COMPSCI 330: Design and Analysis of Algorithms 1/19/2016 and 1/21/2016

Divide and Conquer

Lecturer: Debmalya Panigrahi *Scribe: Tianqi Song, Tianyu Wang*

1 Overview

This set of notes is organized as follows. We begin by introducing an $O(n^2)$ sorting algorithm, which motivates the concept of divide and conquer by introducing QuickSort. We then proceed by analyzing three more examples: merge sort, bit-string multiplication, polynomial multiplication and an $O(n)$-time algorithm for the problem of selection. We conclude by introducing the master theorem method for solving recurrence relations. [1]

2 Insertion Sort

We now investigate a simple sorting algorithm.

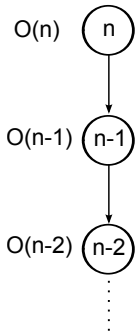
Algorithm 1 Insertion sort

```
1: function INS-SORT(A[1...n], k)
2:     for i ← 1 to n do
3:         INSERT(A[1,...,i−1], i)        ▷ assume INSERT inserts A[i] into sorted position in A[1,...,i−1] and takes O(i) time.
4:     return A[1,...,n]
```

A recurrence relation for Algorithm 1 is

$$T(n) = T(n-1) + O(n).$$

We analyze the running time by computation tree:



We have

[1] Some of the material in this note is from previous notes by Samuel Haney, Yilun Zhou and Nat Kell for this class in Fall 2014.

$$T(n) = O(n) + O(n-1) + O(n-2) + ... + O(1) = O(n^2)$$

Next, we show that we can design a faster sorting algorithm by divide-and-conquer technique.

3 QuickSort

In the following algorithm, A is the underlying array/list; r, l are rightmost (largest), and leftmost (smallest) indices respectively.

3.1 The algorithm

Algorithm 2 QuickSort(A, l, r)

1: $m \leftarrow Pivot(A, l, r)$
2: Swap(A, l, m)
3: QuickSort$(A, l, m-1)$
4: QuickSort$(A, m+1, r)$

The Swap(A, i, j) function just swap the values $A[i]$ and $A[j]$. The Pivot function is described in the following section.

3.2 Picking a pivot point

A pivot point is the point from where we divide the QuickSort into two subproblems. Below is an algorithm for picking a pivot point.

Algorithm 3 Pivot(A, l, r)

1: $i \leftarrow l$, $j \leftarrow r+1$, $x \leftarrow A[i]$,
2: **while** $i < j$ **do**
3: Repeat $i++$ until $x \leq A[i]$
4: Repeat $j--$ until $x \geq A[j]$
5: Swap(A, i, j)
6: Return j

Therefore, the recurrence relation for QuickSort is $T(n) = O(n) + T(i) + T(n-i)$ where i is the chosen pivot point. In the worst case, where $i = n-1$ or $i = 1$, $T(n) = T(n-1) + O(n) = O(n^2)$. In the best case, where $i = \frac{n}{2}$, $T(n) = 2T(\frac{n}{2}) + O(n) = O(n \log n)$.

3.3 Extension

Randomly picking a pivot point

In practice, we often randomly picking a pivot point with the following procedure inserted before picking a pivot. The modified version is often referred to as Randomized QuickSort.

Algorithm 4 R-Select(A, l, r)

1: $i \leftarrow \text{Random}(l, r)$
2: $\text{Swap}(A, i, l)$

Here the $\text{Random}(l, r)$ function uniformly randomly generates an integer that is between l and r. The analysis of the randomized QuickSort is beyond the scope of this course. With this trick, the expected running time is $O(n \log n)$.

4 Mergesort

4.1 Introduction

Mergesort is a sorting algorithm with both worst-case and average-case performance of $O(n \log n)$. The algorithm is also a recursive divide-and-conquer alillustrategorithm, but instead of using a pivot to decide where to partition our subproblems, mergesort always divides the array equally. Mergesort then recursively sorts these two subarrays and then combines (or merges) them into one master sorted array.

