CS 6212 DESIGN AND ANALYSIS OF ALGORITHMS

LECTURE: DIVIDE & CONQUER – PART II

Instructor: Abdou Youssef

OBJECTIVES OF THIS LECTURE

By the end of this lecture, you will be able to:

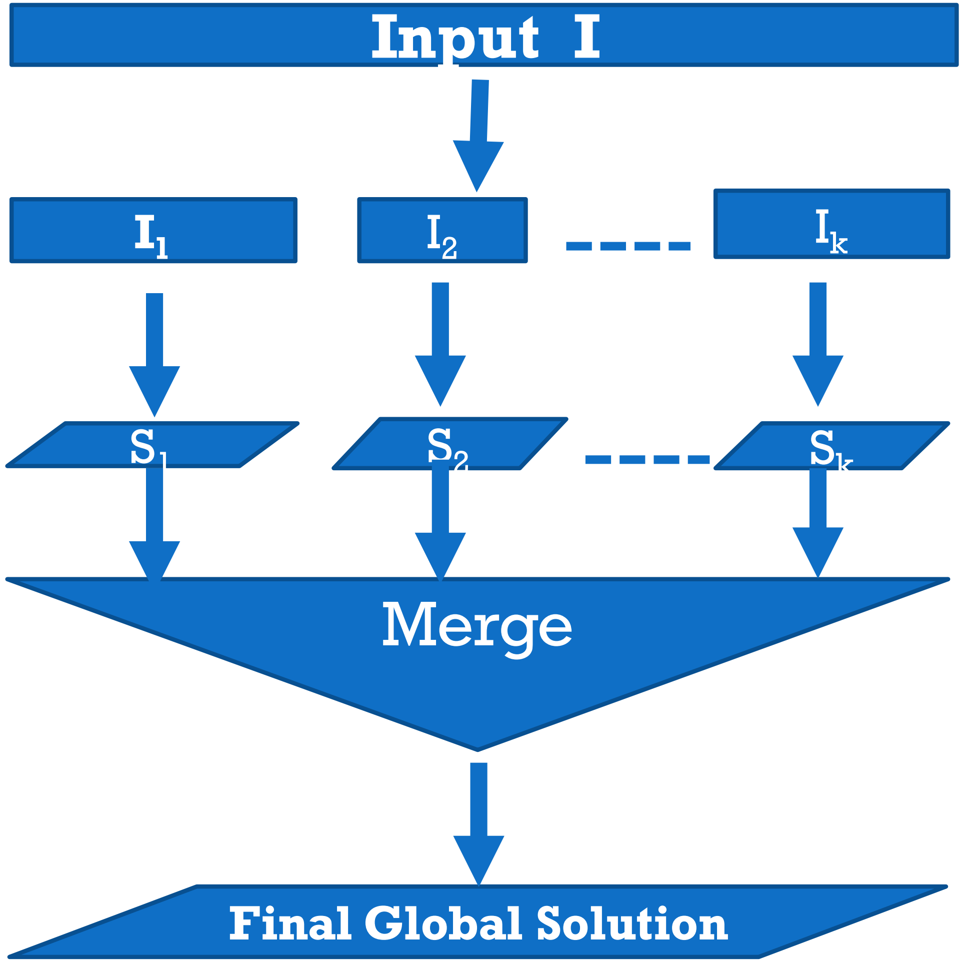
- Apply the Divide & Conquer technique in more elaborate ways to design an algorithm for selecting elements of arbitrary ranks from an input array
- Carry out more involved time complexity analysis using induction

OUTLINE

- Third application of Divide and Conquer: Order Statistics (i.e., finding the kth smallest element in an array)
  - Basic D&C approach
  - More advanced D&C approach
  - Detailed time complexity analysis

DIVIDE & CONQUER
-- TEMPLATE REMINDER --

```
Template divide&conquer (input I)
begin
    if (size or value of input is small enough)
    then
        solve directly and return;
    endif
    divide input I into two or more parts $I_1, I_2, \ldots$;
    $S_1 \leftarrow$ divide&conquer($I_1$) ;
    $S_2 \leftarrow$ divide&conquer($I_2$) ;
    ..................
    Merge the subsolutions $S_1, S_2, \ldots$ into a
        global solution S;
end
```



Input I

I_1 I_2 ---- I_k

S_1 S_2 ---- S_k

Merge

Final Global Solution

DIVIDE & CONQUER

-- THE ORDER STATISTICS PROBLEM --

- Problem:
    - **Input**: An arbitrary array A[1:n] of comparable data (i.e., has a comparator like $\leq$), and an integer k $(1 \leq k \leq n)$
    - **Output**: The k^{th} smallest element of the array A
    - **Task**: Develop a D&C algorithm for finding the k^{th} smallest element of input array A

