Using Cell Phone Keyboards is ($\mathcal{NP}$) Hard

Peter Boothe

Manhattan College
peter.boothe@manhattan.edu

Abstract. Sending text messages on cell phones which only contain the keys 0 through 9 and # and * can be a painful experience. We consider the problem of designing an optimal mapping of numbers to sets of letters to act as an alternative to the standard $\{2 \to \{abc\}, 3 \to \{def\} \ldots\}$. Our overall goal is to minimize the expected number of buttons that must be pressed to enter a message in English. Some variations of the problem are efficiently solvable, either by being small instances or by being in P, but the most realistic version of the problem is $\mathcal{NP}$ hard. To prove $\mathcal{NP}$-completeness, we describe a new graph coloring problem UniquePathColoring. We also provide numerical results for the English language on a standard corpus which describe several mappings that improve upon the standard one. With luck, one of these new mappings will achieve success similar to that of the Dvorak layout for computer keyboards.

Typing on a keyboard which has fewer keys than there are letters in the alphabet can be a painful task. There are a plethora of input schemes which attempt to make this task easier (e.g. [6, 5, 2] and many more), but the one thing they have in common is that all of these input methods use the standard mapping of numeric keys to alphabetic numbers of $\{2 \to \{abc\}, 3 \to \{def\}, 4 \to \{ghi\}, 5 \to \{jkl\}, 6 \to \{mno\}, 7 \to \{pqrs\}, 8 \to \{tuv\}, 9 \to \{wxyz\}\}$. We will consider schemes of rearranging the numbers on the keys to make messages easier to type. Most variants of this problem turn out to be NP-hard, unfortunately.

Before we get any farther, let us sketch the basic problem that we will keep revising and revisiting.

Problem 1 (MinimumKeystrokes).
Instance *A set of letters corresponding to an alphabet A ($|A| = n$), a number of keys k, an input method* IN*, and a set of tuples of words and frequencies W. The frequencies in W are integers, and the words are made up of solely of elements of A. We will treat* IN *as a function which, given a partition of A and a word w, returns how many keystrokes are required to type w.*

Question *What is the best partition of A into k sets, such that the total number of keystrokes to type every word in W its associated frequency times is minimized? Equivalently, what is the partition of P of A ($|P| = k$) that will minimize*

$$\sum_{(w,f) \in W} f * \text{IN}(w, P)$$

Fig. 1. A cell phone keyboard. Each key is mapped to a letter sequence: $2 \rightarrow (abc), 3 \rightarrow (def), 4 \rightarrow (ghi), 5 \rightarrow (jkl), 6 \rightarrow (mno), 7 \rightarrow (pqrs), 8 \rightarrow (tuv), 9 \rightarrow (wxyz)$

We will consider two different real-world schemes for IN (basic typing and T9), and for each variant or sub-variant that is proven $\mathcal{NP}$-hard, we consider the restriction on P that requires that we keep the alphabet in alphabetical order. In all cases where it is computationally feasible we provide results for the case of the English language, on 8-key keyboards, using the British National Corpus[1] (BNC). One feature of note is that no matter what scheme we use, the problem is trivial if the number of keys is not smaller than the number of letters in the alphabet ($k \geq |A|$). Using tiny keyboards is only (computationally) difficult when the keyboards do not have enough keys.

1 Setting a Baseline with the Easiest Problem

The baseline we compare against is the most painful method of text entry. In this method, to type an 'a', the user of the cellphone types the '2' key; to type a 'b' they type '22', to type 'c' they type '222', to type a 'd' they type '3', to type an 'e' they type '33', and so on. To type a word with multiple letters that use the same key, such as 'accept', the user must pause between keystrokes. Thus, the full key entry sequence for 'accept', using '.' to indicate a pause, is '2.22.223378'.

We will neglect the pauses and concern ourselves solely with the number of keystrokes required. This layout and input method imply that 'a' will always require one button press, 'b' will always require two, and so on. Thus, to get a baseline of how many keystrokes are required to enter the entirety of our

corpus of words W, we count the total number of occurrences of each letter, and multiply that number of occurrences by the number of button pushes required by that letter. Running this on the British National Corpus using the standard cell phone keyboard layout, we find that entering the entirety of the 100,106,029 word occurrences in the corpus would require 948,049,046 button presses.

