CS3933 Winter 2007 – Group Assignment 4
Due Wednesday March 21, at 5pm.
Sample Solution

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

1. A set of n test cases need to be run to test a new high-tech sensor. Each case i must first be run on the sensor for time s_i to collect data, and then this data is processed by a computer in time p_i. However, there is only one sensor (and only one test case at a time can be run on it), while there are over n computers available for processing. We want to schedule the test cases so that the entire set will be completed as soon as possible (both data collecting and processing, for all test cases). Completion time is defined as the elapsed time between when the first test case starts running on the sensor to when the last test case completes processing.

   (a) (4 marks) Design a greedy algorithm that will find the minimum completion time. Hint: first devise an expression that will, given a test case order, calculate the completion time.

   Algorithm Bottleneck(n, $\{s_i\}$, $\{p_i\}$)
   - sort the test cases into order by decreasing (nonincreasing) p_i value
   - $time \leftarrow 0$
   - $max \leftarrow 0$
   - for each test case i, in the sorted order from 1 to n
     - $end \leftarrow time + s_i + p_i$
     - if $end > max$
       - $max \leftarrow end$
     - $time \leftarrow time + s_i$
   - return max

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

   The sort, using mergesort, will take time $\in \Theta(n \log n)$. The for loop will have n iterations, and each iteration takes constant time. Since the rest of the algorithm takes constant time, the algorithm as a whole takes time $\in \Theta(n \log n)$.

   (c) (5 marks) Prove that your algorithm works correctly.

   Proof: 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 completion time of jobs in the sequence.

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

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

If $i < r$, nothing before or including job i has changed, so $time(i)$ is unchanged. If $i > k$, the jobs before job 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 $i = k$: $time(k) = p_k + \sum_{j=1}^{k} s_j$ has been changed to $p_k + \sum_{j=1}^{r-1} s_j$, which will be no more than the original $time(k)$ since $r - 1 < k$.

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

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

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

Therefore no job in the modified optimum finishes after the latest job 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. Consider the following recurrence, which includes both initialization and conditional cases (so you would use the minimum of only the cases whose conditions are met):

$$X(i,j) = \min \begin{cases} X(i-1,j) + a_i & \text{if } i > 0 \text{ and } j \geq 0 \\ X(i-1,j-1) + 2 \cdot a_j & \text{if } a_j > i,\ i > 0, \text{ and } j > 0 \\ \text{j} & \text{if } i = 0 \end{cases}$$

Note that this recurrence is not intended to mean anything; just work with it as it is given.

(a) (4 marks) Design a dynamic programming algorithm that, given n input values $\{a_i\}$, will find the value of $X(n,n)$.

```
Algorithm X(n, {a_i})
    Let M be a 2-dimensional table of values, (n+1) × (n+1).
    for j from 0 to n
        M[0, j] ← j
    for i from 1 to n
        M[i, 0] ← M[i − 1, 0] + a_i
        for j from 1 to n
            M[i, j] ← M[i − 1, j] + a_i
            if a_j > i and M[i, j] > M[i − 1, j − 1] + 2 · a_j
              M[i, j] ← M[i − 1, j − 1] + 2 · a_j
    return X[n, n]
```

(b) (3 marks) Analyse the running time and space of your algorithm.

Time Analysis:

The first for loop iterates $n+1$ times. Additionally, there is a pair of nested for loops, each of which iterates n times, so the innermost lines of code are executed n^2 times. The rest of the algorithm provides a constant factor, so the algorithm as a whole runs in time $\in \Theta(n^2)$.

Space Analysis:

This algorithm uses a $(n+1) \times (n+1)$ table, and two additional variables. Thus it runs in $\Theta(n^2)$ space.

3. The post office sells stamps of a variety of different denominations (values), and different sizes. Consider a set of n stamps, where each stamp i has a specific denomination d_i and a size s_i. All denominations are distinct. All input numbers are positive. We can use stamp denominations multiple times.

When buying stamps that total to a given postage amount P, we want to minimize the total size of the stamps (to leave the most addressing space on the envelope).

(a) (6 marks) Design an algorithm that will, given the information about n stamps (denominations $\{d_i\}$ and sizes $\{s_i\}$) and a postage amount P, find the minimum total size of stamps that will add to P (if such a set exists).

Note that we can assume that the input values are integers, since we could adjust the input to accomodate fractions if they are not.

```
Algorithm Postage(n, {d_i}, {s_i}, P)
    Let M be a 2-dimensional table of numbers, (n+1) × (P+1)
    M[0,0] ← 0
    for j from 1 to P
        M[0,j] ← ∞   (no stamps but still need postage)
    for i from 1 to n
        M[i,0] ← 0   (no postage left to make)
        for j from 1 to P
            M[i,j] ← M[i−1,j]
            if d_i ≤ j and M[i,j] > M[i,j−d_i] + s_i
              M[i,j] ← M[i,j−d_i] + s_i
    return M[n,P]
```

(b) (4 marks) Adapt your algorithm from (a) to find the set of stamps that produces the minimum.

```
Algorithm Postage(n, {d_i}, {s_i}, P)
    Let M be a 2-dimensional table of numbers, (n+1) × (P+1)
    Let T be a 2-dimensional table of labels, (n+1) × (P+1)
    Let D be a 1-dimensional array of n integers
    M[0,0] ← 0
    for j from 1 to P
        M[0,j] ← ∞   (no stamps but still need postage)
    for i from 1 to n
        M[i,0] ← 0   (no postage left to make)
        for j from 1 to P
            M[i,j] ← M[i−1,j]
            T[i,j] ← "skip"
            if d_i ≤ j and M[i,j] > M[i,j−d_i] + s_i
              M[i,j] ← M[i,j−d_i] + s_i
              T[i,j] ← "use"
    if M[n,P] = ∞
        print "postage not possible"
```

```
    else
        for i from 1 to n
            D[i] ← 0
        Trace(T, {d_i}, D, n, P)
    return M[n, P]

Algorithm Trace(T, {d_i}, D, i, j)
    if j > 0
        if T[i, j] = "skip"
            Trace(T, {d_i}, D, i − 1, j)
        else
            D[i] ← D[i] + 1
            Trace(T, {d_i}, D, i, j − d_i)
            print "use d_i stamp"
```

(c) (3 marks) Analyse the running time and space of your algorithm from (a).

Time Analysis:

The first for loop iterates P times. Additionally, there is a pair of nested for loops, which iterate n (outer) and P (inner) times, so the innermost lines of code are executed $n \cdot P$ times. The rest of the algorithm provides a constant factor, so the algorithm as a whole runs in time $\in \Theta(nP)$.

Space Analysis:

This algorithm uses a $(n+1) \times (P+1)$ table, and two additional index variables. Thus it runs in $\Theta(nP)$ space

(d) (7 marks: 4 for code, 3 for test cases. subtract 2 if traceback not implemented) Implement your algorithm (from (b), preferably) and test it on three suitable test cases. Submit your source code, test cases, and results.

The implementation should be quite straightforward, except that an appropriate flag value for the "impossible" cases, to be used instead of ∞, needs to be chosen and used carefully (as very large values will likely roll over). The test cases need to include at least one impossible case, and at least one case where more than one of a particular stamp type is needed. It should also include a non-greedy case (where taking the largest type of stamp will not work).