THE ORDER STATISTICS PROBLEM

-- SPECIAL CASES --

- If $k = 1$, the problem reduces to finding the minimum
- If $k = n$, the problem reduces to finding the maximum
- If $k = \frac{n}{2}$, the problem reduces to finding the median
- So, the Order Statistics problem is a generalization of finding the min/max/median problem (to find the k^{th} smallest for arbitrary k)

THE ORDER STATISTICS PROBLEM

-- TIME COMPLEXITY CONSIDERATIONS (1) --

- Finding the min or max can be done in O(n) time
    - Scan the array left to right,
      keeping track of the min/max so far

```
M=A[1];
for i=2 to n do
    M=min(M,A[i]);
endfor;
return M;
```

- If $k = \frac{n}{2}$, the problem reduces to finding the median
    - Can it be done in O(n) time?
- In general, given that the min/max can be found in O(n), can we find the k^{th} smallest in O(n) time no matter what k is?

THE ORDER STATISTICS PROBLEM

-- TIME COMPLEXITY CONSIDERATIONS (2) --

- "Tempting" solution:
  - Sort the array; k^{th}smallest is in the k^{th}position of the sorted array
  - But that takes $O(n\log n)$ time.
- Given the higher $O(n\log n)$ cost of the sorting-based solution, is it reasonable to still hope for an O(n) algorithm?
- Well, the sorting-based solution does too much work: it can find not only the k^{th} smallest value for the given k, but also, the 1st smallest, the 2nd smallest, the 3rd smallest, … .
- So, there is a possibility for an O(n) algorithm that finds the k^{th} smallest for just the given k

THE ORDER STATISTICS PROBLEM

-- A FIRST D&C ATTEMPT --

```
function select(A[1:n],k)  // returns the $k^{th}$ smallest value of A
// it is assumed that $1 \leq k \leq n$
begin
  if n==1 then            // k must then be 1
        return (A[1]);
  endif
  r := partition(A[1:n],1,n);   // same as in Quicksort
  case
    k=r:        return (A[r]);
    k < r:      return (select(A[1:r-1],k));
    k > r:      return (select(A[r+1,n],k-r));  // why k-r, not k
  endcase
end
```

CYU

- Why k-r instead of k in the case k>r

TIME COMPLEXITY OF SELECT

- $T(n) = \max(\text{the times of the the 3 cases}) + \text{partition time}$
- $T(n) = \max\big(c, T(r-1), T(n-r)\big) + cn$
- $\boldsymbol{T(n) = \max\big(T(r-1), T(n-r)\big) + cn}$
    - because $c \leq T(r-1)$ and $T(n-r)$
- The value r is unknown (can be any value in $1{:}\,n$), so it is not possible to solve that recurrence relation. Instead, we can do one or both of the following:
    - Worst-case time complexity
    - Average-case time complexity

TIME COMPLEXITY OF SELECT

-- WORST-CASE TIME COMPLEXITY --

- $\boldsymbol{T(n) = \max\big(T(r-1), T(n-r)\big) + cn}$
- Worst case in D&C happens when the partitioned data is extremely unbalanced: (one part is empty, the other part full)
- In that case, $r = 1$, i.e., $r - 1 = 0$, or $r = n$
- Either way, we'll have: $T(n) = \max\big(T(0), T(n-1)\big) + cn$, that is,
- $\boldsymbol{T(n) = T(n-1) + cn}$
- We saw that recurrence relation in Quicksort: $\boldsymbol{T(n) = O(n^2)}$
- That is a shock: we were trying to beat $O(n \log n)$, but got $O(n^2)$

TIME COMPLEXITY OF SELECT
-- AVERAGE-CASE TIME COMPLEXITY --

- $T(n) = \max\big(T(r-1), T(n-r)\big) + cn$
- Worst case is too expensive and too extreme
- Average-case is more representative
- We will not do it, but you are invited to carry out an average-case time complexity analysis (like the one for Quicksort)
- You will discover that average-case time: $T_A(n) = O(n\,log\,n)$
- **Better, but not good enough**

Remember, we're hoping for O(n)

