0.1 Review (recurrences)

Recurrence: $T(n) = 2T(\lfloor n/2 \rfloor) + n^2$, $T(1) = 1$. This could reflect a situation where we have a recursive algorithm, and it does n^2 units of work at the top level, then calls itself twice on subproblems that are half the size.

Master theorem: First, we solve this using the master theorem. Here, $a = 2$ and $b = 2$ and $f(n) = n^2$. Furthermore, $af(n/b) = 2(n/2)^2 = n^2/2 \leq cf(n)$ for $c = 1/2$. Therefore, case 3 of the master theorem applies, and $T(n) = \Theta(n^2)$.

Recall that case 3 of the master theorem says that

"If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$."

Substitution method: Now, just for practice, we will prove this using the substitution method. In order to do this, we will use strong induction. In strong induction, we first prove that a statement is true for $n = 1$. Then assuming that the statement is true for $n = 1, 2, \ldots, k-1$, we prove the statement is true for $n = k$. Taken together, these two claims show that the statement is true for all n.

Specifically, we will guess that $T(n) \leq cn^2$. The base case holds for $c \geq 1$, since $T(1) = 1 \leq c$.

As our inductive hypothesis, we will assume that $T(k) \leq ck^2$ for all $k < n$. Then

$$\begin{aligned} T(n) &\leq 2c(\lfloor n/2 \rfloor)^2 + n^2 \\ &\leq 2c(n/2)^2 + n^2 \\ &= cn^2/2 + n^2 \\ &= (c/2 + 1)n^2 \end{aligned}$$

Now $(c/2 + 1)n^2 \leq cn^2$ when $c/2 + 1 \leq c$, or when $c \geq 2$. So we can let $c = 2$, and then the inductive step is proven.

Recursion tree: Ignoring floors and ceilings for now, the zeroth level has a single node with cost n^2. The first level has two nodes, each with cost $(n/2)^2 = n^2/4$. The third level has four nodes, with cost $(n/4)^2 = n^2/16$. In general, you can get the cost of a node by taking the subproblem size and squaring it. At the ith level, this cost is $n^2/4^i$. On the other hand, the ith level has 2^i nodes, so the total cost of each level is $n^2/2^i$.

The height of the tree is given by $n/2^i = 1 \to i = \log n$

Summing the costs of each level, we get

$$\sum_{i=0}^{\log n} \frac{n^2}{2^i} = n^2 \sum_{i=0}^{\log n} \frac{1}{2^i} \leq 2n^2$$

by the sum of an infinite geometric series. Therefore, the total cost of the algorithm is $\Theta(n^2)$.

0.2 The selection problem

Problem: Given an array A with n elements, how can we find the ith smallest element of that array?

Exercise: Find a naive upper bound on how long it takes to do this.

Answer: We can get an upper bound of $O(n \log n)$ by sorting the array (using mergesort, heapsort, or another $O(n \log n)$ algorithm), and then choosing the element at the ith index.

Exercise: Find a lower bound.

Answer: Any algorithm must take all n elements into account if we want to guarantee that the algorithm produces the right answer. (Note that some algorithms only guarantee that we get the right answer with high probability, and in that case we don't necessarily have to look at all the elements.) So a lower bound is $\Omega(n)$.

Today we will learn the "median of medians" algorithm, which is an algorithm for solving the selection problem in $O(n)$ time.

1 The Partition procedure

One important subroutine in this algorithm is the Partition procedure, which is described in Section 7.1 of the textbook. The procedure takes as input a slice of an array ($A[p..r]$) plus a pivot element. Then it rearranges the elements in the array slice so that all of the elements less than the pivot element come before the pivot element, and all of the elements greater than the pivot come after the pivot element.

Discrepancies from textbook: The Partition procedure described here is different from the one described in the textbook, in that the one in the textbook just assumes that the pivot element is the last element in the array slice, which makes sense if you are using it for quicksort, but for the median of medians algorithm we use the slightly modified version where you can choose the pivot element (and it's basically the same because we just swap it with the last element at the beginning of the algorithm). Here we assume that the pivot element is inside the array slice.

I also replaced the variables "p" with "leftEdge", "r" with "rightEdge", and "i" with "boundary," because I thought too many one letter variables would make things confusing.

1.1 Description of algorithm

At any point, the array slice is split into three parts: "left," "right," and "unexplored." The "left" part contains elements that are known to be smaller than the pivot. The "right" part is directly to the right of the "left" part, and it contains elements known to be larger than the pivot. To the right of the "right" part is the unexplored region, and at the very right of the array slice is the pivot element.

To populate these regions, we maintain a variable, "boundary", which represents the boundary between the left and the right region. (The variable "j" represents the boundary between the right and the unexplored region.)

On each iteration j, we decide if $A[j]$ is larger or smaller than the pivot. If $A[j]$ is larger than the pivot, we add $A[j]$ to the "right" region by adding 1 to j (this is accomplished through the natural iteration of a for loop).

