CS160 Lab #8: Problem Solving (continued) & Software Engineering

General

Read through the "Background" material below and the online tutorials, then download the Lab #8 question set and answer the questions. Turn in the questions using the instructions posted on the class web site.

For ALL Word processing documents, you must submit your documents in one of the following formats: MS-Word (NOT Works), RTF (most word processors can save in this format), or Open Document (used by the freely available Open Office suite). They will be returned ungraded if submitted in any other format.

Concepts

This lab continues our exploration of problem solving techniques and programming. You will learn a method known as "Divide and Conquer" in this lab. Additionally, we will get an overview of Software Engineering and the Software Lifecycle.

Background

Note: Original source of the background contained below is from the "CS160 Worksheets" by Daniel Balls of the CS department at Oregon State University; updated and revised by Mitch Fry (CS, Chemeketa Community College).

Problem-Solving by Divide-and-Conquer methods

Another problem-solving method that is often utilized in computer science is called divide-and-conquer. In this paradigm, a problem is repeatedly divided into smaller and smaller sub-problems, until the solution to individual sub-problems becomes obvious and are each solved. The solutions to these sub-problems are then re-combined to form a solution to the original problem. Divide-and-conquer algorithms often require fewer steps than greedy algorithms when solving the same problem.

For example, suppose we want to arrange the following words in order from longest to shortest:

"desire to latent sometimes go misfortune awakens overwhelming on"

One method that would surely work is to apply a greedy method called *selection sort*. In this method we seek to find the longest word in the group, and once that is accomplished, we put it in the first position. Then the second longest word is determined and put in the second position and so on.

Here is how selection sort would sort the words from longest to shortest.

First, it would search the list for the longest word:

desire to latent sometimes go misfortune awakens ***overwhelming*** *on*

The longest word is "overwhelming," so it would swap "desire" and "overwhelming."

overwhelming *to latent sometimes go misfortune awakens* ***desire*** *on*

In the second iteration, it would search the remainder of the list for the longest word:

overwhelming to latent sometimes go ***misfortune*** *awakens desire on*

The longest word found is "misfortune," so it would swap "to" and "misfortune."

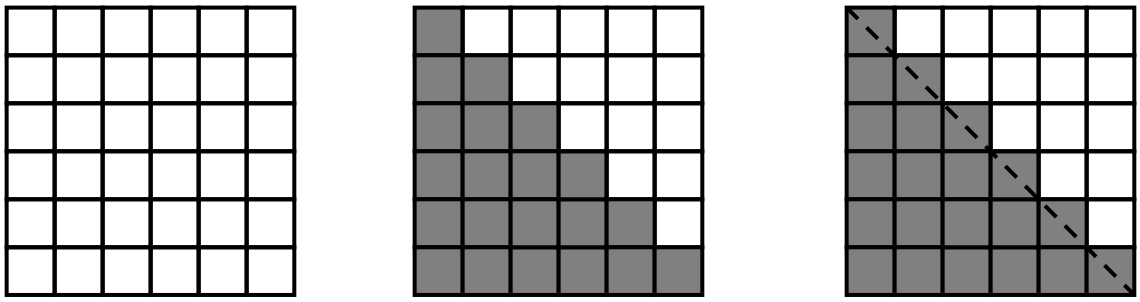
overwhelming ***misfortune*** *latent sometimes go* ***to*** *awakens desire on*

The algorithm would continue in this fashion until the list of words is completely sorted:

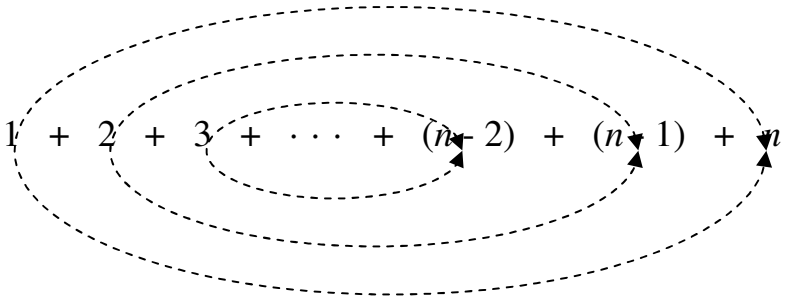
overwhelming misfortune sometimes awakens latent desire to go on

Certainly the selection sort algorithm is effective—it sorts the words correctly—but is it efficient? To determine how efficient the selection sort algorithm is, we need to think about how many times it compared a pair of words. During the first step, the number of comparisons carried out was eight—one less than the number of words in the list. At the next step, there were seven comparisons, and six at the next, and so on. In general, if there are n objects in a list to be compared, there will be $(n - 1)$ comparisons during the first step of the selection sort process, $(n - 2)$ during the second step, and so on until only 1 comparison is made at the $(n - 1)^{st}$ and last step of the process. In general, if the list contains n elements, the number of comparisons being made using the selection sort algorithm is: $(n - 1) + (n - 2) + \cdots + 3 + 2 + 1$.