If we examine a cell phone keyboard (Fig. 1) then we observe some terrible design choices. The frequently-occurring letters 'r' and 's' require more keystrokes than 'q'! Surely, a better designed keyboard can do better than this. If we just reversed the order of the letters on the 7 key, from 'pqrs' to 'srqp', the number of button presses required would drop to 865,118,331 — a savings of more than 8.7%. If we assume that every button press takes an equal amount of time, then this corresponds the users spending 8.7% less time entering their text messages. In an effort to discover how great these savings can be, we define the problem BasicCellPhoneTyping based on the template problem on page 2.

Problem 2 (BasicCellPhoneTyping).
Instance *A set of letters corresponding to an alphabet A ($|A| = n$), a number of keys k, and a set of tuples of words and frequencies W. The frequencies in W are integers, and the words are made up of solely of elements of A.*

Question *What is the best partition P of A into k sequences, such that the total number of button presses to type every word in W its associated frequency times is minimized? The number of button presses is equal to the order of the letter in the sequence assigned to a given key. For example, in the key $2 \rightarrow \{abc\}$, a requires one keystroke and c requires 3. Equivalently, if we denote the number of button presses required to type letter a with partition P as $\mathrm{IN}_P(a)$, then we want to find the P that minimizes*

$$\sum_{(w,f)\in W} \sum_{c\in w} \mathrm{IN}_P(c) * f$$

To solve BasicCellPhoneTyping we construct a provably optimal greedy algorithm which works in time $O(|W| + |A| \log |A|)$. Our algorithm is detailed in Fig. 2, and involves creating a histogram of the letters, and then assigning letters to keys round-robin style in the order from most-popular to least-popular. To prove optimality, we invoke an exchange argument.

Theorem 3. *The greedy algorithm finds an optimal solution to* BasicCellPhoneTyping.

Proof. We will assume, for simplicity of proof, that all letters occur a different number of times in the corpus. We begin by comparing two assignments of letters to keys by, for each assignment, constructing a sequence of sets, or spectrum, S. The set S_1 (layer 1) is the set of all letters which require a single button press (that is, they are the first letter in their sequence on their assigned key), S_2 (layer 2) is the set of all letters which require two button presses, and so on. If two

```
GreedyBasicCellPhoneTyping (A, k, W)
  // A is the set of letters
  // k is the number of keys
  // W is a set of pairs of words and their corresponding frequencies
  lettercount ← new_map()
  for c ∈ A
      lettercount[c] ← 0
  for (word, frequency) ∈ W
      for c ∈ word
          lettercount[c] ← lettercount[c]+frequency
  ordered ← [(lettercount[c], c) for c ∈ lettercount]
  sort(ordered)
  keys ← a array of length k where each element is an empty list
  for i ← 0 . . . length(ordered)
      (count, char) ← ordered[i]
      append(char, keys[i  mod k])
  return keys
```

Fig. 2. The greedy algorithm for Prob. 2.

assignments have equal spectra, then one assignment may be transformed into the other assignment simply by swapping letter between keys without increasing or decreasing the total number of button pushes required to enter the corpus of words. Two assignments are isomorphic iff their spectra are equal.

Now assume for the sake of contradiction that we have assignment D, with spectrum T, that is not isomorphic to the greedy solution G with spectrum S. In a spectrum resulting from the greedy algorithm, all letters at layer i occur strictly more frequently than the letters at layer j, $j > i$. Also, in S, for all layers i except for the last layer, $|S_i| = k$. Because T and S are not isomorphic, in T there must be at least one layer i such that $T_i \neq S_i$. Let T_i be the first layer for which that is true.

