EECS 660: FUNDAMENTALS OF COMPUTER ALGORITHMS

MODULE V: DIVIDE AND CONQUER, PART I

DISCLAIMER

ACKNOWLEDGEMENT

- Many of the figures, unless otherwise stated, come from ***Algorithm Design*** 4th edition, by Jon Kleinberg and Eva Tardos and ***Analysis of Algorithms*** 2nd edition, by Jeffery McConnell.

OUTLINE

- A general introduction of divide and conquer algorithm design schema, and its relation with subproblems and recursion.

- Examples of divide and conquer algorithms
  - Computing Fibonacci number (unrolling the recursion)
  - Merge sort (solving closed form and the master's theorem)
  - Median finding (uneven subproblem division)

SUBPROBLEM AND RECURSION

- Consider the computation of Fibonacci number:
  - $f(n) = f(n-1) + f(n-2)$ for $n > 2$
  - $f(1) = f(2) = 1$

- We see that the solution of the entire problem, i.e. the computation of $f(n)$, depends on the solution of a smaller problem in size, i.e. $f(n-1)$ and $f(n-2)$.

- The smaller (w.r.t the original problem) problems, are called **subproblems**.

SUBPROBLEM AND RECURSION

- The identification of subproblems can sometime leads to another intuitive design principal, called **recursion** or **divide and conquer.**

- The design principal can lead to very simple algorithms, as the algorithm for solving the subproblems are in many cases similar (or even identical) to the one for solving the entire problem. (In this case, we can write a single function and plug in different inputs.)

SUBPROBLEM AND RECURSION

- The idea of this design principal consists (recursively) of three main steps:
  - partition the larger problem into a number of subproblems
  - solve the subproblems (most likely recursively)
  - use the results of the subproblems to solve the larger problem

SUBPROBLEM AND RECURSION

- Now, we can try to develop a divide and conquer algorithm to solve the problem of computing Fibonacci number:

```
Fibo($n$)
If $n < 2$ return 1;
return Fibo($n-1$)+Fibo($n-2$)          // 1: Partition into two smaller problems
                                        // 2: solve them individually through recursion
                                        // 3: take the results of the subproblems and solve
                                        //    the larger problem as their sum
```

SUBPROBLEM AND RECURSION

- We will need to analyze the correctness and efficiency of the algorithm.
  - correctness and efficiency analysis of divide and conquer algorithm could be different than those for the greedy algorithms
  - the correctness analysis if often much simpler
  - the efficiency analysis could be more involving

SUBPROBLEM AND RECURSION

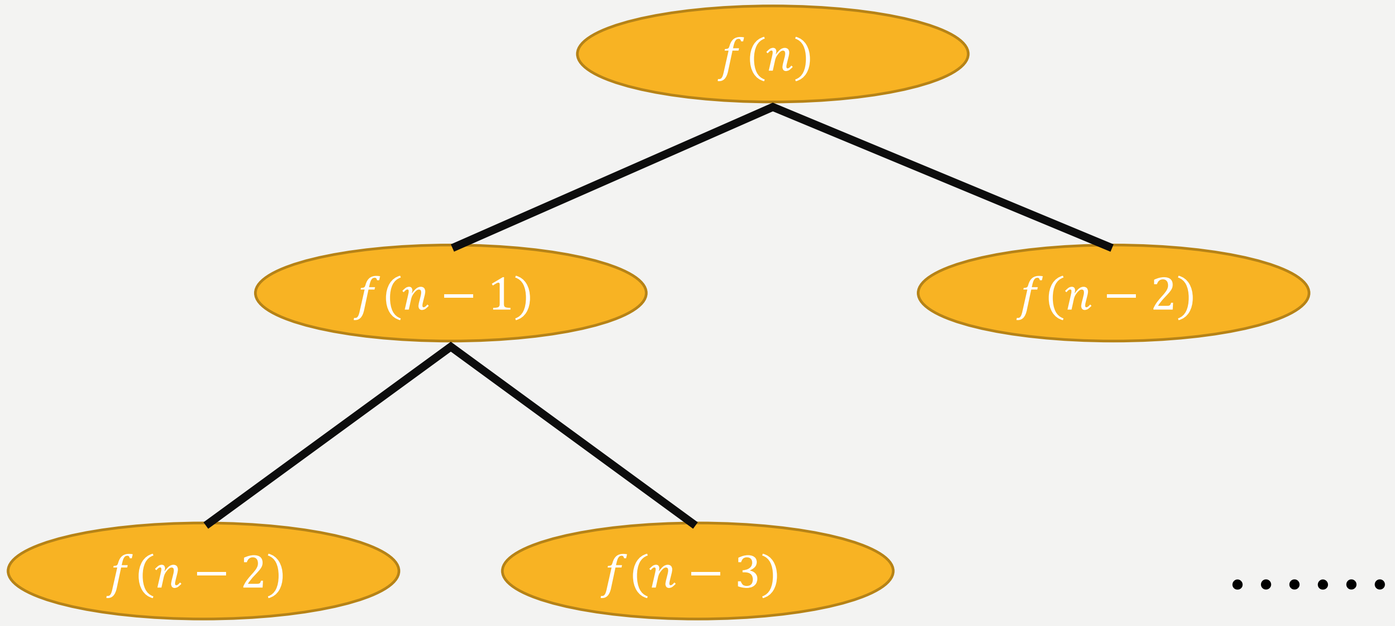
- The correctness of divide and conquer algorithms can often be shown by that the algorithm correctly implements the recursion.
  - for the Fibonacci number problem, the recursion is defined as $f(n) = f(n-1) + f(n-2)$ for $n > 2, f(1) = f(2) = 1$
  - clearly our algorithm correctly implements the recursion

- In many cases we do not need to prove the correctness of divide and conquer algorithms, if the correctness is obvious.
- In some cases, we do need to show that the recursion relation is correct, in a sense that it consider all necessary cases (via partitioning into correct set of subproblems) and correctly computes the larger problem with the results of the smaller problems.