THE ORDER STATISTICS PROBLEM

-- A SECOND D&C ATTEMPT: QUICKSELECT --

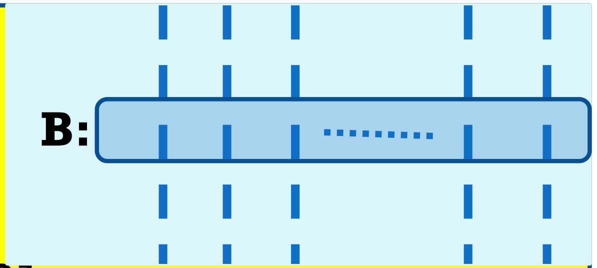
- Same as 1st attempt, but we "fix" partition to avoid **imbalance**

```
function QuickSelect(A[1:n],k)  // returns the $k^{th}$smallest of A
// it is assumed that $1 \leq k \leq n$
begin
  if n==1 then            // k must then be 1
        return (A[1]);
  endif
  r := wise_partition(A[1:n]);
  case
    k=r:        return (A[r]);
    k < r:      return (select(A[1:r-1],k));
    k > r:      return (select(A[r+1,n],k-r));  // why k-r, not k
  endcase
end
```

QUICKSELECT

-- WISE_PARTITION --

- Same as 1st attempt, but we “fix” partition to avoid **imbalance**
- How: Pick a better partitioning element than A[1]. How?

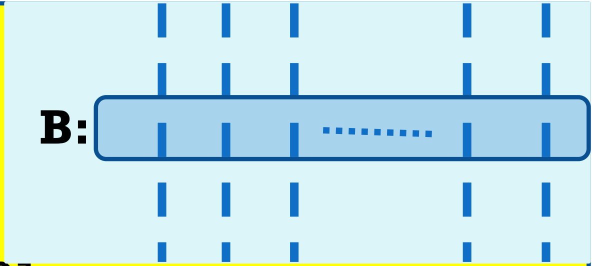


```
Function getWisePartitioningElement(A[1:n])
begin
    int m=n/5;  // integer division
    Divide the array into groups of five each:   A[1:5], A[6:10],...;
    Sort each group; // Now A[1:5] is sorted, A[6:10] is sorted, ...
    B[1:m] := the array of the middles of  the sorted groups: B=A[3],A[8], A[13],...
    mm = the median of B, that is, the m/2-th smallest element of  B[1:m];
    Group discussions: How to find the median of B?
    return (mm);
end
```

QUICKSELECT

-- WISE_PARTITION --

- Same as 1st attempt, but we "fix" partition to avoid **imbalance**
- How: Pick a better partitioning element than A[1]. How?



```
Function getWisePartitioningElement(A[1:n])
begin
    int m=n/5;  // integer division
    Divide the array into groups of five each:   A[1:5], A[6:10],…;
    Sort each group;  // Now A[1:5] is sorted, A[6:10] is sorted, …
    B[1:m] := the array of the middles of  the sorted groups: B=A[3],A[8], A[13],…
    //find the median of B, i.e., the m/2-th smallest  of B
    mm :=  Quickselect(B[1:m],m/2);
    return (mm);
end
```

QUICKSELECT
-- ALGORITHM --

```
function QuickSelect(A[1:n],k)  // returns the $k^{th}$ smallest of A
// it is assumed that $1 \leq k \leq n$
begin
   if n==1 then     // k must be 1
      return (A[1]);
   endif
   r := wise_partition(A[1:n]);
   case
      k=r:        return (A[r]);
      k < r:      return (QuickSelect(A[1:r-1],k));
      k > r:      return (QuickSelect(A[r+1,n],k-r));  // why k-r, not k
   endcase
end
```

```
wise_partition(A[1:n]):
  • mm=getWisePartitioningElement(A[1:n])
  • Swap A[1] with mm
  • r=partition(A[1:n])
  • return (r)
```

```
getWisePartitioningElement(A[1:n])
  •  Get B[1:m]; // O(n) time
  •  mm :=  Quickselect(B[1:m],m/2);
```