There are two cases for layer T_i. In the first case, $|T_i| < k$, and we may remove any element from a later layer place it in layer T_i and create a strictly better layout. In the second case, there is some letter a such that $a \in T_i$ and $a \notin S_i$. We choose the least-frequent such a. Because the greedy layout algorithm creates layers in order of frequency, we know that the frequency of a is less than that of some letter b in layer T_j, $j > i$. Therefore, by creating a new layout with a and b swapped between T_i and T_j, we have strictly decreased the number of button presses. Therefore, in both cases, T was not an optimal layout, and we have reached our contradiction. □

For the BNC, the optimal layout has the spectrum

$$[\{etaoinsr\}, \{hldcumfp\}, \{gwybvkxj\}, \{qz\}]$$

and any layout with that spectrum requires 638,276,114 button presses to entire the entire BNC instead of our original requirements of 948,049,046. This represents a savings of 32.67% over our original layout, but only for the users who type

using this most basic of input methods. Unfortunately, this is the group of users who are likely the least adaptable to change, as almost all "digital natives" use predictive methods to input text messages[1]. To place the least burden of "newness" on these users while still decreasing the number of button presses, we now consider schemes where the only alteration of the keyboard is the rearrangement of the sequence of letters for a given key.

By an argument symmetric to the proof of Theorem 3, we find that the optimal layout in this new scheme is to place the letters of a given key in sorted order, according to frequency. Thus, the keyboard layout changes to $2 \rightarrow [acb], 3 \rightarrow [edf], 4 \rightarrow [ihg], 5 \rightarrow [lkj], 6 \rightarrow [onm], 7 \rightarrow [srpq], 8 \rightarrow [tuv], 9 \rightarrow [wyxz]$ and requires 678,547,463 button presses to enter the whole corpus. This layout represents a 28.43% savings, and it has the added benefit that it does not change which key is mapped to which set of letters. This makes it **legacy preserving**, as creating a keyboard with this layout will not invalidate such advertising gems as 1-800-FLOWERS. Thus, we can speed up users by 28.43% (neglecting inter-letter pauses) and not undermine more than half a century of advertising. This layout represents perhaps the most plausible layout yet[2]. After setting up this baseline for the easy problem, we turn our attention to optimizing cellphone keyboards for predictive input methods.

2 The T9 Input Method

In an effort to minimize the pain of cellphone keyboard typing, cellular telephone manufacturers have created the T9 input method, which attempts to guess which letter (of the three or four possible) is intended when a user hits a single key. Enhancements of this input method also provide speculative lookahead to report, at any given time, what word it is that the user is most likely trying to enter and provide a completion. Unfortunately, because there are fewer keys on the cellphone keyboard than there are letters in the English alphabet, there are words which have different spellings but the same input sequence. As an example, in the traditional mapping of keys to characters both 'me' and 'of' have the input sequence '63'. Two words with the same input sequence force the user to press a third button to cycle through the possibilities in order from most likely ('of') to least likely ('me'). Even worse: 'home', 'good', 'gone', 'hood', 'hoof', 'hone', and 'goof' all have the input sequence '4663'. If the * key is used as the "next match" button, then the user will have to type in '4663******' to type in the word 'goof', for a total of 10 button presses — exactly the same number of button presses required using the default layout and the basic input method, and three

[1] I have no statistics on this except for an informal survey of one of my classes. In that class, every student who used SMS either had a cellphone with a 26+ key alphabetic keyboard or used a predictive method.

[2] The described layout is the only legacy preserving layout in this paper, and therefore should be considered the most practical suggestion. It also has the added benefit of not angering any of the organizations behind the numbers 1-800-FLOWERS, 1-800-THRIFTY, and 1-800-MARINES

more button presses than is required when using the basic input method with an optimized layout.

When two words have the same input sequence (neglecting the *'s at the end) then these words are **t9onyms**. When two or more words are t9onyms, then the less-popular words require extra key presses, raising the expected number of key presses to type in our corpus. To type a given word, one must press one button for every character in the word, followed by pressing the * key as many times as there are t9onyms which are more likely than the desired word. Because typing on a cellphone keyboard is already an unpleasant experience, we would like to minimize the expected number of keystrokes. The number of keystrokes a word requires is equal to the number of letters in the word, plus the number of t9onyms which are more frequent than the word. Formally, we extend Prob. 1 and define the MinimumT9Keystrokes problem as:

Problem 4 (MinimumT9Keystrokes).
Instance *An alphabet A, a set of words and their associated frequencies W, and a number of keys k ($|A| > k$).*

Question *What is the partition P of A into k sets which minimizes the total button presses required to enter the entire corpus using the T9 input method?*

and the corresponding decision problem is

Problem 5.
Instance *A set A, a set W of sequences of elements of A, a mapping f from sequences in W to the integers, and a number t.*

Question *Is there a partition P of A into k sets such that, if we denote $P(w)$ as the sequence of partitions $[p_i | w_i \in p_i, p_i \in P]$,*

$$\sum_{w \in W} (\text{len}(w) + \text{order}(w, P, W, f)) * f(w) \leq t$$

where len(w) is the length of the sequence w and order(w, P, W, f) is the size of $\{s \in W | P(s) = P(w) \wedge f(s) \leq f(w) \wedge s < w\}$, which is the set of sequences which map to the same sequence of partitions, but are not more popular, and are lexicographically smaller than w.

