CSCI E-124-Spring, 2004 Homework 3

Russell Lowke, March 26th 2004

1. Prove that there is a unique minimum spanning tree on a connected undirected graph when the edge weights are unique.

Proof by contradiction using the exchange property of minimum spanning trees.

Let's say there are two minimum spanning trees for a given graph, trees T and T'. Then, according to the exchange property, there is some lightest edge e' that appears in one but not the other, let's say it's in T' but the choice is arbitrary. This creates a cycle when we add it to T. Since we already know that T is a MST, this edge must be the heaviest one in the cycle (else it would've been in there before, and it can't be the same as an existing edge because **all edge weights are unique**). When we remove the lightest edge in the cycle, e, we know that it cannot be in T' (T' has no cycles). We assumed that e' was the lightest edge appearing in one tree and not the other, but e MUST be because it's lighter than e' and must not be in T else T would have a cycle. Thus we have a contradiction, and there must not exist two such trees T and T'.

2. Set Cover. Give a family of set cover problems where the set to be covered has n elements, the minimum set cover is size $k = 3$, and the greedy algorithm returns a cover of size $\Omega(\log n)$.

To satisfy the requirement that there always exists a set cover of size 3, first divide the set into three equal pieces (or as close as possible). To make the greedy algorithm return a set cover on the order of log n, we need to trick the algorithm into NOT taking any of these three "big" sets for as long as possible. In order to do so, we construct new sets following this procedure:

First, we look at the max. size that any one of these big sets can be, which starts off as ceiling (n / 3) <note: I mean actual division were we do count remainders, not integer division>. Then we add 1 to this value (to make sure it's given priority) and create a new set of that size. Then, we distribute this new set as evenly as possible between the three big ones. This results in a new maximum size (of the 3 big sets) which is:

new max. size := old max. size - floor((old max. size) / 3)

Call this function again with the new max. size, and repeat until you have filled one of the sets up completely. Then, stop. Since the max. size of the set is O(n) (approx. 1/3 n) and this number is reduced to 2/3 it's size every round until it's full, we can guarantee that we produce roughly log (1/3 n) (that's to the base 3/2, I believe... but I may be mistaken) and since we've guaranteed (by their construction) that the greedy algorithm will always take them in the order we create them, we can also state that the algorithm runs in O(log n) time.

3. We have two machines, and a set of jobs j_1, j_2, j_3, ..., j_n that we have to process one at a time... Each job j_i has an associated running time r_i. The load on the machine is the sum of the running times of the jobs placed on it. The goal is to minimize the completion time, which is the maximum load over all machines.

We have two machines, and a set of jobs j1, j2, j3, ..., jn that we have to process one at a time... Each job ji has an associated running time r i . The load on the machine is the sum of the running times of the jobs placed on it. The goal is to minimize the completion time, which is the maximum load over all machines.

• Suppose we adopt a greedy algorithm... Give an example where a factor of 3/2 is achieved... Suppose now instead of 2 machines we have *m* machines.

Worst case scenario: items are evenly distributed BEFORE we add the largest item on.

Assuming a perfect, even distribution, the largest running time of the largest job is at most r, where r is the maximum running time (if not we couldn't have divided it up into a perfect, even distribution). We remove this item, and divide up the rest of the jobs perfectly. If we can do so, then the running time on each machine should be 1/2 r (again, we assume a perfect distribution). When we add back in our largest item, of length r, we how have a total running time of 1/2 r (redistributed jobs) + r (the largest item we just added back in) = 3/2r (our upper limit).

The assumption of an even distribution was valid because to get the maximum running time we need to add that max. value onto the largest possible load... but it automatically goes to the one with the MINIMUM load. Therefore, to get the largest running time, we have to maximize the value of the minimum load. This happens when they're all even. If they're not, again, there exists some lower load value on a machine that the algorithm would immediately try to add the largest job to and our answer would ultimately be less.

General solution:

We can expand this idea out further, relying on the fact that the worst case scenario is always when we divide up the times perfectly evenly into stacks of size r (with the largest value in), remove this value, divide things back up evenly, and add back in the largest value (or size r). This consistently yields a max. value of 1/r + r/m, where r itself is the sum of all running times divided by the number of machines. So, to generalize further, we have a max. bound of

(sum of all running times) / m + (sum of all running times)/ m^2

working from this, to construct an input that produces this result, we need to generate a set where all of the value can be perfectly divided among the n machines WITH AND WITHOUT the largest item among them. Also, the largest item is of size (sum of all running times) / m. We feed in all values starting with the second largest job and working down toward the smallest then feeding in the largest job last and we can be guaranteed the maximum completion time.