SUBPROBLEM AND RECURSION

- Now we need to analyze the time complexity of the algorithm.
- One way to see the time complexity is to simply unroll the recursion:
  - we see that to compute $f(n)$, we need to get the results of $f(n-1)$ and $f(n-2)$, and add them together (we assume that the addition takes $O(1)$ time)
  - to compute $f(n-1)$, we need to get the results of $f(n-2)$ and $f(n-3)$, and add them together
  - …
  - the process can simply be modeled as a binary tree

SUBPROBLEM AND RECURSION



Note that the main effort we spend is to compute the parent node by summing up the two children, in which case the summation takes $O(1)$ time. Clearly, the overall time complexity thus corresponds to how many nodes we have in the binary tree. Note that for each layer, we reduce the index by 1, and hence we have n layers. Hence the time complexity of the algorithm is $O(2^n)$.

SUBPROBLEM AND RECURSION

- On a separate note, we can actually compute the Fibonacci number in $O(n)$ time.
- The more efficient algorithm will be developed under the design principal of **dynamic programming**, which will be introduced as the next module.

A SHORT SUMMARY

- Larger problems can sometimes be solved by dividing them into smaller subproblems.
  - We solve the smaller subproblems recursively.
  - We merge the results of the subproblems to solve the larger problem.

- Such a divide and conquer schema often leads to recursive algorithms.
  - The correctness of the algorithm is (in most cases) easy to prove.
  - It could be less obvious to analyze the time complexity (we have introduced here a method called "**unrolling the recursion**").

THE DIVIDE AND CONQUER ALGORITHM FRAMEWORK

```
DivideAndConquer($data, n, solution$)
If $n \leq SizeLimit$ return DirectSolution($data, n, solution$)
DivideInput($data, n, smallData, smallSize, numSubproblems$)
for $i$ in 1 … $numSubproblems$ do
        DivideAndConquer($smallData[i], smallSize, smallSolution[i]$)
endfor
return CombineSolutions($smallSolution, numSubproblems, solution$)
```

MERGE SORT

- Merge sort is a sorting algorithm developed under the divide and conquer schema.
- The entire set of numbers is divided into two equal-sized (not necessarily, but it facilitates efficiency analysis) subsets. Each subset is sorted individually, and then the two sorted subsets are merged to a larger sorted list of numbers.
- You might have already learnt it in EECS 560.

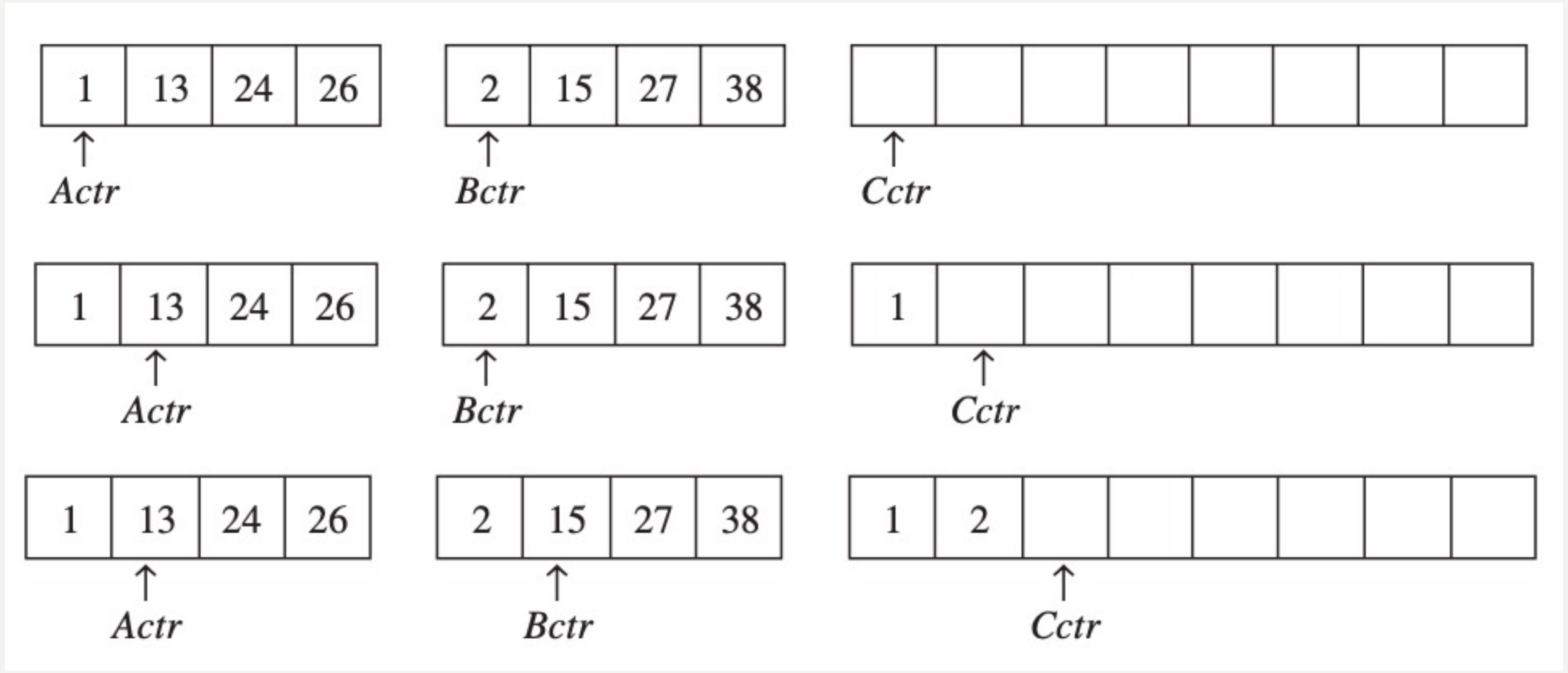
MERGE SORT

$$\begin{tikzcd}
 & & & \text{8,3,12,7 (3,7,8,12)} & & & \\
 & \text{8,3 (3,8)} \arrow[urr] & & & & \text{12,7 (7,12)} \arrow[ull] & \\
\text{8 (8)} \arrow[ur] & & \text{3 (3)} \arrow[ul] & & \text{12 (12)} \arrow[ur] & & \text{7 (7)} \arrow[ul]
\end{tikzcd}$$

Numbers without parentheses indicate the unsorted list.

Numbers in parentheses indicate the sorted list.