4.2 Pseudocode

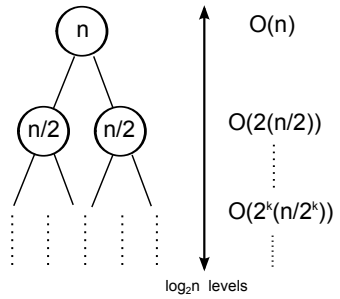
```
1   // Msort(A,i,j) will sort element A[i] through A[j] in A
2   Msort(A,i,j)
3           B=Msort(A,i,(i+j)/2)
4           C=Msort(A,(i+j)/2+1,j)
5           A=merge(B,C)
6
7   // Merge(B,C) assumes sorted arrays B and C
8   // and merges them into one big sorted array
9   Merge(B,C)
10          D=empty array
11          i=j=k=1
12          while(i<|B| and j<|C|)
13                  if(B[i]<C[j]) then D[k]=B[i], i++
14                  else then D[k]=C[j], j++
15                  k++
16          // At this point all elements of one array should be copied to D
17          Copy all elements of the other array into D
18          return D
```

4.3 Example

As an example, consider the following array: 7 1 5 2 0 -2. Mergesort will divide the array into two subarrays 7 1 5 and 2 0 -2. Then it will recursively sort these two arrays and give back 1 5 7 and -2 0 2. To merge these two arrays, we first compare 1 with -2. Because -2 is smaller, it is copied to the final sorted array. Then 1 is compared with 0 and 0 is copied to the final array. Then 1 and 2. This time 1 is smaller so 1 is copied to the final array. Next 5 is compared with 2 and 2 is copied to final array. Now the second array has been used up so 5 and 7 of the first array are copied to the final array. The final array is -2 0 1 2 5 7.

4.4 Performance

For mergesort, the worst-case and average-case performances are the same as the algorithm does not involve any random selection or subarrays of variable length. Line 3 and 4 of the pseudocode each takes $T(n/2)$ time and the `Merge`procedure takes $O(|B|+|C|)$ time, where $|B|$ and $|C|$ are the sizes of the array B and C. The recurrence relation is thus $T(n) = 2T(n/2) + O(n)$. The computation tree is like:



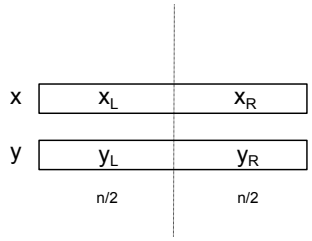
We have $T(n) = O(n \log_2 n)$.

4.5 Comparison to QuickSort

QuickSort is in-place, thus consuming less space. On the other hand, QuickSort is less parallelizable, and can potentially suffer from greater time complexity. Both algorithms are widely used, and in fact, they are both expected to achieve the best running time that a comparison-based sorting algorithm can hope for, which is $O(n \log n)$.

5 Multiplying Bit Strings

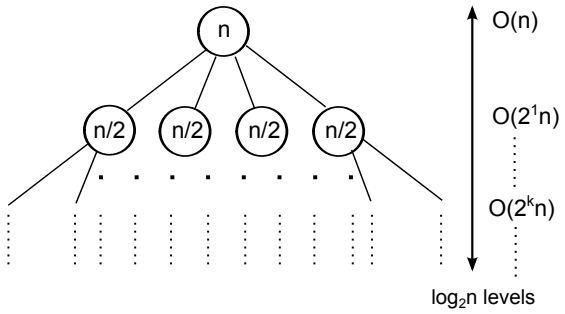
The problem is to multiply two n-bit numbers x and y. The straightforward way is to divide each number into two parts and each part has $\frac{n}{2}$ bit.



We have

$$
\begin{aligned}
x &= x_L 2^{\frac{n}{2}} + x_R \\
y &= y_L 2^{\frac{n}{2}} + y_R \\
xy &= x_L y_L 2^n + (x_L y_R + y_L x_R) 2^{\frac{n}{2}} + x_R y_R
\end{aligned}
$$

The running time $T(n) = 4T(\frac{n}{2}) + O(n)$. This recurrence relation can be solved by computation tree.



