CME 323: Distributed Algorithms and Optimization

Instructor: Reza Zadeh (rezab@stanford.edu)

TA: Yokila Arora (yarora@stanford.edu)

HW#1 – Solution

1. A computation takes 250 seconds to run on a single core, i.e. $T_1 = 250$. Is it possible to design a multithreaded version of the computation that takes time 40 seconds on 5 processors with greedy scheduling? How about 60 seconds on 5 processors? Justify your answer.

   Solution By Brent's theorem, we have

   $$\frac{T_1}{p} \leq T_p \leq \frac{T_1}{p} + T_\infty$$

   Since $T_1 = 250$, using 5 processors, $T_p \geq 50$. Thus, it is not possible to design a multithreaded algorithm which takes only 40 seconds.

   It may be possible to design a multithreaded algorithm which takes 60 seconds, under the assumptions we made for greedy scheduling.

2. The Karatsuba algorithm multiplies two integers x and y. Assuming each has n bits where n is a power of 2, it does this by splitting the bits of each integer into two halves, each of size $n/2$. For any integer x we will refer to the low order bits as x_l and the high order as x_h. The algorithm computes the result as follows:

```
Algorithm 1: Karatsuba Multiply
   Signature: KM(x,y,n)
   Input    : Two integers $x, y$, each having $n$ bits
   Output   : The product $x \cdot y$
1  if $n = 1$ then
2      return $x \cdot y$
3  end
4  else
5      $a \leftarrow \mathbf{km}(x_\ell, y_\ell, n/2)$
6      $b \leftarrow \mathbf{km}(x_h, y_h, n/2)$
7      $c \leftarrow \mathbf{km}(x_l + x_h, y_l + y_h, n/2)$
8      $d \leftarrow c - a - b$
9      return $(b2^n + d2^{n/2} + a)$
10 end
```

   Note that multiplying by 2^k can be done just by shifting the bits over k positions. If you have seen this before, you might have thought of it as a sequential algorithm, but actually it is a parallel algorithm; in particular, the three recursive calls to **km** can be made in parallel.

   (a.) Assuming addition, subtraction, and shifting take $O(n)$ work and $O(n)$ depth what is the work and depth of **km**?

Solution We recall from "Parallel Algorithms" (Blelloch and Maggs) that the *work* of an algorithm is the total number of operations performed, and the *depth* is defined as the longest chain of dependencies among its operations.

Each invocation of `km` with n-bit integers results in `km` being called three times (in parallel), each with input size $n/2$. We are told that $n = 2^k$ for some $k \in \mathbb{Z}^+$. Our recursion bottoms out when $n = 1$, in which case we perform a single-digit multiply in constant time. Notice that the number of single-digit multiplies, i.e. the number of times we bottom-out in our recursion and hit our base-case, is given by 3^k, where each multiply takes $O(1)$ *work*.

Let $W(n)$ define the total work of our algorithm. Since additions, subtractions, and bit-shifts are assumed to require $O(n)$ work, we may express

$$W(n) = 3W\left(\frac{n}{2}\right) + \alpha n$$

for some constant $\alpha \in \mathbb{R}^+$. Using the Master Theorem,[1] we see that $W(n) = \Theta(n^{\log_2 3})$.

With respect to Depth, let $D(n)$ denote the depth of our algorithm. We know that addition, subtraction, and shifting also require $O(n)$ depth. Notice that the recursive calls to `km` are made in parallel and therefore share no dependencies. Hence

$$D(n) = D(n/2) + \alpha n,$$

for some $\alpha \in \mathbb{R}$. Using the Master Theorem,[2] we see that $D(n) = \Theta(n)$.

(b.) Assuming addition, subtraction, and shifting take $O(n)$ work and $O(\log n)$ depth what is the work and depth of **km**?

Solution Work remains the same. But now our Depth is given by

$$D(n) = D(n/2) + O(\log n).$$

Applying the Master Theorem, we fall into Case 2.[3] Thus, $D(n) = \Theta(\log^2 n)$.

