Problem Set 1

Henry Z. Lo

June 17, 2014

Problem 1. *Count the number of inversions (out of order pairs) in an array. Let the elements be $x_1, \ldots, x_n$. There is an inversion if $x_i > x_j$ and $i < j$.*

Naive solution: bubble sort.

A better solution would be similar to merge sort. First, we divide into one element subarrays. When we merge, we add together the inversions in the two subarrays. We also need to calculate the inversions between subarrays, which can be done by sorting the array first.

Note that when counting inversions between subarrays, it doesn't matter how the subarrays are individually ordered. So we can sort the subarrays first, count inversions, then merge the subarrays into one array (as in merge sort).

```
invs = 0
countinvs(a):
  if a.length<=1:
    return a

  left = countinvs(a[1:a.length/2])
  right = countinvs(a[a.length/2:a.length])

  initialize result
  for (i,j,k = 0; i < a.length; i++)
    if k==right.length:
      result[i] = left[j++]
      invs++
    else if j==left.length:
      result[i] = right[k++]
    else if left[j] <= right[k]:
      result[i] = left[j++]
    else if left[j] > right[k]
      result[i] = right[k++]
      invs++
  return result
```

The `invs` variable contains the number of inversions. We can separate `countinvs` into multiple functions to return `invs`.

This algorithm is essentially the same as merge sort, except with inversion counting. Thus, it is $O(n \log n)$.

Problem 2. *Suppose you are investing. You want to buy high, then sell low. You have an array x of integers representing future prices, and can make one buy and one sell. What is the most you can make?*

First, divide. Then, on combining, pick the maximum of the left difference, right difference, and a middle difference. The middle difference is the largest thing in the left, minus the largest thing in the right.

```
maxdiff(a):
  if a.length==1:
    return -INFINITY
  left = a[0:a.length/2]
  right = a[a.length/2:a.length]
  return max(maxdiff(left), maxdiff(right),
             max(left) - min(right), 0)
```

`max(left) - min(right)` is $O(n)$. There are $\log n$ splits. Runtime is $O(n \log n)$.

Problem 3. *There are n files of different sizes, and you want to put them on a tape. In tapes, a file can only be accessed after every other file on the same tape. Place the files on tape to minimize the average access time.*

Let the file size of i be f_i. In order to get to f_i, we have to read in $f_1 + \ldots + f_{i-1}$, plus f_i for reading file i. On average, the expected cost is:

$$\frac{1}{n} \sum_{j=1}^{n} \sum_{i=1}^{j} f_i$$

It should be clear from this equation that we want to place the largest file (with size f_n) at the end, so that it only appears once in this sum.

Code and runtime is essentially equivalent to a sorting algorithm.

Problem 4. *It has been hypothesized that couples which are closer in social desirability are more likely to have a stable relationship. Suppose we have n heterosexual men, and n heterosexual females. Each person has an integer social desirability score. Find the pairings which maximizes overall stability.*

Let m_i be the social desirability of the i^{th} male, and f_i for females. One greedy approach we can use – pair those closest first, then pair everyone else – is not optimal if. Consider this group:

females	6	9	13
males	3	7	8

This greedy strategy would pair f_2 and m_1, then f_1 and m_2, yielding a total difference of 10, when the optimal solution would yield an 8.

The greedy strategy that works is this: first, we sort all males and females by social desirability. Then we just assign the same ranked individuals with each other.

The algorithm is essentially a sorting algorithm, followed by array traversal. We can show evidence that this algorithm works by drawing two points for each gender, in every possible ordering, then showing that every pairing other than ordered is not optimal.

Problem 5. *Suppose you have a rod of length n, which you can cut into sections. You can cut up sections of length $i = 1, \ldots, n$ inches. Each different length fetches a different price. For example:*

length i	*1*	*2*	*3*	*4*	*5*	*6*
price p_i	*2*	*3*	*5*	*6*	*7*	*10*

How can we cut up a rod to maximize profit?

This is a classic dynamic programming problem. Suppose $n = 5$. We can arrive at a given length of rod multiple ways, but the only thing that matters is the one path which maximizes profit.

Let's consider the case when $n = 5$, and the table is:

length i	1	2	3	4	5
price p_i	2	3	4	7	8

The solution here is $4, 1$. We can solve this bottom-up or top-down. Also, draw the computation tree for this problem, and convert it into a graph.

```
p = value array
s = array with all elements set to negative infinity
memoize_rod(n):
  if (s[n] >= 0)
    return s[n]
  else if (n == 1)
    value = p[1]
  else
    value = negative infinity
    for (int i=1; i<n; i++)
      value = max(value, p[i] + memoize_rod(n-i))
  s[n] = value
  return value

p = value array
s = array with all elements set to negative infinity
dp_rod(n):
  s[0] = 0
```

```
  for (int j=1; j<n; j++)
    value = negative infinity
    for (int i=1; i<j; i++)
        value = max(value, p[i] + s[j-i])
    s[j] = value
  return s[n]
```

Both algorithms are $O(n^2)$.

Problem 6. *You have three sets of elements, s_1, s_2, s_3. Find all elements in exactly one set.*

We can do this in two ways: using only hash tables, and using bit strings.

Hash table: Construct hash sets for s_1, s_2, s_3 and s_3. Then, iterate over every element of s_1, and check this element's membership in s_2 and s_3. If it doesn't exist in other, keep it. Repeat for s_2 and s_3. Code:

```
unique_elements(s1, s2, s3):
  initialize hashsets h1, h2, h3, h4
  for i in s1:
    h1.add(i)
  for i in s2:
    h2.add(i)
  for i in s3:
    h3.add(i)
  for i in s1,s2,s3:
    if ((i not in s1) and (i not in s2)) or
       ((i not in s2) and (i not in s3)) or
       ((i not in s1) and (i not in s3)):
      h4.add(i)
  return h4
```

This algorithm iterates over all elements twice.

Bit strings: Construct mapping for every element in s_1, s_2, s_3 to a bit. Construct bit sets. Then perform `xor` on all three bit sets.

```
unique_elements(s1, s2, s3):
  initialize bitsets b1, b2, b3, map
  c = 0
  for i in s1:
    if i not in map:
      map[i] = c++
    b1.set(map[i])
  for i in s2:
    if i not in map:
      map[i] = c++
    b2.set(map[i])
```

```
  for i in s3:
    if i not in map:
      map[i] = c++
    b3.set(map[i])
  return b1 XOR b2 XOR b3
```

This iterates over all elements once.