CS3933 Fall 2007 – Group Assignment 4
Due Wednesday November 14, at 5pm.

Note that, even though desired running times are not specified, faster algorithms are better and thus worth more marks.

1. (12 marks) You are organizing a practice triathlon for a group of your friends. In this triathlon, each person will complete a swim, a bike ride, and a run, in that order, without stopping. Since this is a practice triathlon, each person's time to complete each segment is known, so each person j takes s_j minutes to complete the swim segment, b_j minutes to complete the bike segment, and r_j minutes to complete the run. However, since this is only practice, you only have one lane in the pool, so the swim segment must be completed one person at a time. The bike and run segments are not constrained, so everyone can do those at once.

   Supervising this entire process is quite time consuming, so you want to know what the earliest time is that you could be done (all people having completed all three segments). Assume that the first swimmer starts at time 0, and that you can determine the order of the competitors.

   (a) (5 marks) Design a greedy algorithm that will find the minimum completion time.

   for i from 1 to n
    $c_i \leftarrow b_i + r_i$
   sort the people by nonincreasing c_i value, using mergesort
   $t \leftarrow 0$
   $f \leftarrow 0$
   for i from 1 to n (in sorted order)
    if $f < t + s_i + c_i$
     $f \leftarrow t + s_i + c_i$
    $t \leftarrow t + s_i$
   return f

   (b) (1 mark) Analyse your algorithm's running time.

   The two for loops are independent, and each iterate n times. The mergesort, also independent, runs in $\Theta(n \log n)$ time. The rest of the algorithm contributes a constant factor, so the algorithm as a whole runs in time $\in \Theta(n \log n)$.

   (c) (6 marks) Prove that your algorithm works correctly.

Assume not. Then there is some input for which our greedy algorithm produces a computation order $g_1, g_2, \ldots, g_n$ that is not optimal. Since the greedy order differs from all optimal orders, consider the optimal order $o_1, o_2, \ldots, o_n$ where r is maximised so that $g_i = o_i \;\; \forall 1 \leq i < r$ and $g_r \neq o_r$.

Since these sequences are permutations, there will be some k such that $o_k = g_r$, and since the sequences are identical before position r, we know that $k > r$.

Modify the optimal by moving o_k to between o_{r-1} and o_r. We now need to show that this modification will not increase the maximum finishing time of the people in the sequence.

For any person o_i in the optimal sequence, where we use the indices in the order that the people are listed for the optimum:

$$time(o_i) = c_i + \sum_{j=1}^{i} s_j$$

If $i < r$, nothing before or including person i has changed, so $time(i)$ is unchanged. If $i > k$, the people before person i have been rearranged, but this change in order will not affect the sum of the s_j times, so $time(i)$ is unchanged.

If $i = k$: $time(k) = c_k + \sum_{j=1}^{k} s_j$ has been changed to $c_k + \sum_{j=1}^{r-1} s_j$, which will be no more than the original $time(k)$ since $r - 1 < k$.

If $r \leq i < k$: $time(i) = c_i + \sum_{j=1}^{i} s_j$ has been changed to $c_i + s_k + \sum_{j=1}^{i} s_j \;\leq\; c_i + \sum_{j=1}^{k} s_j$. Since the greedy algorithm orders the people by nonincreasing c_i, and the greedy algorithm selected $o_k = g_r$ before any of the o_i with $i \geq r$ (since the people selected before g_r by the greedy algorithm were the same as the people selected before o_r by the optimum), then $c_k \geq c_i$ for all $r \leq i < k$. Therefore the modified time for i is

$$c_i + s_k + \sum_{j=1}^{i} s_j \;\leq\; c_i + \sum_{j=1}^{k} s_j \;\leq\; c_k + \sum_{j=1}^{k} s_j \;=\; time(k)$$

so the finishing time for person i in the modified optimum is no more than $time(k)$ for the optimum.

Therefore no person in the modified optimum finishes after the latest person from the original optimum, so this modified order is still optimum. However, it contradicts the selection of the optimum that would maximise r, since it agrees with the greedy order for the first r positions instead of only the first $r - 1$ positions.

Since we have a contradiction, our assumption must be wrong, and the greedy algorithm is optimal.

2. (9 marks) Combinations can be calculated directly using a formula involving factorials, but this type of calculation performed using a computer often has serious problems either due to integer overflow from intermediate results or rounding errors from floating-point calculations. Instead, consider the following recurrence that calculates combinations:

$$C(i,j) = \begin{cases} 0 & \text{if } j > i \\ 1 & \text{if } i = j \text{ or } j = 0 \\ C(i-1,j) + C(i-1,j-1) & \text{otherwise} \end{cases}$$

(a) (5 marks) Design a dynamic programming algorithm that, given n and k, will find the value of $C(n,k)$.

```
if k > n return 0
Let C[0..n][0..k] be a 2-dimensional table of integers.
C[0][0] ← 1
for i from 1 to n
    C[i][0] ← 1
    for j from 1 to min(i − 1, k)
        C[i][j] ← C[i − 1][j] + C[i − 1][j − 1]
    C[i][i] ← 1
return C[n, k]
```

Note that they should still get full marks if the loop on j does not include the k cutoff (and so goes to $i-1$ each time). That variant will be a bit easier to analyse, but that's ok.