QUICKSELECT

-- TIME COMPLEXITY ANALYSIS (1/9) --

- Time of QuickSelect $T(n)$ satisfies:

$$T(n) = \max\big(T(r-1), T(n-r)\big) + \text{time of wise\_partition}$$

- Time of wise_partition = time of QuickSelect(B,m/2) +

  time of partition

$$= T\left(\frac{n}{5}\right) + cn$$

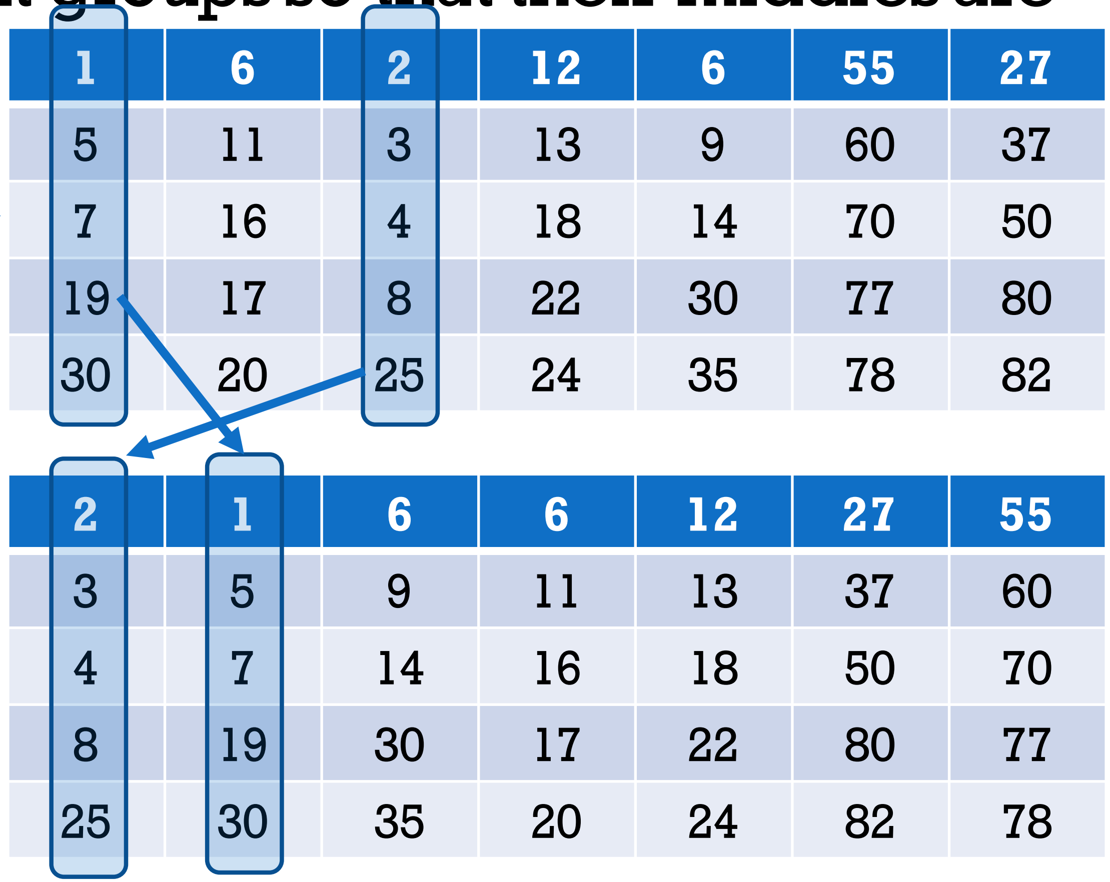
- Therefore, $\boldsymbol{T(n) = max\big(T(r-1), T(n-r)\big) + T\left(\frac{n}{5}\right) + cn}$
- **Theorem**: Due to wise-partitioning, $\frac{n}{4} \leq r \leq \frac{3n}{4}$.

QUICKSELECT

-- TIME COMPLEXITY ANALYSIS (2/9) --

- **Proof**: Re-arrange the 5-element groups so that their middles are sorted
- Example:

groups (columns) before re-arranging



1	6	2	12	6	55	27
5	11	3	13	9	60	37
7	16	4	18	14	70	50
19	17	8	22	30	77	80
30	20	25	24	35	78	82

