**COMPSCI 330: Design and Analysis of Algorithms** 9/2/2014

Lecture #3

*Lecturer: Debmalya Panigrahi* *Scribe: Nat Kell*

# 1 Overview

In this lecture, we will further examine divide-and-conquer algorithms. First we will define and analyze an $O(n)$-time algorithm for the problem of *selection:* given an $n$ element array $A$ and value $k$, return the $k$th smallest element in the array. Next, we examine a DAC algorithm for multiplying two $n \times n$ matrices due to Strassen. By cleverly reducing the number of subproblems we need to perform in each recursive call, this algorithm improves upon the naive $O(n^3)$-time approach.

# 2 Median Finding and Selection

Suppose we are given an $n$ element array $A$ and we wish to find its median. An immediate algorithm to this problem would be to first sort $A$ using your favorite $O(n \log n)$-time sorting algorithm. Then, we can look at $(n/2)$th element in this sorted array to find the median. However, this approach seems to being too much work—the added time it takes to order all the elements is unnecessary since we're only interested in finding the median. Ideally, we could find the median in $O(n)$ time (an algorithm that solves this problem must run in $\Omega(n)$ time; can you argue why?)

Instead of finding the median of $A$, we will actually solve the problem of *selection* instead, which is a slightly more general problem. Specifically, the algorithm is now given a parameter $k$ as input and is asked to find the $k$th smallest item in $A$ (i.e., the element we would find at index $k$ if $A$ was sorted). Clearly, solving selection solves the problem of median finding if we set $k = n/2$. This more general structure will allow us to correctly define our conquer step when devising our divide-and-conquer algorithm.

## 2.1 A DAC Algorithm for Selection

Our goal is to define an algorithm that finds the $k$th smallest element of an $n$ element $A$ in $O(n)$ time. We begin by defining the following correct (but as we'll see) inefficient algorithm SELECTION$(A,k)$:

1. Select a pivot value $p \in A$ (for now, we do this arbitrarily). Then, restructure $A$ so that it is the concatenation of the following three subarrays: all elements less than $p$, followed by elements equal to $p$, and then followed by elements greater than $p$; call these regions $A_1$, $A_2$, and $A_3$, respectively. We'll also denote that $|A_i| = n_i$.

2. If $k \leq n_1$, we return SELECTION$(A_1,k)$. Since $A_1$ contains the $n_1$ smallest elements in $A$ and $k \leq n_1$, we know that the $k$th smallest element in $A_1$ is also the $k$th smallest element in $A$. Thus making a recursive call on $A_1$ that leaves the value of $k$ unchanged will correctly return the desired element.

3. If $n_1 < k \leq n_1 + n_2$, simply return $p$. Here we know that the $k$th smallest element lies in $A_2$. Since this subarray only contains the value $p$, the desired entry must be the pivot $p$.

4. Otherwise, $k > n_1 + n_2$, in which case we return SELECTION$(A_3, k - n_1 - n_2)$. Now we know our desired element is in subarray $A_3$; however, because this array is preceded by $A_1$ and $A_2$, we now must locate the $(k - n_1 - n_2)$th smallest element in $A_3$ if we want to find the $k$th smallest element in the overall array $A$ (this just accounts for the preceding elements that we are essentially "chopping-off" when we make a recursive call that only examines $A_3$).

This completes the algorithm's definition (note that here, step 1 defines the divide step, and steps 2-4 define our conquer step; there is no combine procedure in this case). It should now be clear that this algorithm is indeed correct; however, if we are not careful as to how we pick our pivot $p$, the running time of the algorithm will suffer in the same way we saw quick-sort suffer in the last lecture.