(b) (4 marks) Analyse the running time and space of your algorithm.

Time: (we consider that $n \geq k$; if not, the algorithm runs in constant time.)

Upper bound:

The outer for loop iterates n times. For each of these interations, the inner for loop iterates at most k times. The rest of the algorithm contributes at most a constant factor, so the algorithm as a whole runs in time $\in O(nk)$.

Lower bound:

Consider the i values such that $i > \frac{k}{2}$. There will be at least $n - \frac{k}{2}$ iterations of the outer for loop that have such an i value, and since $k \leq n$, $n - \frac{k}{2} \geq n - \frac{n}{2} = \frac{n}{2}$. For each of these iterations, the inner loop (on j) iterates at least $\frac{k}{2}$ times. Therefore the algorithm runs in time $\in \Omega(\frac{n}{2} \cdot \frac{k}{2}) = \Omega(nk)$.

Since the algorithm runs in time that is both $\in O(nk)$ and $\in \Omega(nk)$, it runs in time $\in \Theta(nk)$.

Note, however, that the input is two numbers; the size of the input is thus $m = 2\log_2 n$, so the running time is $\in \Theta(2^m)$.

Space:

The table takes space for $(n+1)(k+1)$ integers. There are 2 additional integer counters. Thus, assuming that an integer takes constant space, this algorithm takes $\Theta(kn)$ space.

3. (16 marks) You are running a small internet company that needs regular data communications. Your communications needs are predictable. You have access to two possible data communications plans; one has a *flat* fee of X dollars per month for any amount of data, and the other has a *rate* fee of Y for each megabyte transferred. Both plans have an activation fee of Z, which also applies if you switch from one plan to the other.

   You need to design and write a dynamic programming algorithm that will determine which plan (flat or rate) you should have for each month in order to minimise your total cost. This algorithm is given the values for X, Y, and Z, and communication data for n months, where each month i has a megabyte amount m_i.

   (a) (5 marks) Design an algorithm that will, given the input data, find the minimum total cost.

   Let F and R be arrays of length n.
   $F[1] \leftarrow Z + X$
   $R[1] \leftarrow Z + Y \cdot m_1$
   for i from 2 to n
       $F[i] \leftarrow F[i-1] + X$
       if $F[i] > R[i-1] + Z + Y \cdot m_i$
           $F[i] \leftarrow R[i-1] + Z + Y \cdot m_i$
       $R[i] \leftarrow R[i-1] + Z + X$
       if $R[i] > R[i-1] + Y \cdot m_i$
           $R[i] \leftarrow R[i-1] + Y \cdot m_i$
   $result \leftarrow F[n]$
   if $result > R[n]$
       $result \leftarrow R[n]$
   return $result$

(b) (3 marks) Adapt your algorithm from (a) to find the plan schedule that produces the minimum.

Let F and R be arrays of length n.
Let T_F and T_R be arrays of labels of length n.
$F[1] \leftarrow Z + X$
$R[1] \leftarrow Z + Y \cdot m_1$
for i from 2 to n
    $F[i] \leftarrow F[i-1] + X$
    $T_F[i] \leftarrow$ "flat"
    if $F[i] > R[i-1] + Z + Y \cdot m_i$
        $F[i] \leftarrow R[i-1] + Z + Y \cdot m_i$
        $T_F[i] \leftarrow$ "rate"
    $R[i] \leftarrow R[i-1] + Z + X$
    $T_R[i] \leftarrow$ "flat"
    if $R[i] > R[i-1] + Y \cdot m_i$
        $R[i] \leftarrow R[i-1] + Y \cdot m_i$
        $T_R[i] \leftarrow$ "rate"
$result \leftarrow F[n]$
$last \leftarrow$ "flat"
if $result > R[n]$
    $result \leftarrow R[n]$
    $last \leftarrow$ "rate"

for i from n down to 1
    print "month" i "is" $last$
    if $last =$ "flat"
        $last \leftarrow T_F[i]$
    else
        $last \leftarrow T_R[i]$
return $result$

(c) (1 mark) Analyse the running time and space of your algorithm from (a).

The for loop iterates $n - 1$ times. All other parts of the algorithm contribute a constant factor or term, so the algorithm runs in time $\in \Theta(n)$.

(d) (7 marks: 3 for code, 4 for test cases) Implement your algorithm (from (b), preferably) and test it on three suitable test cases. Submit your source code, test

cases, and results.

Marking notes: The code should look like it works, and the test cases should include some cases where the greedy approach of picking the best result for each month (individually or cumulatively) does not work.