We have

$$T(n) = n(1+2+2^2+...2^k+...+2^{\log_2 n}) = n(2^{\log_2 n+1}-1) = O(n^2)$$

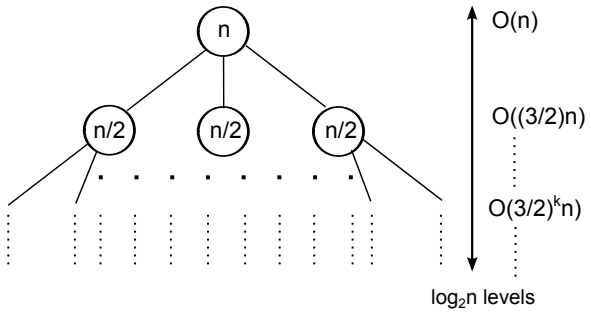
There is a smarter way to do the multiplication based on the fact that

$$\begin{aligned} xy &= x_L y_L 2^n + (x_L y_R + y_L x_R)2^{\frac{n}{2}} + x_R y_R \\ &= x_L y_L 2^n + ((x_L + x_R)(y_L + y_R) - x_R y_R - x_L y_L) + x_R y_R \end{aligned}$$

Only three multiplications are needed: $x_L y_L$, $x_R y_R$ and $(x_L + x_R)(y_L + y_R)$. We have

$$T(n) = 3T(\frac{n}{2}) + O(n)$$

The computation tree is like



We have

$$T(n) = n(1+\frac{3}{2}+(\frac{3}{2})^2+...(\frac{3}{2})^k+...+(\frac{3}{2})^{\log_2 n}) = O(n(\frac{3}{2})^{\log_2 n}) = O(3^{\log_2 n}) = O(n^{\log_2 3}) \approx O(n^{1.585})$$

6 Polynomial Multiplication

Polynomial Multiplication is another example where Divide-and-Conquer technique can be applied. Assume

$$P(x) = \sum_{k=0}^{n} a_k x^k$$

and

$$Q(x) = \sum_{k=0}^{n} b_k x^k.$$

If we naively multiply them term by term, it would be of complexity $O(n^2)$. However, we can apply the following trick. Write

$$P(x) = \sum_{i=0}^{\frac{n}{2}} a_i x^i + x^{\frac{n}{2}} \sum_{i=0}^{\frac{n}{2}} a_{i+n/2} x^i = P_1(x) + x^{\frac{n}{2}} P_2(x)$$

and

$$Q(x) = \sum_{i=0}^{\frac{n}{2}} b_i x^i + x^{\frac{n}{2}} \sum_{i=0}^{\frac{n}{2}} b_{i+n/2} x^i = Q_1(x) + x^{\frac{n}{2}} Q_2(x).$$

Then

$$PQ = P_1Q_1 + x^{\frac{n}{2}}(P_1Q_2 + P_2Q_1) + x^n P_2Q_2. \tag{1}$$

The above equation tells us that we can divide the problem into 4 subproblems with half of the original input size, which gives us the following recurrence relation:

$$T(n) = 4T(\frac{n}{2}) + O(n)$$

However, we can improve this to

$$T(n) = 3T(\frac{n}{2}) + O(n)$$

by the fact that

$$P_1Q_2 + P_2Q_1 = (P_1 + P_2)(Q_1 + Q_2) - P_1Q_1 - P_2Q_2. \tag{2}$$

Note that P_1Q_1 and P_2Q_2 in equation 2 also appear in equation 1. Therefore we only need to compute one more multiplication, which is $(P_1 + P_2)(Q_1 + Q_2)$, giving us the following recurrence relation

$$T(n) = 3T(\frac{n}{2}) + O(n)$$

This recurrence relation gives us $T(n) = O(n^{\log_2 3})$, which is faster than naively multiplying out all terms.

7 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 kth 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.

7.1 A DAC Algorithm for Selection

Our goal is to define an algorithm that finds the kth smallest element of an n element A in $O(n)$ time. We begin by defining the following correct (but as we will 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 will 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 kth smallest element in A_1 is also the kth 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 kth 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 kth 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:

$$\begin{aligned} T(n) &= T(n/2) + O(n) \\ &\leq c \cdot (n + n/2 + n/4 + \ldots + 1) && (3) \\ &= cn \cdot (1 + 1/2 + 1/4 + \ldots + 1/n) && (4) \\ &< cn \cdot 2 = O(n), && (5) \end{aligned}$$

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 (3). To see why inequality (5) is true, we quickly review geometric series. Recall (hopefully) that for some real number $0 \leq r < 1$, we have that

$$1 + r + r^2 + r^3 + \ldots = \sum_{i=0}^{\infty} r^i = \frac{1}{1-r}. \quad (6)$$

For line (4), 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 (6) 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 will 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.