This type of sum occurs so frequently in computer science that we'll briefly divert our attention to understanding its features. We would like to have a formula so that regardless of the value of n, we can always quickly find the result of adding the first n natural numbers: $1 + 2 + 3 + \cdots + (n - 1) + n$. When n is small, say 3, the sum is trivial: $1 + 2 + 3 = 6$. But what if n is 30,000? It would be a waste of time to do 29,999 additions, even with a calculator or computer, because there is a simple formula that will do the job. One way to understand this formula is to look at a geometric argument. Suppose we have an arbitrary n by n square made up of smaller squares of side length 1. In total there would be n^2 of the smaller squares—n rows and n columns (see figure below with $n = 6$). To model the desired sum geometrically we could fill in the first square of the first row, the first two squares of the second row, and so on until we fill in the first n squares of the n^{th} row (i.e. we would fill in each square of the last row of squares). The *area* of the shaded squares is equal to $1 + 2 + 3 + \cdots + (n - 1) + n$, the exact sum in which we are interested. Drawing a diagonal from the top left corner to the bottom right corner will split the n by n square exactly in half, so that the area to the left of the diagonal is $n^2/2$. But there is still some shaded area that has not been counted—the n half-shaded squares. The total area of the half-shaded squares is $n/2$.Thus the entire area of the shaded squares is $\frac{n^2}{2} + \frac{n}{2} = \frac{n^2 + n}{2} = \frac{n(n+1)}{2}$.



Another way to arrive at this formula is to write the numbers in a list and then pair the first and last numbers, then the second and second to last, and so on, as shown in the figure below.



Each of these pairs adds to (n + 1), and if n is one can see that there are n/2 pairs that have been matched up. Thus the sum of the pairs is $(n+1)\cdot\frac{n}{2}=\frac{n^2+n}{2}=\frac{n(n+1)}{2}$, which is the same result obtained using the geometric argument.

We can now see that the sum

(n – 1) + (n – 2) + · · · + 3 + 2 + 1 = $\sum_{i=1}^{n-1} i=\frac{(n-1)^2+(n-1)}{2}=\frac{n^2-n}{2}$.

Since the dominating (largest order) term in this sum is n^2, we say that the selection sort algorithm has order n^2, and often write this fact in the following way: $O(n^2)$.

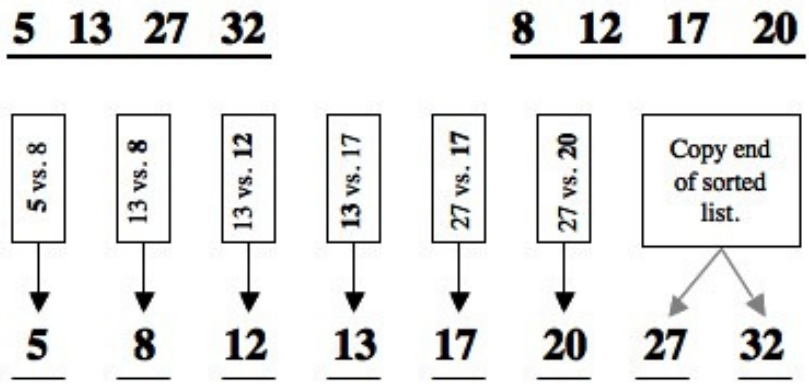
Merging Lists and the Merge Sort Algorithm

At this point we will focus on a different and more efficient way of sorting a list of objects. The algorithm that we will describe uses the divide-and-conquer problem-solving strategy. To begin, we introduce the idea of merging two lists that have already been sorted. For example, the two numerical lists that follow have each been sorted from least to greatest:

5 13 27 32 and 8 12 17 20

Merging these two lists into one in which all eight numbers have been sorted will begin, as you might expect, by comparing the first numbers of each list—in this case 5 and 8. Once these numbers have been compared (and the lesser number, 5, is placed in the first position of the merged list) the second number in the first list—13—is compared with the first number in the other list. The lesser number, 8, takes its place in the second position of the merged list and the next comparison, between the second numbers in both lists, is made. (See the diagram below for a complete illustration of this example.)

This process will go on until one of two things happens: the last number in one of the lists is compared to, and determined to be smaller than, a number from the other list; or the last two numbers in each list are compared. In the best-case scenario, the last number in one of the lists—say list A—is smaller than the first number of the other list—call it list B. In this case we will make four comparisons—comparing each of the numbers in list A with the first number in list B. Once the last number in list A has been compared with, and is determined to be smaller than, the first number in list B, no other comparisons are necessary, for list B is already in order. At this point list B is simply copied and the two lists have been merged after only ½·n comparisons.