3. Suppose a square matrix is divided into blocks:

$$M = \begin{bmatrix} A & B \\ C & D \end{bmatrix}$$

where all the blocks are the same size. The *Schur complement* of block D of M is $S = A - BD^{-1}C$. The inverse of the matrix M can then be expressed as:

$$M^{-1} = \begin{bmatrix} S^{-1} & S^{-1}BD^{-1} \\ -D^{-1}CS^{-1} & D^{-1} + D^{-1}CS^{-1}BD^{-1} \end{bmatrix}$$

[1]Here, $a = 3, b = 2$, hence $\log_b a = \log_2 3$. Then, $f(n) = \alpha n = O(n)$. Thus $c = 1 < \log_2 \approx 1.6$. Case 1.

[2] Here, $a = 1$, $b = 2$, hence $\log_b a = \log_2 1 = 0$. So $f(n) = \alpha n = \Omega(n)$. This places us into Case 3. We check that $f(n/2) \leq kf(n)$ for some constant $k < 1$ – i.e. choose $1/2 < k < 1$, then $\alpha n/2 \leq k\alpha n$ satisfied.

[3]Specifically, $a = 1$, $b = 2$, hence $c = \log_b a = \log_2 1 = 0$. Hence for $k = 1$, $f(n) = \log n = \Theta(n^c \log^k n)$.

This basically defines a recursive algorithm for inverting a matrix which makes two recursive calls (to calculate D^{-1} and S^{-1}), several calls to matrix multiply, and one each to elementwise add and subtrace two matrices. Assuming that matrix multiply has work $O(n^3)$ and depth $O(\log n)$ what is the work and depth of this inversion algorithm?

Solution We are explicitly told what recursive algorithm we are using. It involves two recursive calls, each on a square matrix whose side-length is half as large. The following steps in the algorithm require matrix multiplies which dominate the work and depth of elementwise operators.[4]

Therefore, we set up our recurrence for *work*,

$$W(n) = 2W\left(\frac{n}{2}\right) + \alpha n^3,$$

for some $\alpha \in \mathbb{R}^+$. Using the Master Theorem[5] , we conclude that $W(n) = O(n^3)$.

With regard to the depth of the algorithm, notice that D^{-1} required to compute S^{-1}, i.e. these operations may *not* be done in parallel. Hence

$$D(n) = 2D\left(\frac{n}{2}\right) + O(\log n).$$

We fall into case 1 of the Master Theorem, since $a = b = 2$ and $f(n) = O(\log n) = O(\sqrt{n})$,[6] Thus, $D(n) = O(n)$.

4. Describe a divide-and-conquer algorithm for merging two sorted arrays of lengths n into a sorted array of length $2n$. It needs to run in $O(n)$ work and $O(\log^2 n)$ depth. You can write the pseudocode for your algorithm so that it looks like your favorite sequential language (C, Java, Matlab, ...), but with an indication of which loops or function calls happen in parallel. For example, use `parallel for` for a parallel for loop, and something like:

```
parallel {
    foo(x,y)
    bar(x,y)
}
```

to indicate that `foo` and `bar` are called in parallel. You should prove correctness at the level expected in an algorithms class (e.g. CME305 or CS161).

[4]Specifically, the element-wise add or subtract requires n^2 independent additions. The work is clearly $O(n^2)$. Notice that depth $O(1)$, since with $p = n^2$ processors we can perform exactly one of the n^2 operations on each processor in constant time.

[5]$a = 2, b = 2$, so $\log_b a = 1$; $f(n) = O(n^3)$ implies $c > \log_b a$, i.e. Case 3. We check $2f(n/2) \leq kf(n)$ for some $k \in (0, 1)$, i.e. let $k = 1/2$ then $2\alpha n^3/8 = \alpha n^3/4 \leq \alpha n^3/2$.

[6]To see why, note that $\log x < x$ for all $x > 0$. Then, $\log x = 2\log(\sqrt{x}) < 2\sqrt{x}$.