This problem is in $\mathcal{NP}$, as any assignment may be verified to require fewer than t button pushes in time proportional to the total length of all the words in W. To prove completeness, however, we first prove the $\mathcal{NP}$-completeness of an intermediate problem: UniquePathColoring.

Problem 6 (UniquePathColoring).
Instance *A graph $G = (V, E)$, a set of paths P, and a parameter k. A path p is a sequence of vertices in which adjacent vertices in p are also adjacent in the edge set E.*

QUESTION *Is there a k-coloring of G such that every path $p \in P$ has a unique coloring? If we consider the coloring function $\chi(v)$ to map vertices to colors, then we can extend this notation by having $\chi(p)$ map a path $p = [v_1, v_2, \ldots]$ to the sequence $\chi(p) = [\chi(v_1), \chi(v_2), \ldots]$. Is there a coloring χ of V such that*

$$|\{\chi(p)\ \forall p \in P\}| = |P|\ ?$$

Theorem 7. UNIQUEPATHCOLORING *is $\mathcal{NP}$-complete.*

Proof. To prove that UNIQUEPATHCOLORING is $\mathcal{NP}$-complete, we begin by noting that any coloring of V may be verified, in polynomial time, to map each path to a unique sequence of colors. Therefore, the problem is in $\mathcal{NP}$. To prove completeness, we reduce from GRAPHCOLORING[3].

An instance of GRAPHCOLORING consists of a graph $G = (V, E)$ and a parameter k. We then ask the question of whether there is a k-coloring χ of the vertices of the graph such that $\forall (u, v) \in E$, $\chi(u) \neq \chi(v)$. We transform an instance of GRAPHCOLORING into an instance of UNIQUEPATHCOLORING in the following manner:

Given $G = (V, E)$ and k from GRAPHCOLORING, we create

$$G' = (V \cup \{0, 1\}, E \cup \{(0, 1), (1, 0), (0, 0), (1, 1)\} \cup \{(v, 0)\ \forall v \in V\})$$

We then uniquely number each edge in E with the numbers $1 \ldots |E|$. For each edge $e = (u, v)$ numbered i, we create the path set

$$\begin{aligned} p_i = \{&[v, 0, b_1(i), b_2(i), b_3(i), \ldots b_{\lceil \log_2 i \rceil}], \\ &[u, 0, b_1(i), b_2(i), b_3(i), \ldots b_{\lceil \log_2 i \rceil}]\} \end{aligned}$$

where $b_1(i)$ is the first digit of i in binary, $b_2(i)$ is the second digit of i in binary, and so on. Now we create our path set

$$P = \{[0, 1], [1, 0]\} \cup \bigcup_{i=1}^{|E|} p_i \ .$$

We then ask the question of whether G' and P can be unique-path colored using only k colors.

If there is a k-coloring χ of G, then we use that coloring to generate a k-unique-path coloring of G' and P in the following manner: First, assign vertices 0 and 1 different colors from the range of χ. Next, assign each vertex to the color it received in the k-coloring of G'. Now, every element of our path set corresponds to a unique color sequence. $[0, 1]$ and $[1, 0]$ are the only sequences of length two, and because we assigned these two vertices different colors, $\chi([0, 1]) \neq \chi([1, 0])$. If path x and path y are not from the same p_i, then they have a different sequence of zeroes and ones. Therefore, because 0 and 1 are assigned different colors, the only possible path that a given path in some p_i might be confused with is the

other path in that p_i. But each p_i corresponds to an edge in E, and we know for all edges in $(u, v) \in E$ that $\chi(u) \neq \chi(v)$. Therefore, both paths within a given p_i also have a distinct color sequence. Therefore, given a k-coloring of G, we can generate a k-unique-path coloring of G' and P.