In the worst-case scenario, the merging process ends with the last numbers from list A and list B being compared. In this case there will have been (n – 1) comparisons. In general, if n is even, it will take *at most* (n – 1) comparison to merge two lists of size $n/2$ as we shall demonstrate below.

We will now apply the divide-and-conquer paradigm to the general problem of sorting a large list of numbers. The process that we will outline below is called the merge sort algorithm. For sake of example, let's suppose we have a list of 1024 numbers. We will view the list as 1024 sorted lists of size and note that our objective is to obtain one sorted list of size 1024.

At the first step we will merge the first two single 'sorted lists' (i.e. the single numbers viewed as individual lists of size one) into a list of size two. This will require only one comparison (we've only got two numbers to compare). We'll then do the same with the next pair of numbers, and so on until there are 512 sorted lists of size two. With one comparison per merge and (½)·(1024) = 512 merges, there will be 1·512 = 512 comparisons at the first step.

At the second step we will merge two sorted lists of size two into a sorted group of four numbers. There will be at most 3 comparisons per merge (n – 1 where n = 4 = number of objects being merged) and (½)·(512) = 256 merges. So during the second step there will be *no more than* 3·256 = 768 comparisons made. Recall we may make fewer comparisons, but we'll be conservative and assume the worst-case scenario. We will look at one more step to establish the pattern: during the third step—in which we will merge sorted groups of four numbers into sorted groups of eight numbers—there will be at most 7 comparisons per merge and (½)·(256) = 128 merges. Thus at the third step we are guaranteed to complete all merges using no more than 7·128 = 896 comparisons.

In general, if we begin with an unsorted list of n numbers, at the k^{th} step of the merge-sort process, there will be at most 2^k – 1 comparisons per merge and $\frac{n}{2^k}$ merges. Thus there will be *less than* $2^k \cdot \frac{n}{2^k} = n$ comparisons.

An applet that can compare various sort algorithms is found here:

http://newterra.chemeketa.edu/faculty/mfry5/cs160/applets/SortingAlgorithms.html.

1. Change the sorting method on the left from **Quick Sort** to **Selection Sort**. Change the sorting method on the right from **Bubble Sort** to **Merge Sort**.

2. With your mouse, drag the blue circle in the *Number of Elements* box until the number of elements is 16. Choosing this number of elements will help illuminate the merge sort process.
3. At a steady rate, continually click the step button, which will move the sorting process along a step at a time. Once you get the basic idea, you can click the play button, which will move the process along at a quicker pace. If you want to slow the process down again, simply click the pause button and then the step button.
4. In the **Selection Sort** demonstration, the red bar represents the current position that is being filled and the moving yellow bar represents the bar-by-bar comparisons that are being made to determine the shortest remaining bar.
5. In the **Merge Sort** demonstration, bars 1 and 2 are merge-sorted, followed by the merge-sorting of bars 3 and 4. Now those two 'sorted lists' are merged-sorted. Thus we have an ordered list of bars 1 – 4. The next step the applet takes is to merge-sort the 5^{th} and 6^{th} bars and then the 7^{th} and 8^{th} bars. These two sorted lists of size two are now merged—forming another sorted list of four bars (representing bars 5 – 8). The two lists of four bars are merge-sorted into one list of eight bars. This same process is now applied to bars 9 – 16, until they have been sorted from shortest to tallest. The last step is to merge the two sorted lists of 8 bars, the first one consisting of bars 1 – 8 (in order from shortest to tallest) and the last one consisting of bars 9 – 16 (in order from shortest to tallest).
6. Now change the number of elements to be sorted and compare the algorithms again.

Another piece of information we need to know is how many steps there are going to be in the merge sort process. In our example of 1024 words, we can ask this question: How many times will we decrease the number of sorted lists in half (recall we started with 1024 lists of size one, and then reduced these to 512 lists of size two, and so on) before we get to one sorted list. The nature of this problem suggests we use the log function, as $\log_2(n)$ can be defined as the number of times the quantity n can be cut in half until it has been reduced to the value 1. So $\log_2(1024) = 10$, and we know that there will be 10 steps involved in reducing 1024 sorted lists of size one to one sorted list of size 1024. Recall that at each step there are less than n comparisons which means we are guaranteed to be able to sort the list of size 1024 by using fewer than 10,240 comparisons. This is significantly less than 523,776, which is the number of comparisons required for accomplishing the same task using the greedy selection sort method.

In general, if there are n words to be sorted, there will be $\log_2(n)$ steps and no more than n comparisons at each step. So we are guaranteed that there will be less than $n \cdot \log_2(n)$ comparisons made over the course of the entire process. ***In other words the merge sort algorithm is $O(n \cdot \log n)$. As was mentioned earlier, the selection sort algorithm is $O(n^2)$.***