Solution Recall that on a single machine, we can solve this problem in $O(n)$ time by maintaining a pointer to the head of each sorted array, stepping through both arrays simultaneously, appending the lesser element of the two arrays to the sorted output, and incrementing the corresponding pointer; this iterates for at most $2n$ steps.

Notice that we may find the median of a sorted array in $O(1)$ time. Let $n = |A|$. If n odd, then the median element is uniquely determined by index $(n-1)/2$ (where we *index* starting from 0). If n even, there are two medians with *indices* at $n/2$ and $n/2 - 1$.

Below, $\preceq$ denotes the *element-wise inequality* operator. In our pseudo-code, we index an array just like arrays are sliced in python, e.g. `a[1:j]` means that we start at the *second* element and take all elements up to but *not including* index j.

```
Algorithm 2: Parallel Merge
  Input   : Sorted arrays $A, B$, where $|A| = n$ and $|B| = m$
  Output: Merged and Sorted array $C$ of length $n + m$
1 if $n <= 1$ and $m <= 1$ then return $A$ and $B$ in sorted order ;
2 if $m \mod 2 == 1$ then $j \leftarrow (m-1)/2$ ;
3 else                      $j \leftarrow m/2$ ;
4 $i \leftarrow$ max index of corresponding element in $A$ such that $A[0:i] \preceq B[j:m]$
  /* In Parallel, DO:                                              */
5     $a \leftarrow$ Merge(A[0:i], B[0:j])
6     $b \leftarrow$ Merge(A[i:n], B[j:m])
  /* END Parallel                                                  */
7 return concatenate(a,b)
```

Notice that when j defined as above in our algorithm,

$$\max\{\texttt{a[0:i]}, \texttt{b[0:j]}\} \leq \min\{\texttt{a[i:n]}, \texttt{b[j:m]}\}$$

and that `a` and `b` are sorted. This is why after our recursive calls return `a` and `b`, we *claim* that we may simply concatenate the result and maintain our sort-guarantee. Further, we also *claim* that this algorithm has work $O(n)$ and depth $O(\log^2 n)$.

In the recursion tree at depth d, there are 2^d calls made to the `merge` procedure, denoted by `Merge(`$A_{d,i}$`,` $B_{d,i}$`)` for $i = 1, 2, \ldots, 2^d$. Each of the $B_{d,i}$ are of size *exactly* $m/2^d$, and note that the size of $A_{d,i}$'s are such that $\sum_{i=1}^{2^d} |A_{d,i}| = n$.

Searching for the element in A such that $A[0:i] \preceq B[0:n/2]$ using a binary search on our sorted array A requires $O(\log n)$ work on a single processor. Therefore, we see that total work for taking two sorted arrays of size n is given by

$$W(n,n) = \sum_{d=1}^{\log_2 n} \left[\sum_{i=1}^{2^d} \log |A_{d,i}| + c \right] \qquad \text{for some constant } c$$

But note that since $\log x$ concave, so by Jensen's Inequality (See equation 4). This leads to the Arithmetic-Geometric Mean Inequality, with the consequence that for a set of n inputs

$$\frac{\sum_{i=1}^{n} \log x_i}{n} \leq \log\left(\frac{\sum_{i=1}^{n} x_i}{n}\right).$$

From our observations above regarding the size of $A_{d,i}$'s, and since $2^d \geq 1$ for all $d \in \mathbb{Z}^+$, we see that

$$\frac{\sum_{i=1}^{2^d} \log(|A_{d,i}|)}{2^d} \leq \log\left(\frac{n}{2^d}\right) \implies \sum_{i=1}^{2^d} \log(|A_{d,i}|) \leq 2^d \log\left(\frac{n}{2^d}\right)$$