If $A[j]$ is smaller, then we swap it with the element that is currently at the boundary between the "left" and the "right" region, then add 1 to both "j" and the "boundary" variable. This has the effect of increasing the size of the "left" region by 1, while maintaining the elements in the "right" region.

This figure is an example of Partition in action. Here, p and r are the boundaries of the array slice, and i is the "boundary" variable. The pivot element is currently at $A[r]$, and at the end it is moved into its correct position.

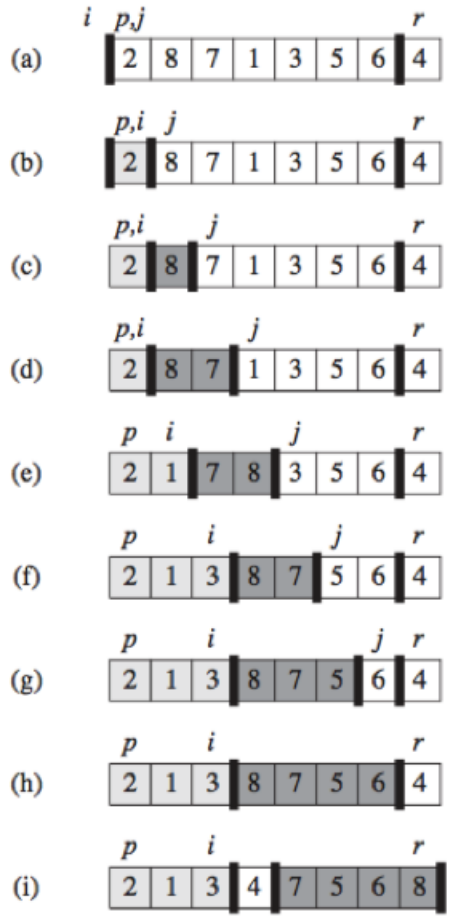

Figure 7.1 The operation of PARTITION on a sample array. Array entry $A[r]$ becomes the pivot element x. Lightly shaded array elements are all in the first partition with values no greater than x. Heavily shaded elements are in the second partition with values greater than x. The unshaded elements have not yet been put in one of the first two partitions, and the final white element is the pivot x. **(a)** The initial array and variable settings. None of the elements have been placed in either of the first two partitions. **(b)** The value 2 is "swapped with itself" and put in the partition of smaller values. **(c)–(d)** The values 8 and 7 are added to the partition of larger values. **(e)** The values 1 and 8 are swapped, and the smaller partition grows. **(f)** The values 3 and 7 are swapped, and the smaller partition grows. **(g)–(h)** The larger partition grows to include 5 and 6, and the loop terminates. **(i)** In lines 7–8, the pivot element is swapped so that it lies between the two partitions.

```
Partition(A, leftEdge, rightEdge, pivot):
    x = A[pivot]
    exchange A[pivot] with A[rightEdge]
    boundary = leftEdge - 1
    for j = leftEdge to rightEdge - 1:
        if A[j] <= x:
            // Move element A[j] into the left side of the partition
            boundary = boundary + 1
            exchange A[boundary] with A[j]
```

```
exchange A[boundary + 1] with A[rightEdge]
return boundary + 1
```

Exercise: Why is the running time of Partition $\Theta(n)$ (for an array slice of length n)?

Answer: The for loop goes through roughly n iterations, and each iteration takes constant time.

1.2 Correctness proof

This will largely be a formalized version of what I said earlier.

Loop invariant: At the beginning of each iteration of the for loop, for any array index k, we have

1. If $leftEdge \leq k \leq boundary$, then $A[k] \leq x$.
2. If $boundary + 1 \leq k \leq j - 1$, then $A[k] > x$.
3. If $k = rightEdge$, then $A[k] = x$.

(This is basically what I was saying about the "left" region and the "right" region.)

Termination: At termination, $j = rightEdge$. Therefore, every entry in the array is in one of the three sets described by the invariant, and we have partitioned the values of the array into three sets: those less than or equal to x, those greater than x, and a singleton set containing x.

Initialization: Prior to the first iteration of the loop, $boundary = leftEdge - 1$ and $j = leftEdge$. The first two conditions are satisfied because there are no values between $leftEdge$ and $boundary$, and no values between $boundary + 1$ and $j - 1$. The swap at the beginning of the algorithm causes the third condition to be satisfied.

Maintenance: We consider two cases, depending on the outcome of the if statement. When $A[j] > x$, the only action in the loop is to increment j. After j is incremented, condition 2 holds for $A[j - 1]$ and all other entries remain unchanged. When $A[j] \leq x$, the loop increments $boundary$, swaps $A[boundary]$ and $A[j]$, and then increments j. Because of the swap, we now have that $A[boundary] \leq x$, and condition 1 is satisfied. Similarly, we also have that $A[j-1] > x$, since the item that was swapped into $A[j-1]$ is, by the loop invariant, greater than x.

The final lines of Partition swap the pivot element with the leftmost element greater than x, thereby moving the pivot into its correct place in the partitioned array, and then returning the pivot's new index.