MERGE SORT



MERGE SORT

```
MergeSort($data, n, sorted$)
If $n \leq 1$ return InsertionSort($data, n$)
DivideInput($data, n, smallData, n/2, 2$)
MergeSort($smallData[0], n/2, smallSorted[0]$)
MergeSort($smallData[1], n/2, smallSorted[1]$)
$sorted =$CombineSorted($smallSorted[0], smallSorted[1]$)
return
```

MERGE SORT

- You can argue the correctness of the merge sort algorithm by showing that:
  - the division is clearly correct since the union of the two subsets is exactly the original set of number, and the intersection of the two subsets is an empty set (no number is missing and no number is duplicated)
  - the merge of two sorted list into a larger sorted list is also correct, since all numbers remained in the arrays A and B must be larger than the number that is put into the array C.

MERGE SORT

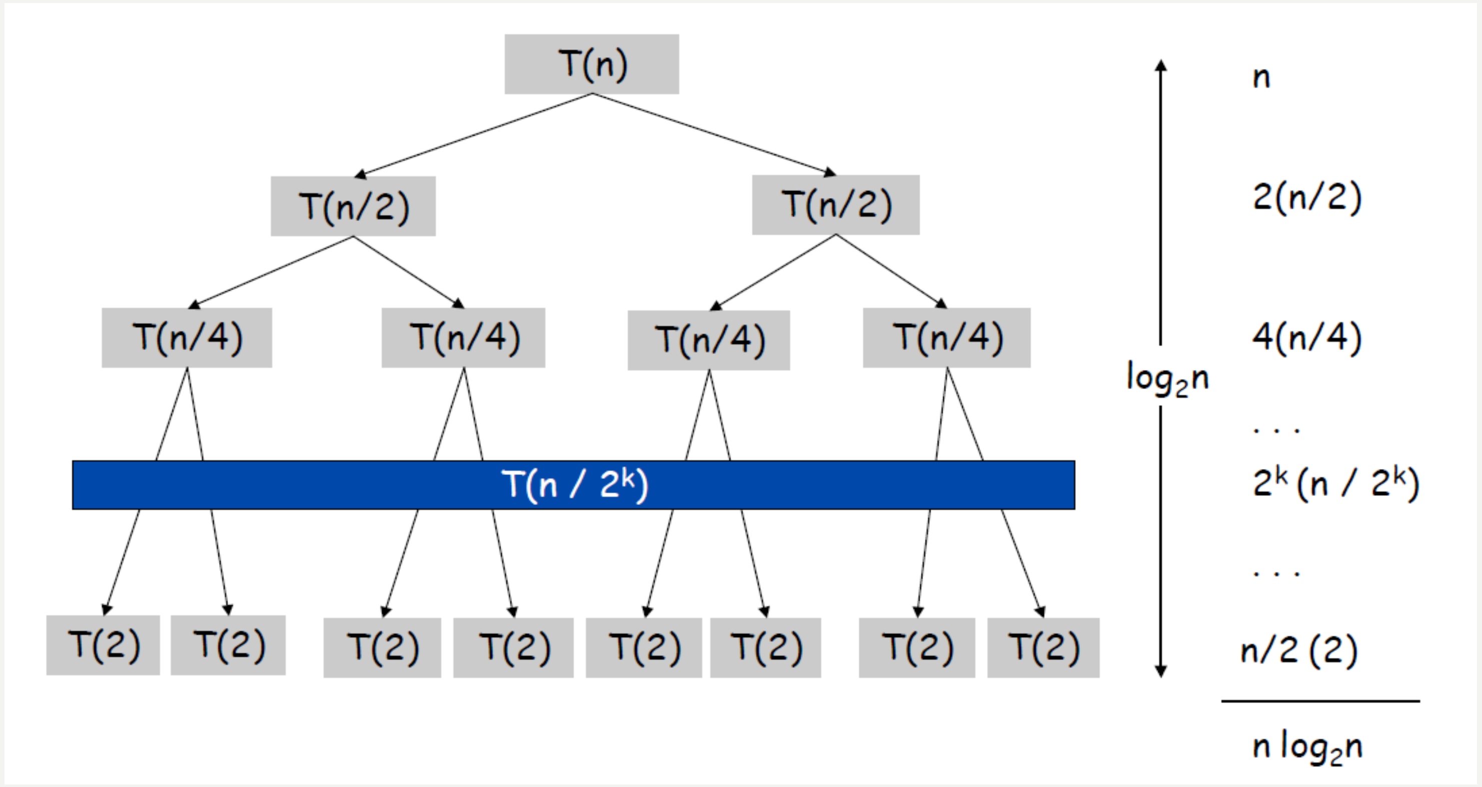
- We can unroll the recursion to analyze the time complexity of merge sort.

- We note that the size of the number set(s) is reduced into half at each recursion, and it takes $O(\log n)$ recursions to reach 1 (or any constant limit).

- Note that at each recursion, we need to merge two sorted subsets into one sorted set. The process clearly takes linear time since we go through the arrays A, B, and C exactly one time. Since the unions of all numbers to be sorted at each recursion are exactly the same (which is exactly the input set as the division does not miss or duplicate any number), it takes $O(n)$ time to merge the results at each recursion.

- It follows that the total time complexity is $O(n \log n)$.

MERGE SORT


T(n)
T(n/2)
T(n/2)
T(n/4)
T(n/4)
T(n/4)
T(n/4)
T(n / 2^k)
T(2)
T(2)
T(2)
T(2)
T(2)
T(2)
T(2)
T(2)
log_2 n
n
2(n/2)
4(n/4)
. . .
2^k (n / 2^k)
. . .
n/2 (2)
n log_2 n

MERGE SORT

- Another way we can use to analyze the time complexity is to convert the **recursion form** into **closed form**.

- Let the time required for sorting n numbers be $T(n)$, and we have the following recursion form:

$$T(n) = \begin{cases} 2T\left(\frac{n}{2}\right) + O(n) \; if \; n > 1 \\ O(1) \; if \; n \leq 1 \end{cases}$$

MERGE SORT

- We can write the recursive form into a set of formulae w.r.t different input sizes:

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$$
$$T(n/2) = 2T\left(\frac{n}{4}\right) + O(n/2)$$
$$\dots \dots$$
$$T(2) = 2T(1) + O(2)$$
$$T(1) = O(1)$$

MERGE SORT

- We can multiply 2^i for the ith equation:

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$$
$$2 * T\left(\frac{n}{2}\right) = 2 * 2T\left(\frac{n}{4}\right) + 2 * O(n/2)$$
$$\dots\dots$$
$$2^{\log n-1} * T(2) = 2^{\log n-1} * 2T(1) + 2^{\log n-1} * O(2)$$
$$2^{\log n} * T(1) = 2^{\log n} * O(1)$$