Hence we see that

$$\begin{aligned} W(n,n) &\leq \sum_{d=1}^{\log n} \left(2^d \log\left(\frac{n}{2^d}\right) + c\right) \\ &\leq \sum_{d=1}^{\log n} 2^d(\log n - d) + c\log n = \log n \sum_{d=1}^{\log n} 2^d - \sum_{d=1}^{\log n} 2^d d + c\log n \\ &= (2(n-1))\log n - 2(n\log n - n + 1) + c\log n \\ &= O(n). \end{aligned}$$

With regard to the *depth* of our algorithm, note that each recursion level has depth $O(\log n)$, due to our binary-search bottleneck; note as well that the recursion stops whenever the elements of $A_{d,i}$ become smaller than 2, which happens by recursion level $O(1 + \lceil \log_2 n \rceil)$. Hence we see that depth is $O(\log^2 n)$.

Remark on Concatenate You may have wondered why we assume that `concatenate` takes constant time. Realize that if we were to naively appending the elements of one array to another, this would require $O(n)$ work, since we must copy or move each element from one address in memory to another. Notice, however, that each element may be moved independent of other elements, hence depth is $O(1)$.

But we can do much better thank this. We can bring work down to $O(1)$ while still maintaining unit depth. There are two ways to do this. The first way is to manage our memory directly so that the two input arrays are placed contiguously in our random access memory. The second is to simply use an `if` statement whenever accessing elements in our output array. This `if` statement costs unit work, and hence we can still maintain our guarantee of constant time access to any element in the output array.

Alternative Solution There is a way to find the median of the union of two sorted arrays in $\log n$ time on one-machine. Hence this sub-routine has $\log n$ work (and depth, as written). It can be proven that this can be done. After which, we can see that

$$W(n) = 2W\left(\frac{n}{2}\right) + O(\log n),$$
$$D(n) = D(n/2) + O(\log n).$$

Using the Master Theorem, the results show $W(n) = O(n)$ and $D(n) = O(\log^2 n)$.

5. Given the price of a stock at each day for n days, we want to determine the biggest profit we can make by buying one day and selling on a later day. For example, the following stock prices have a best profit of 5:

   $[12, 11, 10, 8, 5, 8, 9, 6, 7, 7, 10, 7, 4, 2]$

   since we can buy at 5 on day 5 and sell at 10 on day 11. This has a simple linear time serial solution. Give an algorithm to solve this problem that runs in $O(n)$ work and $O(\log n)$ depth. Give pseudocode as in the previous problem.

   Solution We will use a function called `min_scan`, which takes as input an array of numbers length n (call it A) and outputs an array of length n; in each index i of the output is the minimum number of $A[0 : i]$, i.e. the minimum number up to that point in the original array. We may implement this function as we do other scan functions, including all-prefix sum, to have $O(n)$ work and $O(\log n)$ depth.

   Our entire algorithm for the stock-market problem can be described as follows.

```
Algorithm 3: Buy Low Sell High
  Input  : an array $X$ of stock prices containing $n$ elements
  Output: The max-value we could achieve (non-negative)
1 mins ← min_scan($X$)      // done in parallel
2 max_gains ← X − mins     // element-wise subtraction, in parallel
3 max_val ← max{max_gains}
4 if max_val > 0 then return max_val ;
5 else return 0;
```

   Our algorithm does not involve any explicit recursive calls. Element-wise subtraction of n elements involves n independent computations, hence has work $O(n)$ and depth $O(1)$. Computing the maximal value of an n-array can be done with $O(n)$ work and $O(\log n)$ depth as seen in the following problem. The last step of our algorithm performs a constant time check before returning the output.

6. In this problem, we will look at how fast the maximum of a set of n elements can be computed when allowing for concurrent writes. In particular we allow the arbitrary write rule for "combining" (i.e. if there are a set of parallel writes to a location, one of them wins).

   Perhaps surprisingly, this can be done in $O(\log \log n)$ depth and $O(n)$ work.

(a.) Describe an algorithm for maximum that takes $O(n^2)$ work and $O(1)$ depth (using concurrent writes).