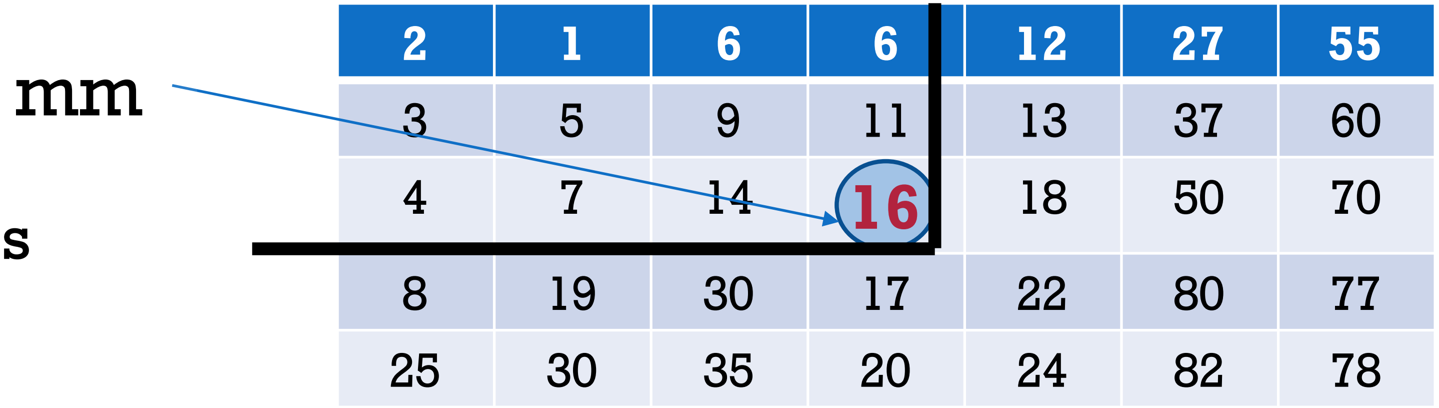
groups after re-arranging

2	1	6	6	12	27	55
3	5	9	11	13	37	60
4	7	14	16	18	50	70
8	19	30	17	22	80	77
25	30	35	20	24	82	78

QUICKSELECT

-- TIME COMPLEXITY ANALYSIS (3/9) --

- Now the median of B is in the middle of the middle column (see example)



2	1	6	6	12	27	55
3	5	9	11	13	37	60
4	7	14	16	18	50	70
8	19	30	17	22	80	77
25	30	35	20	24	82	78

- The middle column is

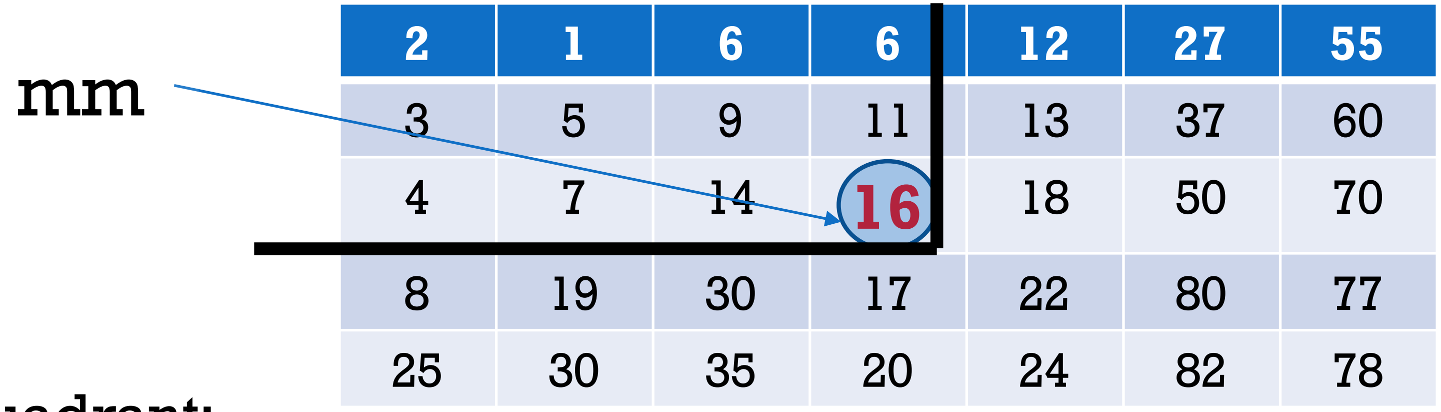
  column m/2

QUICKSELECT
-- TIME COMPLEXITY ANALYSIS (4/9) --

- Now the median of B is in the middle of the middle column (see example)

- The middle column is

  column m/2



2	1	6	6	12	27	55
3	5	9	11	13	37	60
4	7	14	16	18	50	70
8	19	30	17	22	80	77
25	30	35	20	24	82	78

- Consider the top left quadrant:
  - All its numbers $\le mm$
  - Number of elements in that quadrant $= 3\frac{m}{2} = 3\frac{n/5}{2} = \frac{3n}{10} > \frac{n}{4}$

Therefore, all that quadrant except mm will go to the left part of A after partition(A[1:n])

- Since the left part after partitioning A has $r-1$ element and contains all the top left quadrant (except mm), we conclude that $r-1 \ge 3\frac{m}{2} - 1 \ge \frac{n}{4} - 1 \Rightarrow r \ge \frac{n}{4}$