MERGE SORT

- We can add up all the equations, and will notice that some terms canceled out:

$$T(n) = 2\cancel{T\left(\frac{n}{2}\right)} + O(n)$$
$$2 * \cancel{T\left(\frac{n}{2}\right)} = 2 * 2\cancel{T\left(\frac{n}{4}\right)} + 2 * O(n/2)$$
$$\dots\dots$$
$$2^{\log n-1} \cancel{*} T(2) = 2^{\log n-1} \cancel{*} 2T(1) + 2^{\log n-1} * O(2)$$
$$2^{\log n} \cancel{*} T(1) = 2^{\log n} * O(1)$$

MERGE SORT

- The sum becomes:

$$T(n) = 2^{\log n} * O(1) + 2^{\log n-1} * O(2) + \cdots + 2 * O\left(\frac{n}{2}\right) + O(n)$$

- Note that each term on the right-hand side evaluates to $O(n)$, and there are $O(\log n)$ terms. So, the overall time complexity is in $O(\log n)$, the same as what we get with unrolling the recursion.

MASTER’S THEOREM

- The above analysis provides a more well-defined approach to analyze time complexity, which can directly compute the closed form without going through the mental process of unrolling the recursion.

- One limitation of the approach is that it is easier to apply with cases where you have equal-sized subproblems.
  - for example, the merge sort algorithm divides the whole set into two equal-sized subsets; and we have shown how to analyze its time complexity
  - however, the Fibonacci number algorithm divides the problem into two subproblems with two different sizes ($f(n-1)$ and $f(n-2)$); and it is more challenging to compute its closed form using this approach

MASTER'S THEOREM

- If our divide and conquer algorithm divides the larger problem into equal-sized smaller problems, we can formalize the time complexity computation into a theorem called the **Master's theorem**.

- We will need three types of information from the algorithm:
  - how many subproblems the large problem is divided into (denote as a)
  - how much smaller the subproblem is compared to the large problem (denoted as b)
  - what is the exponent of the polynomial time complexity required by the result merge process (denoted as c)

MASTER'S THEOREM

- That is, we can write the recursive form of the algorithm as:

$$T(n) = aT\left(\frac{n}{b}\right) + O(n^c)$$

  - **Caveat:** Not all divide and conquer algorithm can be written in this form! The Master's theorem only applies to algorithms that can.

- Once we get a, b, c, we will be able to get the closed form time complexity directly.

- Let's derive the time complexity and prove the Master's theorem.

MASTER'S THEOREM

- We shall start with imaging the recursion process in a similar way as in unrolling the algorithm: an a-nary tree (note that we divide the larger problem into a subproblems).

- Now, consider the ith layer in the a-nary tree (or the ith unrolling of the recursion; we will use the term "layer" throughout the discussion to make it easier to conceive).
  – we will have a^i subproblems
  – each subproblem will have a size of $\frac{n}{b^i}$
  – obtaining the solution of the subproblem will need $r(\frac{n}{b^i})^c$ time, where r is a constant term that converts the time complexity term into the actual time

MASTER'S THEOREM

- Now, the time required to compute the solution for all subproblems in the ith layer becomes:

$$ra^i\left(\frac{n}{b^i}\right)^c$$

- Rearranging the terms, we get:

$$rn^c\left(\frac{a^i}{b^{ic}}\right) = rn^c\left(\frac{a}{b^c}\right)^i$$

MASTER'S THEOREM

- Then, them overall time for the algorithm is simply the sum of the per-layer time overall all layers. Note that we have $\log_b n$ layers, which leads to an overall time of:

$$T(n) = \sum_{i=1}^{\log_b n} rn^c \left(\frac{a}{b^c}\right)^i$$

- By analyzing the term $rn^c \left(\frac{a}{b^c}\right)^i$, we realize that $r, n,$ and c are constants for a given algorithm and input instance. The only term that is variable between different layers is $\left(\frac{a}{b^c}\right)^i$.

MASTER’S THEOREM

- While computing the sum of a series, it is important to know whether it is an increasing or a decreasing series.
  - that is, we need to discuss whether the term $\left(\frac{a}{b^c}\right)^i$ is increasing, invariable, or decreasing as i increases

- We have three cases to discuss:
  - an increasing series: i.e., $\frac{a}{b^c} > 1 \Longrightarrow c < \log_b a$
  - an invariable series: i.e., $\frac{a}{b^c} = 1 \Longrightarrow c = \log_b a$
  - a decreasing series: i.e., $\frac{a}{b^c} < 1 \Longrightarrow c > \log_b a$

MASTER'S THEOREM

- Case 1: an increasing series: i.e., $\frac{a}{b^c} > 1 \Longrightarrow c < \log_b a$:

$$T(n) = rn^c \sum_{i=1}^{\log_b n} \left(\frac{a}{b^c}\right)^i = O(rn^c \left(\frac{a}{b^c}\right)^{\log_b n}) = O(rn^c \frac{a^{\log_b n}}{(b^c)^{\log_b n}}) = O(rn^c \frac{n^{\log_b a}}{n^{\log_b b^c}})$$

$$= O(rn^c \frac{n^{\log_b a}}{n^c}) = O(rn^{\log_b a}) = \boldsymbol{O(n^{\log_b a})}$$

- $\log_b a^{\log_b n} = \log_b n * \log_b a = \log_b n^{\log_b a} \Longrightarrow a^{\log_b n} = n^{\log_b a}$

MASTER'S THEOREM

- Case 2: an invariable series: i.e., $\frac{a}{b^c} = 1 \Longrightarrow c = \log_b a$:

$$T(n) = rn^c \sum_{i=1}^{\log_b n} \left(\frac{a}{b^c}\right)^i = rn^c \log_b n \left(\frac{a}{b^c}\right)^i = rn^c \log_b n = \boldsymbol{O}(\boldsymbol{n^c\, log_b\, n})$$

MASTER'S THEOREM

- Case 3: a decreasing series: i.e., $\frac{a}{b^c} < 1 \Longrightarrow c > \log_b a$:

$$T(n) = rn^c \sum_{i=1}^{\log_b n} \left(\frac{a}{b^c}\right)^i = rn^c \frac{a}{b^c}\left(\frac{1-\left(\frac{a}{b^c}\right)^{\log_b n}}{1-\frac{a}{b^c}}\right) = rn^c \frac{a}{b^c}\left(\frac{1-0}{1-\frac{a}{b^c}}\right) = \boldsymbol{O(n^c)}$$

- Recall that the sum of a decreasing series is $a_0 \frac{1-r^n}{1-r}$, where a_0 is the initial term, $r < 1$ is the ratio between adjacent terms, and n is the number of terms.
- Note that the discussion of time complexity involves the assumption that $n \to \infty$, and when this happens $\left(\frac{a}{b^c}\right)^{\log_b n} \to 0$ if $\frac{a}{b^c} < 1$.