Solution For each element x_i, $1 \leq i \leq n$, we associate a bit initialized to have unit value. Notice that there are n bits to be initialized, hence work is $O(n)$ and depth is $O(1)$ since bits may be initialized independently.

For each pair of elements $x_i, x_j, i \leq j$, we make a comparison in parallel. Notice that the work is $\binom{n}{2} = \frac{n(n-1)}{2} = O(n^2)$, and the depth is 1, since each of the $\binom{n}{2}$ comparisons may be computed independently. We use $p = \binom{n}{2}$ processors, and for each comparison we attempt to *over-write* $b_i = 0$ if $x_i < x_j$ and $b_j = 0$ if $x_j < x_i$, i.e. if we can definitively say that an element is smaller than some other element in our input, its associated bit gets set to 0. This 0 encodes that the element is *not* a maximum.

Notice that we may end up with two processors writing to the same location in memory at the same time. However, notice that in our algorithm, we only at tempt to overwrite a bit if we turn it off. Hence we may allow arbitrary writes in the event of conflict, since all of the writes are trying to accomplish the same thing.

Notice that all bits whose associated value is 1 at the end of this process must have the same value, for if not we get a contradiction: fix attention to two such values; notice that since they have different values, they cannot be the same element, hence $i \neq j$, and thus at some-point in our algorithm they were considered. But if the element had different values, exactly one of them should have been turned off. Once a bit turns off, it never turns on again. Since we assume both associated bits have value 1, this is a contradiction.

Now, we need to *return* our result. Notice that if we were to naively loop through our bit sequence looking for a bit which is turned on, that this would take $O(n)$ time since there is no guarantee where the maximal element lies in the array. Instead, assign each of our n bits to a particular processor. In constant time, check whether the bit turned on. If it is, fetch the corresponding entry from the array in unit time and write it to output address in unit time. Although we do not have unit depth, we have a constant depth in our DAG which is *not* a function of n. Hence $T_\infty = O(1)$ for this algorithm. Notice that by our argument in the previous paragraph, any writes which are concurrently attempted to output are all trying to write the same value, so again the arbitrary write rule causes no harm.

(b.) Use this to develop an algorithm with $O(n)$ work and $O(\log \log n)$ depth. Hint: use divide and conquer, but with a branching factor greater than 2.

Solution We use a Divide-And-Conquer algorithm with a branching factor of $n^{1/3}$. That is, we divide the array into $n^{1/3}$ blocks each of size $n^{2/3}$ elements, and recursively find their maximum. Notice that the recursion bottoms-out when $n \leq 2$. From the max elements of each of the $n^{1/3}$ blocks, compute the max using our brute-force parallel algorithm described in part (A), requiring $O(n^2)$ work. Then, we see that,

$$W(n) = n^{1/3}\,W(n^{2/3}) + O\left((n^{1/3})^2\right).$$

For depth, notice that the recursive calls from divide-and-conquer may be made in parallel, and the base-case (where we apply algorithm from part (a)) only requires $O(1)$ depth. So,

$$D(n) = D(n^{1/3}) + O(1).$$

Notice that we can *not* use the Master Theorem because in each recursive call we divide our input by $\sqrt[3]{n}$.

Solving recurrence relation for depth We first solve the recurrence relation for depth. We see that we have a constant amount of depth for each level of recursion, so we just need to solve for the number of recursion levels needed. We know $D(2) = O(1)$ (the recursion bottoms out when $n \le 2$), so we use this fact to solve for the number of levels.

$$n^{1/3^k} = 2 \iff \frac{1}{3^k}\log(n) = \log(2) \iff \frac{\log(n)}{\log(2)} = \log_2(n) = 3^k \implies k = \log_3\log_2(n)$$

We have constant depth for each level, so in total depth is $O(\log_3\log_2(n)) = O(\log\log(n))$.