4. Consider an algorithm for integer multiplication of two *n*-digit numbers where each number is split into three parts, each with *n*/3 digits.

a) Design and explain such an algorithm, similar to the integer multiplication algorithm presented in class. Your algorithm should describe how to multiply the two integers using only six multiplication on the smaller parts (instead of the straightforward nine).

by dividing the numbers up into 3 parts, so

$x = (a*10^{2/3} + b*10^{1/3} + c)$, and

$y = (d*10^{2/3} + e*10^{1/3} + f)$, we get a multiplication of

$xy = ad*10^{4/9} + (ae + bd)*10^{2/9} + (af + cd)*10^{2/3} + (bf + ce)*10^{1/3} + be*10^{1/9} + cf$

which is 9 multiplications. However, we can cut a few out by noticing that

$(a + b)(e + d) = (ae + bd) + ad + bd$

so we compute the 3 things besides (ae + bd), subtract, and we have saved a step. Also,

$(a + c)(f + d) = (af + cd) + ad + cf$

so we compute the 2 other values we don't know (we know ad) and subtract to find (af + cd). Also

$(b + c)(f + e) = (bf + ce) + be + cf$

and we only do 1 more multiplication here to get (b+c)(f +e) and we have found every term using only 6 multiplications.

b) Determine asymptotic running time... split it into two parts or three?

Recurrence of $6T(n/3) + O(n)$. The running time is $n^{\log 6}$ where the log is BASE 3. or, as a decimal, $n^{1.631}$ which is WORSE than the other answer... so you'd rather divide it into 2 parts.

c) ... use only five multiplications instead of six...

If you could only use 5 multiplications you'd get a value of $n^{1.465}$ which is better than 2 parts, and thus you'd be better off splitting into 3, because 2 is $n^{1.59}$

5. Suppose we have an array *A* containing *n* numbers, some of which may be negative. We wish to find indices *i* and *j* ...

a) Give a trivial $O(n^2)$ algorithm.

The pseudocode is:

```
Procedure slowPoke (n)
        maxValue = 0;
        maxI, maxJ;
        for all i from 1 to n - 1
                for all j from i to n
                        add up the values between A(i) and A(j)
                        if current sum is greater than previous maxValue
                                replace old value, store indices in maxI and maxJ
                        fi
                rof
        rof
end slowPoke
```

b) Show how using divide and conquer, we can obtain an $O(n \log n)$ algorithm.

First, go through (in a linear pass) and group together your positive values and your negative values summing them up. This reduces the problem down to being a smaller array with only one positive or one negative value beside one another. Now start at the largest positive value and work out to the left and right following this rule:

If the negative value beside where you are is less than the positive value on the other side of it, take them both. If not (or if we run out of space) go in the other direction and do the same again.

c) Now try to come up with an $O(n)$ algorithm.

Start at the leftmost end of the array, and move to the right summing all values (and storing this sum for every index that we are over). When this sum drops below zero, we reset it to zero before adding on the next item. When this is done, the largest value will be the rightmost bound of your final partition (or *bounds*... if this yields more than one largest number you have more than one possible answer.

Do the exact same thing but starting from the right and working toward the left and the maximum value(s) will be the leftmost bound(s) of the partition.

6. A challenge that arises in databases is how to summarize data in easy-to-display formats. A problem in this context is the minimal imbalance problem...

• Give an algorithm for determining the partition with the minimal imbalance.

This can be solved in polynomial time using Dynamic Programming.

A recursive, divide and conquer approach would involve placing 1 index, calculating the imbalance for the partition it creates, and then recursively placing the next k - 1 indices to divide up the rest of the array optimally (assuming that the first partition was placed optimally). Of course, we need to iterate over all possible placements of a given index and compare the results of each one to find the placement that results in the minimum imbalance to get a correct answer and this is costly and involves a lot of recomputation. Instead, we store a 2 dimensional array A(i, j)that stores the imbalance of the partition whose lower and upper bounds are i and j respectively. We are guaranteed to get the right answer (after all, we're still iterating over *every possible* solution) but it will run much faster.

• Explain how your algorithm would change if the imbalance was redefined...

The basic structure of the algorithm would be the same, and we would be storing the same values. However, instead of checking (for each new placement of some index $i < k$) for the answer that would yield the minimum partition among the others, we just sum up the results over all partitions and choose the placement that results in the minimum sum.

7. Suppose we want to print a paragraph... Find a dynamic programming algorithm to determine the neatest way to print a paragraph. Of course you should provide a recursive definition of the value of the optimal solution that motivates your algorithm.