More specifically, let $T(n)$ be the running-time of the algorithm on an $n$ element array. Recall that step 1 (i.e., picking a pivot and restructuring the array) can be done in $O(n)$ time. The running time of steps 2-4 will depend on what case we fall in, but regardless, we can say it is bounded by $T(\max\{n_1, n_3\})$ since we only ever make recursive calls on $A_1$ or $A_3$ (but not both). If we get an even split each time we make a recursive call and $n_1 \approx n_3 \approx n/2$, we will be in good shape. Formally, the running time of the algorithm in this case is given by:

$$\begin{aligned} T(n) &= T(n/2) + O(n) \\ &\leq c \cdot (n + n/2 + n/4 + \ldots + 1) && (1) \\ &= cn \cdot (1 + 1/2 + 1/4 + \ldots + 1/n) && (2) \\ &< cn \cdot 2 = O(n), && (3) \end{aligned}$$

where $c$ is the constant hidden by the $O(n)$ term in the original recurrence. It should be fairly straightforward to see why if we expand the recurrence, we obtain line (1). To see why inequality (3) is true, we quickly review geometric series. Recall (hopefully) that for some real number $0 \leq r < 1$, we have that

$$1 + r + r^2 + r^3 + \ldots = \sum_{i=0}^{\infty} r^i = \frac{1}{1-r}. \tag{4}$$

For line (2), we have that $r = 1/2$ when examining the term $(1 + 1/2 + 1/4 + \ldots + 1/n)$. Since this sum is bounded by $\sum_{i=0}^{\infty}(1/2)^i$ (the latter is an infinite sum whose terms subsumes the finite series in the former), the closed form for an infinite geometric series given by (4) implies that these terms are bounded by $1/(1 - 1/2) = 2$.

Turning our attention back to SELECTION, the case where $A_1$ and $A_3$ are roughly equal in size will result in this ideal running time; however, since we currently have no rule for picking a pivot, there is nothing preventing, say, $n_1 = 0$ and $n_3 = n - 1$. If this worst case occurs for every recursive call we make, the running time is now:

$$\begin{aligned} T(n) &= T(n-1) + O(n) \\ &= c \cdot (n + n - 1 + n - 2 + \ldots + 1) = O(n^2). \end{aligned}$$

Thus, as currently defined, the running time of SELECTION$(A, k)$ is $O(n^2)$, which is worse than the $O(n \log n)$ naive sorting approach we first gave. Clearly, if we want to get any pay dirt out of this algorithm, we'll have to find a way of picking the pivot so that we maintain at least some balance between the sizes of $A_1$ and $A_3$.

## 2.2 Picking a Good Pivot

As outlined above, we would ideally like to pick a pivot so that $n_1 \approx n_3 \approx n/2$. The best pivot to pick then is just the median of $A$; however, this is a bit circular. Selection is a problem that, at least when $k = n/2$, is trying to find the median in the first place! Therefore, the somewhat arcane pivot-finding procedure we are about to define attempts to find an "approximate" median in $O(n)$ time. If there is at least some balance between $A_1$ and $A_3$, hopefully we can discard a large enough fraction of the array in each recursive call to yield our desired $O(n)$ runtime.

Our new pivot procedure, which we'll call PIVOT-FIND$(A)$, is defined as follows (where again $|A| = n$):

1. Divide $A$ into $n/5$ blocks $B_1, \ldots, B_{n/5}$ each of size 5 (assume for sake simplicity that $n$ is divisible by 5).

2. Sort each of these blocks individually, and so afterwards, the 5 elements in a given block $B_i$ are sorted with respect to one another. Note that the median of each block is now the third element within that block.

3. For each block $B_i$, store $B_i$'s median in a new array $C$ of size $n/5$.

4. We then return our pivot to be SELECTION$(C, n/10)$, which is the median of $C$ ($n/10$ comes from the fact there are $n/5$ elements in $C$, and therefore the midway point in this array will be at $n/10$).

The most interesting observation to make about this procedure is the fact we are using a recursive call to SELECTION as our means of picking the pivot. Thus, we are counterintuitively using the very divide-and-conquer algorithm we are attempting to define as a subroutine when defining our divide step (picking a pivot). So for example, consider our first recursive call to SELECTION$(A,k)$. By the time PIVOT-FIND$(A)$ completes, we will have made several cascading calls to SELECTION and PIVOT-FIND on smaller arrays before even reaching the first recursive conquer steps in our top recursive call (steps 2-4 in SELECTION). By no means is this a typical approach when defining divide-and-conquer algorithms, but it is a great example of how one can take an algorithmic paradigm and creatively dovetail standard techniques in order to construct a faster algorithm.