Solving recurrence relation for work Now we solve the work recurrence relation. We may use a similar technique for finding the number of levels in the recursion tree, again using the fact that $W(2) = O(1)$, and we get $k = \log_{3/2}\log_2(n)$ as the number of levels. However, we do not have a constant amount of work at each level, so we need to *un-roll* the relation some.

$$\begin{aligned}
W(n) &= n^{1/3}W(n^{2/3}) + O(n^{(1/3)^2})\\
&= n^{1/3}\left[(n^{2/3})^{1/3}\,W\left(n^{(2/3)^2}\right) + O\left(n^{(2/3)^2}\right)\right] + O(n^{(1/3)^2})\\
&= n^{1/3}\left[(n^{2/3})^{1/3}\left[(n^{(2/3)^2})^{1/3}\,W(n^{(2/3)^3}) + O(n^{(2/3)^3})\right] + O(n^{(2/3)^2})\right] + O(n^{(1/3)^2})\\
&= \underbrace{O(n^{2/3}) + n^{1/3}O(n^{(2/3)^2}) + n^{1/3}(n^{2/3})^{1/3}O(n^{(2/3)^3}) + n^{1/3}(n^{2/3})^{1/3}(n^{(2/3)^2})^{1/3}O(n^{(2/3)^4}) + \dots}_{\log_{3/2}\log_2(n)\text{ terms}}
\end{aligned}$$

We now have a series where each summand a product of terms.[7] We recognize a pattern, and combine terms

$$O(n^{(2/3)^{1}}) + n^{1/3}O(n^{(2/3)^{2}}) + n^{1/3}(n^{(2/3)^{1}})^{1/3}O(n^{(2/3)^{3}}) + n^{1/3}(n^{2/3})^{1/3}(n^{(2/3)^{2}})^{1/3}O(n^{(2/3)^{4}}) + \dots$$
$$= \sum_{j=1}^{\log_{3/2}\log_2(n)} \left(\prod_{i=0}^{j-2}(n^{(2/3)^{i}})^{1/3}\right) O(n^{(2/3)^{j}})$$

Now, we simplify algebra.[8]

$$\left(n^{(2/3)^i}\right)^{1/3} = n^{\frac{1}{3}\left(\frac{2}{3}\right)^i} \implies \prod_{i=0}^{j-2}\left(n^{(2/3)^i}\right)^{1/3} = n^{\frac{1}{3}\sum_{i=0}^{j-2}\left(\frac{2}{3}\right)^i} = n^{\frac{1}{3}\frac{1-(\frac{2}{3})^{j-1}}{1-\frac{2}{3}}} = n^{1-(\frac{2}{3})^{j-1}}$$

So,

$$\begin{aligned}
W(n) &= \sum_{j=1}^{\log_{3/2}\log_2(n)} \underbrace{\left(\prod_{i=0}^{j-2}(n^{(2/3)^i})^{1/3}\right)}_{n^{1-(\frac{2}{3})^{j-1}}} O(n^{(2/3)^j}) = \sum_{j=1}^{\log_{3/2}\log_2(n)} O\left(n^{1-(\frac{2}{3})^{j-1}+(\frac{2}{3})^j}\right) \\
&= \sum_{j=1}^{\log_{3/2}\log_2(n)} O\left(n^{1-(\frac{2}{3})^{j-1}\left(1-\frac{2}{3}\right)}\right) \\
&= \sum_{j=1}^{\log_{3/2}\log_2(n)} O\left(n^{\frac{3^j-2^{j-1}}{3^j}}\right) \\
&= \sum_{j=1}^{\log_{3/2}\log_2(n)} O\left(n^{1-(2/3)^j/2}\right) \\
&= \sum_{j=1}^{\log_{3/2}\log_2(n)} \frac{O(n)}{O(n^{(2/3)^j/2})} \\
&= O(n)
\end{aligned}$$

We claim that the last equality holds. To see this, for given n, let $k = \log_{3/2}\log_2(n)$.

[7]We caveat our notation: $n^{(2/3)^i} = n^{2^i/3^i} \neq (n^{2/3})^i = n^{2i/3}$.