Now we prove that given a k-unique-path coloring of G' and P we can create a k-coloring of G. Given a k-unique path coloring, we are guaranteed that no path in P has the exact same coloring as any other path in P. This means that, for all i, the two paths in p_i are distinct. The only difference between the two paths in p_i is in the first vertex of the path. Therefore, for every p_i, the vertices of corresponding edge in must be given distinct colors, and this is true for all i and $e \in E$. Therefore, for all $(u, v) \in E$, the color assigned to u is distinct from the color assigned to v. Therefore, from a k-unique-path coloring of G' and P, we have a k-coloring of $G = (V, E)$.

□

This proof implies not just that UniquePathColoring is $\mathcal{NP}$-complete, but that it is $\mathcal{NP}$-complete even when we restrict ourself to just 3 colors, because the number of colors in the UniquePathColoring is exactly equal to the number of colors in the instance of GraphColoring, and 3-coloring a graph is an $\mathcal{NP}$-complete problem[4].

Now that we have proven UniquePathColoring to be $\mathcal{NP}$-complete, we immediately use it in a reduction.

Theorem 8. MinimumT9Keystrokes *(as specified in Problem 5) is $\mathcal{NP}$-complete.*

Proof. As we noted on page 6, MinimumT9Keystrokes is in $\mathcal{NP}$. To prove completeness, we reduce from UniquePathColoring. An instance of UniquePathColoring consists of a graph $G = (V, E)$, a set of paths P, and a parameter k. We then ask the question of whether there is a vertex coloring of G using k colors where each path in P ends up with a unique sequence of colors.

We transform an instance of UniquePathColoring into an instance of MinimumT9Keystrokes in the following way: We make A be the set of vertices V, and we make W be the set of paths P, and we assign each of these new 'words' a frequency of 1. We then ask the question of whether there exists a partition of A into k sets such that the total number of button presses required is no greater than $t = \sum_{w \in W} \text{len}(w)$.

Now we prove that if there is a UniquePathColoring using k colors, then there exists a solution to the corresponding instance of MinimumT9Keystrokes. In particular, a k-unique-path coloring if G give us a partition of the vertices V into k sets. We map each set to a single key in our solution to MinimumT9Keystrokes. Because each path has a unique coloring, each word in the corresponding MinimumT9Keystrokes has a unique pattern of button presses. Therefore, the number of more popular words with the same keystrokes as a given word is exactly zero. Thus, because each word in the corpus has a frequency of one, to type in the entire corpus only requires exactly as many keystrokes as there are words in the corpus, and therefore, the k unique path

coloring of V corresponds exactly to an assignment of letters to keys such that the total number of keystrokes is exactly $\sum_{w \in W} \text{len}(w)$, which is exactly t.

Next, we prove that a solution to the MinimumT9Keystrokes problem instance implies a solution to the corresponding UniquePathColoring problem. If there exists a partition of A into k partitions such that

$$\sum_{w \in W} (\text{len}(w) + \text{order}(w, P, W, f)) * f(w) \leq t = \sum_{w \in W} \text{len}(w)$$

then, because $f(w) = 1$, $\forall w \in W$, it must be true that $\text{order}(w, P, W, f) = 0$, $\forall w \in W$. Therefore, every word $w \in W$ has a unique sequence of button presses if we partition A into k partitions according to P. To generate our k-unique-path coloring of G, we color each set in P a single unique color. Because each word in w has a unique sequence of partitions, each path has a unique coloring, and therefore we can color the corresponding instance of G using k colors. Thus, a partition of V into k partitions which allows the corpus to be entered in less than t keystrokes implies a k-unique path coloring of G. Therefore, because UniquePathColoring can be reduced to MinimumT9Keystrokes and is in $\mathcal{NP}$, MinimumT9Keystrokes is $\mathcal{NP}$-complete. □

This proof of $\mathcal{NP}$-completeness is of a relatively strong form: just like UniquePathColoring, our problem remains $\mathcal{NP}$-complete even if we restrict ourselves to $k = 3$ (only 3 buttons on the cell phone keyboard).