Also note that this procedure indeed runs in $O(n)$ time. Step 1-2 will take $O(n)$ time since there are $n/5$ blocks, each of which take $O(1)$ time to sort since they each have a constant number of elements (even if we use bubble-sort or insertion-sort). Steps 3-4 will take $O(n)$ time, as well, since we can step through all the blocks in $n/5$ steps, pluck out each median at a given block's third location, and then add it to $C$; hence, the procedure's overall running time is in $O(n)$.

Now, let's see what this new PIVOT-FIND$(A)$ buys us. To do our analysis, consider reordering the blocks so that they are sorted by their medians. More formally, we specify this ordering as $B_{\phi(1)}, B_{\phi(2)}, \ldots, B_{\phi(n/5)}$, where $\phi$ is a function that remaps the block indices such that if $m_i$ is the median of block $B_i$, we have that $m_{\phi(i)} \geq m_{\phi(j)}$ for all $j < i$ and $m_{\phi(i)} \leq m_{\phi(j)}$ for all $j >$ i. It is important to note that *the algorithm is not actually performing this reordering*—we are just defining this structure for sake of analysis.

Observe that the pivot $p$ returned by SELECTION$(C, n/10)$ is the median of block $B_{\phi(n/10)}$. Now, define the following sets:

- Let $S_1$ be the set of all $x \in A$ such that $x \in B_{\phi(j)}$, $x \leq m_{\phi(j)}$, and $j \leq n/10$ (recall that we defined $m_i$ to be the median of block $B_i$). Informally, these are elements that exist in the first half of this block ordering and are less or equal to the median of the block to which they belong.

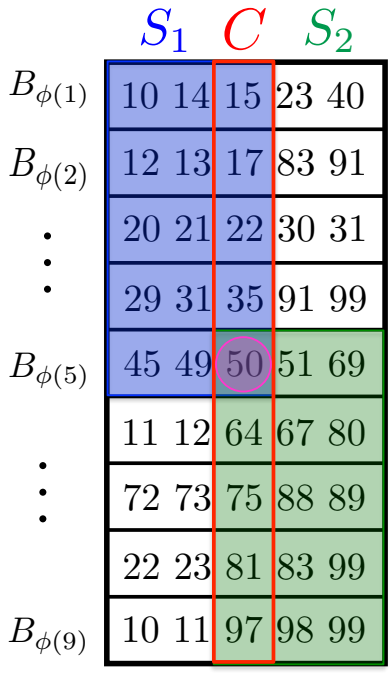


Figure 1: A diagram illustrating the definitions of $B_{\phi(1)},\ldots,B_{\phi(n/5)}$, $C$, $S_1$, and $S_2$. Here $n = 45$, and thus we have 9 blocks in total. Observe that each block is sorted individually and that the blocks themselves are ordered based on their medians. Array $C$ is outlined in red, and the median of $C$, circled in pink, would serve as our pivot $p$. Sets $S_1$ and $S_2$ are highlighted by the blue and green boxes, respectively. Also observe that all the elements in $S_1$ and $S_2$ are less than and greater than the pivot, respectively (which is argued formally in Lemma 1).

- Similarly, let $S_2$ be the set of $y \in A$ such that $y \in B_{\phi(j)}$, $y \geq m_{\phi(j)}$, and $j \geq n/10$. Likewise, this is the set of elements that exist in the second half of our block ordering and are no smaller than the medians of their respective blocks.

Figure 1 illustrates the definitions of these sets. So what is all this structure good for? Remember our goal is to make sure that both $A_1$ and $A_3$ receive a large enough fraction of the elements. The following lemma makes use of our block ordering $B_{\phi(1)},\ldots,B_{\phi(n/5)}$ and our newly defined sets $S_1$ and $S_2$ to argue that if we use PIVOT-FIND to find our pivot, we indeed obtain some balance.

