Final - Due Tuesday 12/12/2000

K. Subramani
Department of Computer Science and Electrical Engineering,
West Virginia University,
Morgantown, WV
ksmani@csee.wvu.edu

1 Asymptotic Notation

1. Yes, because I can multiply $0.2831n$, by a constant such as $\frac{2}{0.2831}$ to get a function that increases faster than n;

2. Let $f(n) = n^2$ and $g(n) = n.\log^5 n$. Using techniques taught in class,

$$\begin{aligned}
\lim_{n\to\infty} \frac{f(n)}{g(n)} &= \lim_{n\to\infty} \frac{n^2}{n.\log^5 n} \\
&= \lim_{n\to\infty} \frac{n}{\log^5 n} \\
&= \lim_{n\to\infty} \frac{\log n}{5.\log\log n} \\
&= \lim_{n\to\infty} \frac{\frac{1}{n}}{5.\frac{1}{n.\log n}} \\
&= \lim_{n\to\infty} \frac{\log n}{5} \\
&= \infty
\end{aligned}$$

   This implies that $n.\log^5 n = o(n^2)$.

2 Divide and Conquer

1. Use the same idea as in MERGE-SORT. It is instructive to note that in the worst-case, there are $O(n^2)$ inversion pairs (a reverse-sorted list, for instance!); hence an approach that enumerates inversion pairs may be sub-optimal. Algorithm (2.1) provides the required Divide-and-Conquer strategy. Assume without loss of generality, that the number of elements is an exact power of 2. In class, I argued why this assumption is not restrictive.

   The running time of the algorithm is clearly described by the recurrence:

$$T(n) = 2.T(\frac{n}{2}) + O(n)$$

   which gives $T(n) = O(n.\log n)$.

Function CARDINALITY-INVERSION-PAIR($\mathbf{A}, p, q$)

1: Let $n = q - p + 1$ {n is the number of elements in $\mathbf{A}$}
2: **if** $(n = 1)$ **then**
3: **return**(0)
4: **end if**
5: Let $mid = \frac{p+q}{2}$
6: l_1 =CARDINALITY-INVERSION-PAIR($\mathbf{A}, p, mid$)
7: l_2 =CARDINALITY-INVERSION-PAIR($\mathbf{A}, mid + 1, q$)
8: l_3 = NEW-MERGE($\mathbf{A}, mid, p, q$)
9: **return**($l_1 + l_2 + l_3$)

Algorithm 2.1: Inversion Pair cardinality

Function NEW-MERGE($\mathbf{A}, mid, p, q$)

1: $l_3 = 0; x \leftarrow p; y \leftarrow (mid + 1); z \leftarrow p$
2: Create temporary array $\mathbf{C}$ of size $[q - p + 1]$
3: {When an element from the left side, i.e. $\mathbf{A}[p..mid]$ is moved into the temporary array $\mathbf{C}$, we add y values to the count l_3.}
4: **while** $(x \leq mid)$ **and** $(y \leq q)$ **do**
5: **if** $(\mathbf{A}[x] \leq \mathbf{A}[y])$ **then**
6: { This is not an inversion pair}
7: $\mathbf{C}[z] = \mathbf{A}[x]; z++; x++$
8: **else**
9: { This is an inversion pair}
10: $\mathbf{C}[z] = \mathbf{A}[y]; l_3 = l_3 + (y - 1); z++; y++$
11: **end if**
12: **end while**
13: **if** $(y > q)$ **then**
14: { Each of the remaining elements in the left array form exactly q inversion pairs}
15: Copy these elements into C
16: $l_3 = l_3 + (mid - p + 1) \times q$
17: **end if**
18: Copy $\mathbf{C}$ back into $\mathbf{A}$
19: **return**(l_3)

Algorithm 2.2: A different type of Merging

Function GREEDY-COIN-CHANGER($\mathbf{A}, mid, p, q$)

1.
1: Divide n by 25; Let n_1 be the quotient and r_1 be the remainder
2: Divide r_1 by 10; Let n_2 be the quotient and r_2 be the remainder
3: Divide r_2 by 5; Let n_3 be the quotient and r_3 be the remainder
4: Output n_1 quarters, n_2 dimes, n_3 nickels and r_3 pennies.