QUICKSELECT
-- TIME COMPLEXITY ANALYSIS (5/9) --

- So far we have $r \geq \frac{n}{4}$, that is, $\boldsymbol{\frac{n}{4} \leq r}$
- Now, if you analogously consider the bottom right quadrant, you find that its size is $> \frac{n}{4}$, and all its elements (except for mm) are > mm, and hence all of it (except mm) goes to the right part of A after partitioning around mm
- Therefore, $n - r \geq size\ of\ bottom\ right\ quadrant - 1 > \frac{n}{4} - 1$
- Thus, $n - r > \frac{n}{4} - 1 \Rightarrow n - r \geq \frac{n}{4} \Rightarrow \boldsymbol{r \leq \frac{3n}{4}}$
- This completes the proof that $\boldsymbol{\frac{n}{4} \leq r \leq} \frac{3\mathrm{n}}{4}$. **Q.E.D.**

QUICKSELECT

-- TIME COMPLEXITY ANALYSIS (6/9) --

- **Theorem:** The time complexity $T(n)$ of QuickSelect(A[1:n],k) satisfies: $T(n) \leq T\left(\frac{3n}{4}\right) + T\left(\frac{n}{5}\right) + cn.$
- **Proof:**
  a. Recall $T(n) = \max\big(T(r-1), T(n-r)\big) + T\left(\frac{n}{5}\right) + cn$
  b. $T(r-1) \leq T(\frac{3n}{4})$ because $r - 1 < r \leq \frac{3n}{4}$
  c. $T(n-r) \leq T(\frac{3n}{4})$ because $\frac{n}{4} \leq r \Rightarrow n - r \leq \frac{3n}{4}$
  d. Therefore, $\max\big(T(r-1), T(n-r)\big) \leq T(\frac{3n}{4})$ using (b) and (c)
  e. Thus, $T(n) \leq T\left(\frac{3n}{4}\right) + T\left(\frac{n}{5}\right) + cn$ using (a) and (d). Q.E.D.

$$\boldsymbol{\frac{n}{4}} \leq \boldsymbol{r} \leq \frac{3\mathrm{n}}{4}$$

QUICKSELECT
-- TIME COMPLEXITY ANALYSIS (7/9) --

- **Theorem**: The time complexity $T(n)$ of QuickSelect(A[1:n],k) satisfies: $T(n) \leq 20cn$.
- **Proof**: By induction on n.
  - Basis steps: for n=1. Need to prove $T(1) \leq 20c1$.

    Well, $T(1) = c < 20c$.
  - Induction step: Assume $T(m) \leq 20cm\ \forall m < n$. (This is called induction hypothesis (I.H.))

    Prove $T(n) \leq 20cn$.

    Recall $T(n) \leq T\left(\frac{3n}{4}\right) + T\left(\frac{n}{5}\right) + cn$ (from last theorem)

QUICKSELECT

-- TIME COMPLEXITY ANALYSIS (8/9) --

- **Theorem**: The time complexity $T(n)$ of QuickSelect(A[1:n],k) satisfies: $T(n) \leq 20cn$.
- **Proof**: … induction step next.

Prove $T(n) \leq 20cn$, assuming $T(m) \leq 20cm\ \forall m < n$

Recall $T(n) \leq T\left(\frac{3n}{4}\right) + T\left(\frac{n}{5}\right) + cn$

Applying I.H. $T(m) \leq 20cm$ on $m = \frac{3n}{4} < n$, we get: $T\left(\frac{3n}{4}\right) \leq 20c\frac{3n}{4} = 15cn$

Applying I.H. $T(m) \leq 20cm$ on $m = \frac{n}{5} < n$, we get: $T\left(\frac{n}{5}\right) \leq \frac{20cn}{5} = 4cn$

Therefore, $T(n) \leq T\left(\frac{3n}{4}\right) + T\left(\frac{n}{5}\right) + cn \leq 15cn + 4cn + cn = 20cn.$ Q.E.D.

QUICKSELECT

-- TIME COMPLEXITY ANALYSIS (9/9) --

- By the last theorem, $T(n) \leq 20cn$ and thus $T(n) = O(n)$ because $20c$ is a constant.
- Therefore, **QuickSelect takes $O(n)$ time**.
- Success!