Because our problem is $\mathcal{NP}$-complete, we will not try to find a general solution. In our specific instance, with 26 letters and an eight key keyboard (1 and 0 are reserved for other uses), we must choose the best layout from among

$$\binom{26}{8}\binom{18}{8}\binom{10}{8} = 3,076,291,325,250$$

different possibilities. Given that, currently, it takes around a tenth of a second to count the button presses required to enter the BNC, an exhaustive search will take 307,629,132,525 seconds, or 9.75 millennia. Therefore, we have two remaining avenues of attack: find the best answer possible using stochastic methods, or solving a simpler problem. Using stochastic methods (a simple genetic algorithm), the best layout we found was

$$\{\{eb\}, \{lcd\}, \{sh\}, \{zmx\}, \{fayg\}, \{toj\}, \{unpv\}, \{irkwq\}\}$$

which required 441,612,049 button presses to enter the entire corpus. Our genetic algorithm was quite simple: From an initial gene pool of random layouts, we discarded all but the best layouts (the survivors). Then, we added to the gene pool random mutations of the survivors to fill out the gene pool back to its original size, and repeated the process of selection and mutation for some time. As compared to the requirements of the default layout (443,374,079), this represents a savings of less than 1%, and is thus not a very attractive alternative to existing layouts. This low amount of improvement, while not good news for our proposed reordering, is excellent news for all current cell phone users: the default cell phone keyboard layout seems relatively efficient for T9 text entry in English!

2.1 Alphabetic Order

Our arbitrary remappings of the keyboard may be quickly deemed unusable, simply because it is almost impossible to tell where a given key is located without memorizing the entire new layout. Therefore, we define a refinement of the keyboard layout problem, in which the characters are required to be kept in alphabetical order as a sequence, and our only choices are about how to divide the subsequence into key assignments. The default layout can be described as the partition $abc|def|ghi|jkl|mno|pqrs|tuv|wxyz$. Is this partition optimal? To decide this, we define the problem MinimumKeystrokePartition as:

Problem 9 (MinimumKeystrokePartition).
Instance *A sequence of letters $A = \{a_1, a_2, \ldots\}$, a set of words and their associated probabilities W, and a set of keys $K = \{2, 3, 4, \ldots\}$ ($|A| > |K|$).*

Question *What is the mapping f of $A \to K$ which minimizes the expected number of characters to type a word from W, and where $f(a_1) = 2$ and if $f(a_i) = k$, then either $f(a_{i+1}) = k$ or $f(a_{i+1}) = k + 1$?*

The number of partitions of a sequence of size n into k subsequences is equal to $\binom{n-1}{k-1}$, and in the particular case of the 26 letter alphabet and the eight keys available on a cell phone keyboard, we find that there are $\binom{26-1}{8-1} = 480,700$ possible sequences. This means that, in the case of our corpus which requires tenths of a second to test against a proposed solution, we can test every possible sequence in just 480,700 seconds. Therefore, our final result comes from simply generating and testing every single partition of the alphabet into 8 sets. We found that the best partition was

$$\{\{ab\}, \{cde\}, \{fghij\}, \{klm\}, \{nop\}, \{qrs\}, \{t\}, \{uvwxyz\}\}$$

which required 442,717,436 button presses, yet again for a savings of less than 1% over the default layout.

3 Conclusion

We have developed three problems based on the idea of making cell phone keyboard more efficient for typing text messages. Different text entry methods yield problems with very different levels of solvability. If we restrict ourselves to the basic text entry method, then a greedy algorithm will work for finding the optimal layout. Even better, the greedy algorithm works whether we want to restrict ourselves to rearranging the order of letters on a single key, or to rearranging all letters on all keys. On the other hand, if we try to optimize against the more complex T9 input method, then we find that the problem is $\mathcal{NP}$-complete. After proving completeness, we gave numerical results which indicate that it will be difficult (or perhaps impossible) to significantly improve on the default layout of letters. This stands in sharp contrast to the simpler input method, where we

were able to improve by more than 27% just by reordering the letters on the keys, but never moving a letter from one key to another.

We have also discovered a new $\mathcal{NP}$-complete problem UniquePathColoring, which may be of use in other contexts. In particular, it is rare in the literature to find an $\mathcal{NP}$-complete problem that contains both partitions and sequences, which is why we had to invent a new one here. The hope, as always, is that this problem will prove useful to others.