**Lemma 1.** *If we use* PIVOT-FIND *to pick the pivot $p$, then* $\max\{n_1,n_3\} \leq 7n/10$.

*Proof.* Our key observations are that $S_1 \subseteq A_1 \cup A_2$ and $S_2 \subseteq A_2 \cup A_3$. Here, we will argue that the former claim is true and show it implies that $n_3 \leq 7n/10$. We'll leave establishing $S_2 \subseteq A_2 \cup A_3$ and showing that this implies $n_1 \leq 7n/10$ as an exercise (although, it should be symmetric to the argument we present here).

Let $x \in S_1$. Therefore, $x$ exists in some block $B_{\phi(j)}$ where $j \leq n/10$ *and* is less than or equal to the median of its block $m_{\phi(j)}$. Since $j \leq n/10$, $m_{\phi(j)}$ must lie in the first half of $C$ and therefore is no greater than the median of $C$. Since PIVOT-FIND ensures that $p$ is the median of $C$, we have that $m_{\phi(j)} \leq p$. Thus, it follows that $x \leq p$, implying that $x \in A_1 \cup A_2$, as desired.

To complete the proof, observe that $S_1$ contains $3/5$ of the elements in blocks $B_{\phi(1)},\ldots,B_{\phi(n/10)}$, since for each of these blocks, $S_1$ includes the median and the two preceding elements. Since these blocks account

for half the elements in entire array $A$, it follows $|S_1| = 3n/10$. However, we just showed $S_1$ is a subset of $A_1 \cup A_2$, implying that $A_1 \cup A_2$ cannot be smaller than $3n/10$. This implies that $|A_3| = n_3$ can be no larger than $7n/10$ since $n_1 + n_2 + n_3 = n$.

As mentioned at the start of the proof, a symmetric argument can be made to show $n_1 \leq 7n/10$. Hence, we have that $\max\{n_1, n_3\} \leq 7n/10$. □

We are now ready to complete our run-time analysis for SELECTION with PIVOT-FIND. Let's briefly recall the running times of all the components in a given recursive call of SELECTION$(A,k)$, where again we denote $T(n)$ to be the total running time of this call if $|A| = n$.

- Steps 1-3 of PIVOT-FIND, where we divide our array into blocks and create the array of medians $C$, takes $O(n)$ time.
- Step 4 of PIVOT-FIND makes a call to SELECTION$(C, n/10)$; since $C$ is an array of size $n/5$, this will take $T(n/5)$ time.
- Restructuring the array around the pivot in steps 1-2 of SELECTION takes $O(n)$ time.
- As we argued earlier, steps 2-4 of SELECTION take $T(\max\{n_1, n_3\})$ time. By Lemma 1, we know that this will be at most $T(7n/10)$.

Thus, the overall running time of the algorithm is as follows (we'll again use $c$ as the constant that is hidden by the $O(n)$ term that bounds the running times of creating array $C$ and pivoting the array around $p$).

$$\begin{aligned} T(n) &= T(7n/10) + T(n/5) + cn \\ &< cn \cdot \sum_{i=0}^{\infty} \left(\frac{9}{10}\right)^i \qquad (5) \\ &= cn \cdot 10 = O(n), \qquad (6) \end{aligned}$$

as desired. You should try to verify inequality (5) (use the tree expansion method; you should get that the total work done on the $k$th level of the tree is $cn \cdot (9/10)^k$). Equation (6) follows by the closed form for a geometric series we saw earlier in equation (4).

A final note: observe that if we had picked the size of each block to be 4 instead of 5, the recurrence would now become $T(n) = T(3n/4) + T(n/4) + O(n)$. When we expand the recursion tree for this recurrence, we are doing $cn$ work at each level of the tree. Since the tree will have $O(\log n)$ levels, we instead get a running time of $O(n \log n)$. Thus, the decision to use blocks of 5 was carefully made when designing the algorithm to avoid getting this extra $O(\log n)$ factor. Picking anything greater than 5 will also work, but we will still get an $O(n)$ time algorithm since doing this will only improve the constant we get on line (6) (and remember, any algorithm for selection must take $\Omega(n)$ time).