DIVIDE AND CONQUER RECAP

- We saw how D&C works
- We saw several applications of it
- We carried out several time complexity analyses, including average case and worst case analyses
- In all cases, an intermediate recurrence relation for the time was derived and solved
- There are many more applications of D&C
- Although there are many other algorithmic design techniques, D&C is one of the first techniques that algorithm designers try when they want to solve non-trivial computational problems

A FEW OTHER QUICK D&C APPLICATIONS

-- BINARY SEARCH IN A SORTED ARRAY --

- Input: A **sorted** array $X[1:n]$ and a number a
- Output: Whether a is in $X[1:n]$, and if so, find k where
- Function BinarySearch($X[1:n]$, a) {

```
If(n==1)
        if(X[1]==a) return 1;
        else return -1;
Endif
If($a == X\left[\frac{n}{2}\right]$)Return (k);
Else if ($a < X\left[\frac{n}{2}\right]$)
        BinarySearch($X[1:\frac{n}{2}-1]$, $a$) ;
Else
        BinarySearch($X[\frac{n}{2}+1:n]$, $a$) ;
Endif
}
```

- Here is a situation where after the data is split into two halves, the algorithm is called on only one half.
- Therefore, a D&C algorithm need not call itself on each part of the split data

- Time Complexity T(n)=?; Assume $n=2^k$

  $T(n)=T(n/2)+c$
  $T(n/2)=T(n/4)+c$
  $T(n/4)=T(n/8)+c$
  …
  $T(n/2^{k-1})=T(n/2^k)+c$
  Add and simplify: we get:
  $T(n)=T(n/2^k)+c+c+\ldots+c=T(1)+ck$
  $T(n)=T(1)+c\log n = O(\log n)$

A FEW OTHER QUICK D&C APPLICATIONS
-- POWER x^n--

- Input: A numerical value x and a non-negative integer n
- Output: the value of x^n

Simple method:

```
Func pow(x,n){
    y=1;
    For i=1 to n do
        y=y*x;
    Endfor
    Return y;
}
```

Time: T(n)=O(n)

D&C method:

```
Func pow(x,n){
    If (n==0) return 1;
    Else if (n==1) return x;
    Else
        z=pow(x,n/2);
        y=z*z;
        If (n is odd) y=y*x;
        Return y;
    Endif
}
```

Time:
T(n)=T(n/2)+c
Therefore,
T(n)=O(log n);

A FEW OTHER QUICK D&C APPLICATIONS
-- POLYNOMIAL EVALUATION (1/3) --

- Input:
  - A polynomial $P(x) = a_0 + a_1 x + a_2 x^2 + a_3 x^3 + \cdots + a_{n-1}\, x^{n-1}$ represented simply by the array $a[0{:}\, n-1]$
  - A number x // example 2, 5, 3.6, etc.
- Output: the value of $P(x)$ evaluated at the input value of x
- D&C method:
  - Begin with the partitioning of the input into two halves, next
  - Let $m = \frac{n}{2}$
  - $P(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_{m-1}\, x^{m-1} + a_m x^m + \cdots + a_{n-1}\, x^{n-1}$

A FEW OTHER QUICK D&C APPLICATIONS

-- POLYNOMIAL EVALUATION (2/3) --

- D&C method:
  - Let $m = \frac{n}{2}$
  - $P(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_{m-1}\, x^{m-1} + a_m x^m + \cdots + a_{n-1}\, x^{n-1}$
  - $P(x) = [a_0 + a_1 x + a_2 x^2 + \cdots + a_{m-1}\, x^{m-1}] + [a_m x^m + a_{m+1} x^{m+1} + \cdots \ldots + a_{n-1}\, x^{n-1}]$
  - $P(x) = [a_0 + a_1 x + a_2 x^2 + \cdots + a_{m-1}\, x^{m-1}] + x^m [a_m + a_{m+1} x^1 + \cdots + a_{n-1}\, x^{n-m-1}]$
  - $P(x) = \qquad Q(x) \qquad + x^m R(x)$

    Where $Q(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_{m-1}\, x^{m-1}$, represented by $a[0{:}\, m-1]$

    and $R(m) = a_m + a_{m+1} x^1 + \cdots + a_{n-1}\, x^{n-m-1}$, represented by $a[m{:}\, n-1]$
  - Now we can call the algorithm recursively on $Q(x)$ and $R(x)$
  - Merging: compute x^m and then $P(x) = Q(x) + x^m R(x)$ and return $P(x)$;

A FEW OTHER QUICK D&C APPLICATIONS
-- POLYNOMIAL EVALUATION (3/3) --

```
Function Poly(a[0,n-1] , x)
Begin
    If (n=0) then
        return a[0];
    Else
        m=n/2;
        Q=Poly(a[0,m-1], x);
        R=Poly([0,a[m,n-1]], x)
        y=pow(x,m);
        Return Q+y*R;
    Endif
End Poly
```