An easy definition of the solution is that one picks the optimal line break for a given line in such a way that the line breaks for the rest of the lines in the document are optimal. A simple recursive algorithm would choose a value for the line break (between 1 word on the line and the max. value that can fit) and then call the same algorithm on the rest of the remaining document, making sure to iterate through all possible values for the line break and comparing the imbalances to make sure you choose the minimal ones. To make this a dynamic programming algorithm, store for each word in the document the best way to divide the *rest* of the document beyond it that to produce the smallest imbalance. So, as you're shifting around the values for the line breaks in your first few function calls and asking the rest of the document to divide itself you can take advantage of the fact that it might *already* know how to optimally divide itself beyond that point.

7 [continued]. For this problem, besides explaining/proving your algorithms as for other problems on the set, you should also code up your algorithm to print an optimal division of words to lines. The output should be the text split into lines appropriately, and the numerical value of the penalty.

[CODE]

```
//
//
//  dynamic.cpp
//
//       AUTHOR: Russell Lowke
//
//           Date: 3/26/2004
//


#include <iostream>
#include <cstring>

#include <string>
using std::string;

#include "Document.h"

int main(int argc, char* argv[]) {

    int m;
    string filename;
    if (argc == 3) {
        m        = atoi(argv[1]);
        filename = argv[2];
    } else {
        cerr << "No arguments: using some defaults" << endl;
        m = 72;
        filename = "Buffy.txt";
    }
    Document d(filename, m);
    d.flowDocument();
    d.printDocument();

    cout << endl;
}

//
//
//  Document.h
//
//       AUTHOR: Russell Lowke
//
//           Date: 3/26/2004
//

#ifndef DOCUMENT_H
```

```
#define DOCUMENT_H

#include <iostream>
#include <vector>
#include <string>
#include <fstream>
using std::ifstream;

using namespace std;

typedef vector<int>       IntList;
typedef vector<string>    StringList;


class Document {

    public:

        void printDocument();
        void flowDocument();

        // the key recursive function
        int calculatePenalty(int startIndex);

         Document (string filename, int lineLength);

        // gotta clean up after myself
        ~Document() {
            if (_knownMinPenalty)
                delete[] _knownMinPenalty;
            if (_breaks)
                delete[] _breaks;
        }

    private:

        int           _documentPenalty;
        int           _lineLength;
        IntList       _goodBreaks;                // error flag exists to give up
                                                  // the capability to bail out if
                                                  // things mess up during initialization

        bool          _error;
        string        _filename;                  // to store the indices that we're using
                                                  // for breaks;
        IntList*      _breaks;                    // text of the file, as a vector of strings
        StringList    _text;                      // will store the minimum penalty we know
                                                  // to exist if we flow the document from
                                                  // that point on

        int*          _knownMinPenalty;
        const static int UNKNOWN = -1;

};

#endif /* DOCUMENT_H */

//
//
```

```
//   Document.cpp
//
//       AUTHOR: Russell Lowke
//
//                Date: 3/26/2004
//

#include "Document.h"
#include <cmath>
using namespace std;

void Document::printDocument() {

    cout << "Optimal division for " << _filename << endl << endl;
    for (int i = 0; i < _text.size() ; ++i) {
        if (_goodBreaks.front() == i) {
             _goodBreaks.erase(_goodBreaks.begin());
            cout<<endl;
        }
         cout<< _text[i]<<" ";
    }
}


Document::Document (string filename, int lineLength) {

    cout << "Creating document from " + filename << endl;
    _filename   = filename;
    _lineLength = lineLength;
    _error      = false;

     fstream inputFile(filename.c_str(), ios::in);

    if (!inputFile) {
         cerr << "ERR: File not found" << endl;
         _error = true;
    } else {
         string word;
        while(inputFile>>word) {
         _text.push_back(word);
    }
         _knownMinPenalty = new int[_text.size()];
         _breaks = new IntList[_text.size()];

        for (int i = 0; i < _text.size(); ++i) {
            // array holds the minimum penalty when the document is flowed from that
            // point on a value of -1 means that the value is currently unknown
             _knownMinPenalty[i]= UNKNOWN;
        }
    }
}


//
//    method takes an int: the starting word, and returns an int, the total penalty
//    when everything's been optimally broken up beyond that point.
//
```

```
int Document::calculatePenalty(int startIndex) {

    // if we already know how to divide from this point on, return that knowledge
    if (_knownMinPenalty[startIndex] != UNKNOWN)
        return _knownMinPenalty[startIndex];

    // first calculate the maximum number of words to have on the line: we work back
    //   from there
    int numWords = 0;
    int currentLength = 0;

    for (int i = startIndex; i < _text.size() ; ++i) {

        // currentlength + " " + word > max. allowed length means we must put
        // this word on the next line.  if not, just add it and move on.
        if (!(( currentLength + 1 + _text[i].length() ) > _lineLength)) {

            // on first word, don't add a space
            if (currentLength != 0)
                currentLength += 1;

             currentLength += _text[i].length();
            ++numWords;

        } else
            break;
    }

    // check if we're on the last line.  If so, the penalty's ZERO
    if (startIndex + numWords == _text.size()) {
        for (int i = startIndex; i < _text.size(); ++i)
            _knownMinPenalty[i] = 0;

        return 0;

    } else {

        int penalty    = UNKNOWN;
        int newPenalty = UNKNOWN;
        int goodIndex  = _text.size() - 1;

        int trailingSpaces = _lineLength - currentLength;



        //
        // Now the clincher: total penalty = penalty of this line +
        //                                   penalty of rest of document
        // Note:  the clause (trailingSpaces + _text[startIndex + (i - 1)].length()) + 1)
        // exists so that the value for the trailingSpaces updates as we successively
        // remove words from the end of the line and calculate the penalty for the rest.
        // _text[startIndex + (i - 1).length is the length of the word on the end of the
        // line, and the +1 is, of course, for the space that was between it and the
        // previous one
        //
        for (int i = numWords; i >= 1; --i) {
            if (i != numWords)
```

```
                trailingSpaces += 1 + _text[startIndex + (i)].length();

            newPenalty = (int) pow((double) (trailingSpaces), 3) +
                 calculatePenalty(startIndex+i);

            if ((newPenalty < penalty) || (penalty == UNKNOWN)) {
                 penalty = newPenalty;
                 goodIndex = startIndex + i;
            }
        }

        // we're done, so store the minimum penalty and the knowledge of how to break
        // up the rest of the document
         _knownMinPenalty[startIndex] = penalty;
         _breaks[startIndex].push_back(goodIndex);
        for (IntList::iterator i = _breaks[goodIndex].begin();
                               i != _breaks[goodIndex].end(); ++i)
             _breaks[startIndex].push_back(*i);

        return penalty;
    }
}

void Document::flowDocument() {

    cout << "About to calculate penalty with " << _text.size() <<
            " words and a line length of " << _lineLength << endl;
    _documentPenalty = calculatePenalty(0);
    cout << "Penalty of document is: " << _documentPenalty << endl;
    cout << "breaks before words numbers: ";
    for (IntList::iterator i = _breaks[0].begin(); i != _breaks[0].end(); ++i)
        cout << " " << *i;

    _goodBreaks = _breaks[0];
    cout << endl << endl;
}
```