2 The median of medians algorithm

The Select algorithm is an algorithm for finding the ith smallest element of an array in $O(n)$ time. The idea of the algorithm is to choose a special pivot (the "median of medians"), and use the Partition procedure to put elements that are smaller than the pivot on the left, and elements that are bigger than the pivot on the right. Then we recursively search on either the left side or the right side of the array, depending on how many elements there are in each part of the array.

(If there are more than i elements in the left part of the array, we just have to find the ith element in the left part of the array, and if there are $k - 1 < i$ elements in the left part of the array, then we have to find the $(i - k)$th element in the right part of the array.)

The secret sauce is in the choice of pivot. We choose a pivot that is somewhere in the "middle portion" of the sorted version of the array. As a result, we are guaranteed to throw away a sizable portion of the array on each recursive call, which gives us the linear runtime.

(Note that ideally we would want to choose the median element as the pivot, because then we would throw away half of the array on each call. However, it is kind of hard to find the median, and if we wanted to find the median we would probably want to use the algorithm we are writing right now. Our special pivot takes less time to find, so we are using that instead of the median.)

2.0.1 Precise description of algorithm

Assume that the numbers are distinct.

1. Divide the n elements of the input array into $\lfloor n/5 \rfloor$ groups of 5 elements each (plus one extra group with the leftover elements, if any). This produces $\lceil n/5 \rceil$ groups total.
2. Find the median of each of these groups by insertion-sorting the elements and then picking the median from the sorted list of group elements. (Note that finding the median of a 5 element group takes constant time.)
3. Use Select recursively to find the median x of the $\lceil n/5 \rceil$ medians found in step 2. (If there are an even number of medians, we let x be the smaller of the two.)
4. Use the "Partition" procedure to move all of the elements that are smaller than x to the left of x, and all of the elements that are bigger than x to the right of x. Let k be one plus the number of elements on the low side of the partition, so that x is the kth smallest element and there are $n - k$ elements on the high side of the partition.
5. If $k = i$, then return x. Otherwise, use Select recursively to find the ith smallest element on the low side (if $i < k$), or the $(i - k)$th smallest element on the high side (if $i > k$).

2.1 Runtime analysis

2.1.1 Bounding the subproblem sizes

The first thing we want to do is get a lower bound on the number of elements we are throwing away on each recursive call. This will give us a guarantee on how fast our subproblem sizes are decreasing.

Recall that we have $\lceil n/5 \rceil$ medians. At least half of them are greater than or equal to our pivot x (the median-of-medians). Most of those groups contain two other elements that are greater than their medians, so they contribute at least 3 elements that are greater than x. (The only groups that don't are (1) the group that has fewer than 5 elements, and (2) the group containing x itself.)

This gives us at least

$$3\left(\left\lceil \frac{1}{2}\left\lceil \frac{n}{5}\right\rceil \right\rceil - 2\right) \geq \frac{3n}{10} - 6$$

elements that are greater than x.

By the same logic, at least $3n/10 - 6$ elements are less than x.

So on each recursive call, we throw away at least $3n/10 - 6$ elements, which means step 5 calls Select recursively on at most $7n/10 + 6$ elements.

2.1.2 Finding a recurrence

Steps 1, 2, and 4 take $O(n)$ time. (Step 2 consists of $O(n)$ calls of insertion sort on sets of size $O(1)$.) Step 3 takes time $T(\lceil n/5 \rceil)$, and step 5 takes time at most $T(7n/10 + 6)$, assuming that T is monotonically increasing.

For the base case, we assume that any input of fewer than 140 elements requires $O(1)$ time.

Then the recurrence is

$$T(n) \leq \begin{cases} O(1) & \text{if } n < 140 \\ T(\lceil n/5 \rceil) + T(7n/10 + 6) + O(n) & \text{if } n \geq 140 \end{cases}$$

2.1.3 Solving the recurrence

We use the substitution method to solve the recurrence. Specifically, we will show that $T(n) \leq cn$ for some constant c and all $n > 0$.

For the base case, we choose c large enough that $T(n) \leq cn$ for all $n < 140$. Since 140 is a constant, we can achieve this by declaring c to be a relatively large constant.

Now for the inductive step, we assume $T(k) \leq ck$ for all $k < n$.

Also, let a be the constant from the $O(n)$ term (i.e. a constant such that the function described by the $O(n)$ term is bounded above by an for all $n > 0$).

By the induction hypothesis, we have

$$\begin{aligned} T(n) &\leq c\lceil n/5 \rceil + c(7n/10 + 6) + an \\ &\leq cn/5 + c + 7cn/10 + 6c + an \\ &= 9cn/10 + 7c + an \\ &= cn + (-cn/10 + 7c + an) \end{aligned}$$

which is at most cn if

$$-cn/10 + 7c + an \leq 0.$$

When $n > 70$, this is equivalent to the inequality $c \geq 10a(n/(n-70))$. Because we assume that $n \geq 140$, we have $n/(n-70) \leq 2$, so we can choose $c \geq 20a$ to satisfy the inequality. (In addition, recall that c must be large enough to satisfy the base case.)

Thus, the median of medians algorithm runs in $O(n)$ time.