MASTER'S THEOREM

- In summary, we have:
  - If $c > \log_b a$ then $T(n) = O(n^c)$;
  - If $c = \log_b a$ then $T(n) = O(n^c \log n)$;
  - If $c < \log_b a$ then $T(n) = O(n^{\log_b a})$.

MASTER'S THEOREM

- Let's try to apply the Master's theorem to analyze the time complexity of the merge sort algorithm.

- We know that for merge sort, we have:
  - $a = 2$ (we have two subproblems for each recursion)
  - $b = 2$ (the larger problem is twice as large as each subproblem)
  - $c = 1$ (merging two sorted lists into one sorted list needs linear time)
  - since $c = \log_b a$, we know that it is in Case 2, which has a form of $O(n^c \, log_b \, n)$
  - plug in b and c we get $O(n^1 \, log_2 \, n)$, which is exactly what we have come up with using recursion unrolling and direct computation

MEDIAN FINDING

- Now, we are going to introduce the third problem, median finding.

- We are going to learn:
  - in some cases divide and conquer can speedup (in theoretical time complexity) direct solution
  - how to analyze time complexity for divide and conquer algorithms with uneven-sized subproblems using the substitution method

MEDIAN FINDING

- <u>**The median finding problem**</u>: find the median of a set of unsorted numbers.

- Naïve solution: sort the numbers and find the median:
  - sorting: $O(n \log n)$ time
  - overall time complexity: $O(n \log n)$ time

- Can we do better?

MEDIAN FINDING

- Let's see from which aspect we can improve the naïve algorithm:
  - denote the median as m
  - we know that based on definition of median, it implies that the whole set N can be partitioned into two subsets N_s and N_L, such that $|N_s| = |N_L| = |N|/2$, and for each $i \in |N_s|$ we have $i \leq m$ and for each $j \in |N_L|$ we have $j \geq m$
  - how much time do we need to detect such a condition, given that m, N_s, and N_L are well defined? Clearly we can do this in $O(n)$ time, as we simply compare each number in N_s, and N_L with m to verify the condition
  - in other words, <u>we do not care whether the numbers in N_s or N_L are actually sorted</u>; and here is some unnecessary information computed by the naïve algorithm
  - we may try to eliminate the computation of such unnecessary information and speedup the overall algorithm

MEDIAN FINDING

- Blum, Floyd, Pratt, Rivest and Tarjan developed a divide and conquer algorithm for median finding, which has a time complexity of $O(n)$.

- Let's discuss the crux of the idea:
  - if we partition the set of numbers N into two subsets, say N_1 and N_2, and we have recursively found their respective medians m_1 and m_2 (without loss of generality we assume $m_1 \leq m_2$)
  - we know that the median of the entire set N, i.e., m, must be between m_1 and m_2. That is, we have $m_1 \leq m \leq m_2$
  - (proof of the claim): if $m_2 < m$, it means m is larger than more than $|N_2|/2$ numbers in N_2, and also because $m_1 < m$, m is larger than more than $|N_1|/2$ numbers in N_1. It follows that m is larger than more than $|N|/2$ numbers in N, hence m cannot be the median. (We omit the discussion of $m_1 > m$ since the symmetricity is obvious.)
  - Also note that we did not assume $|N_1| = |N_2|$; i.e., the two subsets may be uneven

MEDIAN FINDING

- Let’s discuss the crux of the idea (continued):
  - since $m_1 \leq m \leq m_2$, the feasible set of number that contains m now has a smaller size (as compared with $|N|$)
  - we may gain some computational efficiency as we are trying to find the median from a smaller subset (which needs to be proved)
  - by counting the numbers in N that are between m_1 and m_2, and knowing the sizes of N_1 and N_2, we can actually find m

MEDIAN FINDING

```
SELECT($A$,$i$)      // $A$ is the set of unsorted numbers, $i$ is the rank of the number we wish to find
Divide the $n$ items into groups of 5;        // we will discuss why 5
Find the median of each group of 5;
Use SELECT recursively to find the median (call it $x$) of these $n/5$ medians;
Partition the $n$ items around $x$, and let $k$ be the ranking of $x$;
If $i = k$
        then return $x$;
else if $i < k$
        use SELECT recursively by calling SELECT($A[1,\ldots,k-1], i$);
else if $i > k$
        use SELECT recursively by calling SELECT($A[k+1,\ldots,|A|], i-k$);
endif
```

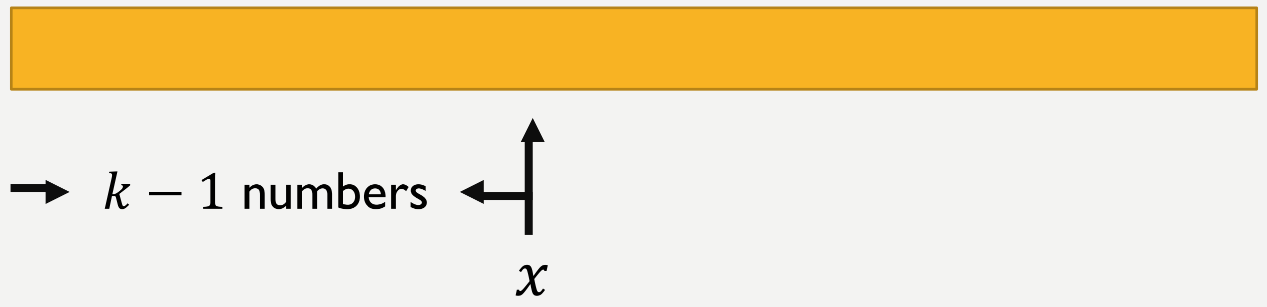
MEDIAN FINDING

- Since the correctness of the algorithm is less obvious than the algorithms we studied before, we need to look deeper to prove its correctness.