We left as future work any exploration of the issue of needing to pause between letters in the basic typing method, and, while there do exist even more sophisticated input methods[6], we have left them unmentioned and unexamined. Optimizing a keyboard for the basic text input method was relatively straightforward, but, for the T9 input method, optimally using a cell phone keyboard is $\mathcal{NP}$ hard.

References

1. British National Corpus Consortium. British National Corpus, 2000.
2. F. Evans, S. Skiena, and A. Varshney. Resolving ambiguity in overloaded keyboards, 2004. `http://cs.smith.edu/~orourke/GodFest/`.
3. M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman, 1979.
4. M. R. Garey, D. S. Johnson, and L. Stockmeyer. Some simplified NP-complete problems. In *STOC '74: Proceedings of the sixth annual ACM symposium on theory of computing*, pages 47–63, New York, NY, USA, 1974. ACM.
5. J. Gong, P. Tarasewich, and I. S. MacKenzie. Improved word list ordering for text entry on ambiguous keypads. In *NordiCHI '08: Proceedings of the 5th Nordic conference on human-computer interaction*, pages 152–161, New York, NY, USA, 2008. ACM.
6. S. B. Nesbat. A system for fast, full-text entry for small electronic devices. In *ICMI '03: Proceedings of the 5th international conference on multimodal interfaces*, pages 4–11, New York, NY, USA, 2003. ACM.

A A Genetic Algorithm for Discovering Efficient T9 Layouts

```
#include <algorithm>
#include <iostream>
#include <fstream>
#include <map>
#include <string>
#include <vector>
#include <sys/time.h>

#define POPULATION 32
#define GENERATIONS 20

using namespace std;

typedef pair<int, string> pis;
vector<pis> words;
long long total;

bool piscompare(pis a, pis b) { return a.first > b.first; }

long long quality(vector<string> &partition)
{
    map<char,string> rmap;
    int i = 0;
    string buttons = "23456789";
    for (vector<string>::iterator s = partition.begin();
            s != partition.end();
            ++s, ++i) {
        for (string::iterator c = s->begin(); c != s->end(); ++c) {
            rmap[*c] = buttons[i];
        }
    }

    map<string, vector<pis> > typing;
    for (vector<pis>::iterator fw = words.begin();
            fw != words.end();
            ++fw) {
        int freq = fw->first;
        string word = fw->second;
        string presses = "";
        for (string::iterator c = word.begin(); c != word.end(); ++c) {
            presses += rmap[*c];
        }

        if (typing.count(presses) == 0)
            typing[presses] = vector<pis>();
        typing[presses].push_back(*fw);
    }

    for (map<string, vector<pis> >::iterator w = typing.begin();
            w != typing.end();
            ++w) {
        sort(w->second.begin(), w->second.end(), piscompare);
    }

    long long totalpresses = 0;
    for (map<string, vector<pis> >::iterator w = typing.begin();
            w != typing.end();
            ++w) {
        int wordsize = w->first.size();
        for (int i = 0; i < w->second.size(); i++) {
            totalpresses += (wordsize + i) * w->second[i].first;
        }
    }
    return totalpresses;
}

void printpart(vector<string> &p)
{
    for (vector<string>::iterator i = p.begin();
            i != p.end();
            ++i)
        cout << *i << " ";
    cout << endl;
}

vector<string> randompartition(string in)
{
    vector<string> p;
    for (int i = 0; i < 8; i++)
        p.push_back("");

    while (in.size() > 0) {
        int pos = random() % in.size();
        int key = random() % 8;
        p[key].append(1, in[pos]);
```

```
        in.erase(pos, 1);
    }

    return p;
}

typedef pair<long long, vector<string> > pllvs;
bool lvscompare(pllvs a, pllvs b) { return a.first < b.first; }

vector<vector<string> > cull(vector<vector<string> > &pop)
{
    vector<pllvs> fitness;
    for (vector<vector<string> >::iterator i = pop.begin();
            i != pop.end();
            ++i) {
        fitness.push_back(pllvs(quality(*i), *i));
    }

    sort(fitness.begin(), fitness.end(), lvscompare);

    vector<vector<string> > newpop;
    for (int i = 0;
            i < fitness.size() / 4;
            ++i) {
        cout << fitness[i].first << " ";
        printpart(fitness[i].second);
        newpop.push_back(fitness[i].second);
    }

    return newpop;
}

vector<string> mutate(vector<string> in)
{
    for (int count = 0; count < 2; count++) {
        int i;
        do { i = random() % in.size(); } while (in[i].size() == 0);

        int c = random() % in[i].size();
        in[random() % in.size()].append(1, in[i][c]);
        in[i].erase(c, 1);
    }

    printpart(in);
    return in;
}

int main()
{
    ifstream win("words/wordlist");
    win >> total;

    string word;
    win >> word;

    int freq;
    while (win >> freq) {
        win >> word;
        words.push_back(pis(freq,word));
    }

    // Seed it
    timeval t1;
    gettimeofday(&t1, NULL);
    srandom(t1.tv_usec * t1.tv_sec);

    string alphabet = "abcdefghijklmnopqrstuvwxyz";
    vector<vector<string> > population;
    for (int i=0; i < POPULATION; i++)
        population.push_back(randompartition(alphabet));

    for (int i=0; i < GENERATIONS; i++) {
        population = cull(population);
        while (population.size() < POPULATION) {
            population.push_back(mutate(population[random() % population.size()]));
        }
    }

    return 0;
}
```

