it involves copying a pointer rather than the characters themselves).

We sort by one character in linear time K times, giving $\Theta(NK) = \Theta(N \lg N)$ time regardless of the data.

- b. Suppose instead we sort the strings in part (a) using MSD radix sort. Now what are the best and worst-case times? Give your reasoning.

We do linear-time sorts of subsequences of the whole sequence. If the sequences differ in their initial characters (best case), then after we have sorted the first $\lg N$ characters, all sublists will be 1 string long and we'll be done (time $\Theta(N \lg N)$). This only a constant-factor difference from the LSD case, of course, but we have examined fewer characters. If the characters differ at the end (worst case), we do the same amount of work as in the LSD case, because the sorts by the initial characters keep giving us a single sublist.

5. Fill in the blank to correctly complete the quotation below:

She got the which of what-she-did,
Hid the bell with a blot, she did,

But she fell in love with a hominid,
Where is the which of the what-she-did?

From "The Ballad of Lost C'mell" by Cordwainer Smith.

6. [8 points] In discussing random numbers, I remarked that the following is *not* a good general solution to the problem of randomly selecting K different numbers in the range $0..N-1$ when N is much larger than K :

```
/** Fill B with a random selection of A.length different integers in
    the range 0 .. N-1, where N >= B.length, using R as a random source. */
void choose1 (int[] B, int N, java.util.Random R) {
    int K = B.length;
    int[] A = new int[N];
    for (int i = 0; i < N; i += 1)
        A[i] = i;
    for (int k = 0; k < K; k += 1) {
        swap A[N-1-k] with A[R.nextInt (N-k)];
        B[k] = A[N-1-k];
    }
}
```

(`.nextInt(p)` selects a pseudo-random number in the range 0 to $p-1$.) This algorithm fills an array with $0..N$, and then swaps the last item with a random item in the array, the second to last with a random item in the first $N-1$ items, and so forth for a total of K items.

- a. [1 point] Give a time bound on `choose1` as a function of N , K , or both, assuming that `.nextInt` requires constant time.

Creating and filling the array: $\Theta(N)$. Doing K swaps: $\Theta(K)$. Since $K < N$, this is $\Theta(N)$ total.

- b. [6 points] With a better choice of data structure in place of array `A`, this algorithm is not so bad. Specifically, I'd like to write it as:

```
void choose2 (int[] B, int N, java.util.Random R) {
    int K = B.length;
    Ints A = new Ints (N); /* A is initialized to 0, 1, ..., N-1 */
    for (int k = 0; k < K; k += 1) {
        A.swap (N-1-k, R.nextInt (N-k));
        B[k] = A.get (N-1-k);
    }
}
```

implemented in such a way that it uses only $O(K)$ space.

Fill in the class definition on the next page to give the desired effect. Use anything in the Java library you want. Don't bother to detect errors.

```

/** A sequence of ints, indexed (like an array) from 0.  Uses space
 * proportional to min(N, M), where N is the size of the sequence and
 * M is the number of swap operations applied. */
public class Ints {
    /** An N-element sequence initialized to { 0, 1, ..., N-1 }. */
    public Ints (int N) {
        size = N;
    }

    /** The length of this sequence. */
    public int size () {
        return size;
    }

    /** The Ith item in this sequence (numbering from 0). */
    public int get (int i) {
        if (changed.containsKey (i))
            return changed.get (i);
        else
            return i;
    }

    /** Swap the Ith and Jth item in this sequence. */
    public void swap (int i, int j) {
        int t = get (i);
        changed.put (i, get (j));
        changed.put (j, t);
    }

    private HashMap<Integer, Integer>
        changed = new HashMap<Integer, Integer> ();
    private int size;
}

```

- c. [1 point] Give a time bound on **choose2** with your implementation of **Ints**, as a function of N , K , or both, and show your reasoning. State any assumptions you make about the speed of methods in the Java library.

*Assuming we get essentially constant-amortized-time performance from HashMaps, the time required to execute **choose2** is $O(K)$.*