# 3 Matrix Multiplication

Next, we'll examine a DAC algorithm for the problem of *matrix multiplication*. Recall that the *dot product* of two vectors $n$-dimensional vectors $u = \langle u_1, \ldots, u_n \rangle$ and $v = \langle v_1, \ldots, v_n \rangle$ is defined as $u \cdot v = u_1 v_1 + u_2 v_2 + \ldots + u_n v_n$ (the sum of the pairwise products). To now define matrix multiplication, let $A$ and $B$ be a $m \times t$

matrix and a $t \times n$ matrix, respectively. We define the matrix product $AB$ as a $m \times n$ matrix $C$ whose $i$th row and $j$th column entry is the dot product of the $i$th row of $A$ with the $j$th column of $B$.

If we want to compute such a matrix $C$, we can do so naively in $O(nmt)$ time. For each of the $mn$ entries in $C$, we just directly compute the dot product for this entry. Since we need to sum $t$ scalar products for each of these dot products, each entry in $C$ takes $O(t)$ time to compute, giving us an overall running time of $O(nmt)$. When both matrices being multiplied are square $n \times n$ matrices, this running time becomes $O(n^3)$. For the rest of the lecture, we will focus on this square-matrix case.

To begin to define a DAC algorithm for this problem, we first need a way of recursively defining matrix multiplication. Observe that we can essentially treat any $n \times n$ matrix like a $2 \times 2$ matrix. Namely, let $A_{ij}$ be the $n/2 \times n/2$ matrix given by the $i$th half of the rows and $j$th half of the columns of a matrix $A$ (so $i, j \in \{1,2\}$). So our situation looks like the following:

$$AB = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = C.$$

One can verify that in order to compute $C$, we can treat these quadrant versions of $A$ and $B$ as $2 \times 2$ matrices and multiply their sub-matrices as if there were just four scalars in each matrix. Namely, we have that

$$\begin{aligned} A_{11}B_{11} + A_{12}B_{21} &= C_{11}, \\ A_{11}B_{12} + A_{12}B_{22} &= C_{12}, \\ A_{21}B_{11} + A_{22}B_{21} &= C_{21}, \\ A_{21}B_{12} + A_{22}B_{22} &= C_{22}. \end{aligned} \tag{7}$$

Since this way of expressing a matrix multiplication is defined in terms of multiplying smaller matrices, this gives us a DAC algorithm. Our divide step splits both matrices $A$ and $B$ into quadrant matrices, the conquer step computes the necessary products of these quadrant matrices (as specified above), and on the combine step we add these products together in order to compute each quadrant of $C$.

To analyze this algorithm's running time, let $T(n)$ be the time required to multiply two $n \times n$ matrices. Observe that adding two $n \times n$ matrices takes $O(n^2)$ time since there are $n^2$ pairwise additions to compute and each scalar addition takes constant time. Since the algorithm divides the problem such that we are doing eight $n/2 \times n/2$ matrix multiplications and four additions, we can express our running time as follows.

$$\begin{aligned} T(n) &= 8T(n/2) + 4\dot{(}n/2)^2 \\ &= 8T(n/2) + n^2 \\ &= \sum_{i=0}^{\log_2 n} 8^i (n/2^i)^2 \qquad (8) \\ &= n^2 \cdot \sum_{i=0}^{\log_2 n} 2^i \\ &= n^2 \cdot \left( \frac{2^{\log_2 n + 1} - 1}{2 - 1} \right) \qquad (9) \\ &\leq 2n^3 = O(n^3). \end{aligned}$$