7.2 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 will 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 ϕ 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.
- 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 will 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 $|A| = n$.

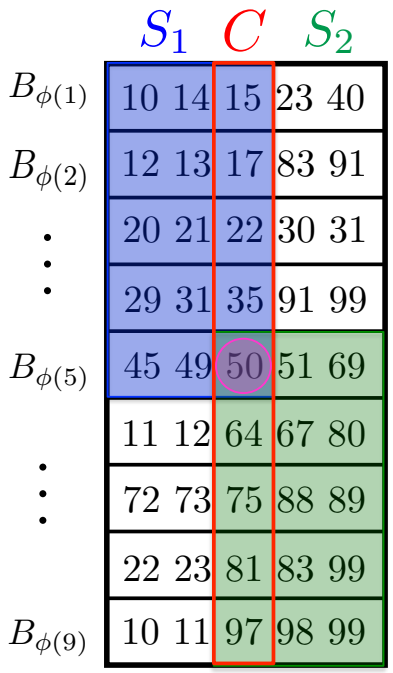


Figure 1: A diagram illustrating the definitions of $B_{\phi(1)}, \ldots, B_{\phi(n/5)}$, C, S_1, and S_2. Here $n = 45$, and thus we have 9 blocks in total. Observe that each block is sorted individually and that the blocks themselves are ordered based on their medians. Array C is outlined in red, and 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).

- 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 will 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).

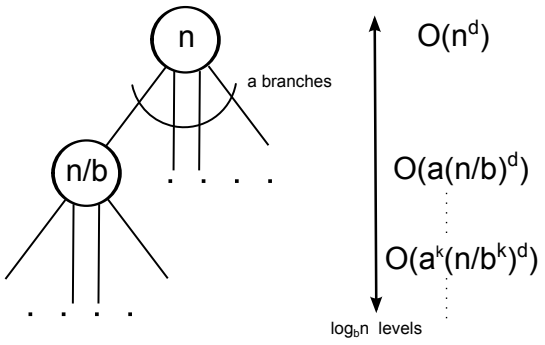
$$\begin{aligned} T(n) &= T(7n/10) + T(n/5) + cn \\ &< cn \cdot \sum_{i=0}^{\infty} \left(\frac{9}{10}\right)^i \qquad (7) \\ &= cn \cdot 10 = O(n), \qquad (8) \end{aligned}$$

as desired. You should try to verify inequality (7) (use the tree expansion method; you should get that the total work done on the kth level of the tree is $cn \cdot (9/10)^k$). Equation (8) follows by the closed form for a geometric series we saw earlier in equation (6).

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 (8) (and remember, any algorithm for selection must take $\Omega(n)$ time).

8 Master Theorem

Let us analyze a recurrence relation for a general divide-and-conquer algorithm: $T(n) = aT(\frac{n}{b}) + O(n^d)$. The computation tree is like:



We have $T(n) = O(n^d + a(\frac{n}{b})^d + ... + a^k(\frac{n}{b^k})^d + ... + a^{\log_b n}(\frac{n}{b^{\log_b n}})^d)$, where the item $a^{\log_b n}(\frac{n}{b^{\log_b n}})^d = a^{\log_b n} = n^{\log_b a}$. The solution is based on the comparison between d and $\log_b a$. There are three situations:

- $d > \log_b a$: $T(n) = \sum_{j=0}^{\log_b n - 1} a^j(\frac{n}{b^j})^d = \frac{n^d - \frac{a}{b^d} n^{\log_b a}}{1 - \frac{a}{b^d}}$. The item n^d dominates and we have $T(n) = O(n^d)$.
- $d < \log_b a$: $T(n) = \sum_{j=0}^{\log_b n - 1} a^j(\frac{n}{b^j})^d = \frac{n^d - \frac{a}{b^d} n^{\log_b a}}{1 - \frac{a}{b^d}}$. The item $n^{\log_b a}$ dominates and we have $T(n) = O(n^{\log_b a})$.
- $d = \log_b a$: All items have the same power and we have $T(n) = O(\sum_{j=0}^{\log_b n - 1} \frac{a^j}{b^{dj}} n^d) = O(n^d \log_b n)$.