- Time Complexity analysis:
  //Assume we have already computed the
  // powers of x: $x, x^2, x^3, \ldots$ in O(n) time
- T(n)=2T(n/2)+c
- Therefore, T(n)=O(n)

- Could we compute the polynomial in a straightforward way (without D&C) in O(n) time?
- Yes (An exercise)
- So why bother with D&C (see later)

ANY ADVANTAGE TO D&C WHEN SIMPLER METHODS ARE AS FAST? (1/2)

- We just saw that computing a polynomial can be done in a simple fashion in O(n) time, i.e., as fast as the D&C method
- Similarly, for a given array X[1:n], you can find its min, max, and sum:
  - in a simple fashion in O(n) time, and also
  - using D&C in O(n) time
- Is there any advantage to using D&C in such situations?

ANY ADVANTAGE TO D&C WHEN SIMPLER METHODS ARE AS FAST? (2/2)

- Is there any advantage to using D&C in such situations?
- Answer: YES, YES
  - Suppose you have a multicore machine (of many processors)
  - The D&C method produces an algorithm where the (recursive) calls on the subparts of the data can be executed in parallel (i.e., simultaneously) on the different cores
  - This results in some serious speedup of the algorithm, by a factor of k, where k is the number of cores
- The other, simpler methods may be too serial, i.e., **unsplittable** into processes that can utilize the different cores simultaneously

ANOTHER “KILLER” APPLICATION OF D&C
-- DISCRETE FOURIER TRANSFORM (1/5) --

- The Discrete Fourier Transform (DFT)
- It transforms an input column vector X (i.e., array) of length n to another (output) column vector Y (of the same length n)
  - by multiplying X by a specific $n \times n$ matrix A
  - That is, $Y = AX$
- You probably have learned matrix multiplication, but if not, we will cover it later in the semester
- For now, suffice it to say that the transform (AX) takes O(n^2) time
- For n fairly large (in the thousands/millions), O(n^2) is quite slow

ANOTHER "KILLER" APPLICATION OF D&C
-- DISCRETE FOURIER TRANSFORM (2/5) --

- The Discrete Fourier Transform (DFT)
- The transform (AX) takes O(n^2) time
- For n fairly large (in the thousands/millions), O(n^2) is quite slow
- In practice, the DFT is used to
  - Filter signals like audio (phone calls, Radio, TV) and video (TV)
  - Reduce/eliminate noise in phone calls, radio/TV broadcast, etc.
- Such things have to be done in **real time** (e.g., during a phone call), on small devices (e.g., smart/regular phones, car radio)
- Therefore, they must on "small computers" yet fast enough for real-time user experience

ANOTHER "KILLER" APPLICATION OF D&C
-- DISCRETE FOURIER TRANSFORM (3/5) --

- Such things (Filtering, noise removal) have to be done in **real time** (e.g., during a phone call), on small devices (e.g., smart/regular phones, car radio)
- Therefore, they must on "small computers" yet fast enough for real-time user experience
- The lengths of such digital signals, even when divided into short chunks (like one second worth of digital sound), are in the 10,000's of numbers per chunk
- Transforming such chunks using DFT take 100M's operations/chunk if done the straightforward way
- Also, every chunk has to be processed in less than one second in order for the filtering and the ongoing phone call to [proceed hand-in-hand
- No phone/radio can perform at such a speed in real time, especially in older years

ANOTHER "KILLER" APPLICATION OF D&C
-- DISCRETE FOURIER TRANSFORM (4/5) --

- No phone/radio can perform at such a speed in real time, especially in older years
- Therefore, scientists/engineers had to find a faster algorithm for DFT
- Such an algorithm was found (by **Cooley** and **Tukey)** in 1965
    - The algorithm is **Divide & Conquer** algorithm!
    - It takes $O(n \log n)$ time, which, as we have seen multiple times already, is much faster than $O(n^2)$
    - The algorithm is referred to as Fast Fourier Transform (FFT)

ANOTHER "KILLER" APPLICATION OF D&C
-- DISCRETE FOURIER TRANSFORM (5/5) --

- The Fast Fourier Transform (FFT)
- To understand the algorithm, you need to know:
  - Matrix multiplication
  - Complex numbers
  - A bit of Trigonometry
  - Divide & Conquer
- We will not cover it in this course, but if you're interested you can find much coverage on it on the Web
- Suffice it to say that FFT was a revolutionary discovery with great impact on modern electronic technology and engineering applications