[OUTPUT, M = 72]

```
Creating document from Buffy.txt
About to calculate penalty with 176 words and a line length of 72
Penalty of document is: 2104
breaks before words numbers:  14 24 37 48 61 73 84 96 106 117 127 139 151 162 172

Optimal division for Buffy.txt

Buffy the Vampire Slayer fans are sure to get their fix with the DVD
release of the show's first season. The three-disc collection includes
all 12 episodes as well as many extras. There is a collection of
interviews by the show's creator Joss Whedon in which he explains
```

his inspiration for the show as well as comments on the various cast members. Much of the same material is covered in more depth with Whedon's commentary track for the show's first two episodes that make up the Buffy the Vampire Slayer pilot. The most interesting points of Whedon's commentary come from his explanation of the learning curve he encountered shifting from blockbuster films like Toy Story to a much lower-budget television series. The first disc also includes a short interview with David Boreanaz who plays the role of Angel. Other features include the script for the pilot episodes, a trailer, a large photo gallery of publicity shots and in-depth biographies of Whedon and several of the show's stars, including Sarah Michelle Gellar, Alyson Hannigan and Nicholas Brendon.

dynamic has exited with status 0.

[OUTPUT, M = 40]

```
Creating document from Buffy.txt
About to calculate penalty with 176 words and a line length of 40
Penalty of document is: 2183
breaks before words numbers:  6 14 20 25 32 38 45 51 59 66 73 79 86 92 97 102 108 114 120 126
132 139 145 152 158 165 170 175

Optimal division for Buffy.txt

Buffy the Vampire Slayer fans are
sure to get their fix with the DVD
release of the show's first season.
The three-disc collection includes all
12 episodes as well as many extras.
There is a collection of interviews
by the show's creator Joss Whedon in
which he explains his inspiration for
the show as well as comments on the
various cast members. Much of the same
material is covered in more depth with
Whedon's commentary track for the show's
first two episodes that make up the
Buffy the Vampire Slayer pilot. The
most interesting points of Whedon's
commentary come from his explanation
```

of the learning curve he encountered shifting from blockbuster films like Toy Story to a much lower-budget television series. The first disc also includes a short interview with David Boreanaz who plays the role of Angel. Other features include the script for the pilot episodes, a trailer, a large photo gallery of publicity shots and in-depth biographies of Whedon and several of the show's stars, including Sarah Michelle Gellar, Alyson Hannigan and Nicholas Brendon.

dynamic has exited with status 0.