- First, we will initially call **<u>SELECT</u>**$\boldsymbol{(N, |N|/2)}$, which will correctly identify the median of the set N if the function **<u>SELECT</u>** works correctly.

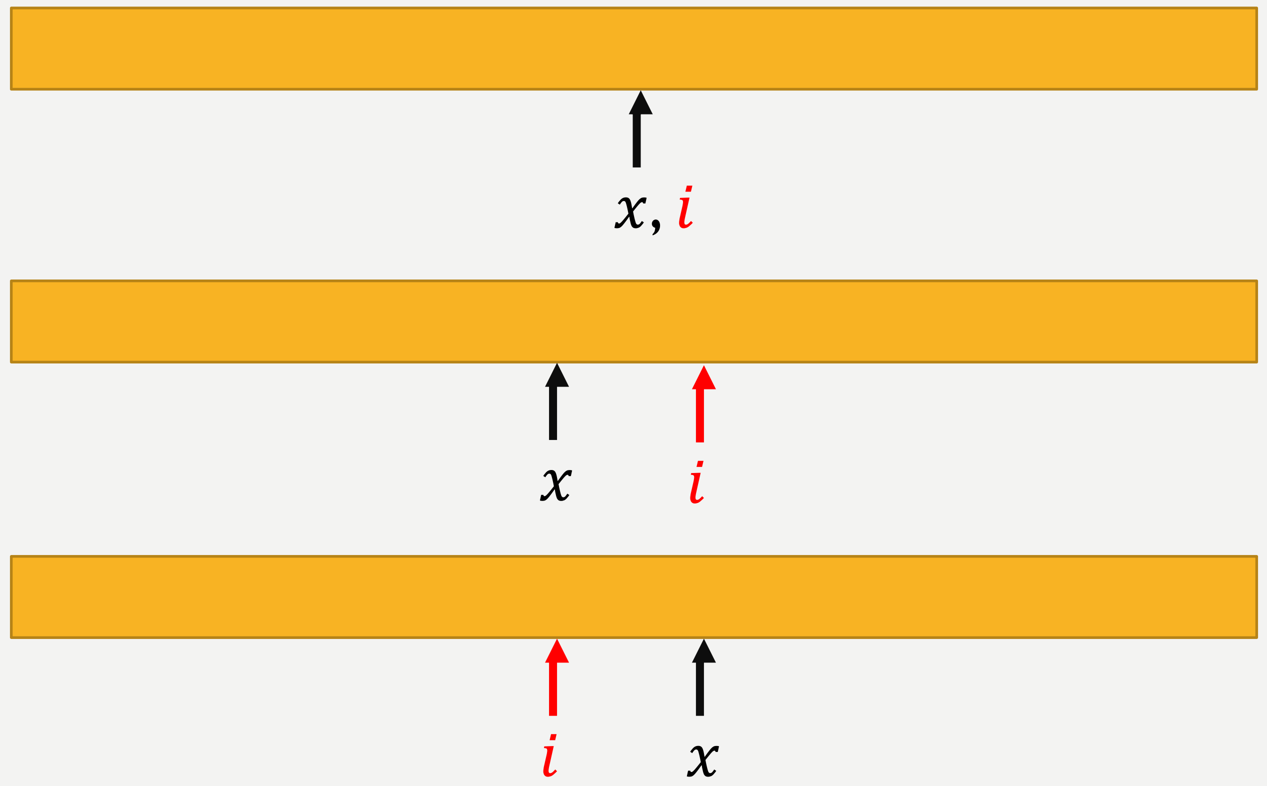
MEDIAN FINDING

- Note that the algorithm "**Partition the n items around x, and let k be the ranking of x**". So, k is the rank of x of the entire set A.



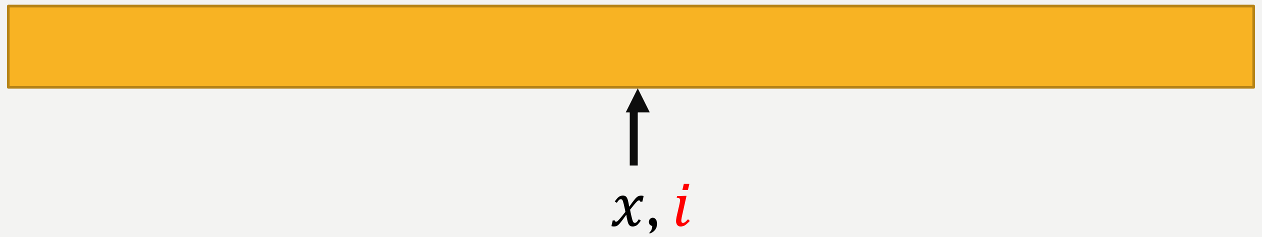
MEDIAN FINDING

- And we need to discuss three cases (depending the relative relation between k and i):



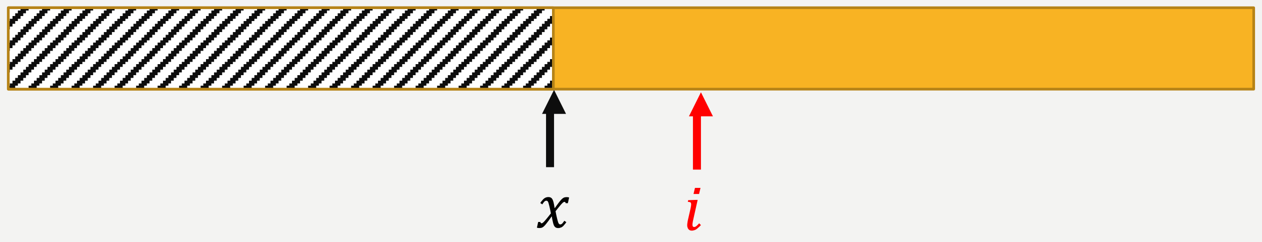
MEDIAN FINDING

- Case 1: if x is exactly the ith element, we will simply return x.