B Code for Analyzing All Alphabetic-Order T9 Layouts

```
#include <algorithm>
#include <iostream>
#include <fstream>
```

```
#include <map>
#include <string>
#include <vector>

using namespace std;

typedef pair<int, string> pis;
vector<pis> words;
long long total;

bool piscompare(pis a, pis b) { return a.first > b.first; }

long long quality(vector<string> &partition)
{
    map<char,string> rmap;
    int i = 0;
    string buttons = "23456789";
    for (vector<string>::iterator s = partition.begin();
            s != partition.end();
            ++s, ++i) {
        for (string::iterator c = s->begin(); c != s->end(); ++c) {
            rmap[*c] = buttons[i];
        }
    }

    map<string, vector<pis> > typing;
    for (vector<pis>::iterator fw = words.begin();
            fw != words.end();
            ++fw) {
        int freq = fw->first;
        string word = fw->second;
        string presses = "";
        for (string::iterator c = word.begin(); c != word.end(); ++c) {
            presses += rmap[*c];
        }

        if (typing.count(presses) == 0)
            typing[presses] = vector<pis>();
        typing[presses].push_back(*fw);
    }

    for (map<string, vector<pis> >::iterator w = typing.begin();
            w != typing.end();
            ++w) {
        sort(w->second.begin(), w->second.end(), piscompare);
    }

    long long totalpresses = 0;
    for (map<string, vector<pis> >::iterator w = typing.begin();
            w != typing.end();
            ++w) {
        int wordsize = w->first.size();
        for (int i = 0; i < w->second.size(); i++) {
            totalpresses += (wordsize + i) * w->second[i].first;
        }
    }
    return totalpresses;
}

int start = 0;
int end = -1;
void genallpartitions(string &s, int count, int index,
        vector<string> &part)
{
    if (count == 0) {
        if (--start > 0) {
            --end;
        } else {
            // Analyze it
            cout << quality(part) << " ";
            for (vector<string>::iterator i = part.begin();
                    i != part.end();
                    ++i)
                cout << *i << " ";
            cout << endl;
            if (--end == 0) exit(0);
        }
    } else if (count == 1) {
        part.push_back(s.substr(index, s.size()-index));
        genallpartitions(s, count-1, s.size(), part);
        part.pop_back();
    } else {
        for (int size = 1; size < s.size() - index - count + 2; size++) {
            part.push_back(s.substr(index, size));
            genallpartitions(s, count-1, index+size, part);
            part.pop_back();
        }
    }
}
```

```
int main(int argc, char** argv)
{
    if (argc == 3) {
        start = atoi(argv[1]);
        end = atoi(argv[2]);
    }

    ifstream win("words/wordlist");
    win >> total;

    string word;
    win >> word;

    int freq;
    while (win >> freq) {
        win >> word;
        words.push_back(pis(freq,word));
    }

    string alphabet = "abcdefghijklmnopqrstuvwxyz";
    vector<string> partition;
    genallpartitions(alphabet, 8, 0, partition);
    return 0;
}
```