Again, you should verify that expanding the recurrence yields line (8). In line (9), we are now using the closed form for a *finite* geometric series. In general, if we sum $1 + a + a^2 + \ldots + a^n$ for some $a \geq 0$ and $a \neq 1$, we obtain:

$$\sum_{i=0}^{n} a^i = \frac{a^{n+1}-1}{a-1}. \tag{10}$$

As a brief aside, let's show why this formula is true. Observe that if we multiply the lefthand side of (10) side by $(a-1)$, we get

$$\begin{aligned}(a-1)\cdot(1+a+a^2+\ldots a^n) &= (a+a^2+\ldots+a^{n+1}) - (1+a+a^2+\ldots+a^n) \\ &= -1+(a-a)+(a^2-a^2)+\ldots+(a^n-a^n)+a^{n+1} \\ &= a^{n+1}-1. \end{aligned} \tag{11}$$

Therefore, dividing both the LHS and RHS of (11) by $(a-1)$ yields the formula in (10). You will need to use this finite form anytime your base (in this case $a$) is at least 1. In cases where $a$ is less than one, the formula for infinite geometric series given by (4) should almost always suffice when doing asymptotic analysis (as we saw previously in Section 2).

Turning our attention back to our divide-and-conquer algorithm for matrix multiplication, notice that this algorithm also gives a $O(n^3)$ time bound, which is no better than if we naively compute the product using the triple for-loop approach we mentioned earlier. To improve upon this, we need to reduce the number of new sub-calls we spawn at each recursive call. Notice we can do this at the cost of extra additions, since doing so only changes the constant hidden by the additive $O(n^2)$ term in the recurrence. Here, we give an algorithm due to Strassen, which uses seven multiplications at each step.

Specifically, for each recursive call, we will compute the following seven matrices $M_1, \ldots, M_7$ (note that we are still maintaining the definitions for our quadrants $A_{ij}$ and $B_{ij}$).

$$\begin{aligned} M_1 &= (A_{11}+A_{22})(B_{11}+B_{22}) & M_2 &= (A_{21}+A_{22})B_{11} \\ M_3 &= A_{11}(B_{12}-B_{22}) & M_4 &= A_{22}(B_{21}-B_{11}) \\ M_5 &= (A_{11}+A_{12})B_{22} & M_6 &= (A_{21}-A_{11})(B_{11}+B_{12}) \\ M_7 &= (A_{12}-A_{22})(B_{21}+B_{22}). \end{aligned}$$

One can verify (against (7)) that we have:

$$\begin{aligned} C_{11} &= M_1+M_4-M_5+M_7 \\ C_{12} &= M_3+M_5 \\ C_{21} &= M_2+M_4 \\ C_{22} &= M_1-M_2+M_3+M_6. \end{aligned}$$

It is important to observe that, despite doing several more matrix additions and subtractions, we only do one multiplication for each $M_i$ we compute. Since our new equations for each $C_{ij}$ only require matrix additions and no multiplications, this entire computation only preforms seven matrix multiplications.

Hence, our running time for this algorithm is expressed as follows (where $c$ is the constant hidden by the $O(n^2)$ term that accounts for the matrix additions we do for a given recursive call):

$$\begin{aligned} T(n) &= 7T(n/2) + cn^2 \\ &= cn^2 \cdot \sum_{i=0}^{\log_2 n} \left(\frac{7}{4}\right)^i \\ &= O(n^{\log_2 7}) \approx O(n^{2.81}). \end{aligned}$$

(We'll leave working out the details of this recurrence as an exercise; again, you need to use (10) to obtain the closed-form, and from there, simplify using logarithmic change-of-base).

Thus, this gives us a better asymptotic bound than our previous $O(n^3)$ time algorithms. Not much intuition can be given for how we defined these "magical" matrices $M_1, \ldots, M_7$. Subject to the constraint that we could only do one multiplication in each $M_i$, we just needed to define the $M_i$s to be pieces in a jig-saw puzzle where it was possible to construct each $C_{ij}$ using these seven pieces.

# 4 Overview

In this lecture, we further examined divide-and-conquer algorithms. We first looked at algorithms for the problem of selection, and showed that by carefully picking a pivot in our divide-step, we were able to drastically improve our initial $O(n^2)$-time DAC algorithm to a desired $O(n)$-time algorithm. We then examined the problem of square-matrix multiplication, and showed how we can reduce of number of sub-calls spawned at each recursive call by cleverly defining our conquer step.