MEDIAN FINDING

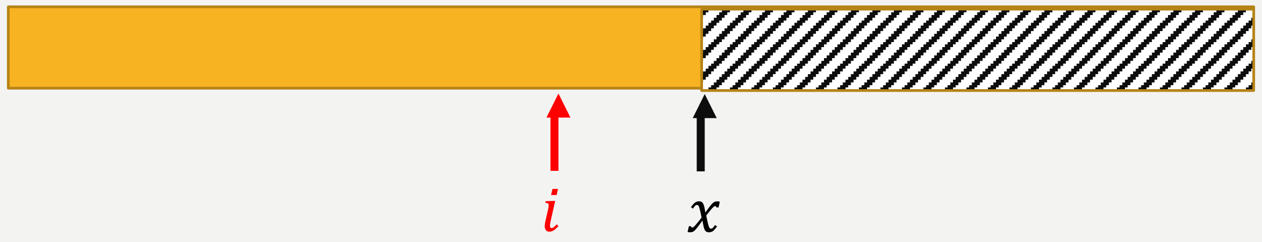
- Case 2: if x ranks less than i (or $k < i$), we will eliminate all numbers less than x.
  - Note that we only partitioned the numbers around x (but did not sort the numbers). So, the numbers in the yellow regions are unsorted. That means we will not be able to know which number corresponds to the ith rank, and the best we can do is to take them all. (It explains why we cannot simply take $A[k+1 \ldots i]$.)



<u>SELECT</u>$\boldsymbol{(A[k+1, \ldots, |A|], i - k)}$

MEDIAN FINDING

- Case 3: if x ranks larger than i (or $k > i$), we will eliminate all numbers less than x and all numbers larger than i.



SELECT$(A[1, \ldots, k - 1], i)$

- The algorithm correctly finds the number with the expected rank in all three cases. Hence, the algorithm is correct.

MEDIAN FINDING

- Now, let's analyze the efficiency of the algorithm.
- Let's write the time required for each recursion:

$$T(n) = T\left(\frac{n}{5}\right) + T(|Subset|) + O(n)$$

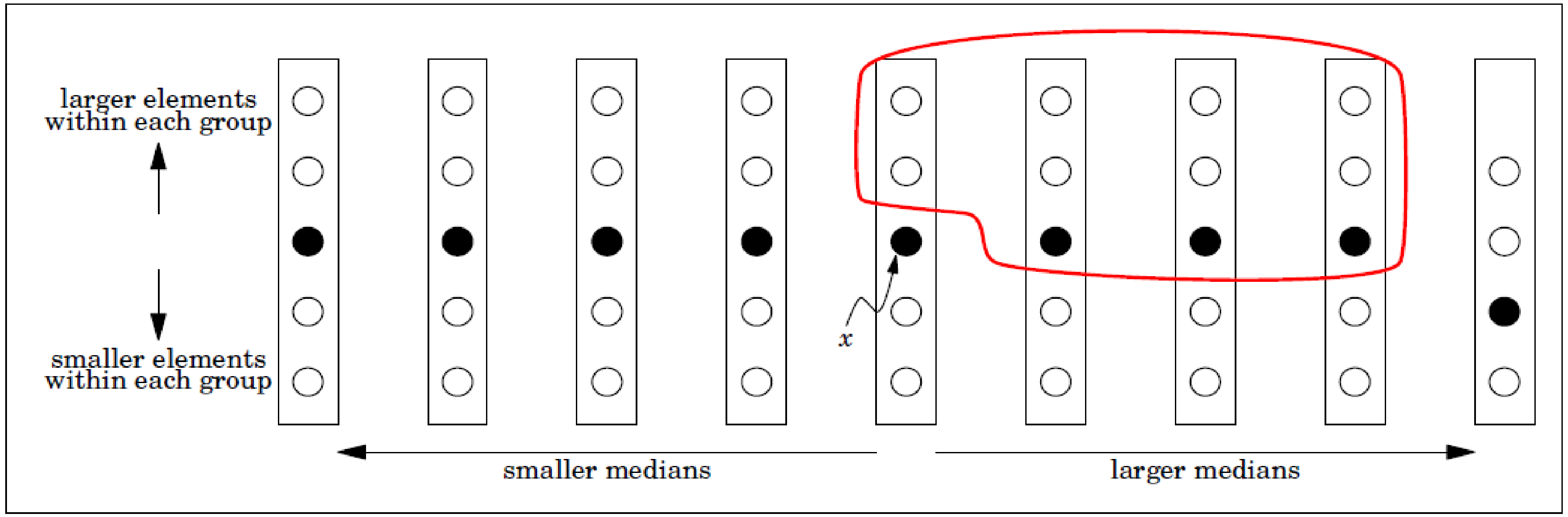
  - the term $T\left(\frac{n}{5}\right)$ comes from "**Use <u>SELECT</u> recursively to find the median (call it x) of these $n/5$ medians**"
  - $Subset$ corresponds to $A[k\ +1,\ldots,|A|]$ or $A[1,\ldots,k\ -1]$, depending which case applies (i.e., the yellow regions in the previous slides)
  - the linear time $O(n)$ comes from "**Partition the n items around x, and let k be the ranking of x**", where we need to compare each number in A with x

MEDIAN FINDING

- To complete the formula, we still have to figure out how large $Subset$ will be.
  - in other words, how much smaller $Subset$ is compared to A

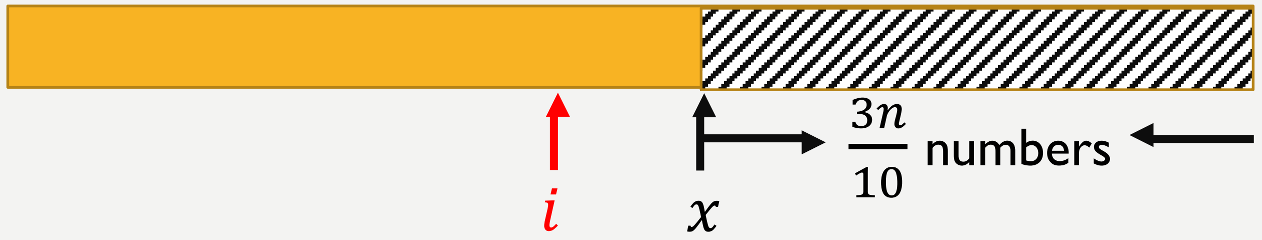
- We know that the "cut" is determined by x, which is the median of the medians of all groups of 5.
  - therefore, we know that x must be smaller/larger than half of the medians of all groups of 5
  - and each median of groups of 5 must be smaller/larger than half of the numbers in its corresponding group
  - by transitivity we can infer the worst-case scenario of the size of $Subset$

MEDIAN FINDING


larger elements within each group
smaller elements within each group
x
smaller medians
larger medians

- (Assuming the discussion of "less than" cases.) x, as the median of the medians, is smaller than $\frac{n}{5*2}$ medians of groups of 5.
- Each median of 5 is also smaller than $floor(\frac{5}{2})$ numbers in its group.
- x is guaranteed to be smaller than $\frac{n}{5*2} + \frac{n}{5*2} * floor\left(\frac{5}{2}\right) = \frac{3n}{10}$ numbers

MEDIAN FINDING



The yellow region, which corresponds to $Subset$, is at most $\frac{7n}{10}$

Discussion of the "larger than" case, which corresponds to the elimination of smaller numbers, is omitted due to obvious symmetricity.