[8]Recall for any geometric series with $x \in \mathbb{R}$, that $\sum_{i=0}^{k} x^i = \frac{1-x^{k+1}}{1-x}$.

Let $\delta = (2/3)^{(k-1)/2} > 0$. We may re-write our equation for work as

$$
\begin{aligned}
W(n) &= \sum_{j=1}^{\log_{3/2}\log_2(n)} \frac{O(n)}{O(n^{(2/3)^j/2})} \\
&= \frac{O(n)}{O(n^{(2/3)^k/2})} + \sum_{i=1}^{k-1} \frac{O(n)}{O(n^{(2/3)^i/2})} \\
&= \frac{O(n)}{O((n^{(2/3)^k})^{1/2})} + \sum_{i=1}^{k-1} \frac{O(n)}{O(n^{(2/3)^i/2})} \\
&= \frac{O(n)}{O((2)^{1/2})} + \sum_{i=1}^{k-1} \frac{O(n)}{O(n^{(2/3)^i/2})} \\
&\le O(n) + (k-1)\frac{O(n)}{O(n^{(2/3)^{(k-1)/2}})} \\
&= O(n^{1-k/3}) + O\left((\log\log n)n^{1-\delta}\right) \\
&= O(n)
\end{aligned}
$$

7. In this set of problems, we review how to *select* data from relational databases.

(a.) Write a SQL statement to find the total purchase amount of all orders. Sample table: `orders`.

ord_no	purch_amt	ord_date	customer_id	salesman_id
70001	150.5	2012-10-05	3005	5002
70009	270.65	2012-09-10	3001	5005
70002	65.26	2012-10-05	3002	5001
70004	110.5	2012-08-17	3009	5003
70007	948.5	2012-09-10	3005	5002

Solution

```
SELECT SUM (purch_amt)
  FROM orders;
```

(b.) Write a SQL statement which selects the highest grade for each of the cities of the customers. Sample table: `customer`.

customer_id	cust_name	city	grade	salesman_id
3002	Nick Rimando	New York	100	5001
3005	Graham Zusi	California	200	5002
3001	Brad Guzan	London		5005
3004	Fabian Johns	Paris	300	5006
3007	Brad Davis	New York	200	5001

Solution

```
 SELECT MAX (grade)
  FROM customer
  GROUP BY city;
```

(c.) Write a SQL statement to find the highest purchase amount on a date "2012-08-17" for each salesman with their ID. Sample table: `orders`, used in (a).

Solution

```
SELECT salesman_id, MAX(purch_amt)
  FROM orders
  WHERE ord_date = "2012-08-17"
  GROUP BY salesman_id;
```

8. In this problem, we review how to *merge* two tables together.

(a.) Write a SQL statement to know which salesman are working for which customer. Use the sample table `customer`, used in previous problem, and also `salesman`.

```
salesman_id  name        city        commission
-----------  ----------  ----------  ----------
5001         James Hoog  New York    0.15
5002         Nail Knite  Paris       0.13
5005         Pit Alex    London      0.11
5006         Mc Lyon     Paris       0.14
5003         Lauson Hen              0.12
```

Solution

```
 SELECT c.cust_name, c.city, s.name, s.commission
  FROM customer c
  INNER JOIN salesman s
  ON c.salesman_id = s.salesman_id;
```

(b.) Write a query to display all salesmen and customers located in London.

Solution

```
SELECT salesman_id, name
  FROM salesman
  WHERE city = 'London'
  UNION
  SELECT customer_id, customer_name
  FROM customer
  WHERE city = 'London';
```

(c.) Write a SQL statement to make a cartesian product between `salesman` and `customer` i.e. each salesman will appear for all customer and vice versa for those salesmen who belongs to a city and the customers who must have a grade.

Solution

```
SELECT *
  FROM salesman
  CROSS JOIN customer
  WHERE salesman.city IS NOT NULL;
```

(d.) Write a SQL statement to make a report with customer name, city, order number, order date, and order amount in ascending order according to the order date to find that either any of the existing customers have placed no order or placed one or more orders. Use `customer` and `orders` tables.