Algorithm 3.1: Changing n cents into coins

3 Greedy

Let Chris' algorithm (denoted by C) break up n cents using $\{p_C, n_C, d_C, q_C\}$ coins, where

(a) p_C stands for the number of pennies (1 cents)

(b) n_C stands for the number of nickels (5 cents)

(c) d_C stands for the number of dimes (10 cents)

(d) q_C stands for the number of quarters (25 cents)

Likewise let the greedy strategy (denoted by G) break up n cents using $\{p_G, n_G, d_G, q_G\}$ coins. If Chris's strategy is better than the greedy strategy, then we must have

$$p_C + n_C + d_C + q_C < p_G + n_G + d_G + q_G$$

We have to build up the case, using increasing denominations.

Observation: 3.1 *If the only denomination allowed was the penny, C cannot beat G*

Proof: *Obvious, since both strategies would have exactly n coins each.* □
Let us now consider the case, when both pennies and nickels are allowed i.e. the denomination set is $\{p, n\}$.

Lemma: 3.1 *$n_G \geq n_C$, i.e. the greedy strategy has at least as many dimes as Chris' strategy.*

Proof: *If Chris' strategy had more nickels, than the greedy strategy did not grab all the available nickels and hence was not greedy.* □
If $n_C = n_G$, then Chris' strategy is no better than G. Now consider the case, when $n_G > n_C$. Let $n_G - n_C = m$, where $m \geq 1$. So Chris is keeping the number of nickels low, while pushing $5m$ cents onto the penny stack. But by doing so, he increases the number of pennies by $5m$ as compared to the greedy strategy, while the greedy strategy needs exactly m more nickels to account for the $5.m$ difference. Thus, if $n_C < n_G$, the greedy strategy beats Chris' strategy by $5m - m = 4m$ coins

We increase the denomination set to include pennies, nickels and dimes i.e. $\{p, n, d\}$. Arguing as in the previous case, we must $d_G \geq d_C$. If $d_G = d_C$, then Chris' cannot beat the greedy strategy, since we get a version of the problem considered above. Let us therefore, consider the case, when $d_G - d_C = m \geq 1$. Chris saves on dimes, by pushing $10.m$ cents onto the nickel and penny stacks. However, for each dime he gains by pushing it on the lower denomination stacks, he could lose in the following ways:

(a) He chooses 10 pennies; Greedy beats Chris by 9 coins

(b) He chooses 2 nickels; Greedy beats Chris by 1 coins

(c) He chooses 1 nickel and 5 pennies; Greedy beats Chris by 5 coins.

Thus in all eventualities, the greesy strategy is better. (This is because 10 divides 5 exactly!)

An identical argument can be made to extend the denomination set to include quarters.

2. Take the coin denominations, $1, 5, 10, 11$. Assume that $n = 15$. The greedy approach gives the break-up $11 + 1 + 1 + 1$, with cardinality 4; the optimal cardinality is 2 with break-up $10 + 5$.

4 Dynamic Programming

This is identical to the Quiz 2 problem. Please make substitutions as necessary. A simple analysis gives running time of $O(n.L^2)$.

5 Graph Algorithms

The simplest test that I can think of is computing D^n and comparing it with a copy of D^{n-1} (using the first dynamic programming algorithm). If the graph has no negative weight cycles, then relaxing the edges one more time should not decrease the length of the shortest path path to any vertex, since the shortest path between any pair of vertices has at most $(n-1)$ edges. Thus there is no negative weight cycle if and only if $D^n = D^{n-1}$.