MEDIAN FINDING

- Now we can rewrite the formula for the overall runtime of the algorithm:

$$T(n) \leq T\left(\frac{n}{5}\right) + T\left(\frac{7n}{10}\right) + O(n)$$

- The formula is not in the standard form which allows us to apply the Master's theorem. Unrolling the recursion also seems difficult because the subproblem sizes are non-equal nor correspond to integer fractions.
- We will need to use another method, called **substitution method**, to analyze its time complexity.

MEDIAN FINDING

- The substitution method works as the follows:
  - we "<u>guess</u>" the time complexity of the recursion
  - we substitute the guessed time complexity into the running time formula
  - check if the substitution leads to an upper bound that is consistent with what we have guessed

- It sounds like a circular reasoning, but it is actually not the case.
  - if we "guess" wrong, it does not lead to a consistent upper bound
  - and we will show it later

MEDIAN FINDING

- Since the naïve algorithm (simple sorts the numbers) is in $O(n \log n)$, it makes sense to guess a lower time complexity to see if it works. We will start with guessing $O(n)$.

- In other words, we assume $T(n) \leq cn$ where c is a constant.

MEDIAN FINDING

- Proposition: The divide-and-conquer median-finding algorithm runs in $O(n)$ time.
- Proof:

(Base case): For an input size of 1, we can complete the median finding in time c by making it sufficiently large, such that $T(1) \le c * 1 = c$.

(Induction): We assume running time of the algorithm $T(k) \le ck$ for $1 \le k < n$. Now, substitute the assumption to the running time formula and we get:

$$T(n) = T\left(\frac{n}{5}\right) + T\left(\frac{7n}{10}\right) + O(n) \le \frac{cn}{5} + \frac{7cn}{10} + an = \frac{9cn}{10} + an = cn + \left(a - \frac{c}{10}\right)n$$

Since a is a constant (which associated with the running time for the partitioning process around x), we can make c a large-enough constant where $c > 10a$ and $\left(a - \frac{c}{10}\right)n$ be negative.

MEDIAN FINDING

- Proof (cont.): If $\left(a - \frac{c}{10}\right)n$ is negative, then

$$T(n) \leq cn + \left(a - \frac{c}{10}\right)n \leq cn$$

which is consistent with our hypothesis at the beginning.

(Conclusion): Hence, the algorithm runs in $O(n)$ time.

MEDIAN FINDING

- Now, we will show, if we did not guess the time complexity right, it will not lead to a consistent upper bound.
  - note that the big-O notation corresponds to upper bound; and in theory if we guess a higher-than-necessary time complexity it is still correct, just not tight
  - so, we will guess a lower time complexity, say $O(1)$

MEDIAN FINDING

- Assuming the algorithm runs in $O(1)$ and $T(n) \leq c$ for some constant c, we have:

$$T(n) \leq T\left(\frac{n}{5}\right) + T\left(\frac{7n}{10}\right) + O(n) \leq c + c + an = 2c + an$$

- The formula clearly indicates an $O(n)$ time complexity, no matter how we manipulate the constants a and c. The $O(n)$ complexity is inconsistent with what we have guessed, i.e., $O(1)$; hence, the algorithm does not run in $O(1)$.

- Note: this is merely a case to show that if we guess wrong, we will not be able to complete the proof; and hence <u>the substitution method is not a circular reasoning</u>.

MEDIAN FINDING

- Finally, let’s figure out why we would partition the set of number into groups of 5. We shall redo the efficiency analysis while assuming that the group size is a variable d.

- <u>Assume d is an odd number</u>, then we can write the running time as:

$$T(n) = T\left(\frac{n}{d}\right) + T\left(n - \frac{d+1}{2} * \frac{n}{2d}\right) + O(n) = T\left(\frac{n}{d}\right) + T\left(\frac{(3d-1)n}{4d}\right) + O(n)$$
$$\leq \frac{cn}{d} + \frac{(3d-1)cn}{4d} + an = \frac{(3d+3)cn}{4d} + an = cn + (a - \frac{(d-3)c}{4d})n$$

MEDIAN FINDING

- To make $cn + (a - \frac{(d-3)c}{4d})n$ a proper upper bound, we shall make $(a - \frac{(d-3)c}{4d})n$ negative. In other words, $\frac{(d-3)c}{4d}$ should be positive.

- It is clear that $d > 3$. And since we assumed d is an odd number, the smallest value d can take is 5.

- When $\frac{(d-3)c}{4d}$ is positive, we can make c large enough such that $(a - \frac{(d-3)c}{4d})n$ becomes negative, and thus make the algorithm remain in $O(n)$ time.

MEDIAN FINDING

- In another case, when d is an even number, we can perform a similar analysis:

$$T(n) = T\left(\frac{n}{d}\right) + T\left(n - \frac{d}{2} * \frac{n}{2d}\right) + O(n) = T\left(\frac{n}{d}\right) + T\left(\frac{3dn}{4d}\right) + O(n) \le \frac{cn}{d} + \frac{3dcn}{4d} + an$$
$$= \frac{(3d+4)cn}{4d} + an = cn + (a - \frac{(d-4)c}{4d})n$$

- Similarly we must have $d > 4$ if d is an even number. Taken together, $d = 5$ works as the smallest choice that makes the algorithm run in $O(n)$ time.
- Why don't we make d as large as possible? This is because we will still need to find the median among the d numbers. Even through d is treated as a constant, in practice finding the median among many numbers is time consuming. Hence we should use small d if possible.

SUMMARY

- Subproblems and recursion.
- Divide and conquer algorithms:
  - number of subproblems (a)
  - size of subproblems (b)
  - how to combine the solutions of subproblems (c)
- Examples:
  - Computing Fibonacci number (unrolling the recursion)
  - Merge sort (solving closed form and the master's theorem)
  - Median finding (uneven subproblem division)