Solution

```
SELECT c.cust_name, c.city, order.ord_no, order.ord_date,
       order.purch_amt
  FROM customer c
  LEFT OUTER JOIN order
  ON c.customer_id = order.customer_id
  ORDER BY order.ord_date;
```

9. In this problem, we consider *aggregation* of data.

(a.) Write a SQL statement to find the highest purchase amount ordered by the each customer on a particular date with their ID, order date and highest purchase amount. Sample table: `orders`, used in problem 6 part (a).

Solution

```
SELECT customer_id, ord_date, MAX(purch_amt)
  FROM orders
  GROUP BY customer_id, ord_date;
```

(b.) Write a SQL query to display the average price of each company's products, along with their code. Sample table: `item_mast`.

```
PROD_ID   PROD_NAME     PROD_PRICE    PROD_COMPANY
-------   ---------    -----------    ------------
101      Mother Board     3200          15
102      Key Board         450          16
103      ZIP drive         250          14
104      Speaker           550          16
105      Monitor          5000          11
106      DVD drive         900          12
```

Solution

```
SELECT AVG(prod_price), prod_company
  FROM item_mast
  GROUP BY prod_company
```

10. **Joins with multiple keys** The point of this question is to explore how SQL handles cases where a join is performed on tables containing duplicate rows. Consider the following table `item_mast`.

PROD_ID	PRODUCT	PROD_PRICE	PROD_COMPANY
101	Mother Board	3200	1
101	Mother Board	2900	999
103	ZIP drive	250	14
106	DVD drive	900	12

and a corresponding table of customer purchases, `purchases`.

PROD_ID	CUSTOMER	PRODUCT	city
101	James Hoog	Mother Board	New York
101	James Hoog	ZIP drive	Los Angeles
103	Mc Lyon	ZIP drive	Pittsburgh

Notice that in `item_mast`, the same product can appear multiple times (listed under different manufacturers). Also, in database `purchases` the same customer can appear multiple times. If we `join` carefully using select columns, we can identify observations uniquely in the resulting output table. However, suppose we join the two tables only on `item_mast: product, product price` and `purchases: customer, product`.

Draw a sample table describing what the output looks like, and explain the result.

Solution Suppose we execute the following command:

```
SELECT item_mast.PRODUCT, item_mast.PROD_PRICE, purchases.CUSTOMER
FROM item_mast
INNER JOIN purchases
ON item_mast.PRODUCT=purchases.PRODUCT;
```

then we get the following table as output:

```
PRODUCT        PROD_PRICE     CUSTOMER
-------        ----------     ---------
Mother Board     3200        James Hoog
Mother Board     2900        James Hoog
ZIP drive         250        James Hoog
ZIP drive         250        Mc Lyon
```

We can see that in the table `item_mast`, there are 2 products named 'Mother Board' from different companies, and they both are shown in this output table. They have the same product ID 101, and since the table `purchases` doesn't mention the company name, it is correct to show both of these in the output table.

The product 'ZIP drive' has ID 103 in the table `item_mast`, and 2 entries with ID 101 and 103 in the table `purchases`. When we join the tables, only the entry with

ID 103 should be output. But we see that both the entries have been output. So, this is incorrect.

To give the correct output, we need to join on both the prod_id and product columns.

```
SELECT item_mast.PRODUCT, item_mast.PROD_PRICE, purchases.CUSTOMER
FROM item_mast
INNER JOIN purchases
ON item_mast.PRODUCT=purchases.PRODUCT
AND item_mast.PROD_ID=purchases.PROD_ID;
```

then we get the following table as output:

```
PRODUCT        PROD_PRICE     CUSTOMER
-------        ----------     ---------
Mother Board     3200        James Hoog
Mother Board     2900        James Hoog
ZIP drive         250        Mc Lyon
```